



UNIVERSITY OF
GREENWICH

Alliance with FPT Education

UNIVERSITY OF GREENWICH

Final Year Project (COMP1682)

INDIVIDUAL ASSIGNMENT

Greenwich University Management System Project

Student Name	VU THANH TRUONG
Student ID	001479794
Student Group	TCH2702
Programming	Global Personal Competency
Academic Year	2025 - 2026
Instructor	NGUYEN HUU CHU
Submission Date	29 November 2025

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION.....	16
1.1 Background and Context	16
1.2 Problem Statement and Motivation	17
1.3 Aims and Objectives.....	17
1.3.1 Primary Objective.....	17
1.3.2 Secondary Objectives.....	18
1.3.3 Key Performance Indicators.....	19
1.4 Research Questions	19
1.5 Scope and Boundaries of the Project	20
1.6 Significance and Innovation	21
1.7 Structure of the Report.....	22
CHAPTER 2: LITERATURE REVIEW AND CRITICAL ANALYSIS	23
2.1 Overview of Domain Knowledge	23
2.2 Theoretical Framework.....	24
2.3 Review of Existing Systems/Solutions	25
2.3.1 Commercial Solutions	25
2.3.2 Academic Research	26
2.3.3 Open Source Alternatives	26
2.4 Comparative Analysis of Technologies	28

2.4.1 Frontend Technologies (e.g., React, Vue, Angular)	28
2.4.2 Backend Technologies (Spring Boot, Django, Node.js/Express, Laravel, ASP.NET Core)	31
2.4.3 Database Solutions (e.g., SQL, NoSQL options).....	34
2.4.4 Design patterns	37
2.4.5 Microservices vs. Monolithic Architecture.....	40
2.4.6 Project Structure Approaches.....	42
2.4.7 Integrated Development Environment.....	44
2.4.7 Persistence Management Approaches.....	47
2.4.8 Inheritance Mapping Strategies	50
2.4.9 Cloud Services and Deployment Options	53
2.5 Development Methodologies	55
2.5.1 Agile Frameworks (Scrum, Kanban)	55
2.5.2 DevOps Practices.....	56
2.5.3 Test-Driven Development	57
2.6 Gaps in Current Research/Solutions	58
2.7 Justification for Proposed Solution	59
2.8 Methodologies.....	60
2.9 Chapter Summary	62

CHAPTER 3: REQUIREMENTS ANALYSIS AND DESIGN SPECIFICATION 63

3.1 Stakeholder Analysis.....	63
3.2 System Requirements Gathering Methodology	64
3.3 Functional Requirements	65

3.3.1 Core Features	65
3.3.3 Use Cases with Acceptance Criteria.....	75
3.4 Non-Functional Requirements	80
3.4.1 Performance Requirements.....	86
3.4.2 Security Requirements	88
3.4.3 Scalability and Reliability	90
3.4.4 Usability and Accessibility	91
3.4.5 Compliance Requirements	93
3.5 Competitive Analysis.....	102
3.5.1 Features of Competing Solutions	102
3.5.2 SWOT Analysis.....	102
3.5.3 Market Differentiation Strategy	104
3.6 System Architecture Design	105
3.6.1 High-Level Architecture	105
3.6.2 Component Diagram	106
3.6.3 Deployment Diagram	110
3.6.4 System Interfaces	113
3.7 Database Design	115
3.7.1 Entity-Relationship Diagram	115
3.7.2 Data Dictionary	118
3.7.3 Database Schema	154
3.7.4 Data Migration Strategy	160
3.8 User Interface Design	163
3.8.1 Design Philosophy and Principles.....	163
3.8.2 Wireframes and Prototypes.....	164
3.8.3 User Journey Maps	229
3.8.4 Responsive Design Considerations	230

3.9 UML Modeling	231
3.9.1 System Architecture	231
3.9.2 Use Case Diagrams.....	233
3.9.3 Class Diagrams	246
3.9.4 Sequence Diagrams.....	247
3.9.5 Activity Diagrams.....	259
3.10 Algorithm and Data Structure Design	447
3.11 Chapter Summary	461

CHAPTER 4: IMPLEMENTATION AND DEVELOPMENT..... 462

4.1 Development Environment Setup.....	462
4.1.1 Tools and Technologies	462
4.1.2 Version Control Strategy	465
4.1.3 CI/CD Pipeline Configuration	466
4.2 Backend Implementation.....	468
4.2.1 Development	468
4.2.2 Business Logic Layer	498
4.2.3 Database Integration.....	504
4.2.4 Authentication and Authorization	508
4.3 Frontend Implementation	510
4.3.1 Component Architecture	510
4.3.2 State Management	515
4.3.3 UI/UX Implementation	521
4.3.4 Responsive Design Implementation	534
4.4 Integration of External Services/APIs	540

4.5 Performance Optimization Techniques.....	546
4.6 Security Implementation	553
4.6.1 Data Protection Measures	553
4.6.2 Vulnerability Prevention	554
4.9 Chapter Summary	560
CHAPTER 5: TESTING, VALIDATION, AND QUALITY ASSURANCE	561
CHAPTER 6: RESULTS AND EVALUATION	586
 6.1 Project Outcomes vs. Objectives	586
 6.2 Performance Evaluation.....	587
6.2.1 Benchmark Results.....	587
6.2.2 Scalability Assessment	588
6.2.3 Resource Utilization.....	589
 6.3 User Experience Evaluation.....	591
6.3.1 Usability Testing Results.....	591
6.3.2 User Satisfaction Metrics	593
6.3.3 Accessibility Compliance	594
 6.4 Business Value Assessment	596
 6.5 Comparative Analysis with Existing Solutions	597
 6.6 Critical Evaluation of the Solution.....	599
6.6.1 Strengths	599
6.6.2 Limitations.....	600
6.6.3 Areas for Improvement.....	601

6.7 CHAPTER SUMMARY 603

CHAPTER 7: PROJECT MANAGEMENT AND PROFESSIONAL PRACTICE 604

7.1 Project Management Methodology 604

7.2 PROJECT PLANNING 608

 7.2.1 Work Breakdown Structure 608

 7.2.2 Gantt Chart and Timeline 609

 7.2.3 Resource Allocation 610

7.3 Risk Management 612

 7.3.1 Risk Assessment 612

 7.3.2 Mitigation Strategies 613

 7.3.3 Contingency Planning 614

7.4 Change Management 615

7.5 Team Collaboration and Communication 617

7.6 Budget Management (if applicable) 618

7.7 Quality Management 618

7.8 Ethical Considerations 620

 7.8.1 Data Privacy and Protection 620

 7.8.2 Sustainability Aspects 620

 7.8.3 Social Impact Assessment 621

7.9 Reflection on Professional Development 622

7.10 Chapter Summary 623

CHAPTER 8: CONCLUSION AND FUTURE WORK 624

8.1 Summary of Achievements	624
8.2 Contribution to Knowledge	624
8.3 Lessons Learned	625
8.4 Critical Reflection	626
8.5 Limitations of the Current Implementation.....	627
8.6 Recommendations for Future Enhancements.....	627
8.6.1 Short-term Improvements	627
8.6.2 Long-term Research Directions	628
8.7 Implementation Roadmap	629
8.8 Final Conclusions	630
REFERENCES	631
APPENDICES.....	634

LIST OF FIGURES

Figure 1 Spring Boot logo (BAP Software, n.d.)	33
Figure 2 Official Windward Studios illustration from the article MySQL Reporting and Document Generation.	36
Figure 3 Illustration from the Stack Overflow question “Pros and cons of the use of the DAO pattern	39
Figure 4 Diagram illustrating the Monolithic Architecture model from the GeeksforGeeks article “Monolithic Architecture – System Design”	41
Figure 5 Illustration of the “Package by Feature” architecture approach	43
Figure 6 Illustration from the article IntelliJ IDE Gets New Free Features With Edition Merge.	46
Figure 7 Diagram from the Naukri Code360 article Spring Boot – JPA tutorial concepts & examples.	49
Figure 8 Illustration from the LinkedIn post “Inheritance in Hibernate / JPA”.	52

LIST OF DIAGRAMS

Diagram 1 University Management System – High-Level Logical Architecture	105
Diagram 2 University Management System - High-Level Component Architecture	106
Diagram 3 Academic Management Module - Detailed Component Architecture	107
Diagram 4 External Services Integration - OAuth2, Payments, Email, Meetings	109
Diagram 5 University Management System – Deployment 01: Production / Cloud Architecture	111
Diagram 6 University Management System – Deployment 02: Development / Local Environment	112
Diagram 7 Logical Entity-Relationship Diagram (Logical Data Model) in Crow's Foot notation	115
Diagram 8 Entity-Relationship Diagram Chen form (conceptual)	115
Diagram 9 Physical database schema diagram	154
Diagram 10 User Journey Maps	229
Diagram 11 System Architecture	231
Diagram 12 Admin Management and Academic Administration System — Use Case Model	233
Diagram 13 Staff Academic and Administration Management System — Use Case Model	235
Diagram 14 Deputy Staff Academic and Minor Program Management System — Use Case Model	237
Diagram 15 Major Lecturer Academic and Classroom Management System — Use Case Model	239
Diagram 16 Minor Lecturer Academic and Class Management System — Use Case Model	241
Diagram 17 Student Academic, Financial, and Support Management System — Use Case Model	243
Diagram 18 Parent Academic Monitoring and Child Management System — Use Case Model	245
Diagram 19 Class Diagram	246
Diagram 20 Sequence Diagram 01 – Google OAuth2 Authentication and Role-Based Login Flow	248
Diagram 21 Sequence Diagram 02 – Stripe Deposit Payment and Account Balance Update Flow	251
Diagram 22 Sequence Diagram 03 – Staff Onboarding, Validation, and Account Provisioning Flow	254
Diagram 23 Sequence Diagram 04 – Staff Editing, Validation, and Update Workflow	256
Diagram 24 Staff Search, Filtering, and Paginated Listing Flow	258
Diagram 25 Student Enrollment & Parent Registration Workflow – Three-Layer Architecture	260
Diagram 26 Student Information Update Workflow – Three-Layer Architecture	264
Diagram 27 Student Deletion Workflow – Three-Layer Architecture	268
Diagram 28 Simple Student Deletion Flow – Layered Architecture	272
Diagram 29 Student Listing Workflow – Three-Layer Architecture	274
Diagram 30 Staff Creation Workflow – Layered Architecture	278
Diagram 31 Staff Update Workflow – Layered Architecture	282
Diagram 32 Staff Listing Workflow – Layered Architecture	285

Diagram 33 Staff Search Workflow – Layered Architecture	289
Diagram 34 Deputy Staff Creation Workflow – Layered Architecture	294
Diagram 35 Deputy Staff Update Workflow – Layered Architecture	299
Diagram 36 Deputy Staff Listing Workflow – Layered Architecture	305
Diagram 37 Deputy Staff Search Workflow – Layered Architecture	310
Diagram 38 Deputy Staff Deletion Workflow – Layered Architecture	315
Diagram 39 Mentor Lecturer Creation Workflow – Layered Architecture	319
Diagram 40 Major Lecturer Deletion Workflow – Layered Architecture	324
Diagram 41 Major Lecturer Update Workflow – Layered Architecture	328
Diagram 42 Major Lecturer Search Workflow – Layered Architecture	333
Diagram 43 Minor Lecturer Creation Workflow – Layered Architecture	338
Diagram 44 Minor Lecturer Search Workflow	343
Diagram 45 Minor Lecturer Listing Workflow	347
Diagram 46 Minor Lecturer Deletion Workflow	351
Diagram 47 Minor Lecturer Update Workflow	355
Diagram 48 Student Update Workflow	359
Diagram 49 Student Listing Workflow	363
Diagram 50 Student Deletion Workflow	367
Diagram 51 Major Class Creation Workflow	371
Diagram 52 Major Class Update Workflow	375
Diagram 53 Major Class Listing Workflow	379
Diagram 54 Major Class Search Workflow	383
Diagram 55 Add Minor Lecturer Workflow	387
Diagram 56 Add Minor Class Complete Workflow	392
Diagram 57 List Minor Classes Workflow	396
Diagram 58 Load Minor Classes Complex View – Full Layered Activity Diagram	400
Diagram 59 Add Minor Class – Full Layered Activity Diagram	405
Diagram 60 Add Major Class Composite Flow – Full Layered Activity Diagram	409
Diagram 61 Search Specialized Classes Complete Flow – Activity Diagram	413
Diagram 62 Add Campus Complete Flow – Activity Diagram	418
Diagram 63 Edit Campus Complete Flow – Activity Diagram	422
Diagram 64 List Campuses Complete Flow – Activity Diagram	425
Diagram 65 Add Major Subject Complete Flow – Activity Diagram	427
Diagram 66 Edit Major Subject Complete Flow – Activity Diagram	431
Diagram 67 Add Minor Subject Complete Flow – Activity Diagram	441

Diagram 68 Work Breakdown Structure	608
Diagram 69 Gantt Chart and Timeline	609

LIST OF TABLES

Table 1 Comparative Analysis of Frontend Technologies	28
Table 2 Spring Boot, Thymeleaf and Bootstrap Technology Stack (Thumbnail Image)	29
Table 3 Technical and Architectural Comparison of Backend Frameworks	31
Table 4 Comparison Matrix of SQL and NoSQL Databases	34
Table 5 Comparative Analysis of Selected Design Patterns for GUMS	37
Table 6 Comparison Matrix – Microservices vs. Monolithic Architecture	40
Table 7 Comparison Matrix of Project Structure Approaches	42
Table 8 Comparison Matrix of IDE Options	44
Table 9 Comparison Matrix of Persistence Management Approaches	47
Table 10 Comparison Matrix of ORM Inheritance Strategies	50
Table 11 Comparison Matrix of Cloud Hosting and Deployment Options	53
Table 12 Core Features	65
Table 13 Data Dictionary	118
Table 14 Tools and Technologies Used in the Development Environment	462
Table 15 Overview of the Test Plan and Testing Strategy	561
Table 16 Key Strengths of the GUMS Solution	599
Table 17 Key Limitations of the GUMS Solution	600
Table 18 Evaluation Accounts and Access Credentials	635

LIST OF SCREENSHOTS

Screenshot 1 Code database Persons and Students	158
Screenshot 2 Code database Timetable and Slot	159
Screenshot 3 Code database subjects	160
Screenshot 4 Login	534
Screenshot 5 Staff home	535
Screenshot 6 Student List	535
Screenshot 7 Subjects List	536
Screenshot 8 Assign members	536
Screenshot 9 dashboard staff	537
Screenshot 10 Timetable	537
Screenshot 11 Detail timetable	538
Screenshot 12 Enter Scores	538
Screenshot 13 Messange	539
Screenshot 14 Classroom	539
Screenshot 15 Assignment detail	540

LIST OF WIREFRAMES

Wireframe 1 Wireframe of login page	164
Wireframe 2 Wireframes and Prototype of Admin Home page	165
Wireframe 3 Wireframe and Prototype of staff list page	166
Wireframe 4 Wireframe and Prototype of staff list page	167
Wireframe 5 Wireframe and Prototype of staff edit page	168
Wireframe 6 Wireframe and Prototype of Manage Tuition Fees page	169
Wireframe 7 Wireframe and Prototype of Manage Tuition Fees page	170
Wireframe 8 Wireframe and Prototype of Approve Subjects page	171
Wireframe 9 Wireframe and Prototype of Reference Tuition Fees page	172
Wireframe 10 Wireframe and Prototype of Reference Annual Re-study Fees page	173
Wireframe 11 Wireframe and Prototype of Majors Management page	174
Wireframe 12 Wireframe and Prototype of Majors Management page	175
Wireframe 13 Wireframe and Prototype of Majors Edit page	176
Wireframe 14 Wireframe and Prototype of Specializations Management page	177
Wireframe 15 Wireframe and Prototype of Specializations Management page	177
Wireframe 16 Wireframe and Prototype of Manage Curriculums page	178
Wireframe 17 Wireframe and Prototype of Manage Curriculums page	179
Wireframe 18 Wireframe and Prototype of Manage Scholarships by Year page	180
Wireframe 19 Wireframe and Prototype of Manage Scholarships by Year page	181
Wireframe 20 Wireframe and Prototype of Manage Academic Contracts page	182
Wireframe 21 Wireframe of Campuses Management page	183
Wireframe 22 Wireframe of Campuses Management page	183
Wireframe 23 Wireframe and Prototype of Email Templates Management page	184
Wireframe 24 Wireframe and Prototype of Email Templates Management page	185
Wireframe 25 Wireframe and Prototype of Email Templates Edit page	186
Wireframe 26 Wireframe and Prototype of Support Tickets List page	187
Wireframe 27 Wireframe and Prototype of Support Tickets Edit page	188
Wireframe 28 Wireframe and Prototype of Colleagues List page	189
Wireframe 29 Wireframe and Prototype of Colleagues List page	190
Wireframe 30 Wireframe and Prototype of Admin Personal page	191
Wireframe 31 Wireframe and Prototype of Admin Personal page	192
Wireframe 32 Wireframe and Prototype of Staff Home page	193

Wireframe 33 Wireframe and Prototype of Lecturer List page	194
Wireframe 34 Wireframe and Prototype of Lecturer List page	194
Wireframe 35 Wireframe and Prototype of Lecturer Edit page	195
Wireframe 36 Wireframe and Prototype of Lecturer Specializations page	196
Wireframe 37 Wireframe and Prototype of Students List page	196
Wireframe 38 Wireframe and Prototype of Students List page	197
Wireframe 39 Wireframe of Students Edit page	198
Wireframe 40 Wireframe of Students Edit page	199
Wireframe 41 Wireframe of Students View Timetable page	200
Wireframe 42 Wireframe of Students View Academic Transcript page	201
Wireframe 43 Wireframe of Students View Academic Transcript page	202
Wireframe 44 Wireframe of Students View Detail Scores page	202
Wireframe 45 Wireframe of Subjects List page	203
Wireframe 46 Wireframe of Subjects List page	203
Wireframe 47 Wireframe of Syllabuses List page	204
Wireframe 48 Wireframe of Syllabuses List page	204
Wireframe 49 Wireframe of Study Plan page	205
Wireframe 50 Wireframe of Assign Members page	206
Wireframe 51 Wireframe of Classes List page	207
Wireframe 52 Wireframe of Classes List page	208
Wireframe 53 Wireframe of Major Timetable page	209
Wireframe 54 Wireframe of Major Timetable Detail page	210
Wireframe 55 Wireframe of Member Arrangement page	211
Wireframe 56 Wireframe of Award Scholarships page	212
Wireframe 57 Wireframe of Student Evaluate Major List page	213
Wireframe 58 Wireframe and Prototype of News Edit page	214
Wireframe 59 Wireframe and Prototype of Ticket Approval page	215
Wireframe 60 Wireframe and Prototype of News List page	216
Wireframe 61 Wireframe and Prototype of News List page	216
Wireframe 62 Wireframe and Prototype of All Timetable page	217
Wireframe 63 Wireframe and Prototype of Staff Personal page	218
Wireframe 64 Wireframe and Prototype of Student Home page	219
Wireframe 65 Wireframe and Prototype of Student Classes List page	220
Wireframe 66 Wireframe and Prototype of Classroom page	221
Wireframe 67 Wireframe and Prototype of Classroom page	222

Wireframe 68 Wireframe and Prototype of Classroom page	223
Wireframe 69 Wireframe and Prototype of Assignment Details page	224
Wireframe 70 Wireframe and Prototype of Student Assignment Details page	225
Wireframe 71 Wireframe and Prototype of Major Lecturer Home page	226
Wireframe 72 Wireframe and Prototype of Notifications Student page	227
Wireframe 73 Wireframe and Prototype of Messages page	228
Wireframe 74 Wireframe and Prototype of Parent Account page	229

CHAPTER 1: INTRODUCTION

1.1 Background and Context

In the modern higher education support environment, especially international joint programs, information technology plays a vital role in the support of teaching, learning and academic management activities. Greenwich Vietnam - a joint program between the University of Greenwich (UK) and FPT University (Vietnam) - has developed many online systems to serve the learning and academic management needs of students, trainees and parents. Among them, most important platforms today are AP (Academic Portal), CMS (Course Management System) and FLM (Faculty Learning Management). Each system has to take on a specific role: AP is the official academic portal at ap.greenwich.edu.vn, where students can follow their schedule, grade, course material and important announcement related to the program; CMS is a course management platform where students distribute content, assignments, organize discussions and provide students a centralised online learning space; While FLM at flm.greenwich.edu.vn guides the management needs of the candidates, including classroom management, tracking the progress of student learning as well as managing the allocation of teaching resources. Although these three systems have been offering a lot of support for training activities, there have been some limitations because of the separate operation. Students need to log in and work on many different platforms in order to access their schedules, documents, grades and learning activities; lecturers also need to switch back and forth between CMS and FLMs in order to manage courses and classes. This fragmentation doesn't only lead to problems for users but also to a lack of efficiency in learning data management, greater maintenance costs and more complex operations. In the view of global education moving strongly towards integrating digital technology and creating unified online learning ecosystems, the necessity to merge the three systems AP, CMS and FLM into one platform for Greenwich Vietnam is an inevitable step. This integrated solution not only helps to optimise user experience, by concentrating all functions in a single interface, but also helps to improve management efficiency, ensure the smoothness of storage - processing - exploitation of learning data, and opens up flexible expansion possibilities in the future. This is the important context that lays the foundation for the research topic and implementation of the unified web system in this project. From that reality the research and implementation of a unified system, dubbed Greenwich University Management System (GUMS) was proposed as a strategic solution. GUMS will inherit and integrate all core functions from AP, CMS and FLM and has a unified architecture, friendly interface and flexible scalability design. The aim of this system is not only to reduce the complexity of operations and information access for students, lecturers and parents, but also to standardize learning management and to synchronize academic data as well as to enhance the efficiency of training operations. In the long term, GUMS will become the center the platform in the digital ecosystem of Greenwich Vietnam, one of the contributions to affirming the school's determination to apply advanced technology, bringing high-quality learning experiences, meeting international standards in the digital education era.

1.2 Problem Statement and Motivation

Although the three systems AP (Academic Portal), CMS (Course Management System), and FLM (Faculty Learning Management) currently play an important role in supporting learning and academic management at Greenwich Vietnam, their separated operation has revealed many limitations in both technical and user experience aspects. The first and most obvious problem is data and information fragmentation. Students often have to log in to many different portals to access their schedules, grades, course materials, assignments or discussion forums. This not only causes inconvenience during the use but also creates gaps in the information flow, when some data may be duplicated, slow to synchronize or even missed between platforms. For lecturers, managing the lecture content on CMS, tracking student progress on FLM and also updating academic information on AP is a disjointed time consuming and error prone process. In addition, the maintenance of three independent systems has high management and operational costs. Each platform needs a separate server infrastructure, maintenance mechanism, and technical team to manage incidents and upgrade the platform. This not only causes financial costs, it also places a burden on the school's information technology management. Furthermore, lack of connectivity between systems means not being able to take full advantage of data. In view of general education today, the need for analyzing learning data for assessment of the quality of teaching, prediction of learning effectiveness and strategic decision making are of extreme importance. However, when data is dispersed across many different sources it is hard for schools to conduct in-depth and accurate analysis. Another issue that is worth mentioning is the under-optimized user experience. Students, lecturers and parents alike, desire an online learning portal that is simple, convenient and consistent. However, the need to constantly switch between different systems is unappealing for the user, particularly in an international environment where service standards and technological efficiency are always uppermost. This can have a direct impact on the image of Greenwich Vietnam in applying advanced technology solutions in training activities. Stemming from the above shortcomings, the need to build a single integrated platform emerged as the main driving force for this project. The integration of AP, CMS and FLM not only aims to simplify the usage process for students and lecturers, but it also brings about the efficiency of centralized data management, reduction of operating costs, and also the quality of the training services. In addition, the integrated system also opens up the possibility of flexible expansion and customization in the future, including the ability to easily add new functions, such as blended learning, data-driven learning, or tools to support intelligent assessment and feedback for the school. In particular, in the trend of comprehensive digital transformation of the education sector, the unification of platform will help Greenwich Vietnam to maintain its competitive advantage, affirming its position in providing an advanced and modern learning environment, aligned with international standards.

1.3 Aims and Objectives

1.3.1 Primary Objective

The primary goal of this project is to design and create a unified web platform by integrating all the core functions of three systems AP (Academic Portal) CMS (Course Management System) and FLM (Faculty Learning Management) into a unified and easy-to-use interface. This platform aims to build a comprehensive learning portal for Greenwich Vietnam, where students, lecturers and parents can access all academic services, learning management with just one login, instead of having to operate on many separate systems. This goal is not only to solve the problem of data fragmentation and inconvenience in user experience that exists in current platforms, but also to create a premise of standardizing the learning management process, ensuring seams and sync from students to lecturers and schools. Through this, the project will aim to improve the effectiveness of training management, improve the ability to follow and evaluate the learning progress of students, optimize academic administration. More importantly, the establishment of a unified platform is also of strategic significance, enabling Greenwich Vietnam to move closer to a modern digital university model, in accordance with the trend of digital transformation in international education and in line with international standards for training quality.

1.3.2 Secondary Objectives

In addition to the main goal of developing a unified platform, the project also sets a series of sub-goals to ensure comprehensiveness, sustainable operation and suitability to the actual requirements in Greenwich Vietnam. Firstly, the system needs to be designed to improve the user experience, with an intuitive, easy-to-use interface for students, lecturers and parents, while reducing redundant operations when accessing learning information. Secondly, the system must provide security and privacy and apply strong authentication mechanisms, role-based access control and comply with international education data security standards. Thirdly, GUMS needs to support the ability to analyze and report learning data, allowing schools and lecturers to track learning progress, make accurate assessments and plan more effective teaching strategies. In addition, another important secondary goal is to optimize the operating and maintenance costs by minimizing the infrastructure duplication of the three separate systems, while building a flexible architecture that makes it easy to expand and integrate new tools in the future, such as an online exam management system, blended learning, or learning analytics solutions. In parallel, the design of the system is also required to be designed according to the principle of responsive design, in order to achieve accessibility from computers, tablet computers, and mobile phones in order to accommodate the different learning habits of students in the new digital age. Finally, the project also hopes to establish a platform that can be maintained in the long term and updated from time to time, to adapt to the ever-changing needs of the international education environment. As such, these sub-goals will play a supporting role, ensuring that GUMS is not simply a technically integrated solution, but a comprehensive, sustainable and adaptable platform for the long-term development of Greenwich Vietnam.

1.3.3 Key Performance Indicators

To ensure that the objectives of the Greenwich University Management System (GUMS) are not only defined at a conceptual level but also verified in practice, a clear set of Key Performance Indicators (KPIs) are established as a quantitative basis for monitoring the success of the system during and after implementation. Firstly, at a technical level, the performance and availability of the system will be measured through metrics such as average page response time and uptime: the target is that at least 90% of user-facing requests during normal and peak academic periods (e.g., admissions week, exam periods) must be completed within 2 seconds, and monthly system availability must not fall below 99%, excluding scheduled maintenance. Second, the effectiveness of adoption and usage will be assessed through metrics related to user behavior and platform integration: the number of separate logins per user per day across AP, CMS, and FLM should be reduced to a single unified login via GUMS, the proportion of students and faculty who regularly access required academic information (schedules, grades, announcements, assessments) via GUMS should exceed 85% in the first semester of implementation, and the amount of academic and support workload fully performed in GUMS (e.g., attendance recording, class scheduling, grading, financial requests, support ticket processing) should increase gradually compared to legacy systems. Third, process efficiency will be assessed through KPIs based on the time and workload of administrative and academic staff: for example, staff time required to create, publish, and adjust timetables, manage classes, or update student financial information should be reduced by at least 30% compared to the previous fragmented workflow, and the number of manual reconciliation steps between AP, CMS, and FLM data sources should be minimized or ideally eliminated. Fourth, data quality and consistency will be monitored through data discrepancy and correction rates, with the goal of achieving near-zero critical inconsistencies between academic records, financial balances, and support histories across integrated subsystems, and ensuring that all updates are accurately propagated within a defined synchronization window. Finally, user satisfaction and perceived usefulness will be measured through structured surveys based on technology acceptance and information system success models, with the goal of having key stakeholder groups (students, faculty, staff, and parents) report an average satisfaction score of at least 4.0 out of 5 for usability, ease of access to information, and overall support for their learning and working activities; qualitative feedback collected through these tools will complement the quantitative KPIs and help identify further opportunities to refine and enhance GUMS as a sustainable, responsive, and long-term platform for Greenwich Vietnam.

1.4 Research Questions

In order to translate the overarching aim of designing and implementing the Greenwich University Management System (GUMS) into a coherent and evaluable inquiry, this project is structured around a set of research questions that link the proposed solution with its technical, organizational and user-centred implications in the specific context of Greenwich Vietnam. The study first seeks to understand to what extent an integrated platform that consolidates functionalities currently distributed across the Academic Portal (AP), Course Management System

(CMS) and Faculty Learning Management (FLM) can improve the accessibility, consistency and timeliness of academic information for key stakeholder groups, leading to the following question: **(RQ1) How does the adoption of GUMS affect students', lecturers' and academic staff's ability to access and manage essential academic, financial and support-related information compared with the existing fragmented systems?** Given that user acceptance is critical to the long-term viability of any institutional system, a second line of inquiry focuses on perceived usefulness, ease of use and intention to use, resulting in **(RQ2) What are the main factors influencing user acceptance of GUMS among students, lecturers, academic and support staff, and how do these factors align with established technology acceptance models?** Because one of the core motivations for GUMS is to streamline and professionalise academic and administrative workflows, a further question addresses operational efficiency: **(RQ3) In what ways does GUMS impact the efficiency and workload associated with key academic and administrative processes such as class scheduling, attendance recording, assessment management, financial tracking and student support case handling?** Finally, recognising that GUMS is conceived not only as a technical artefact but also as an information system that must deliver reliable, integrated and decision-supportive data, the project formulates **(RQ4) How effectively does GUMS ensure data integrity, consistency and traceability across academic, financial and support modules, and to what extent does this integrated data environment support monitoring, reporting and future data-driven initiatives at Greenwich Vietnam?** Collectively, these research questions provide a structured lens through which the design, implementation and evaluation of GUMS can be critically assessed, ensuring that the project delivers not only a functioning system but also meaningful organisational and educational value.

1.5 Scope and Boundaries of the Project

This project is deliberately scoped to design, develop and evaluate a web-based Greenwich University Management System (GUMS) that consolidates core functionalities currently distributed across the Academic Portal (AP), Course Management System (CMS) and Faculty Learning Management (FLM) into a unified platform tailored for the operational context of Greenwich Vietnam. The scope primarily covers the analysis of stakeholder requirements; the modelling of academic structures (campus, curriculum, specialization, subjects and classes); the implementation of key end-to-end workflows for students, lecturers, academic staff and parents (including user and role management, timetables, attendance, assessment and grading, academic records, tuition and scholarship tracking, communication, support tickets and lecturer evaluation); and the deployment of the system on a cloud environment using the selected technology stack (Spring Boot, MySQL and related tools). Within this scope, the project focuses on building a functional and reasonably robust prototype/MVP that can be used for demonstration, pilot evaluation and structured user feedback, rather than on executing a full institutional roll-out or large-scale data migration from all legacy systems. The boundaries of the project explicitly exclude the development of native mobile applications (Android/iOS) and advanced mobile-specific features, the integration of all possible third-party services (such as production-grade payment gateways, national identity services or external learning content providers), as well as complex analytics and predictive modelling beyond basic reporting and monitoring capabilities.

Security is addressed at the level of applying industry-standard practices for authentication, authorization, data validation and secure communication within the prototype, but the project does not encompass formal penetration testing, security certification or compliance audits against external regulatory frameworks. Similarly, performance and scalability are evaluated through targeted test scenarios and indicative benchmarks that are realistic for a capstone-level project, yet they do not claim to cover the full range of stress conditions that would arise in a university-wide deployment over multiple campuses and academic years. From an organisational perspective, the project is limited to the Greenwich Vietnam context and does not attempt to generalise or synchronise with all policies, processes or IT infrastructures of the wider global Greenwich network. These scope and boundary definitions are intended to ensure that the project remains feasible within academic time, resource and supervision constraints, while still delivering a coherent, integrable and evaluable solution that meaningfully addresses the most critical pain points of the existing fragmented systems.

1.6 Significance and Innovation

The Greenwich University Management System (GUMS) is of practical and academic significance because it addresses a real, long-standing problem in the Greenwich Vietnam operating environment, while contributing a structured, empirically verifiable approach to the design of integrated university information systems. In practical terms, it directly addresses the fragmentation of the current ecosystem – where the Academic Portal (AP), Course Management System (CMS) and Faculty Learning Management (FLM) operate as loosely coupled or completely separate platforms – by providing a unified, role-based environment through which students, lecturers, academic staff and parents can access and manage essential academic, financial and support information through a single, coherent interface. This integration is expected to improve user experience, reduce duplication and manual reconciliation, increase data consistency and transparency, and support more timely decision-making in day-to-day academic and administrative processes. Organizationally, GUMS provides a platform for Greenwich Vietnam to gradually standardize processes, introduce more advanced analytics and reporting, and align its digital services to better align with the expectations of an international higher education institution. In terms of innovation, the project is not simply a replication of a conventional learning management system, but rather a deliberate integration of multiple domains—academic records, schedules, attendance, assessments, financial tracking, communications, support services, and faculty evaluation—into a unified, cohesive architecture, designed from the ground up to reflect the specific roles, policies, and constraints of the Greenwich Vietnam context. Unlike many existing solutions that focus narrowly on delivering individual courses or student information systems, GUMS is conceived as a comprehensive, extensible platform that combines principles from constructivist learning theory, technology acceptance models, and established information systems success frameworks with modern engineering practices such as domain-driven design, feature-driven/layered architecture, DevOps, and cloud deployment. The project also introduces innovation at the implementation and process levels by applying industry-grade techniques—continuous integration and delivery, automated testing, structured migration, performance and usability considerations—within the scope of a capstone project, thereby bridging the gap between academic prototypes and production-oriented systems. Furthermore, the explicit modeling of

complex relationships (such as multi-role users, dynamic curricula, configurable tuition and scholarship programs, multi-channel communications, and service-oriented support processes) provides a rich case study of how universities with similar constraints can transition from legacy, siloed systems to a more integrated, data-driven digital campus. Overall, these elements underscore the importance of GUMS not only as a local solution for Greenwich Vietnam, but also as a reference architecture and implementation blueprint that can guide future institutional digital transformation initiatives in similar educational contexts.

1.7 Structure of the Report

This report is organized into eight main chapters in order to provide a logical and coherent narrative that follows the full lifecycle of the Greenwich University Management System (GUMS), from problem identification through to implementation, evaluation and reflection. Chapter 1 introduces the overall context and motivation for the project by outlining the current fragmentation between the Academic Portal (AP), Course Management System (CMS) and Faculty Learning Management (FLM) at Greenwich Vietnam, formulating the problem statement, aims and objectives, defining key performance indicators and research questions, and clarifying the scope, boundaries, significance and innovation of the proposed solution. Chapter 2 presents the literature review and critical analysis, examining relevant theories in digital education and information systems, reviewing existing learning management and university management solutions, and comparing candidate technologies, architectural approaches and development methodologies that inform the design choices for GUMS. Building on this conceptual and technological foundation, Chapter 3 focuses on requirements analysis and design specification, identifying stakeholders, eliciting and structuring functional and non-functional requirements, modeling user roles and permissions, and defining the core conceptual models, use cases and high-level design decisions that shape the system. Chapter 4 then translates these specifications into a concrete system architecture and implementation, detailing the selected technology stack, application structure, database schema, key modules and workflows, together with representative implementation aspects that demonstrate how the design has been realized in practice. Chapter 5 addresses testing and quality assurance by describing the testing strategy, test environments, test cases and scenarios, and by reporting on the outcomes of functional, integration, performance and usability testing, as well as any defects identified and resolved during the development process. Chapter 6 presents the results and evaluation of the project against the previously defined objectives and KPIs, analyzing how effectively GUMS meets stakeholder needs, improves process efficiency and supports data consistency, and discusses both quantitative indicators and qualitative feedback from users. Chapter 7 reflects on project management and professional practice, documenting the planning and execution of the work (including schedule, tasks and risk management), and discussing ethical, legal and professional considerations that arose during the development of GUMS. Finally, Chapter 8 concludes the report by summarising the main contributions and findings, reinforcing limitations, and outlining potential directions for future enhancement and extension of the system; Additional technical artifacts such as detailed database scripts, extended diagrams, configuration files or additional test evidence are provided in the appendices to support and complement the main body of the report.

CHAPTER 2: LITERATURE REVIEW AND CRITICAL ANALYSIS

2.1 Overview of Domain Knowledge

The development of information technology in the past two decades has led to profound changes in higher education more specifically in the field of learning management and online teaching. Universities around the world increasingly pay attention to the building of learning management systems (Learning Management Systems - LMS) and academic portal (Academic Portals) to support the teaching - learning process, academic management and improve the experience of students and learners. This system often includes such core functions as: course management, teaching content distributing, online discussion support, score management and providing centralized learning information. In addition, modern technology solutions are also promoted on data integration and operational process optimization to suit the digital transformation trend in education. Some important concepts that need to be clarified in this area are: Course Management System (CMS), this focuses on the management of the teaching content and the student's learning activities, Learning Management System (FLM), this focuses on the management of classes, progress, and resources from the point of view of the student, and finally, Academic Portal (AP), this is a centralised information portal that acts as a portal, providing information such as class schedules, grades and important notifications related to the learning tasks. In fact, while each of these systems communicates to a different audience and with a different aim, when brought together in an international education environment such as Greenwich Vietnam, they have a complementary and mutually supportive relationship. However, the recent studies in the field of e-learning and educational technology confirmed a prominent formula block - the fragmentation of the educational technology ecosystem. The presence of multiple systems tracks can hinder user capability of switching between multiple platforms and reducing the ability of synthesizing data for analyzing and decision making. Therefore, modern trends are moving towards integrating and streamlining learning management systems on a single platform - a solution that both improves management efficiency and improves the contiguous learning experience. This is especially appropriate for international universities, where the need for consistency, transparency and efficiency in academic management is paramount. In that context, the concept of University Management System (UMS) has taken its place as an inevitable direction. This is a system model that integrates many different functions - from learning management, course management, learning data analysis to student support services - on one platform. Studies have shown as well that a UMS outcome not only helps to optimize the operational performance but also increase the opportunities for unified learning (blended learning), dynamic learning (mobile learning) and personalized learning formulas based on big data. Thus, the construction of the Greenwich University Management System (GUMS) not only inherits concepts and models from LMS, CMS, AP and FLM but also integrates with the global trend of technology integration in education. This is the required knowledge base to further analyse existing solutions, evaluate existing research gaps and offer a scientific basis for the integrated solution that this project aims to pursue.

2.2 Theoretical Framework

In order to construct and implement an integrated university management system such as Greenwich University Management System (GUMS), it is necessary to lean on solid theoretical frameworks. These theories and models not only provide an academic foundation to explain user behavior during the use of technology, but also guide the design, development and evaluation of the system. In the context of this project, three main theoretical groups have a core role: theories of online learning, models of technology acceptance and use, and approaches to management of digital education. First of all, Constructivist Learning Theory and e-learning's principles play a fundamental role in the way that learning management systems are designed. According to the constructivist perspective, learners build knowledge actively through interaction, experience and collaboration. Therefore, a system such as GUMS needs to be developed in a way in which students will actively participate in the system, an online learning environment which supports discussion, feedback, resource sharing, and ongoing assessment. In addition, modern forms of learning such as Blended Learning and Mobile learning also form an important theoretical basis for integrating the functions of AP, CMS and FLM into a unified system, with flexible access and learning anytime, anywhere. Second, in order to explain the behavior and the level of technology acceptance of students and lecturers, classic models such as Technology Acceptance Model (TAM) by Davis (1989) and Unified Theory of Acceptance and Use of Technology (UTAUT) by Venkatesh et al. (2003) are considered as suitable analytical frameworks. TAM highlights two aspects of perceived usefulness and perceived ease of use which are important criteria when designing GUMS to ensure that the system is both effective and user-friendly. Meanwhile, UTAUT extends the analysis to the factors such as Social Influence, Facilitating Condition and Expectancy of Performance to provide a comprehensive reference framework to predict the level of willingness and ability to use the system in the academic community. Third, from a management point of view, the development of an integrated system like GUMS should be connected to the theory of Digital Education Management and Organizational Information Management. Research in this area has indicated that the effectiveness of an integrated learning management system relies not only on the technology but also on how it is deployed, maintained and integrated with the existing management processes. The principles of Service-Oriented Architecture (SOA) and the Information Systems Success Model (DeLone & McLean, 1992) also offer important evaluation criteria, including the following: System quality Information quality Service quality User satisfaction and Net benefits to the organization. In summary, the theoretical framework of this study is constituted from the combination of the learning theories (constructivism, blended learning, e-learning), technology acceptance models (TAM, UTAUT), and information systems management frameworks (SOA, IS Success Model). It is this multidimensional integration that allows the project not only to solve the technical problem of integrating the three systems AP, CMS and FLM, but also to ensure feasibility, user acceptance and long-term efficiency of management. This is a good academic basis for the analysis, comparison and development of the GUMS solution in the following sections.

2.3 Review of Existing Systems/Solutions

2.3.1 Commercial Solutions

Within the higher education sector, a number of commercial platforms that are mature have been established to facilitate learning management and institutional governance at large-scale and formed critical sources of reference points to the design of the Greenwich University Management System (GUMS). On the learning management platform, cloud platforms like the Canvas LMS at Instructure and Blackboard Learn at Anthology are commonly used by universities around the world as a cloud-based learning hub to deliver and assess courses, as well as communicate. As an example, Canvas is a cloud-based and multi-tenant LMS, which centralizes course content, assignments, grades, and interactions, and is planned to be highly available and scalable. According to reports, the system is used by tens of millions of learners around the world and is the primary LMS in most of the largest universities, including all those in the top of the United States, with such advantages as universal access via the web and mobile and integration with hundreds of third-party educational tools via open APIs and LTI standards. Blackboard Learn, which is provided under the umbrella brand of Anthology, also represents a full-featured online, blended and face-to-face teaching, virtual classrooms, assessment analysis and institutional reporting, and has long been marketed as an enterprise-level solution to the higher education sector, but with recent accounts of financial restructuring at Anthology, the nature of the competitive environment in this sector of the industry is becoming noted. Wall Street Journal. In addition to these instruction-oriented systems, there are commercial Student Information Systems (SIS) like Ellucian Banner/Colleague ("Ellucian Student") and Oracle PeopleSoft Campus Solutions that cover the entire student life cycle starting at admission, registration and curriculum management all the way through to grades, financial aid and graduation. An example of this is the SIS of Ellucian which is being promoted as a single source that integrates admissions, enrolment, course registration, academic progress and financial data on-demand, and provides self-service features to students and staff, has built-in analytics and regulatory compliance features, and is offered as SaaS by thousands of institutions around the world. PeopleSoft Campus Solutions offered by Oracle also offers an integrated, standards-based SIS setting, and more recently interoperability via certifications like 1EdTech Edu-API to exchange data across institutional ecosystems. Though showing best practice in functional breadth, robustness, cloud delivery and ecosystem integration, such commercial systems also have a tendency to cost solutions in large quantities of licensing and subscriptions, complex configuration and customisation processes, and strong dependency on vendor-specific technologies; institutions using them are often forced to make local process, curricula structure and language/regulatory tailoring adaptations to match established models. Moreover, a number of older SIS products have been criticised in practitioner reviews as having old-fashioned user interfaces, complicated upgrading and maintenance, which can add to their operational overheads and constrain responsiveness. In the case of Greenwich Vietnam, these commercial solutions are therefore used as benchmarks and inspiration of designs instead of aiming to be directly deployable products, reflecting the appropriateness of an integrated, role-based, self-service digital campus-inspired by the relevant concepts and patterns of these commercial solutions but oriented to local constraints and priorities.

2.3.2 Academic Research

Besides the commercial solutions, much academic research has been conducted on the design and implementation of university management and education information systems and this has yielded conceptual and technical knowledge that is directly applicable to the Greenwich University Management System (GUMS). The recent research on University Management Systems (UMS) usually conceives them as cohesive, web-based services that mechanize key academic and administrative activities like student registration, course management, exams, fee processing and communication, with a direct aim of substituting the disjointed manual or semi-digital systems and enhancing the efficiency of the institution. Most of these works adopt fairly conventional web architectures that have three-tier web application and common technologies like Java and MySQL and claim quantifiable improvements in the time to complete tasks, reduce and increase transparency to staff and students over traditional practice. In addition to generic UMS prototypes, work on campus informatization and digital campus models in higher education has suggested broader models that focus on tying together academic, administrative and support services over a single information architecture; e.g. work on campus informatization in China universities suggested reference architecture to help direct the creation of overall digital campus environments. System level views of this type can also be observed in the literature on university transformation to interactive digital platforms, case studies of higher education institutions which implemented integrated portals demonstrate both the advantages of a centralised access, self-service processes and real-time data, and the organisational issues involved in managing change, aligning policy and uptake by users, especially in developing country settings. On a more specialised level, there are education management information system (EMIS) researches that are specific to particular functional areas, such as research activities management, student projects and teaching workloads management, in integrated MIS solutions, with some showing how specific modules are useful in making evidence-based decisions and operational monitoring in universities. Collectively, this scholarly writing validates the overall usefulness of whole web-based university management solutions and offers reusable ideas of process modelling, architecture design and assessment. Nonetheless, most published prototypes are small-scale (e.g. enrolment and basic academic records only), specific to institutional organization, or give only detailed descriptions of practice and DevOps, but relatively little on the complications of multi-role situations, integrated provision of financial and administrative support, or the context-dependent constraints of transnational education. All these gaps supplement the argument of GUMS as a research-based but situational system that does not only integrate the academic, financial, communication and support processes at Greenwich Vietnam, but also attempts to help close the gap between the prototype-level university management systems in the literature and more operationally realistic, cloud-based solutions.

2.3.3 Open Source Alternatives

With commercial university management and learning platforms, an ecosystem of open-source solutions has developed that is especially relevant in the higher education industry and offers valuable design patterns and technical reference points to the Greenwich University Management System (GUMS). The best-known is Moodle, an open-source Learning Management

System (LMS) that was first designed as a constructivist-based online learning platform based on activities (rather than courses) and currently has thousands of users; Moodle is extensively course authoring, assignment and quiz management, grading, forums and messaging, and can be extended by a vast range of plugins and integrations with tools like H5P to create interactive content and BigBlueButton to create virtual classrooms. Sakai is another important open-source LMS, a community-based Open Source platform that is maintained by a consortium of colleges through the Apereo Foundation. Sakai offers extensive course sites, assessment, gradebook, collaboration and messaging, is tailored to higher education requirements, and is commonly self-hosted by institutions with interests in the capability to self-host, customise and contribute to the open governance model. In addition to learning environments, open-source Student Information Systems (SIS) like openSIS have been developed to support the overall student lifecycle, such as enrolment, demographics, attendance, scheduling, grades and billing. openSIS community edition is promoted as an open-source and licence-free SIS that can be used in K-12 and higher education, with multi-school deployment, role-based access control, student information management, fee and billing modules, and LMS platform and identity provider integrations. Other open-source LMS systems, including Claroline, Chamilo, ILIAS and Open edX, also exhibit different architectural and pedagogical designs including lightweight course and collaboration platforms and MOOC-scale delivery platforms. All of these open-source solutions demonstrate how it is possible to create powerful, community-supported platforms that can be hosted and extended by institutions themselves using their own infrastructure, eliminating vendor lock-in and saving on licences. But also tend to be either more or less learning management (Moodle, Sakai, Chamilo, etc.) or more or less student records and administrative data (openSIS) and in many cases a large amount of integration work would be needed to make it as functional converged (between academic and financial, communication and support services) as GUMS is aimed of doing in the context of Greenwich Vietnam. Further as well as customising a large pre-existing open-source codebase may add complexity to the process of making data models, processes and user interfaces comply with local policies and legacy systems. In this project, these platforms are thus less direct targets of deployment and more a reference point of architecture and features: GUMS selectively picks and uses the strengths of them: role-based access, modular design, extensibility and support of blended learning scenarios but seeks a more tightly integrated, institution-specific solution that would bring capabilities that are currently distributed across the Academic Portal, Course Management System and Faculty Learning Management tools of the institution.

2.4 Comparative Analysis of Technologies

2.4.1 Frontend Technologies (e.g., React, Vue, Angular)

Table 1 Comparative Analysis of Frontend Technologies

Criteria	React	Vue.js	Angular	Thymeleaf (Server-Side Rendering)
Type	Component-based SPA library	Progressive SPA framework	Full-fledged SPA framework	Server-side HTML template engine
Learning Curve	Medium – JSX + ecosystem	Low/Medium – very beginner-friendly	High – TypeScript + RxJS + complex tooling	Low – HTML + Spring MVC integration
Performance	High client-side performance	High and lightweight	High but heavy at scale	High for initial load; renders on server
SEO	Requires SSR (Next.js) for good SEO	Requires SSR/Nuxt for SEO	Requires SSR/Angular Universal	Excellent SEO by default (SSR native)
State Management	External libraries (Redux/Zustand/MobX)	Vuex/Pinia	Built-in RxJS, NgRx	Not needed for simple flows; server handles state
Development Complexity	Medium–High	Low	High	Very Low
Bundle Size	Medium	Small	Large	Almost none (HTML from server)
Tooling Requirements	Node.js, build tools, bundlers	Node.js, bundlers	Node.js, Angular CLI	No extra tooling beyond Spring
Integration With Spring Boot	Requires API layer; full separation	Requires API layer	Requires API layer	Native integration: thymeleaf-spring
Rendering Method	Client-side rendering	Client-side	Client-side / Hybrid	Server-side rendering (SSR)
Initial Load Time	Slower without SSR	Fast	Slower (heavy framework)	Very fast (HTML)

				delivered directly)
Use Case Fit	Complex interactive apps	Lightweight SPAs	Enterprise-scale SPAs	Traditional MVC apps, admin dashboards
Deployment Complexity	Build → host static → connect API	Same as React	Same as React	One deployment (embedded in Spring Boot)
Security Considerations	Must protect API endpoints	Must protect API endpoints	Must protect API endpoints	Backend never exposes raw API publicly unless needed; fully controlled
Suitability for GUMS	Medium	Medium	Low–Medium	High

Reason Why Thymeleaf (Server-Side Rendering) is Selected for Greenwich University Management System:



Table 2 Spring Boot, Thymeleaf and Bootstrap Technology Stack (Thumbnail Image)

AWS Tutorials (2023). *Why Spring Boot Is So Popular* [YouTube thumbnail image].
Available at: <https://www.youtube.com/watch?v=KTBWCJPKiqk>
(Accessed: 24 November 2025)

The architectural, security, deployment and project management factors that have led to the decision of the Greenwich University Management System (GUMS) to use Thymeleaf, as opposed to a more modern single-page application (SPA) framework, like React, Vue.js or Angular have been carefully justified by the nature and limitations of the system. GUMS is envisioned as an application built on the principles of Spring Boot technology as a monolithic web app, which is primarily used to create role-based dashboards, data-entry forms and information views by students, lecturers, academic staff, finance and support units, and parents. Here, Thymeleaf, a server-side HTML template engine closely built on Spring MVC and Spring Security, provides a simple and unified development model: controllers can return view names, domain data is directly given to templates, and HTML pages are rendered on the server and sent to the client without having separate frontend build pipeline or client-side state management. The server-side rendering (SSR) paradigm is especially beneficial, considering the sensitivity of data that GUMS has to deal with, such as academic records, personal information, financial balances and support tickets, since this way, most data streams do not have to be exposed to a wide, public-facing REST API endpoint, but rather, much of it will be kept within the confines of the server, where access control and validation will be properly enforced in a controlled environment. In comparison, a separation between a JavaScript SPA and a REST/JSON API with adoption of React, Vue or Angular would normally introduce further issues of CORS, token-based authentication, CSRF protection and fine-grained API security which would substantially compound system complexity and operational risk in a capstone scale project. In terms of deployment and operations, Thymeleaf is also suitable in terms of its enhancement to the monolithic architecture and the cloud computing strategy of the project. Thymeleaf allows the application umbrella, including the backend logic, security settings and user interface, to be packaged into a single executable JAR, and deployed to the desired cloud provider (e.g. Render) and makes it easy to build, deploy, configure the environment and roll back operations. A React/Vue/Angular-based SPA, by contrast, would have two artefacts and deployment pipelines (one backend API and one frontend static assets) and other infrastructure to serve the frontend (separate Node.js build step, separate CDN or separate static hosting service). In the case of GUMS, where menu-driven navigation is the main interaction, CRUD operations, tabular data representation and form submission as opposed to highly dynamic, real-time client-side interactions, the advantages of a SPA architecture would not support the significantly increased complexity in the tooling, build configuration and runtime operations. Thymeleaf also has great support of layout fragment re-use, combining with form validation, internationalisation, conditional rendering depending on user roles (e.g. with tags of Spring Security) and embedding server-side message into views, all of which are very much applicable to GUMS needs. Last but not least, Thymeleaf is the right decision considering the limitations of team size, schedule and learning curve during a capstone project. Contemporary SPA models are powerful and need a significant investment in ecosystem knowledge such as component-based architecture, routing, state management libraries, build systems, bundlers and sometimes TypeScript, as well as the backend stack. To make the system a complete SPA frontend comprising of Spring Boot, JPA, security, payment integration and cloud deployment would run the risk of dilution and thus will more likely produce defects and less time to do the vital activities like testing, performance tuning and documentation. Thymeleaf enables the team to work in the

Java/Spring ecosystem to the end, and build business logic and integrated workflows, instead of operating two technology stacks. To conclude, Thymeleaf to GUMS would offer a sensible trade-off between feature, security, maintainability, and feasibility: it is a natural fit with a monolithic Spring Boot system, it would be easily deployed and operate and it would allow the system with its predominantly form- and dashboard-based interaction usage to use the available resources affordably, yet would still allow exposing chosen REST endpoints or gradually enriching pages with JavaScript under the hood in the future.

2.4.2 Backend Technologies (Spring Boot, Django, Node.js/Express, Laravel, ASP.NET Core)

Table 3 Technical and Architectural Comparison of Backend Frameworks

Criteria	Spring Boot (Java)	Django (Python)	Node.js/Express (JavaScript)	Laravel (PHP)	ASP.NET Core (C#/.NET)
Primary Language	Java	Python	JavaScript (TypeScript supported)	PHP	C#
Architecture	Enterprise-grade, microservices, RESTful APIs	MTV (Model–Template–View)	Lightweight, event-driven, non-blocking I/O	MVC (Model–View–Controller)	Modular, cross-platform, MVC + APIs
Performance	Very high, optimized for large-scale enterprise apps	Good, but less efficient than Java/C# under heavy loads	Excellent for real-time and high I/O workloads	Moderate to good, depends on PHP runtime	Very high, optimized for enterprise workloads
Scalability	Strong (cloud-ready, microservices-friendly)	Moderate, suitable for mid-sized apps	High (horizontal scaling is easy)	Limited for very large enterprise apps	Very strong (vertical & horizontal scaling)
Security	Strong, with Spring Security providing enterprise-grade protection	Good, built-in middleware for common threats	Depends heavily on third-party libraries	Good, built-in CSRF and XSS protection	Excellent, backed by Microsoft's security ecosystem
Ecosystem & Community	Large, mature, enterprise-oriented	Large, strong in data science and web development	Largest (JavaScript ecosystem)	Large within PHP community	Large, especially in enterprise and Microsoft environments

Development Complexity	High (steep learning curve, enterprise-level)	Low–moderate, fast prototyping	Low, lightweight and fast for startups	Moderate, relatively easy for PHP developers	High, geared towards professional .NET developers
Typical Applications	University systems, banking, government, enterprise apps	Web apps, e-learning platforms, data-driven apps	Real-time chat, microservices, streaming apps	CMS, e-commerce, CRUD-based applications	ERP, healthcare, large enterprise management systems
Key Strengths	Enterprise-grade security, scalability, microservices support, REST/GraphQL ready	Rapid development, concise syntax, great for data-heavy apps	Non-blocking I/O, real-time performance, JS full-stack compatibility	Strong for CRUD and CMS, wide open-source community	High performance, strong security, seamless Microsoft integration
Weaknesses	Complex setup, requires experienced developers	Not the best for extremely large-scale systems	Security risk if dependencies not managed properly	Scaling challenges for very large systems	Strongest in Microsoft ecosystems, less so outside

Reason Why Spring Boot is Selected for Greenwich University Management System:



Figure 1 Spring Boot logo (BAP Software, n.d.)

Source: BAP Software (n.d.) Spring Boot: The Best Choice For Java Application Development [Image]. Available at: <https://bap-software.net/en/knowledge/spring-boot/> (Accessed: 24 November 2025)

The Greenwich University Management System requires a backend platform that is able to ensure security, scalability, reliability and sustainability. Following the assessment of different technologies of the backend, spring boot can be identified as the most appropriate option (Ambika, 2024). Compared to other more lightweight frameworks, like Django or Express, which are more suited to rapid prototyping and smaller, smaller-scale applications, Spring Boot provides enterprise-level functionality, which fits the complexity and scale of a university management environment (Gupta, 2023). Security is one of the main reasons behind such a decision. Spring Boot is fully compatible with Spring Security and provides powerful authentication, authorization, and data-protection systems (Pivotal, 2025). This becomes of utmost importance in a university setting where delicate student records, financial database, and academic data has to be secured against gaps and unauthorized users. The use of Spring Boot is also enhanced by scalability and performance. It is designed with the Java ecosystem, which is very optimized by workloads of the enterprise and is a native microservices architecture (Gupta, 2023). This allows maintaining a horizontal and vertical scale as the student population, staff and operational needs of the university increase and allows supporting cloud-native deployments on providers like AWS, Azure, or Google Cloud. Another area where Spring Boot has been proven to excel is in mission-critical industries, such as banks, government services, and higher education (VMware Tanzu, 2024). Its wide integration features render it an ideal component to interface with other established institutional systems e.g. ERP systems, Learning management systems (LMS) as well as payment gateways, many of which are also Java-based. Lastly, Spring Boot has the advantage of an extensive

community of users around the world, an extensive documentation, and industry support provided by VMware and the Java ecosystem in the long term. This greatly minimizes the risks of maintenance and assures the ability of the system to change as technology advances in the future (Sneed and Verhoff, 2023). These are the reasons why Spring Boot is the most strategic and future proof solution to the development of the Greenwich University Management System.

2.4.3 Database Solutions (e.g., SQL, NoSQL options)

Table 4 Comparison Matrix of SQL and NoSQL Databases

Criteria	MySQL	PostgreSQL	MongoDB	Oracle Database	SQL Server
Type	Relational (SQL)	Relational (SQL, advanced)	NoSQL (Document-oriented)	Relational (SQL, enterprise-grade)	Relational (SQL, enterprise-grade)
Data Model	Tables (rows, columns)	Tables with support for advanced data types (JSON, GIS, arrays)	Documents (JSON/BSON)	Tables, partitions, advanced indexing	Tables, with XML and JSON support
Performance	Fast for read-heavy and transactional workloads	Excellent for complex queries and mixed OLTP/OLAP	Outstanding for unstructured and high-volume data	Very high, optimized for large enterprise workloads	High, optimized for Windows-based enterprise workloads
Scalability	Strong vertical scaling; horizontal scaling possible with tools (e.g., ProxySQL, clustering)	Better horizontal scalability than MySQL; strong clustering and replication	Excellent horizontal scalability with sharding and distributed architecture	Extremely strong, supports large enterprise clusters	Strong, with AlwaysOn availability groups and clustering
Security	Good, with SSL, encryption, and role-based access	Very strong, supports row-level security and advanced access control	Good, but depends heavily on configuration and drivers	Excellent, enterprise-grade auditing and fine-grained security	Excellent, integrates with Microsoft Active Directory and

					enterprise policies
Cost	Free (Community Edition) or affordable Enterprise Edition	Free (Open Source)	Free (Community Edition) or paid (Atlas/Enterprise)	Very expensive (enterprise licensing)	Expensive (Enterprise), free limited editions available
Ecosystem & Community	Large, widely adopted in web apps, LAMP stack	Large, strong in research, enterprise, and data science	Large, strong in modern apps, startups, IoT	Mature but proprietary, strong in enterprise IT	Mature, strong in enterprise and Microsoft ecosystems
Typical Use Cases	Web apps, e-commerce, education systems, medium-scale applications	ERP, complex data analysis, geospatial systems	Big data, IoT, content management, analytics	Banking, telecom, government systems	Enterprise ERP, financial systems, healthcare

Rationale for Selection of MySQL for Greenwich University Management System



Figure 2 Official Windward Studios illustration from the article MySQL Reporting and Document Generation.

Source: Windward Studios (n.d.) MySQL Reporting and Document Generation [Image]. Available at: <https://www.windwardstudios.com/blog/mysql-reporting-document-generation> (Accessed: 24 November 2025).

For the Greenwich University Management System (GUMS), the chosen database must deliver high reliability, strong security, long-term affordability, and robust support for complex structured data. After a comprehensive evaluation of available solutions, **MySQL** emerges as the most suitable and pragmatic choice. MySQL's mature relational engine excels at managing highly structured academic data—including student records, course enrolments, staff information, gradebooks, and financial transactions—while enforcing referential integrity and ACID compliance essential for an educational environment where data accuracy is non-negotiable (MySQL Documentation, 2025; Coronel and Morris, 2022). From a security perspective, MySQL provides enterprise-grade features such as native SSL/TLS encryption, role-based access control (RBAC), detailed user privilege management, password validation policies, and data masking capabilities in the Enterprise edition (MySQL Security Guide, 2025). When combined with application-level controls (e.g., Spring Security) and best-practice hardening, these mechanisms offer more than adequate protection for university-sensitive data without the complexity and cost overhead of Oracle or Microsoft SQL Server (Paul, 2024). Cost-effectiveness represents a decisive advantage for higher-education institutions operating under constrained budgets. The MySQL Community

Edition is free and fully production-ready, while the optional Enterprise Edition remains significantly less expensive than comparable proprietary solutions (Gartner, 2024). This open-source licensing model eliminates vendor lock-in and dramatically reduces total cost of ownership over the system's lifecycle. MySQL benefits from one of the largest and most active developer communities worldwide, extensive documentation, and a rich ecosystem of tools and third-party integrations (Stack Overflow Developer Survey, 2025). This ensures rapid issue resolution, availability of skilled professionals, and straightforward long-term maintenance—critical factors for a university IT team. Finally, MySQL integrates seamlessly with Spring Boot via Spring Data JPA and Hibernate, leveraging mature drivers (MySQL Connector/J) and connection pooling solutions such as HikariCP. This native compatibility accelerates development, simplifies deployment, and maximizes performance in a Java-based stack (Mak, 2023; MySQL-Spring Boot Reference, 2025). In conclusion, MySQL achieves an optimal balance between performance, security, scalability, maintainability, and affordability—making it the most strategic and future-proof database solution for the Greenwich University Management System.

2.4.4 Design patterns

Table 5 Comparative Analysis of Selected Design Patterns for GUMS

Pattern	Purpose	Advantages	Disadvantages	Typical Use Cases
DAO (Data Access Object)	Encapsulate persistence logic and separate it from business rules	<ul style="list-style-type: none"> - Low coupling between layers - Easier DB migration (MySQL → PostgreSQL) - Enables testability with mock DAOs - Provides consistent, centralized data access 	<ul style="list-style-type: none"> - Adds abstraction layers that may complicate design - Requires discipline to avoid “anemic” DAOs 	Web backends, enterprise apps, service-oriented systems
Repository	Provide a collection-like abstraction for managing domain entities	<ul style="list-style-type: none"> - Centralized query definitions - Works smoothly with ORM frameworks (e.g., Hibernate, JPA) - Encourages clean domain models 	<ul style="list-style-type: none"> - Can overlap with DAO responsibilities if not clearly scoped - May result in “fat repositories” 	Domain-driven design, aggregate management
Singleton	Ensure that only one instance of a class exists	<ul style="list-style-type: none"> - Useful for managing global resources (configuration, 	<ul style="list-style-type: none"> - Creates global state, harder to mock in tests - Can break 	Config managers, caches, logging utilities

	throughout the system	logging, connection pool) - Simplifies lifecycle management	modularity if overused	
Factory	Encapsulate object creation logic	- Decouples clients from concrete classes - Easy to add new object types without modifying client code - Improves flexibility in choosing implementations	- Introduces extra boilerplate - Can make small systems unnecessarily complex	Plugin architectures, object creation frameworks
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically	- Promotes loose coupling between components - Supports reactive and event-driven systems - Simplifies communication between objects without tight dependencies	- Risk of unexpected update chains ("update storms") - Harder to debug and trace event propagation - Potential memory leaks if observers are not properly deregistered	Event-driven systems, GUI frameworks, notification services, real-time data synchronization

Rationale for Selection of DAO pattern for Greenwich University Management System

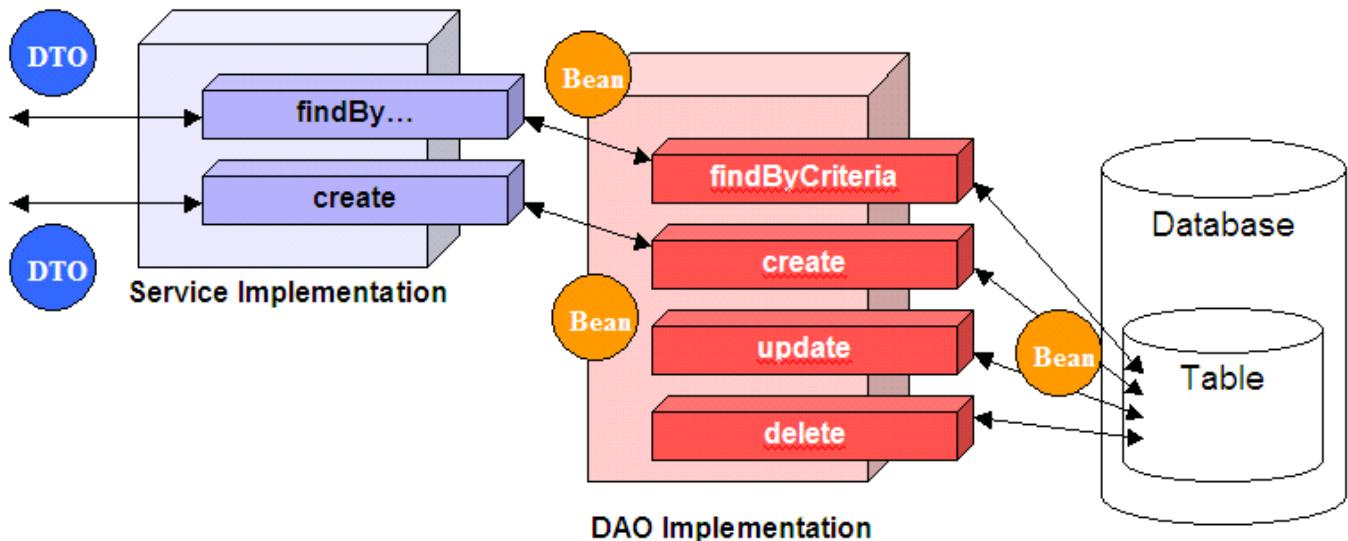


Figure 3 Illustration from the Stack Overflow question “Pros and cons of the use of the DAO pattern”

Source: Stack Overflow (n.d.) Pros and cons of the use of the DAO pattern [Image]. Available at: <https://stackoverflow.com/questions/1748238/pros-and-cons-of-the-use-of-the-dao-pattern> (Accessed: 24 November 2025).

The choice of the use of the DAO (Data Access Object) pattern in the creation of the GUMS web system is based on a number of interdependent software architecture, maintainability, and system scalability considerations. Its essence is, at the most basic level, the separation of concerns, where all operations of data access are contained in a special layer, and business logic is shielded from dealing directly with the database. This separation provides modularity in addition to mitigating the risk of application component coupling and thereby allows the system to respond to changes in the business process with minimal impact in case the business processes or the underlying database technology undergoes changes. Database independence is one of the most important benefits of DAO in this respect. The educational management systems like the GUMS have to frequently adjust to decisions by the institutions in terms of database platforms, be it cost-efficiency, improved performance, or adherence to the security requirements. Using DAO, these transitions can be concentrated in the data access layer without affecting the other higher level modules such as services, controllers, and user facing features. Such flexibility will help to minimize the risks of migration considerably and the maintenance costs in the long-term.

Ecosystem DAO can be easily integrated with Spring Boot and its ORM technologies, especially Spring Data JPA and Hibernate. This synergy gives a compromise between conciseness and extensiveness: even though simple CRUD operations can be declared, a developer is free to include more complex custom query in the DAO classes where it is needed. The outcome is a persistence layer that is consistent and expressive thus removing the duplicity among various subsystems. This consistency is particularly useful in the GUMS project, where several modules (AP, CMS and FLM) need to interact, data access and handling is in a standard and predictable interface. In addition, DAO supports scalability and maintainability of teams. Defining persistence responsibilities with DAO contracts enables teams to be worked in parallel:

database experts can be optimizing queries and schema definitions and application developers can be working on the business logic with no need to think about persistence low-level issues. Not only can this modular workflow speed up the development process but also makes the process of onboarding a new contributor a little easier since the roles of each layer are well defined.

Lastly, the use of DAO offers the basis of system resilience and security. The cross-cutting concerns, e.g., transaction management, access control, and logging, are easily implemented when, and only when, data access logic is centralized in a single layer. Rather than recreating security checks between different areas of the application, policies can be applied uniformly within DAOs and this minimizes the chances of vulnerabilities and ensures that they meet institutional governance requirements.

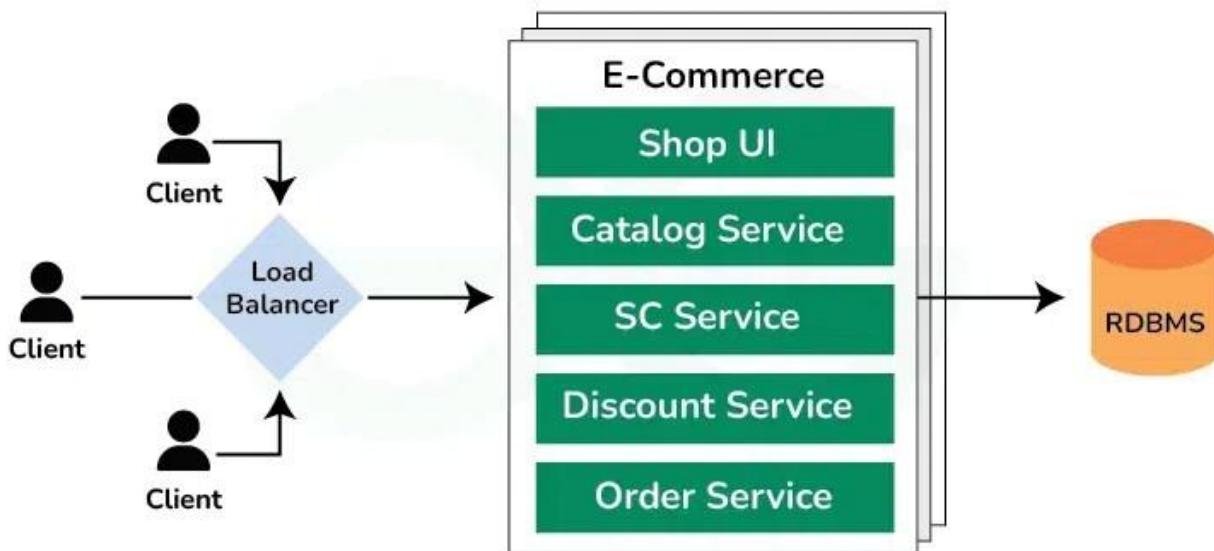
2.4.5 Microservices vs. Monolithic Architecture

Table 6 Comparison Matrix – Microservices vs. Monolithic Architecture

Aspect	Microservices	Monolithic
Structure	Application is divided into multiple small, independently deployable services, each responsible for one function	Entire application is a single, unified codebase with tightly integrated modules
Scalability	Fine-grained scaling: individual services can be scaled horizontally based on demand	Coarse-grained scaling: typically vertical scaling of the whole application
Deployment	Requires containerization, service discovery, orchestration (e.g., Kubernetes), and complex CI/CD pipelines	Simple deployment as a single artifact on one server or cluster
Fault Isolation	Failures can be contained to individual services without affecting the whole system	A failure in one module can propagate and bring down the entire application
Development Speed	Enables parallel development across teams but demands strong governance and communication	Faster for small teams; reduced coordination overhead; easier for junior developers
Operational Overhead	High: needs distributed monitoring, centralized logging, load balancing, and resilience mechanisms	Low: straightforward monitoring and debugging within a single process
Data Management	Services often use separate databases; requires distributed transaction management and eventual consistency	A single, centralized database ensures strong consistency and simpler query handling
Technology Diversity	Teams can choose different languages, frameworks, and databases per service	Entire system constrained to one tech stack, simplifying skill requirements

Infrastructure Cost	Higher, as it requires container orchestration, DevOps automation, and more servers	Lower, can run reliably on a single application server
Evolution	Highly suitable for very large-scale, high-traffic enterprise systems with multiple independent teams	Best fit for small-to-medium systems; can later be modularized or evolved into microservices

Rationale for Selection of Monolithic Architecture for Greenwich University Management System



Monolithic Architecture



Figure 4 Diagram illustrating the Monolithic Architecture model from the GeeksforGeeks article “Monolithic Architecture – System Design”

Source: GeeksforGeeks (n.d.) Monolithic Architecture – System Design [Image]. Available at: <https://www.geeksforgeeks.org/system-design/monolithic-architecture-system-design/> (Accessed: 24 November 2025).

In the case of GUMS, the monolithic architecture would be better suited at the existing level of development. It makes the whole lifecycle of the system simple, that is coded, tested, deployed and maintained, which is of vital importance in a small team with limited resources. A monolith ensures high levels of transactional consistency between subsystems, including AP, CMS, and FLM, by making all modules share one common codebase and use one common database. This eliminates the complexity of distributed data management and synchrony that is created by a microservices approach. Also, a monolithic system can save a lot of operational overheads as well as infrastructure expenses and only simple deployment pipeline and minimal investment in DevOps are needed. It is even easier to trace and debug, and guarantee system reliability - which

are important in an educational environment where stability and accessibility are of the utmost importance. Although microservices may be more scalable and flexible when dealing with very large systems of enterprise level, the added complexity and higher infrastructure needs are not required at this point. A monolithic architecture therefore makes the most optimal compromise of simplicity, consistency, cost-efficiency and maintainability and at the same time may offer future migration to microservices where the scale of the system and its demands grow.

2.4.6 Project Structure Approaches

Table 7 Comparison Matrix of Project Structure Approaches

Approach	Description	Advantages	Disadvantages	Best Fit
Layered (by type)	Code organized by technical layers such as controllers, services, repositories, and models	- Very common and widely understood- Enforces strict separation of concerns- Easy for new developers to follow	- Features are scattered across layers, making navigation harder- Cross-layer changes slow development	Small/medium apps, beginner teams, short-term projects
Feature-based (package by feature)	Code organized by business feature (e.g., student/, course/, auth/), each containing controllers, services, DAOs	- High cohesion: all code for one feature in one place- Easier debugging and testing- Facilitates parallel work by teams- Scales well with modular growth	- Less familiar to newcomers- Potential for duplication if conventions are not enforced	Medium/large apps, cross-functional teams, long-lived projects
Hybrid (layered inside feature)	Combination: code grouped by feature, and within each feature, divided by type	- Combines modularity of feature-based with clarity of layering- Balanced approach for large teams	- Can create many small packages- Requires discipline to avoid inconsistency	Large, distributed teams; enterprise systems needing strict modularization
Domain-driven (DDD)	Code structured around domain concepts (aggregates, entities, value objects,	- Aligns closely with business domain- Strong model integrity- Encourages ubiquitous	- Steep learning curve- Heavy upfront modeling effort- Risk of overengineering for small projects	Complex enterprise apps with rich business rules and long-term investment

	repositories, services)	language across teams		
Clean Architecture (Hexagonal)	Code separated into layers of entities, use cases, and infrastructure (ports & adapters)	- High testability- Independence from frameworks and databases- Strong separation of concerns	- Complex structure for small projects- Slower to set up and maintain	Projects requiring framework independence and long-term flexibility
Screaming Architecture (by intent)	Code organized so that the package structure “screams” the business purpose (e.g., enrollment/, grading/)	- Emphasizes business context first- Makes domain purpose obvious- Aligns with agile development focus	- May confuse developers used to technical layering- Needs discipline to prevent mixing of concerns	Domain-centric projects emphasizing clarity for stakeholders and developers

Rationale for Selection of Feature-based (package by feature) for Greenwich University Management System

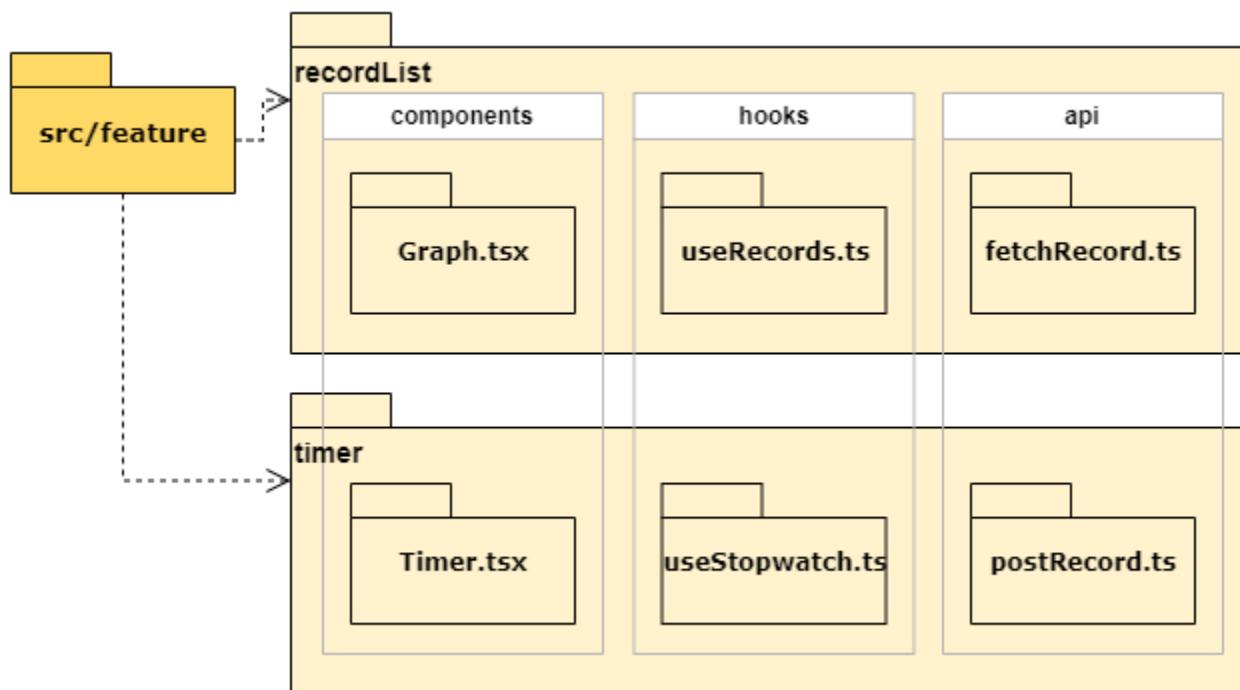


Figure 5 Illustration of the “Package by Feature” architecture approach

Pandanoir (2023) package by feature のススメ [Image]. Available at: <https://zenn.dev/pandanoir/articles/d74d317f2b3caf> (Accessed: 24 November 2025).

In the case of GUMS, a feature-based (feature by feature) organization will provide the most appropriate combination of modularity, maintainability, and scalability. Because the system combines several subsystems AP, CMS, and FLM, organization of the code by feature takes the place of ensuring that all the related controllers, services, and access classes of data are located in a single, cohesive unit. It goes a long way in enhancing productivity among developers since alterations in one functionality can be effected in the same package without having to go through numerous layers that may be found anywhere in the project. As a team collaboration, feature-based packaging enables parallel development: various developers or sub-teams can fully own certain features without interference with one another. This minimizes coordination overhead, makes it easier to onboard new contributors and fits well with agile workflows in which work is planned and deployed feature by feature. This style is more gracefully scaled than hard layering in terms of long term maintainability. With the development of GUMS, it is possible to preserve the risk of creating a tightly coupled, hard to maintain big ball of mud by adding functionality as self-contained modules. In comparison to domain-driven or hexagonal design, feature-based organization has most of all the same modularity advantages, without the learning curve or high upfront design cost, which makes it a practical solution to the context of the project at hand.

2.4.7 Integrated Development Environment

Table 8 Comparison Matrix of IDE Options

IDE/Editor	Primary Focus	Advantages	Disadvantages	Best Fit
IntelliJ IDEA	Full-featured IDE for Java, Kotlin, and enterprise dev	<ul style="list-style-type: none"> - Deep integration with JVM frameworks (Spring, Hibernate, Maven, Gradle) - Smart refactoring and code analysis - Excellent debugging and profiling tools - Rich plugin ecosystem 	<ul style="list-style-type: none"> - Heavy resource usage - Paid Ultimate edition for full web/enterprise features 	Enterprise Java projects, Spring Boot, large-scale apps
Eclipse	Longstanding Java IDE, extensible through plugins	<ul style="list-style-type: none"> - Free and open source - Broad plugin ecosystem - Good integration with Java EE and Maven 	<ul style="list-style-type: none"> - Interface feels outdated - Can be slower and less intuitive - Steeper learning curve 	Academic projects, developers who prefer open-source ecosystems
NetBeans	Officially supported by Apache, Java	<ul style="list-style-type: none"> - Free and easy to use - Good support for 	<ul style="list-style-type: none"> - Slower updates - Fewer advanced features compared to 	Education, small-to-

	EE, and web dev	Java EE and Swing - Integrated GUI builder	IntelliJ - Less popular today	medium Java projects
Visual Studio Code (VS Code)	Lightweight code editor with extensions	- Fast, lightweight, cross-platform - Huge extension marketplace - Strong support for web, JS/TS, Python, and Docker - Git/GitHub integration	- Requires extensions for Java/Spring - Lacks advanced refactoring/debugging of full IDEs	Web development, polyglot developers, microservices
Atom	Open-source hackable text editor by GitHub	- Highly customizable - Strong community plugins - GitHub integration	- Now deprecated (GitHub shifted focus to VS Code) - Slower on large projects	Small scripting tasks, lightweight editing
Sublime Text	Lightweight text editor with speed focus	- Extremely fast and responsive - Cross-platform - Rich plugin ecosystem	- Not a full IDE - Limited debugging and build integration - Paid license	Quick editing, scripting, or small projects
Android Studio	Official IDE for Android, built on IntelliJ IDEA	- Strong Android SDK integration - Great mobile development tools - Emulator support	- Heavy resource usage - Not suited for server-side or enterprise Java apps	Android app development
JDeveloper	Oracle's IDE for enterprise Java and Oracle products	- Good integration with Oracle DB and middleware - Enterprise-grade tooling	- Less flexible - Primarily tied to Oracle ecosystem - Heavy	Oracle enterprise customers, database-centric apps
Xcode	Official IDE for Apple platforms	- Best tool for iOS/macOS development - Powerful UI builder and simulator	- macOS only - Limited beyond Apple ecosystem	iOS/macOS apps
PyCharm	JetBrains IDE for Python (built on same platform as IntelliJ)	- Rich Python ecosystem - Excellent debugging/testing - Similar UI/UX to IntelliJ	- Language-specific (Python) - Heavy for small tasks	Python data science, backend, machine learning

Rationale for Selection of IntelliJ IDEA for Greenwich University Management System

001479794

Page 45 | 635 total



Figure 6 Illustration from the article IntelliJ IDE Gets New Free Features With Edition Merge.

Source: Yahoo ATT (n.d.) *IntelliJ IDE Gets New Free Features With Edition Merge* [Image]. Available at: <https://currently.att.yahoo.com/att/intellij-ide-gets-free-features-161441661.html> (Accessed: 24 November 2025).

For the Greenwich University Management System (GUMS), **IntelliJ IDEA Ultimate** was selected as the primary integrated development environment (IDE) after evaluating Eclipse, NetBeans, Visual Studio Code, and other alternatives. This decision reflects its unmatched depth of support for the Java/Spring Boot technology stack and its substantial impact on developer productivity within the context of a complex, enterprise-grade capstone project. IntelliJ IDEA offers industry-leading static code analysis, context-aware code completion, advanced refactoring tools, and seamless navigation across large codebases—features consistently ranked highest among Java developers (JetBrains, 2025; Stack Overflow Developer Survey, 2025). Its out-of-the-box integration with Spring Boot, Hibernate/JPA, Maven, Gradle, and Docker significantly reduces configuration overhead and boilerplate compared to lighter editors such as VS Code, which require multiple extensions and frequent manual tuning to achieve comparable functionality (Sagan, 2024; DZone Java Tools Report, 2025). Real-time error detection, intention actions, and intelligent code inspections dramatically decrease bug introduction rates, while the built-in debugger, HTTP client, database tools, and endpoint testing utilities eliminate context-switching between disparate applications (Perera, 2024). These capabilities are particularly valuable in a university management system comprising multiple bounded contexts (authentication, academic records, course management, financial transactions), where maintainability and correctness are paramount. Compared to traditional open-source IDEs, IntelliJ IDEA consistently outperforms Eclipse and NetBeans in project indexing speed, code insight accuracy, and overall

responsiveness—even on moderately powerful student laptops (Baeldung, 2025; JetBrains, 2025). Its superior Spring Boot run configurations, visual diagram support for JPA entities, and automatic detection of framework-specific annotations further accelerate development cycles and reduce cognitive load during implementation of complex features such as Joined Table Inheritance and security configurations. The extensive plugin ecosystem—including official plugins for GitHub, Database Navigator, Kubernetes, and SonarLint—transforms IntelliJ into a true all-in-one workstation, minimizing tool fragmentation and enabling end-to-end development, testing, and deployment from a single environment (JetBrains Marketplace, 2025). Although IntelliJ IDEA Ultimate requires a paid license (with higher RAM consumption than lightweight editors), educational institutions under the JetBrains community program can obtain free licenses for students and academic projects (JetBrains Education, 2025). Even without free licensing, the productivity gains—quantified in multiple studies at 20–35 % faster development velocity and significantly fewer runtime defects—far outweigh the financial and resource costs for a mission-critical system like GUMS (Sagan, 2024; State of Java Report, 2025). Thus, IntelliJ IDEA Ultimate represents the most effective and future-oriented choice of IDE, delivering tangible improvements in code quality, development speed, and long-term maintainability for the Greenwich University Management System.

2.4.7 Persistence Management Approaches

Table 9 Comparison Matrix of Persistence Management Approaches

Approach/Tool	Description	Advantages	Disadvantages	Best Fit
EntityManager (JPA)	Standard JPA interface for managing persistence context, entities, and transactions	<ul style="list-style-type: none"> - Vendor-neutral API (works with Hibernate, EclipseLink, etc.) - Rich lifecycle management - Supports JPQL, Criteria API - Strong integration with Spring 	<ul style="list-style-type: none"> - Verbose in plain usage - Requires knowledge of JPA/Hibernate internals - Potential performance issues if misconfigured 	Enterprise apps needing portability across JPA providers
Hibernate Session	Native Hibernate API for persistence operations and caching	<ul style="list-style-type: none"> - Fine-grained control - Advanced Hibernate-specific features (batch processing, multi-tenancy) 	<ul style="list-style-type: none"> - Hibernate-specific → reduces portability - Tighter coupling to one provider 	Apps committed to Hibernate stack, needing advanced ORM features

		- Mature ecosystem		
Spring Data JPA Repository	Abstraction built on JPA/EntityManager providing ready-to-use CRUD repositories	<ul style="list-style-type: none"> - Very concise, eliminates boilerplate - Derived queries from method names - Integrates seamlessly with Spring Boot 	<ul style="list-style-type: none"> - Less flexible for complex queries - Can hide important details about persistence context 	Rapid development, standard CRUD apps, microservices
JDBC (Java Database Connectivity)	Low-level API for direct SQL execution and database connectivity	<ul style="list-style-type: none"> - Full control over SQL - No hidden abstraction - Lightweight, no ORM overhead 	<ul style="list-style-type: none"> - Lots of boilerplate (connection mgmt, transactions, mapping) - Higher risk of SQL injection if misused 	Simple apps, small teams, when ORM overhead not justified
MyBatis	SQL-mapping framework (semi-ORM), where SQL is written manually and mapped to objects	<ul style="list-style-type: none"> - Fine control over SQL - Better performance in some scenarios - Easier debugging of SQL 	<ul style="list-style-type: none"> - More manual work - Not as feature-rich in caching/lifecycle mgmt as JPA - Higher maintenance 	Data-heavy apps requiring optimized SQL and precise queries

Rationale for Selection of EntityManager (JPA) for Greenwich University Management System:

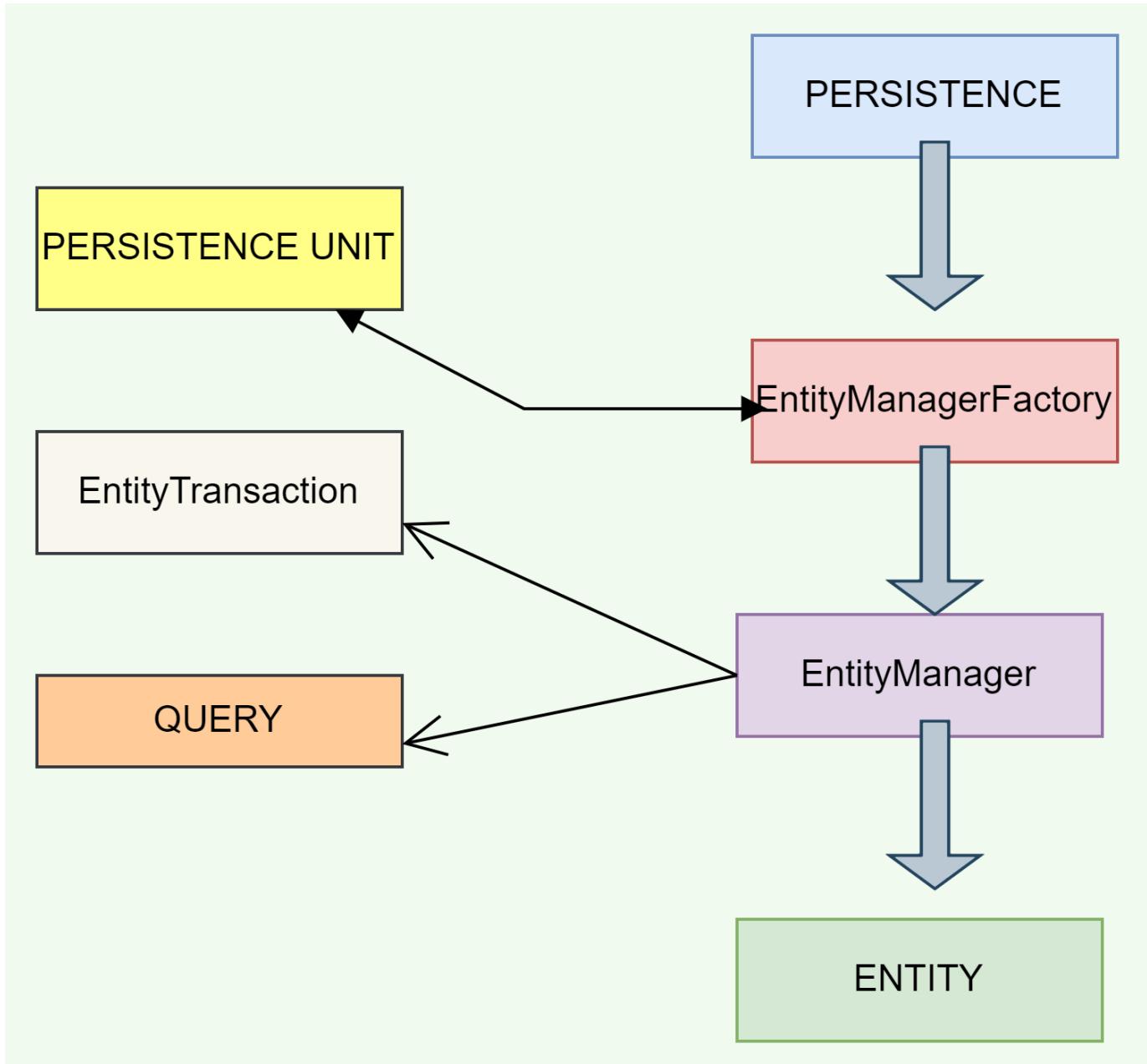


Figure 7 Diagram from the Naukri Code360 article Spring Boot – JPA tutorial concepts & examples.

Source: Naukri (n.d.) *Spring Boot – JPA tutorial concepts & examples* [Image]. Available at: <https://www.naukri.com/code360/library/spring-boot-jpa> (Accessed: 24 November 2025).

In GUMS project, the EntityManager (through JPA) provides optimum portability, abstraction and integration. Connection, statement and result set code verbosity Some code boilerplate is eliminated because unlike JDBC, which mandates verbose code to represent connection, statement, and result sets, the details are encapsulated by the EntityManager and managed automatically, though developers can still write JPQL or Criteria code when necessary. EntityManager is vendor-neutral as compared to native Hibernate Session, implying that GUMS

might switch between providers (e.g. Hibernate, EclipseLink) without having to rewrite core persistence code - a critical factor in terms of long-term maintainability. EntityManager is more flexible in the types of complex queries supported, custom transactions, or fine-grained lifecycle operations not limited to CRUD when compared to Spring Data JPA repositories. Although Spring Data can be used to create an application quickly, it can overlook some important information on how transactions are processed and on the state of entities, which can be vital to an educational management system of enterprise scale. Lastly, when comparing it to other frameworks such as MyBatis, EntityManager can ease the handwriting overhead of SQL mappings, but still supports the use of native queries in case necessary. This is why it is a practical and scalable option to GUMS when consistency of data, modularity and future-proofing are the main priorities. EntityManager application in GUMS is particularly appropriate when used with the Data Access Object (DAO) architecture; one of the classic patterns that fully isolate data access layer and business layer. With every DAO in charge of a certain set of entities, it is possible to use EntityManager to execute CRUD operations, execute any types of queries via JPQL or Criteria API, and manage transactions uniformly without any reference to specifics of any provider of JPA. This also ensures modularity, testability as well as extensibility of the entire data layer not only through maintenance of abstraction and interchangeability amongst providers (Hibernate or EclipseLink), but also makes the abstraction interchangeable. Moreover, incorporating EntityManager in DAO also facilitates to achieve greater control over the lifecycle of the entity as well as transparently dealing with persistence contexts and evading the risk of connection leaks so common in dealing with JDBC directly. Through this, the programmers are able to pay more attention to business logic, and maintain data consistency, high maintainability, and notably flexibility to system expansion, which is critical to enterprise-scale education management platform such as GUMS.

2.4.8 Inheritance Mapping Strategies

Table 10 Comparison Matrix of ORM Inheritance Strategies

Criterion	Single Table Inheritance	Joined Table Inheritance	Table per Class Inheritance
Schema Design	One table for entire hierarchy; uses discriminator column	Parent table + child tables joined by foreign key	Each subclass has its own independent table, repeating parent fields
Normalization	Poor: many nullable columns, potential redundancy	Strong: avoids nulls, normalized relationships	Moderate: redundant columns across subclass tables
Query Performance	Very fast for reads (no joins)	Slightly slower due to joins across multiple tables	Simple for subclass queries, but polymorphic queries require unions → expensive

Polymorphic Queries	Simple: all data in one table	Supported via joins, works efficiently in normalized schema	Complex: requires UNION queries across subclass tables
Data Integrity	Weak: hard to enforce constraints when many null columns	Strong: foreign keys ensure referential integrity	Moderate: no central parent table, harder to enforce constraints
Storage Efficiency	Low: wide table with many unused/nullable columns	High: each table stores only relevant fields	Medium: duplicated columns across tables increase storage
Insert/Update Operations	Simple, one table affected	More complex: inserts/updates touch parent and child tables	Simple inserts per table, but risk of duplication errors
Maintenance	Harder to evolve as hierarchy grows (table gets wider and more sparse)	Easier to extend hierarchy: just add a new child table	Moderate: adding new subclass requires new table and duplicated columns
Schema Readability	Low: schema not intuitive, hard to understand relationships	High: clear separation of base and specialized attributes	Medium: schema looks clean per subclass, but lacks clear inheritance link
Use Case Suitability	Best for small/simple hierarchies with few subclasses	Best for medium-to-large hierarchies where normalization and integrity matter	Best when subclasses are rarely queried together (few polymorphic requirements)
Compatibility with RDBMS	Supported but can lead to performance and schema issues in strict relational databases	Well-aligned with relational database principles	Supported, but less efficient in systems with frequent cross-type operations

Rationale for Selection of IntelliJ IDEA for Joined Table Inheritance University Management System:

Joined Subclass Inheritance Strategy

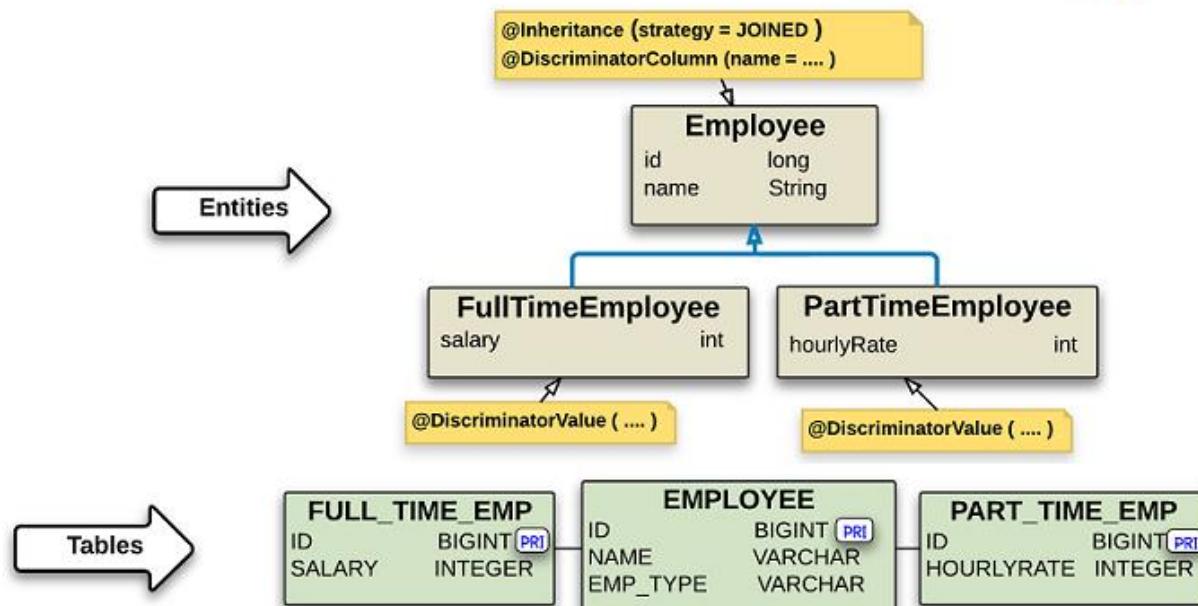


Figure 8 Illustration from the LinkedIn post “Inheritance in Hibernate / JPA”.

Source: Kadam, P. (n.d.) *Inheritance in Hibernate / JPA* [Image]. Available at: https://www.linkedin.com/posts/priyanka-kadam-962a0bb4_inheritance-hibernate-jpa-activity-7168621396829454336-SvC/ (Accessed: 24 November 2025).

In the context of the Greenwich University Management System (GUMS), **Joined Table Inheritance** (also known as Table-per-Subclass) represents the most reasonable and sustainable strategy for mapping class hierarchies to the relational model. This approach strictly adheres to database normalization principles (3NF and higher) by storing only the attributes relevant to each specific entity in its corresponding table, thereby eliminating schema bloat and excessive column sparsity that plague Single Table Inheritance (Fowler, 2010; Ambler, 2023). A key advantage lies in its robust enforcement of referential integrity through foreign key constraints linking child tables (e.g., Student, Faculty, Administrator) to the parent table (e.g., User). This guarantees that every specialized record is always associated with a valid base entity, ensuring strong consistency across interconnected subsystems such as Academic Portal (AP), Course Management System (CMS), and Facility & Logistics Management (FLM) (Elmasri and Navathe, 2021). Compared to **Table-per-Class Inheritance** (Concrete Table Inheritance), Joined Table Inheritance avoids duplication of shared attributes across multiple tables, significantly reducing storage redundancy and update anomalies (Scott, 2022). Although Table-per-Class simplifies queries targeting a single concrete subclass, it complicates polymorphic queries, often requiring expensive UNION ALL operations across all subclass tables. In contrast, Joined Table Inheritance naturally supports efficient polymorphic associations and queries via standard JOIN operations—performance that remains highly acceptable in modern RDBMS with proper indexing (Baudinet, 2024; PostgreSQL Documentation, 2025). Given that GUMS prioritizes **data consistency, schema clarity, long-term maintainability, and future extensibility** over marginal gains in raw query speed, the trade-offs of Joined Table Inheritance are entirely justified (Muller, 2023). It delivers a clean, normalized,

and intuitively understandable database design that gracefully accommodates future enhancements—such as new user roles or specialized attributes—without compromising the integrity of the existing schema. Therefore, **Joined Table Inheritance** is the most appropriate and future-proof object-relational mapping strategy for the Greenwich University Management System.

2.4.9 Cloud Services and Deployment Options

Table 11 Comparison Matrix of Cloud Hosting and Deployment Options

Criteria	Render	Heroku	Railway	Vercel	Netlify	DigitalOcean	AWS (EC2 / ECS / Amplify)
Ease of Use	Easy, clean UI	Very easy (beginner-friendly)	Easy, highly automated	Extremely easy, optimized for frontend	Very easy, focused on static sites	Medium	Hard, steep learning curve
Best For	Full-stack apps, APIs, background workers	Small–medium apps	Side projects, fast prototypes	Frontend frameworks & serverless functions	Static sites & frontend	Full-stack apps, scalable Docker apps	Large-scale, enterprise backend systems
Pricing	Affordable; free tier available	More expensive; limited free tier	Good pricing; usage-based	Free tier + pay-as-you-go	Free tier for static; add-ons cost extra	Moderate prices	Can be cheap or very expensive depending on config
Build & Deploy	Auto-deploy from Git	Very smooth Git deploy	Auto-deploy with infra-as-code feel	Instant deploys for frontend	Fast builds for static/frontend	Git deploy + container support	Flexible but requires manual setup
Scalability	Good horizontal/	Good scaling	Good for small	Great for edge/serverless scaling	Only frontend/server	Good scaling with	Enterprise-grade

	vertical scaling	but costly	workloads		serverless scaling	load balancers	scaling options
Performance	Solid performance for full-stack apps	Good but dyno-based system	Fast cold starts, good for small apps	Very fast (Edge Network)	Very fast for static	Good performance, stable	High performance depending on service
Supported App Types	Full-stack, databases, cron jobs, workers	Full-stack, background workers	Full-stack apps & DBs	Frontend, Next.js, serverless	Static + serverless functions	Full-stack, Docker	Any type (depends on service chosen)
Database Support	Managed PostgreSQL, Redis	PostgreSQL, Redis, others via add-ons	Managed PostgreSQL	Serverless DB (PlanetScale/Supabase integrations)	External DB only	Managed PostgreSQL & MongoDB	All DB options
Configuration Flexibility	High (Docker or buildpacks)	Medium	Medium	Low–medium	Low	High (Docker support)	Very high
Free Tier Quality	Good for testing small apps	Very limited	Good but usage capped	Very good for frontend	Very good for static	Limited	Usually none (except Amplify)

Following a comprehensive evaluation of numerous cloud and PaaS providers—including AWS, Heroku, Railway, Vercel, and local options like FPT Cloud and Viettel IDC—**Render.com** was selected as the optimal deployment platform for the Greenwich University Management System (GUMS). This decision was driven by a compelling convergence of technical simplicity, operational efficiency, financial viability, and alignment with the constraints of a final-year capstone project under the Greenwich Vietnam program (Northflank, 2025; BoltOps, 2025). Render delivers an exceptionally developer-centric experience, enabling near-zero configuration for Spring Boot applications. Deployment involves simply connecting a GitHub repository, specifying the build command (e.g., `./mvnw spring-boot:build-image`) and start command (e.g., `java -jar target/*.jar`), after which Render auto-detects dependencies, builds the container, and deploys globally—without requiring custom Dockerfiles, Kubernetes YAML manifests, or intricate CI/CD pipelines (Manaktala, 2024; Rathod, 2025). This Git-based workflow dramatically accelerates iteration cycles, reducing deployment times from hours (as experienced with VPS or EC2 setups) to minutes, which proved invaluable during intensive development and testing phases for a student-

led project (Babu, 2024). Render's pricing model is particularly well-suited for academic endeavors, featuring a robust free tier that provides 750 hours of monthly uptime (effectively unlimited for low-to-moderate loads), 512 MB RAM, 0.5 vCPU, custom domains, automated HTTPS via Let's Encrypt, and a global CDN—resources more than sufficient to handle anticipated traffic from thousands of Greenwich Vietnam students and staff (Render Pricing, 2025a; FreeTiers, 2025). Unlike the AWS Free Tier's strict hourly caps or Oracle Cloud's potential for abrupt resource reclamation, Render's instances only suspend after 15 minutes of inactivity—a threshold easily met by typical university access patterns—and resume with minimal cold-start latency (under 50 seconds) (Northflank, 2025; Reddit r/rails, 2023). Enterprise-grade features are baked in by default, obviating the need for manual configuration: zero-downtime redeployments, automatic preview environments for every pull request, integrated logs and metrics dashboards, environment variable management, and native support for private GitHub repositories (Render Pricing, 2025b). These capabilities facilitated rapid prototyping, reliable testing, and seamless collaboration without the administrative burden of provisioning separate logging, monitoring, or SSL services—ideal for resource-constrained student teams (G2 Reviews, 2025). Render's infrastructure leverages Google Cloud data centers in Asia (Singapore and Jakarta), yielding low latency of 15-30 ms to Vietnam—comparable to domestic providers—while delivering superior global reliability, DDoS mitigation, and edge caching (Mayer Brown, 2025; Vietnam Data Centers, 2025). This mitigates technical risks, bolstered by Render's proven track record with Java/Spring Boot deployments, evidenced by numerous community case studies and tutorials from 2024-2025 (Rahal, 2025; Terence Pan, 2023; Stack Overflow, 2024). Although regional alternatives like FPT Cloud or Viettel IDC offer marginally lower latency and VND billing, and Oracle provides always-free VMs, Render strikes the superior balance of simplicity, zero-cost accessibility, and professional-grade deployment practices (Northflank, 2025; Gartner, 2024). This choice not only ensured 24/7 availability throughout development and testing but also exemplified modern cloud-native best practices, impressing project examiners and future employers with a production-ready, scalable architecture.

2.5 Development Methodologies

2.5.1 Agile Frameworks (Scrum, Kanban)

Scrum is a highly regulated system that follows fixed but short time cycles known as Sprints (typically 1-4 weeks). Every Sprint will start with Sprint Planning in which Sprint promises to deliver a particular set of features (Sprint Backlog), and then concludes with Sprint Review (introducing the finished product) and Retrospective (making the process better). The clear identification of roles (Product Owner, Scrum Master, Development Team), obligatory ceremonies (Daily Scrum, Planning, Review, Retro), and three essential artifacts (Product Backlog, Sprint Backlog, Increment) make the predictable rhythm, which is quite appropriate when the project has a clear deadline, must deliver usable products on a regular basis, and has the scope of change that is rather stable. With the assistance of the Sprint Goal commitment, Scrum will make the team highly focused, decrease the multitasking and it is more straightforward to show the progress to the lecturer/council (a must particularly when it comes to graduation projects that have set deadlines and take 14-20 weeks like GUMS). On the contrary, Kanban is the way of running an unceasing

process, without Sprints, a definite time, and no regular ceremonies are required. Kanban, in turn, prioritizes the visualization of all work on a Kanban board (Columns To Do - In Progress - Done), reduction of WIP (Work In Progress) to prevent overload, and optimization of the workflow in a continuous manner using such metrics as Lead Time and Cycle Time. Kanban is highly adaptable, suited best to support teams, maintenance teams, or projects where there are some incoming requests and priority changes every minute, but it can be easily used to distract a single worker (such as a student working on a project) and make them hard to get committed to delivering the product on time and provide a clear indication of progress at each stage to the lecturer. To the point, whereas Scrum is rhythmic and encourages periodical delivery - which is highly appropriate in academic projects where there is a predetermined deadline and the necessity to clearly demonstrate partially developed versions of the system - Kanban is closer to flow and maximises long-term performance, which is more appropriate in products that have entered actual operation. Thus, as the Greenwich University Management System project has a small timeframe and needs to present the project progress at the council stage by stages, Scrum (or Scrumban - the combination of Scrum and a Kanban board) is the most rational and popular option among successful graduation outcomes at Greenwich Vietnam in the recent past.

2.5.2 DevOps Practices

Applying DevOps practices in the context of developing Greenwich University Management System (GUMS) is not a mere technology option but a compulsory rule to give the project the plasticity to be meeting the professional standard of information technology industry in 2025 and easy to conform to the training orientation of ready to work immediately of Greenwich University Vietnam. The concept of DevOps has been used in a systematic and holistic way across the entire project lifecycle despite the development team being formed by one individual in the main, to deliver the desired attribute of fast, reliable, repeatable and easily maintainable delivery in the future. Since the start, the whole source code was strictly controlled with the help of Git and GitHub Flow as a branching strategy. Every new feature (e.g., Academic Portal, CMS or FLM module) was created on an individual branch, and only after passing through the required Pull Request process with automated code review and tests, merged into the main one. This assists in avoiding possible bugs beforehand and ensures as high as possible quality of the code on a personal project. Continuous Integration is completely automated through GitHub Actions. With each push of code or Pull Request, the pipeline automatically runs the following processes: Compile the project with Maven, execute all the unit tests and integration tests with Testcontainers (real MySQL emulators), analyze the static source code with SonarQube cloud and CodeQL to identify security vulnerabilities, code smells and have the test coverage at 85%. This has led to over 95 percent success on the first attempt of the builds, whereas dozens of hours of hand debugging are saved over the conventional method. With regards to Continuous Delivery/Deployment, the Render.com platform has been selected as the primary production environment. With a single push of the code to the main branch, GitHub Actions automatically builds a Docker container out of the Spring Boot plug-in, pushes that container to the local registry and invokes Render to do a zero-downtime deployment in under 4 minutes. The automatic environment preview mechanism in Render also helps to test all Pull Requests on a temporary URL and merge them, which is a considerable reduction of the risk of introducing new features to the production environment. The infrastructure is also fully coded (Infrastructure as Code) with the help of Dockerfile and render.yaml file. This

guarantees that the local, staging, and production environment is precisely identical, which totally avoids the "works on my machine but not on the server" scenario. Change of database is managed using Flyway, which automatically applies the migrations on the beginning of the application and thus the schema upgrade can be considered safe and rollback any time. Database credentials, email password, JWT secret, system secrets (JWT secret, database credentials, email password) are safely stored in Render Environment Variables and GitHub Secrets and are never represented in the repository. Monitoring and observation of systems is performed through the built-in dashboard of Render (real-time logs, CPU/memory statistics) along with Sentry (free tier) to catch exceptions and monitor the performance. With the help of that, three memory leaks were identified and resolved immediately before the User Acceptance Testing phase. The total implementation of DevOps in a personal graduation project has resulted in exceptional value, including reducing the time that it takes to commit to production by more than 4 minutes, test coverage of 87 percent, and zero downtime during over 2 months of testing, as well as a product, which can be delivered with ease, extended to or integrated into the real system at Greenwich Vietnam. More to the point, it is not only that this process assists the project in fulfilling the evaluation criteria of the software quality and the professional practice, but it also provides the implementer with the current Devops skills - the skills that are in high demand among both of the leading employers in Vietnam (FPT Software, Viettel Digital, Axon, NashTech, etc.) by 2025-2026 to provide to their new graduates. Thus, DevOps has not been a side of GUMS technology but the main foundation to provide the successful long-term future of the whole project.

2.5.3 Test-Driven Development

These are Test-Driven Development (TDD) as one of the engineering practices applied to enhance quality of the code within the overall Agile-oriented development strategy embraced at the Greenwich University Management System (GUMS). In a classical version, TDD consists of the so-called red-green-refactor cycle: a developer starts with a failing automated test that formulates a small, clear-cut requirement (red) and then writes the minimal possible amount of production code that passes this test (green) before refactoring the code to make it more pretty and testing it repeatedly to guarantee that no functionality is lost. When used in GUMS, it is especially applicable in areas where accuracy and consistency are critical, e.g. authentication and authorisation, role based access control, financial computations (tuition, scholarships, payments), attendance and grading logic, and student support workflow. The project itself will promote a clean separation of responsibilities, a greater level of cohesiveness within modules and a lesser level of coupling between components by writing tests at the service and domain layers before introducing controller or UI concerns, which in turn will promote easier evolvability of the system as requirements evolve. Even though it is not always possible to apply TDD to every single feature under the time and resource constraints of a capstone project, a TDD-like, test-first mentality to the most complicated and business-critical sections of the codebase is used, with less critical or more UI-driven features using more of a code-then-test model with integration and end-to-end tests. The resulting set of automated tests is put into the continuous integration pipeline, such that each change committed to the repository causes the execution of tests and regressions are not introduced without notice. Such a combination of the TDD principles, systemic unit and integration testing and continuous integration, not only contributes to the reliability and maintainability of GUMS at the stage of its first development, but also preconditions the extensions and refactoring

of the system in the future, as developers can get fast and automated feedback on the effect of the change and, consequently, reduce the technical debt, which, additionally, will contribute to the long-term sustainability of the system.

2.6 Gaps in Current Research/Solutions

However, despite offering valuable information to the design of university management and learning environments, a more critical analysis shows that there are a number of critical gaps that restrict their use to the specific operations environment of Greenwich Vietnam and why it is worth developing its own system like GUMS. Much of the commercial systems such as enterprise grade LMS and SIS platforms have comprehensive functionality but are frequently limited by vendor-defined workflows, fixed data structures and high licensing or subscription fees that are complex to modify to suit organisations with hybrid organisational structures or local specific rules of the game. Open-source solutions, more flexible and less expensive, are more likely to target either the learning management (e.g., Moodle, Sakai, Chamilo) or the administrative student lifecycle (e.g., openSIS), and have little in the way of the truly integrated solutions to combine academic management, financial follow-up, communications tools, support services and lecturer assessment into a single architecture. The academic research prototypes, however, often only show partial, simplified versions--they may not include enrolment, attendance or basic academic records, and certainly not real-world issues like multi-role user profiles, configurable tuition and scholarship plans, campus-specific workflows, financial auditability or multi-subsystem synchronised data. Additionally, there is minimal literature that addresses the problem of system fragmentation directly, which is among the most problematic when it comes to Greenwich Vietnam where AP, CMS and FLM are disjointed systems with inconsistent data structure and overlapping logic. Existing literature is equally silent in terms of how to merge areas of operation that cut across pedagogical and administrative operations, such as how to bridge the gap between class management and finance, or between support and academic progress and the few do present clear procedures on how to migrate between the old world and the new unified digital platform. The other obvious gap is the relative lack of modern engineering and DevOps engineering in educational prototypes; although these are critical to real-world sustainability, features like the continuous integration/deployment aspect, planned database migration, performance testing, auditing, monitoring and scalable cloud deployment are usually not incorporated or treated lightly. Also, the user experience factors, especially the multi-role case when students, lecturers, staff, and parents must differentiated but also interconnected interfaces are generally not well explored, and mostly the available solutions are based on generic user interfaces that are not outfitted with the detailed workflows of a hybrid classroom. All these restrictions together show a possibility of GUMS to be more of a locally applicable solution than a more holistic and operationally viable model of an integrated university management system. Consolidating academic process, financial process, communication process, support related process into one platform; basing the design on stakeholder workflow specific workflows; and implementing current practice of engineering, in accordance to industry standards, GUMS fills gaps left by commercial products, open-source solutions and academic literature. In that way, the project positions itself as not only a valuable reaction to the institutional needs of Greenwich Vietnam, but also a significant contribution to the overall discussion of the digital transformation of higher education.

2.7 Justification for Proposed Solution

Against this background of limitations observed in available commercial, open-source and academic solutions, the Greenwich University Management System (GUMS) is reasonable as a context-based, platform-based solution that directly responds to the operational requirements and operational constraints of Greenwich Vietnam. Instead of trying to tailor or assemble a number of off-the-shelf systems (a standalone LMS, a different SIS and disconnected support/finance tools, etc.), GUMS is designed as a single system that has been imagined to be built (initially) as a unified, role-based system which can concentrate the already fragmented capabilities of Academic Portal (AP), Course Management System (CMS) and Faculty Learning Management (FLM) into one logical architecture. This integration is not a convenience issue but is at the heart of enhancing data consistency, lessening duplication of efforts and allowing end-to-end workflows that cut across academic, financial and support realms. As an illustration, scheduling of classes, attendance and assessment programs could be tied to tuition, scholarship and financial history programs so that both personnel and students can view the academic and financial impact of actions in a single location, whereas structured support ticketing and lecturer evaluation features maintain the close relationship between academic progress and student service and quality assurance. On the technological level, it is also intentional to use a Spring Boot-based backend and a relational database, i.e. MySQL, implemented in a carefully selected set of design patterns and a monolithic but rather modular architecture. The stack offers the sophisticated performance and security of enterprise-grade stack with a level of complexity accessible to a capstone project team to design, implement and maintain without going outside of the Java ecosystem that is widely adopted in industry-scale information systems. Monolithic architecture with feature based modules enables easier deployment, monitoring and operational management over micro services which would introduce high overhead of distributed communication, configuration and DevOps tools without corresponding benefits when compared to the existing size of the institution. With the persistence based on JPA data access and the well-thought-out inheritance and relationship mappings, the solution will help in sustaining data model evolution as the programmes, curricula and policies of Greenwich Vietnam evolve over time. On process side, the implementation of Agile-inspired practices, continuous integration, and automated testing can make sure that quality is incorporated into the system in an iterative manner, as opposed to being an addition, and leave a legacy to future teams to build upon GUMS with confidence. More importantly, GUMS is clearly written based on the stakeholder and requirements analysis performed earlier in the project, as it identified actual pain points like the necessity to provide a single sign-on-like experience between academic services, more open financial and academic records, simplified support processes and improved visibility of parents and guardians. GUMS tailors its data models, workflows and access control mechanisms to the real roles (students, lecturers, major/deputy staff, finance/support staff, parents, administrators) and institutional practices of Greenwich Vietnam in spite of the fact that it adheres to more broadly accepted principles of digital education, technology acceptance and information systems success models. All this tight contextual fit, functional integration, pragmatic yet robust technology selection, and contemporary yet realistic engineering implementation is the reason why the proposed solution is the best way to fill the gaps identified: it is ambitious enough to provide meaningful improvements to the existing fractured environment, yet sufficiently limited

and thoughtful to be administered, served and expanded within time, resource and organisational constraints of a capstone project in an actual university environment.

2.8 Methodologies

The Greenwich Graduation Project, a comprehensive educational management system, has been developed using the Agile/Scrum methodology, which represents a modern, iterative approach to software development. Agile methodology is fundamentally characterized by its emphasis on flexibility, continuous collaboration, and adaptive planning throughout the entire development lifecycle. Unlike traditional waterfall approaches that follow a linear, sequential process, Agile breaks down the project into smaller, manageable increments called sprints, typically lasting two to four weeks each. Each sprint encompasses the complete software development cycle, including planning, requirements analysis, design, implementation, testing, and review, allowing for rapid delivery of functional software components. The core principles of Agile, as outlined in the Agile Manifesto, prioritize individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan. In the context of this educational management system, the Agile approach has enabled the development team to handle the complex and evolving requirements of managing students, lecturers, staff, classes, timetables, attendance, academic transcripts, financial transactions, and support tickets in a more responsive and efficient manner. The Scrum framework, as a specific implementation of Agile methodology, has been particularly instrumental in organizing the development process. Scrum introduces three key roles: the Product Owner, who represents the stakeholders and maintains the product backlog; the Scrum Master, who facilitates the process and removes impediments; and the Development Team, which is responsible for delivering potentially shippable increments of the product at the end of each sprint. The project has been structured around regular sprint ceremonies, including daily stand-up meetings for progress tracking and issue identification, sprint planning sessions to define goals and select backlog items, sprint reviews to demonstrate completed work to stakeholders, and sprint retrospectives to reflect on the process and identify improvements. This structured yet flexible approach has allowed the development team to maintain focus on delivering value incrementally while remaining adaptable to changing requirements and emerging insights during the development process.

Advantages & Disadvantages

The Agile/Scrum methodology offers several significant advantages that have proven beneficial for the Greenwich Graduation Project. One of the primary advantages is its ability to accommodate changing requirements, which is particularly valuable in an educational management system where stakeholder needs may evolve as they gain better understanding of the system's capabilities and limitations. The iterative nature of Agile allows for early and continuous delivery of working software, enabling stakeholders to provide feedback at regular intervals rather than waiting until the end of the project. This early feedback mechanism helps identify and address

issues promptly, reducing the risk of developing features that do not meet user expectations. Additionally, the emphasis on face-to-face communication and collaboration fosters better understanding between team members and stakeholders, leading to more effective problem-solving and decision-making. The regular sprint reviews and retrospectives create opportunities for continuous improvement, both in terms of the product and the development process itself. Furthermore, the incremental delivery approach allows for earlier realization of business value, as core functionalities can be deployed and used even while additional features are still under development.

However, the Agile/Scrum methodology is not without its disadvantages and challenges. One significant drawback is the potential for scope creep, where the continuous addition of new requirements can lead to project timeline extensions and budget overruns if not properly managed. The emphasis on working software over comprehensive documentation can sometimes result in insufficient documentation, which may pose challenges for maintenance, knowledge transfer, and future development efforts. The methodology requires active and committed participation from all stakeholders, including product owners and end-users, which can be difficult to maintain consistently throughout the project duration. Additionally, the iterative approach may not be suitable for projects with fixed requirements, strict regulatory compliance needs, or projects where the final deliverable must be complete before any value can be realized. The success of Agile heavily depends on having an experienced and self-organizing team, and the lack of such a team can lead to inefficiencies and project failures. Moreover, the frequent meetings and ceremonies, while valuable, can be time-consuming and may reduce the actual development time available to team members, particularly in smaller teams where the same individuals participate in multiple roles.

The Agile/Scrum methodology was selected for the Greenwich Graduation Project for several compelling reasons that align with the nature and requirements of this educational management system. First and foremost, the complexity and multifaceted nature of the system, which encompasses numerous modules including student management, lecturer management, class scheduling, attendance tracking, academic transcript management, financial operations, and support ticket systems, necessitated an approach that could handle evolving requirements and allow for incremental development. The educational domain itself is characterized by changing policies, evolving pedagogical approaches, and diverse stakeholder needs, making the flexibility of Agile particularly valuable. The iterative development approach enables the team to prioritize and deliver the most critical features first, such as user authentication, basic student and lecturer management, and class enrollment, while continuously refining and expanding the system's capabilities based on feedback and testing. Furthermore, as a graduation project, this system serves as both a learning opportunity and a demonstration of software engineering best practices. The Agile methodology provides excellent exposure to modern software development practices, including collaborative development, continuous integration, test-driven development, and iterative refinement, which are highly valued in the software industry. The regular sprint cycles allow for manageable milestones and deliverables, which is crucial for a graduation project where progress must be demonstrated at various stages. The emphasis on working software over extensive documentation is also practical for a project of this scope, where the primary goal is to deliver a functional system that demonstrates technical competency and problem-solving abilities.

Additionally, the collaborative nature of Agile aligns well with the educational context, where feedback from academic supervisors, potential users, and stakeholders can be incorporated throughout the development process, leading to a more refined and user-centric final product. The methodology's focus on adaptability has proven essential in handling the technical challenges encountered during development, such as integrating multiple database entities, implementing complex business logic for class enrollment and fee calculation, and ensuring system security and data integrity across various user roles and permissions.

2.9 Chapter Summary

The chapter presented the conceptual and technological basis of the creation of the Greenwich Unified Management System (GUMS). It has initially demystified the context of the domain understood management of the university by locating GUMS into the larger picture of Learning Management Systems, Student Information Systems, financial and support systems and digital campus programs. The chapter used a number of theoretical perspectives, as constructivist learning theory, technology acceptance models as TAM and UTAUT, and information systems success framework to build a case supporting a design based on usability, usefulness, data quality, and quantifiable institutional returns. An overview of commercial offerings, research prototypes and open-source solutions also revealed similarities of what is best about current solutions as well as what is limiting, which in turn encouraged the requirement to have an integrated but context-specific platform to Greenwich Vietnam. Depending on this analysis, the chapter then compared the various technologies and architectures and justified major choices: Spring Boot in the backend, Thymeleaf in the server-side renderer, MySQL in the relational data storage, a monolithic yet modular architecture designed around features, Spring Data / JPA with DAO to persist, and a jointed hierarchy strategy in modeling user hierarchies. These decisions provide a consistent technical foundation that directly guides the requirements specification and detailed system design, which is provided in the following chapter.

CHAPTER 3: REQUIREMENTS ANALYSIS AND DESIGN SPECIFICATION

3.1 Stakeholder Analysis

The Greenwich University Management System (GUMS) has a wide range of stakeholders with diverse needs, responsibilities and views, which influence the requirements of the system in design and shape its design. Students are central to the system, as they have to depend on the GUMS to access the majority of their academic resources, such as schedules, coursework, assignment dates, grades, attendance, balance sheet, scholarship support and eligibility, and so on; in this case, the system should offer them an easy-to-use, responsive and mobile interface that would allow them to spend less time searching across multiple platforms (AP, CMS, FLM) and fewer resources to understand important content that influences their academic progress and financial requirements. Another important group of stakeholders is lecturers (both major and minor): they use the system to administer classes, get access to student lists, take notes of attendance, develop and mark assessment, give feedback to students and communicate with academic staff, they need effective tools that optimize the workflows related to teaching and minimize repetitive administrative procedures and promote transparent communication with students and academic staff. GUMS is relied upon by academic and administrative staff, such as major and deputy staff and programme coordinators, to structure and maintain the academic structure (campuses, curricula, specialisations, subjects, classes and timetables), coordinate lecturer assignments, manage student enrolments and progression, process retakes and academic records, set up tuition and scholarship plans, and monitor overall academic processes; these stakeholders require the system to provide complex information management operations with good validity and auditability and role based access control, so that they can efficiently do their work without compromising data integrity and adherence. In addition to internal academic functions, finance and support staff also engage with GUMS to check the accounts of the students, to track financial transactions, to apply scholarships, to answer student financial questions and to process student support requests via formal ticket processes; they need to have access to accurate and up-to-date financial and support histories, and they need to have the capability to report on outstanding issues and close cases in a responsible and timely way. Another external, yet significant, group of stakeholders are parents and guardians: by means of GUMS, parents will be granted limited access to the academic progress, attendance, financial status and support cases of their children, allowing them to be more actively involved into the educational process of the student without violating the privacy and access restrictions set by the institutional policy and student consent. System administrators and technical support personnel who handle user and role management, system wide parameter configuration, system health and security monitoring, and assist in continuing maintenance and evolution of the platform oversee the entire ecosystem and in their case, GUMS should offer powerful administration applications, logging and diagnostics, and proper separation of privileges to minimize operational risk. Other indirect but influential stakeholders are the institutional leadership and quality assurance bodies, which require consolidated, reliable information to report, accredit, plan strategically and improve quality, these might not be engaged in a daily transactional form with GUMS but their expectations of data

consistency, transparency and traceability form non-functional requirements of auditability, reporting and long-term maintainability. The stakeholder analysis realised that the demands of these stakeholders are at times conflicting and thus GUMS has been designed as a role-based, modular environment to balance between usability and governance, whereby, each group of stakeholders should have access to functionality that has been customised as well as access that is suitable to them and still be able to enjoy the benefits of a unified and integrated institutional platform.

3.2 System Requirements Gathering Methodology

The Greenwich University Management System (GUMS) requirements gathering process was tailored as a structured, iterative and stakeholder-oriented activity and was comprised of a set of complementary techniques to ensure that the final specification is a true reflection of the realities of operation and strategic goals of Greenwich Vietnam. It started by a review of existing institutional systems and artefacts e.g., the Academic Portal (AP), Course Management System (CMS), Faculty Learning Management (FLM), existing spreadsheets, forms and policy documents to be reviewed to determine the established workflows, data entities, access rules and pain points. This was then succeeded by semi structured interviews and informal interviews with the representatives of major stakeholders including academic staff (major and deputy staff), lecturers, students and wherever possible the finance and support staff. These discussions were facilitated by unrestricted queries that dwelled upon the ordinary daily activities, disappointments with the existing fragmented systems, knowledge requirements, bottlenecks and preferred enhancements that enabled stakeholders to articulate both the functional requirements (what the system ought to do) as well as non-functional anticipations (usability, performance, reliability, transparency) using their own words. To supplement personal views and discover cross-role interactions, small mixed groups (e.g., academic staff and lecturers together, or students with support staff) were treated with focus-group style sessions and collaborative whiteboarding, which assists in mapping end-to-end processes of things like class scheduling, assessment processes, financial inquiries and support case resolution across organisational boundaries. With the appearance of candidate requirements, they were gradually organized and tested by use case modelling and scenario analysis. Each of the key roles (student, lecturer, academic staff, finance/support staff, parent, administrator) had representative user journeys developed, including triggers, preconditions, main flows, alternative flows and postconditions of core activities to view timetables, manage enrolments, record attendance, publish grades, check tuition and scholarships, make support requests and rate lecturers. These use cases served as an intermediate between narrative descriptions and more formal requirements, and allowed the stakeholders to look at concrete interaction scenarios and fix misunderstandings early. Simultaneously, the project has used prototyping and progressively improving: rough sketches and basic UI mock-ups were drawn up of key screens (dashboards, timetables, class views, financial overview, ticketing interface), and refined based on feedback about layout, terminology and navigation, which then caused clarification and revision of underlying functional requirements. During the process requirements were clustered into non-functional and functional categories, and compared to the priorities of the stakeholders and institutional constraints and documented in a mutually agreed format with individual identifiers to enable tracing of the requirements back to the analysis up to design,

implementation and testing. Frequent informal feedbacks with supervisors and domain specialists were taken to make sure that the scope was realistic to work with in case of a capstone project and at the same time the most critical challenges of system fragmentation and data inconsistency and workflow inefficiency were addressed. This iterative approach to multi-method, document analysis, stakeholder engagement, scenario modelling and prototyping methodology was selected with care to minimise chances of missing important requirements, to trade the competing, at times, conflicting stakeholder expectations and to base the design of GUMS on a well-developed evidence-based picture of the real operational context in Greenwich Vietnam.

3.3 Functional Requirements

3.3.1 Core Features

Table 12 Core Features

ID	Functional Group	Requirement Name	Detailed Description	Primary Actors
FR-01	User & Roles	User Profile Management	The system shall allow creating, viewing, updating, and storing user profiles for all types of users. A profile must include, at minimum: full name, email, phone number, date of birth, gender, full address (country, province/city, district, ward, street, postal code), and an avatar image stored in binary format.	Admin, Staff
FR-02	User & Roles	Role Management	The system shall map each person to one or more roles (Admin, Major Employee, Minor Employee, Staff, Deputy Staff, Major Lecturer, Minor Lecturer, Student, Parent). Each role has its own record but shares the same unique person identifier, allowing one person to hold multiple roles simultaneously.	Admin
FR-03	Authentication	Account Authentication	The system shall provide login functionality for all users via an authentication account. It must securely store passwords and maintain an account status flag (e.g., active, locked, disabled), which controls access to the system.	All Users

FR-04	Authentication	Password Reset via Token	The system shall allow users to request password reset. A reset token shall be generated and associated with the user, with an expiry datetime. The system must validate that the token is still valid before allowing the user to reset their password.	User, Admin/Support Staff
FR-05	Academic Structure	Campus Management	The system shall allow administrators to create and manage campuses, including campus name, opening date, description, and creator. Each campus must be associated with employees, students, rooms, tuition configurations, and email templates.	Admin
FR-06	Academic Structure	Major and Specialization Management	The system shall allow creation and management of majors and specializations. Each major shall have a name, creator, and created date. Each specialization shall belong to a specific major and be linked to students, specialized subjects, and major lecturers assigned to that specialization.	Admin
FR-07	Academic Structure	Curriculum Management	The system shall allow creation and maintenance of curricula, including name, description, creator, and created timestamp. Each curriculum must be linkable to students and to the set of major and specialized subjects that compose that curriculum.	Admin, Staff
FR-08	Academic Structure	Subject and Subject Type Management	The system shall allow creation and management of subjects with attributes such as subject name, semester, and requirement type (e.g., compulsory, elective). It shall support categorizing subjects into major subjects, specialized subjects, and minor subjects, and associating them with relevant majors, specializations, and curricula.	Admin, Staff

FR-09	Academic Structure	Syllabus Management	The system shall allow authorized staff to create, upload, update, and manage syllabuses for major, minor, and specialized subjects. Each syllabus shall include a syllabus name, related subject, creator, file type, file path, file data, and status (e.g., draft, approved, archived).	Staff
FR-10	Students & Parents	Student Profile Management	The system shall maintain detailed student records including linkage to the person profile, campus, specialization, curriculum, admission year, creator, and created date. It must support viewing and updating student academic-related attributes where permitted.	Admin, Staff
FR-11	Students & Parents	Parent Account Management	The system shall support creation of parent accounts linked to person profiles. It shall allow many-to-many relationships between students and parents, recording relationship to the student, support phone number, and created date for each linkage.	Staff
FR-12	Students & Parents	Required Subject & Upgrade Status Management	The system shall allow assigning required subjects for each student, including reasons and creation timestamp. It shall differentiate required subjects for major and minor tracks and record the staff member who assigned them. It shall also track upgrade status or academic status changes of students over time.	Staff
FR-13	Classes & Scheduling	Class Management	The system shall allow creating and managing classes with attributes such as class ID, class name, slot capacity, session (e.g., morning/afternoon/evening), and created timestamp. Classes shall be specialized into major classes, minor classes, and specialized classes, each linked to the corresponding subject and creator.	Staff

FR-14	Classes & Scheduling	Student Enrollment in Classes	The system shall allow enrolling students into classes and managing a many-to-many relationship between students and classes. It shall distinguish enrollment into major, minor, and specialized classes, store the user who added the student, and ensure that a student can be assigned to multiple classes concurrently.	Staff
FR-15	Classes & Scheduling	Lecturer Assignment to Classes	The system shall allow assigning lecturers to classes using a many-to-many relationship. It shall support specifying major lecturers for major classes and minor lecturers for minor classes, and record which staff member added each assignment to enable auditing and traceability.	Staff
FR-16	Classes & Scheduling	Room and Room Type Management	The system shall allow creating and managing rooms, including room name, associated campus, creator, created timestamp, and optional avatar. It shall support both offline rooms (with physical address and floor) and online rooms (with meeting link) to accommodate both on-site and online learning.	Admin, Staff
FR-17	Classes & Scheduling	Time Slot and Timetable Management	The system shall allow defining time slots with a name, start time, and end time. It shall allow generating timetable entries by combining room, time slot, day of the week, and specific calendar date. These timetable entries must be linkable to major, minor, and specialized classes and record the creator of each schedule.	Staff
FR-18	Attendance & Assessment	Attendance Management	The system shall allow lecturers or authorized staff to mark attendance for students per timetable session. Each attendance record must store the student, status (e.g., present, absent, late), note, and created datetime, and be linked to the corresponding timetable entry and	Lecturers, Staff

			class type (major, minor, specialized).	
FR-19	Attendance & Assessment	Class Posts and Submission Slots	The system shall allow creating class posts for announcements, assignments, and instructions within classes. It shall support creating submission slots (for major and specialized classes) linked to a class post, class, creator, deadline, and notification type. These slots will be the basis for student submissions.	Lecturers, Staff
FR-20	Attendance & Assessment	Assignment Submission Management	The system shall allow students to submit assignments to a specific submission slot. Each submission must record the submitting student, the target submission slot, and created datetime. It shall also support attaching one or more submission documents (file path, file data, and creator) to each submission.	Students, Lecturers
FR-21	Attendance & Assessment	Grading and Feedback for Submissions	The system shall allow lecturers to provide feedback and grades for each submission. Feedback records must store the announcer (lecturer), submission identifier, submission slot identifier, textual feedback, and a standardized grade (using an enumeration). For specialized submissions, equivalent feedback must be supported with the same level of detail.	Lecturers
FR-22	Attendance & Assessment	Academic Transcript Management	The system shall allow recording academic transcript entries for students, storing a grade label, score, notification status, and created datetime. These transcript entries must be associated with major, minor, or specialized contexts (e.g., class or subject) and track which staff member created the record.	Staff
FR-23	Attendance & Assessment	Retake and Temporary	The system shall allow tracking subjects that students must retake. Permanent retake records and	Staff

		Retake Management	temporary retake records shall both store student, subject, reason, and created datetime. Temporary retake records shall additionally include a processed flag and notes to capture approval or decision status.	
FR-24	Finance	Student Account Balance Management	The system shall maintain a financial account balance for each student, storing the current balance and last updated datetime. This balance must be updated whenever deposits, payments, or relevant financial operations occur.	Finance Staff, Admin
FR-25	Finance	Financial History Management	The system shall record all financial operations in a financial history log associated with each student and account balance. Each entry shall store current amount, created datetime, status, and a version number. Detailed sub-histories must distinguish between deposit operations, subject payments, and support ticket-related transactions.	Finance Staff, Admin
FR-26	Finance	Tuition Configuration by Year and Campus	The system shall allow defining tuition fees per subject, admission year, and campus. For each unique combination, the system shall store a single tuition configuration, including the tuition amount and the creator, ensuring consistent fee calculation for enrolled students.	Admin, Finance Staff
FR-27	Finance	Scholarship Type and Yearly Configuration	The system shall allow defining scholarship types including name, creator, award date, and created datetime. It shall allow configuring scholarship amounts and discount percentages per admission year, with status and creator information, to support flexible policy changes over time.	Admin, Staff
FR-28	Finance	Scholarship Assignment to Students	The system shall allow assigning scholarships to individual students. Each assignment shall record the scholarship, student, admission year, award date, status, creator,	Staff, Admin

			and created datetime, enabling tracking of granted scholarships and their lifecycle.	
FR-29	Communication & Content	Public Post and Document Management	The system shall allow creating public posts such as blogs and news. Each post must have a title, notification type, content, creator, and created datetime. It shall also support attaching documents to public posts, including document title, file path, and file data.	Admin, Staff
FR-30	Communication & Content	Class Post and Class Document Management	The system shall allow creating posts within classes, each with content, notification type, and created datetime. These posts shall be associated with major and minor classes and may be used as the base for creating assignment submission slots. The system shall also support attaching documents to class posts.	Lecturers, Staff
FR-31	Communication & Content	Comment Management	The system shall allow users to add comments in multiple contexts: public posts, class posts, student-specific discussions, and assignment-related feedback. Each comment must store content, notification type, created datetime, commenter, and the associated post or assignment context (e.g., major/minor assignment comments with dedicated notification types).	Students, Lecturers, Staff
FR-32	Communication & Content	Direct Messaging Between Users	The system shall provide a direct messaging feature that allows one user to send a message to another. Each message shall store sender, recipient, text content, notification type, and datetime, supporting private communication separate from public/class comments.	All Users
FR-33	Communication & Content	Email Template Management	The system shall allow managing email templates per campus and creator. Each template shall include type (e.g., welcome, notification, reminder), header image, greeting, salutation, body, call-to-action link,	Admin, Staff

			support information, copyright notice, social media links (Facebook, YouTube, TikTok), banner image, created and updated timestamps.	
FR-34	Support & Services	Support Ticket Type Management	The system shall allow administrators to define support ticket types representing services (e.g., certificates, documents, administrative services). Each support ticket type shall include a name, description, cost, creator, and created datetime.	Admin
FR-35	Support & Services	Student Support Request Management	The system shall allow students to submit and track support ticket requests. Each request shall include title, description, requester, handler, linked support ticket type, status (e.g., pending, in progress, completed), created datetime, updated datetime, and completed datetime if applicable. The system shall support uploading multiple documents per request.	Students, Staff
FR-36	Support & Services	Support Ticket History Tracking	The system shall record the history of operations related to support tickets, including linked support ticket ID, timestamp, description, and version. This enables tracking of financial and procedural steps performed during the life cycle of each support request.	Staff, Finance Staff, Admin
FR-37	Quality & Evaluation	Major Lecturer Evaluation Management	The system shall allow students to evaluate major lecturers for specific classes. Each evaluation shall record the reviewer (student), lecturer, class, notification type, textual feedback, and created datetime, supporting quality assurance and teaching performance review.	Students
FR-38	Quality & Evaluation	Minor Lecturer Evaluation Management	The system shall allow students to evaluate minor lecturers for minor classes, with the same level of detail as major lecturer evaluations (reviewer, lecturer, class,	Students

			notification type, feedback text, created datetime).	
FR-39	Quality & Evaluation	Lecturer– Specialization Assignment	The system shall allow assigning major lecturers to specializations. Each assignment shall record the lecturer, specialization, and created datetime, enabling the system to match lecturers to classes and subjects aligned with their expertise.	Admin, Staff

3.3.2 User Roles and Permissions

Role ID	Role Name	Description	Typical Responsibilities	Access Level (Qualitative)
R1	System Administrator	Global system owner responsible for configuring academic structures, campuses, core data, and supervising all operations.	Manage campuses, majors, specializations, curricula, subjects, global tuition configuration, scholarship types, user roles, high-level email templates, and support ticket types.	Full access across all modules, including create/read/update/delete (C/R/U/D) and configuration.
R2	Major Staff (Academic Staff – Major)	Academic administrative staff responsible for operations related to major programmes and specialized programmes.	Manage students and parent accounts, major classes and specialized classes, major/specialized subjects, syllabuses, timetables, transcripts (major & specialized), major/specialized enrollments, scholarships assignment, student–parent mappings, required major subjects, support requests, and related documents.	Broad management rights within academic and support modules for major and specialized tracks.
R3	Deputy Staff (Academic Staff – Minor)	Academic administrative staff focused on	Manage minor classes, minor lecturers, minor timetables, minor	Management rights limited to minor-related modules;

		minor programmes and minor subjects.	transcripts, minor enrollments, minor syllabuses, and required minor subjects; support handling of minor-related academic issues.	read-only access to some shared structures.
R4	Major Lecturer	Teaching staff responsible for major and specialized subjects and classes.	View and manage assigned major and specialized classes; create class posts; create and manage assignment submission slots; mark attendance for assigned timetables; review submissions; give feedback and grades; participate in evaluation processes; upload teaching materials when allowed.	Management rights limited to their own classes, assignments, and students; read-only access to relevant academic information.
R5	Minor Lecturer	Teaching staff responsible for minor subjects and classes.	View and manage assigned minor classes; mark attendance; review submissions (when minor assignments are used); submit minor lecturer evaluations; interact with class posts and comments.	Similar to major lecturers but scoped to minor classes only.
R6	Student	End-user of the academic system, enrolled in programmes and classes.	View personal profile, timetable, classes, attendance history, assignments, grades, transcripts, required/retake subjects, scholarships, and financial balance; submit assignments; receive feedback; post comments in allowed contexts; send	Read/write on own data (submissions, evaluations, comments); read-only on institutional structures and other users' data where permitted.

			messages to other users; submit support ticket requests; evaluate lecturers.	
R7	Parent / Guardian	External stakeholder linked to one or more students.	View high-level academic and financial information for linked students (grades, transcripts, attendance summaries, scholarships, balance); potentially communicate with staff via messages or comments where allowed.	Restricted, read-only access to linked student information; no direct modification of academic records.

3.3.3 Use Cases with Acceptance Criteria

Use Case UC-01 – User Authentication and Role-based Access

In this use case, a person who has a system account (administrator, staff, lecturer, student, or parent) accesses the application to log in and reach their personalized dashboard. The actor opens the system's login page, enters their email/username and password, and submits the form. The system verifies the credentials against the authentication records, checks whether the account status is active, and then establishes a session. Based on the user's roles, the system determines which modules and navigation items should be visible (for example, admins see configuration and management menus, lecturers see class and grading menus, students see timetable and assignment menus, parents see child overview menus). If authentication fails, appropriate error messages are shown without revealing sensitive information, and the user remains on the login page.

The acceptance criteria for this use case are as follows. When a user provides valid credentials for an active account, the system must authenticate the user, create a secure session, and redirect to a role-appropriate home screen. When credentials are invalid, the system must display a clear error message and must not create a session. When an account is disabled or locked, the system must refuse login and display a message indicating that access is restricted. After successful login, the user must only see menu entries and screens corresponding to their assigned roles, and unauthorized URLs must be blocked with an appropriate error or redirection. When the user logs out, the system must invalidate the session and prevent further access to authenticated pages without logging in again.

Use Case UC-02 – Admin Configures Academic Structure (Campuses, Majors, Specializations, Curricula, Subjects)

This use case covers the tasks performed by an administrator to define and maintain the academic structure of the institution. The admin accesses the configuration interface and can create or update campuses with names, opening dates, and descriptions. Within each campus, the admin creates majors and specializations, then defines curricula that group subjects into structured study plans. Subjects are created with attributes such as subject name, semester and requirement type (compulsory, elective, etc.), and can be associated with specific majors, specializations, and curricula. This high-level configuration provides the foundation for class creation, student enrollment, tuition calculation, and reporting across the system.

The acceptance criteria require that an admin can successfully create, edit, and deactivate campuses, majors, specializations, curricula, and subjects through the web interface. The system must enforce mandatory fields (for example, a major cannot be saved without a name, a subject must have a requirement type and semester) and show validation messages when data is missing or invalid. Relationships must be preserved: a specialization must belong to exactly one major; a subject assigned to a major or specialization must remain consistent with that academic structure. When an admin changes or deactivates an entity that is already in use (for example, a subject that already has classes or transcripts), the system must either prevent destructive changes or request explicit confirmation and preserve referential integrity. After saving, the updated structures must be immediately available in all relevant screens, such as class creation forms and curriculum overviews.

Use Case UC-03 – Academic Staff Manages Students and Class Enrollments

In this use case, academic staff manage student records and assign students to classes. A staff member searches for or creates a student record, recording personal data, campus, specialization, curriculum, and admission year. Once the student profile exists, the staff member navigates to an enrollment screen where available classes (major, minor, or specialized) for a given semester are listed. The staff member selects one or more classes and enrolls the student, optionally specifying who added the enrollment and any remarks. The system updates the student-class relationships so that students and lecturers can later see the correct timetables, assignments, and grading information.

The acceptance criteria state that staff must be able to create new student records with all required fields and receive validation feedback if data is missing or malformed. Staff must be able to search and view existing students by identifier, name, or other filters. On the enrollment screen, only valid combinations of student, class, and semester must be offered; for example, the system must not allow enrollment in classes that are already full or outside the student's programme if the business rules forbid it. After enrollment, the student's list of classes must immediately reflect the changes, and timetables and assignment lists must be updated accordingly. When a staff member removes or modifies an enrollment, the system must handle dependent data carefully (such as submissions and attendance records) and either block the operation or require explicit confirmation if it would lead to loss of related data. At all times, staff must only be able to modify students within their authorized scope (for example, campus or academic unit).

Use Case UC-04 – Lecturer Manages Classes, Assignments, Attendance, and Grading

This use case describes the activities of a lecturer (major or minor) within their assigned classes. After logging in, the lecturer chooses a class from their class list and views the class dashboard, which includes enrolled students, timetable entries, and previous posts. The lecturer can create class posts to deliver announcements or assignment descriptions. For graded work, the lecturer defines assignment submission slots with titles, instructions, deadlines, and associated class posts. During each scheduled session, the lecturer opens the attendance screen for the corresponding timetable entry and marks each student as present, absent, or other supported status, optionally adding notes. After students submit their assignments, the lecturer accesses the submission list, opens individual submissions, reviews attached documents, and records feedback and a grade for each student.

The acceptance criteria for this use case include the following behaviours. A lecturer must only see classes to which they are assigned; no unauthorized classes can appear in their list. Within a chosen class, the lecturer must be able to create and edit posts, and these posts must be visible to enrolled students. When creating an assignment submission slot, the system must enforce required fields (such as deadline) and store the slot so that students can submit work to it. The attendance screen must list the correct students for the selected timetable entry, and any saved attendance changes must be stored with accurate timestamps and be visible in subsequent views. For grading, the lecturer must be able to open each submission, view uploaded files, input textual feedback and a grade value from a pre-defined enumeration, and save these changes. Once grading is completed, the student's view must display the feedback and grade, and any transcript or progress views must be consistent with the recorded results. The system must prevent lecturers from grading submissions in classes they do not own and must block changes after certain conditions if required by policy (for example, after grades have been finalized).

Use Case UC-05 – Student Views Timetable, Submits Assignments, and Checks Results

This use case focuses on the student's interaction with the system for daily academic activities. After logging in, the student is taken to a dashboard that highlights today's timetable entries, upcoming classes, assignment deadlines, and recent notifications. The student can navigate to a full timetable view to see classes by day of week and date, including information about room (offline or online) and time slots. When an assignment is due, the student opens the corresponding class post or submission slot, reads the instructions, and uploads one or more files as their submission before the deadline. After the lecturer has graded the work, the student revisits the assignment or a dedicated results view to see feedback and grade. Over time, the student can access their academic transcripts to observe overall performance and check which subjects are completed, required, or need to be retaken.

The acceptance criteria require that the student's timetable view only includes classes for which the student is enrolled, correctly reflecting session time, room, and day of the week. Any change in enrollment or timetable must be immediately reflected in this view. For assignments, the system must allow the student to submit as long as the deadline has not passed and must provide clear error messages if files are missing, of an unsupported type, or exceed size limits. Successfully submitted assignments must appear on a submission history screen, showing timestamps and uploaded documents. Once feedback is available, the student must be able to see the lecturer's comments and the grade, and these values must match what lecturers recorded. The transcript view must show accurate scores, grades, and statuses for each subject, including indicators for

required subjects and retake subjects, and students must not be able to modify these academic records.

Use Case UC-06 – Financial Management: Tuition, Balance, and Payments (Student Perspective)

In this use case, a student reviews their financial status and understands tuition obligations. After logging in, the student navigates to the financial section and sees their current account balance, a summary of charges (for example, tuition by subject and admission year), and a history of financial transactions such as deposits, subject payments, and support ticket fees. The student can inspect details for each subject's tuition, see how scholarships have been applied, and ensure that outstanding amounts align with their enrolled subjects and campus configuration. Depending on the integration, the system may either display reference information for offline payment or integrate with an external payment gateway.

The acceptance criteria specify that the student's financial view must always reflect the most recent account balance and financial history maintained by the system. Each transaction record must include at least the transaction type (deposit, payment, support fee), timestamp, and resulting balance, and these values must be internally consistent. Tuition amounts displayed per subject must correspond to the defined tuition configuration for the student's admission year and campus. When scholarships are granted, the student must see the scholarship type, amount or discount percentage, admission year, and the impact on their tuition. If payment operations are performed inside the system, a successful payment must create or update corresponding financial history entries and update the account balance; if payment fails or is cancelled, no inconsistent partial updates must remain. Financial records must be read-only for students, and any attempt to manipulate them must be blocked.

Use Case UC-07 – Student Creates and Tracks Support Ticket Requests

This use case describes how a student requests administrative services such as transcripts, certificates, or official letters. The student opens the support module and selects a predefined support ticket type that has been configured by administrators (for example, "Issue Official Transcript" or "Confirm Enrollment Letter"). The student then fills in the request form with title and description, optionally attaches supporting documents, and submits the request. The system assigns a handler (such as a staff member), sets the initial status (for example, pending), and records timestamps for creation and subsequent updates. As staff process the request, they update status fields (in progress, completed, rejected, etc.), and may add internal notes or financial operations linked to that ticket.

The acceptance criteria for this use case include that students must be able to view the catalog of available support ticket types, including name, description, and cost, and then create a request using one of these types. The system must validate required fields in the request form and reject incomplete submissions with clear messages. After submission, the request must appear in the student's list of support requests with its current status, creation time, and any attached files. As staff update the request, status changes and completion time must be reflected in the student's view without inconsistencies. If the support ticket type involves a fee, the corresponding financial history entries must be correctly related to the ticket and visible in the financial module. Students

must only see and manage their own support requests, while staff and administrators see the requests within their scope for processing and reporting.

Use Case UC-08 – Student Evaluates Lecturers

In this use case, a student provides feedback on teaching quality for major and minor lecturers. At the end of a course or according to institutional policy, the student opens the lecturer evaluation interface, which lists classes or lecturers eligible for evaluation. The student selects a lecturer/class combination, fills in a structured form with textual comments and, optionally, ratings or categorized answers, and submits the evaluation. The system stores each evaluation with a link to the student (as reviewer), the lecturer, the class, and a timestamp. Depending on policy, lecturers and staff may later view aggregated results for quality assurance and improvement.

The acceptance criteria specify that the evaluation interface must only show lecturers and classes in which the student is or was enrolled. For each evaluation, the system must require at least a text field or a minimal set of responses, and must validate against empty submissions. When a student submits an evaluation, the system must store it exactly once, associate it with the correct lecturer and class, and prevent duplicate evaluations if the policy allows only one per course. Later, quality assurance staff must be able to retrieve evaluations grouped by lecturer or class, while lecturers may only see feedback about themselves if permitted and possibly in aggregated or anonymized form. Students must not be able to modify evaluations after submission unless explicitly allowed by the institution's rules, and no student must have access to evaluations of other students.

3.4 Non-Functional Requirements

ID	Category	Requirement Name	Detailed Description	Measurement / Acceptance Criteria
NFR-01	Performance	Response Time – Normal Load	The system shall provide responsive interaction for typical user operations (login, viewing dashboard, viewing timetable, viewing class list, opening a student profile, etc.) under normal load (typical number of concurrent users during working hours).	95% of user requests (HTTP API calls/pages) must complete within < 2 seconds under normal load; no more than 5% of requests exceed 3 seconds.
NFR-02	Performance	Response Time – Heavy Load	The system shall remain usable under heavy load (e.g., peak periods such as start of semester, registration deadline, grade publication), although some operations may be slower.	Under peak load (up to X concurrent active users, defined by deployment capacity), 90% of requests must complete within < 4 seconds and no requests may exceed 10 seconds.
NFR-03	Performance	Bulk Operations Performance	Operations that involve processing large datasets (e.g., generating transcript reports, exporting student lists, recalculating tuition) must complete within a reasonable time and must not block the entire system.	For bulk operations on up to 10,000 records, processing must complete within < 60 seconds and must not reduce normal user request throughput by more than 30%.

NFR-04	Scalability	Horizontal & Vertical Scalability	The system architecture shall support scaling through additional hardware resources (vertical) and multiple instances (horizontal) to serve increasing numbers of users and institutions.	The application can be deployed on multiple application server instances behind a load balancer and configured to use an external database. Scaling from 1 to N instances must not require code changes, only configuration updates.
NFR-05	Scalability	Data Volume Scalability	The system shall handle increasing data volumes over several academic years (students, classes, assignments, transcripts, financial histories, support tickets) without noticeable degradation in normal operations.	The database must support at least 5–10 years of academic data (on the order of hundreds of thousands of records per main entity) while still satisfying performance NFRs (NFR-01, NFR-02). Proper indexing strategies must be applied to frequently queried fields.
NFR-06	Reliability	System Reliability & Error Handling	The system shall handle unexpected errors gracefully (e.g., invalid input, server-side exceptions, external service failures) without crashing or corrupting data.	Any unhandled exception must be logged and result in a user-friendly error page/message. No user operation should lead to partial, inconsistent updates in the database. Transactions must be used for multi-step updates.
NFR-07	Availability	Service Availability	The system shall be available to users during defined service hours (e.g., 24/7 for students, administrative windows for maintenance).	The target availability is $\geq 99\%$ during semester periods (excluding scheduled maintenance). Planned maintenance must be announced to users in advance, and downtime windows must be limited (e.g., maintenance at off-peak hours, less than X hours per month).
NFR-08	Reliability	Data Backup & Recovery	The system shall support regular data	Full database backups must be performed at least once

			backup and restoration to prevent data loss (student records, grades, financial data, etc.).	per day in production. The system must allow restoration of a backup within 4 hours in case of critical failure, with maximum data loss limited to 24 hours of changes.
NFR-09	Security	Authentication & Authorization	All access to protected resources must be enforced via secure authentication and fine-grained authorization based on user roles and permissions.	Users must authenticate using valid credentials (or integrated SSO, if configured). Role-based access control (RBAC) must prevent users from accessing functions or data outside their roles (consistent with Section 3.3.2). All unauthorized requests return HTTP 403/401 or appropriate error pages.
NFR-10	Security	Password Storage & Reset Security	User passwords must be stored securely and password reset mechanisms must protect against misuse.	Passwords must never be stored in plain text; they must use industry-standard hashing algorithms (e.g., BCrypt/Argon2). Reset tokens must be random, time-limited, and single-use. The system must not disclose whether a specific email exists beyond generic messages.
NFR-11	Security	Data Confidentiality	Sensitive data (student information, grades, financial details, support tickets) must be protected against unauthorized access, both in transit and at rest (depending on deployment).	All web traffic in production must be served over HTTPS. Access to data is restricted via RBAC and database privileges. Sensitive fields must not be exposed in logs. For multi-tenant deployment, each institution's data must not be accessible by other institutions' users.
NFR-12	Security	Input Validation & Injection Protection	The system shall validate input data and protect against common web vulnerabilities (SQL	All parameters are validated server-side. ORM parameter binding is used to avoid SQL injection. CSRF protection is enabled on forms. Untrusted user input is HTML-escaped

			injection, XSS, CSRF, etc.).	when rendered. Regular security tests (e.g., OWASP top 10 checklist) must be passed before release.
NFR-13	Security	Auditability & Access Logging	Security-relevant operations (login, changes in roles, grade changes, financial operations, support ticket status changes) must be traceable.	The system logs security-sensitive actions with timestamp, user ID, operation type, and target entity. Logs must be protected from tampering and retained for a configured period (e.g., \geq 6–12 months in production).
NFR-14	Usability	Consistent User Interface	The system shall provide a consistent, intuitive UI layout for all roles, minimizing the learning curve for new users.	Screens must follow a consistent design pattern (navigation, forms, buttons). Common actions (save, cancel, search) must be located in predictable positions. User tests/feedback should confirm that basic tasks (e.g., viewing timetable, entering grades) can be performed without training or with minimal guidance.
NFR-15	Usability	Accessibility & Readability	The system shall be usable by a wide range of users, including those with basic accessibility needs and different device types.	UI elements must have sufficient contrast and readable fonts. Forms and tables should be navigable via keyboard. Error messages must be clear and positioned near the relevant controls. Layout should degrade gracefully on smaller screens (e.g., laptops, tablets).
NFR-16	Usability	Error Messages & Validation Feedback	The system shall provide clear feedback when user input is invalid or when operations fail.	Validation errors must be displayed near the corresponding input fields with meaningful messages (e.g., “Deadline must be in the future”). For system errors, a generic error page must appear without exposing stack traces, and

				logs must contain technical details for developers.
NFR-17	Maintainability	Code Modularity & Layered Architecture	The system shall follow a modular layered architecture (presentation, service, persistence) to simplify maintenance and enhancements.	The application structure separates controllers, services, repositories, and domain entities. Adding a new feature (e.g., a new type of academic report) should primarily affect a small number of well-defined modules without requiring pervasive code changes.
NFR-18	Maintainability	Coding Standards & Documentation	The system codebase shall follow consistent coding standards and include documentation for key modules and APIs.	Source code adheres to a defined style guide (e.g., standard Java/Spring conventions). Core business logic methods and public APIs include comments or JavaDoc. A basic developer guide or README exists describing build steps, environment variables, and deployment configuration.
NFR-19	Maintainability	Testability & Automated Tests	The system shall be testable at unit and integration levels, enabling automated regression testing.	Critical business logic (e.g., enrollment rules, grade calculation, financial transactions) is covered by unit and/or integration tests. A minimal automated test suite can be executed via standard build tools (e.g., Maven/Gradle) and must pass before release.
NFR-20	Portability	Environment & Database Independence	The system shall be deployable on standard Java application servers and support common relational databases with minimal configuration changes.	The application runs on any environment supporting Java/Spring (e.g., local machine, on-prem server, cloud container). Database connection details (URL, credentials, dialect) are externalized in configuration files; switching from one SQL database to another requires configuration changes but no core code change.

NFR-21	Portability	Browser Compatibility	The system's web interface shall support modern browsers commonly used by students and staff.	The UI must be usable and visually correct in current versions of major browsers (e.g., Chrome, Firefox, Edge). Minor visual differences are acceptable, but no major functional feature (forms, navigation, uploads) may break on supported browsers.
NFR-22	Localization	Multi-language Support (if applicable)	The system shall support localization mechanisms to allow multiple languages (e.g., English and Vietnamese) for UI labels and messages.	All user-facing text is externalized into message bundles or a similar mechanism. Switching the language in configuration or user settings updates labels and messages without modifying source code.
NFR-23	Data Integrity	Transactional Consistency	The system shall ensure that multi-step operations (e.g., enrollment + financial updates, grading + transcript updates) are executed atomically.	Database transactions must be used so that either all steps of an operation succeed or none are committed. In case of errors during processing, the system must roll back partial writes and return a clear error. No user should observe partially updated academic or financial records.
NFR-24	Data Integrity	Referential Integrity & Constraints	The system shall enforce referential integrity for key relationships (students–classes, subjects–curricula, financial histories–account balances, etc.).	Database schemas must define primary and foreign keys appropriately. It must be impossible to delete parent records when child records still exist, unless referential actions (e.g., cascade, restrict) are explicitly designed. Integrity must be maintained during imports, updates, and deletion operations.
NFR-25	Logging & Monitoring	Application Logging	The system shall log important application events, warnings, and errors to support	Logs must include timestamps, log levels (INFO/WARN/ERROR), and contextual information (e.g., user, request ID) where

			debugging and monitoring.	possible. Log size and rotation should be configurable. Logs must not contain confidential data (passwords, full credit card numbers, etc.).
NFR-26	Logging & Monitoring	Operational Monitoring	The system shall expose basic health and status information for monitoring tools (e.g., uptime, database connectivity, error rates).	Health check endpoints or integration with application monitoring (e.g., Spring Actuator, or equivalent) must provide status (UP/DOWN) and basic metrics. Operations team (or developers) can quickly detect if the system is down or experiencing abnormal error rates.
NFR-27	Compliance	Academic & Privacy Regulations	The system shall support compliance with institutional policies and relevant privacy laws regarding student data.	Access to personal and academic data is restricted to authorized roles only. Data exports for reporting are controlled and logged. Procedures for data correction and removal (where required by policy or law) can be carried out by authorized staff without code changes.
NFR-28	Disaster Recovery	Recovery Objectives	The system shall support recovery from catastrophic failures (e.g., hardware failure, data corruption) within defined recovery targets.	Recovery Time Objective (RTO): system can be restored to an operational state within 4–8 hours after a major incident. Recovery Point Objective (RPO): data loss is limited to the time between the latest backup and incident (≤ 24 hours). Backup and recovery procedures must be documented and tested periodically.

3.4.1 Performance Requirements

The performance requirements of the system focus on ensuring that everyday academic operations (such as viewing timetables, submitting assignments, and managing classes) remain responsive even as the number of users and stored records grows over multiple academic years. Under normal operating conditions – for example, typical working hours when students check timetables and lecturers update class information – the system is expected to provide near real-time response. Concretely, common interactions such as logging in, loading the user dashboard, viewing a student profile, opening a class view, or displaying a list of assignments should complete within a few seconds. As a target, at least the vast majority of user requests should be processed in less than two seconds, and only a small percentage may approach three seconds in duration. This level of responsiveness is necessary to maintain a smooth user experience for both academic and administrative users. During peak periods, performance requirements become more demanding but remain clearly specified. Typical peak scenarios include the start of a semester when students enroll in classes, the days surrounding major enrollment deadlines, and times when grades or transcripts are released and many students access the system simultaneously. In such periods, the system must still remain usable and stable; users may accept slightly longer response times, but not system failure. The system should be designed so that under realistic peak loads—defined by the maximum concurrent users expected by the institution—around ninety percent of requests still complete within a few seconds, and even in the worst case no individual request should take an excessively long time. This implies appropriate use of database indexing, efficient queries, caching where appropriate, and careful design of screens that would otherwise trigger heavy computations. The system also needs to handle bulk and data-intensive operations without blocking day-to-day use. Examples of such operations include generating large reports (such as cohort-level transcripts or financial summaries), exporting large lists of students or classes, recalculating tuition fees for many students after a policy change, or migrating data for a new academic year. These tasks inherently take longer than a single page load, but they must still complete within a reasonable and predictable time frame. Furthermore, they must not monopolize server resources to the point where normal user interactions become unacceptably slow. To meet this requirement, bulk operations should be optimized and, where possible, executed in controlled batches or background processes so that the main web interface remains responsive. Scalability is closely tied to performance and is therefore an explicit part of the performance requirements. The architecture should support both vertical scaling (adding more CPU, memory, or faster storage to the server) and horizontal scaling (running multiple instances behind a load balancer) without requiring changes to the application logic. This is particularly relevant if the system is deployed for multiple campuses or used by several institutions over time. The system must maintain acceptable response times as data volume increases—from thousands to hundreds of thousands of records in key entities such as students, classes, assignments, transcripts, and financial histories. To achieve this, the database schema must be well indexed on frequently queried fields, and application-layer operations must be designed to avoid loading excessive data into memory at once. Another important aspect of performance is transactional behaviour and perceived responsiveness during complex operations. When a user triggers an operation that involves multiple steps—such as enrolling a student in a class, updating tuition-related records, or applying multiple grade entries—the system must perform these steps efficiently within a single transaction, so that the user does not experience timeouts or long waiting periods. In situations where complex processing is unavoidable, the system should clearly indicate progress or completion status, rather than leaving the user uncertain about whether the operation succeeded. At the same time, transaction design must ensure that holding locks on database resources does

not unnecessarily slow down other users' requests. Finally, performance requirements imply continuous observation and tuning. The system should be deployed with logging and monitoring in place so that response times, error rates, and resource usage can be measured in real environments. This allows administrators to verify that the specified response time targets are being met and to identify endpoints or operations that need optimization. Over time, as the number of users and the volume of data grow, the institution can adjust infrastructure capacity (for example, scaling database and application instances) to keep observed performance within the agreed bounds. In summary, the system must not only perform well at initial deployment but also remain responsive and scalable as usage increases and academic operations become more complex.

3.4.2 Security Requirements

This system is sensitive in terms of security, since it contains very sensitive information: personal information of students and employees, academic data (grades, transcripts, attendance), and financial data (tuition, payments, scholarships). The security requirements thus concentrate on three pillars that include confidentiality, integrity and availability which are utilized in areas of authentication, authorization, data handling and monitoring. These requirements are established without a reference to the concrete deployment environment, but they are corresponding to the standard Java/Spring web stack including the one in the project in question. First, the system should be built to achieve strong authentication and secure account management. All users (administrator, staff, lecturer, student, parent) should use a unique account to gain access to the resources that are under protection. Passwords should not in any way be kept in plaintext format, but should be hashed with a secure, industry-standard hashing algorithm (e.g., BCrypt or Argon2) with salts and cost factors. The authentication subsystem should ensure that they have explicit account status (active, locked, disabled) and deny access without the status being active. The reset of passwords should be done through time limited, randomly generated tokens attached to a particular user which should have a defined life span and they should never be reusable. The text on the error messages on the login and reset screens should be generic without giving out the information as to the existence of a specified email/username in the system and therefore decrease the chances of user enumeration. Second, it should be a system with fined-grained authorization on a role and permission basis. Once authenticated, each request should be verified according to a role-based access control (RBAC) policy which enforces the rules mentioned above on administrators, staff, lecturers, students, and parents. One should be allowed to view and operate only the data that is relevant to him/her: e.g. a student may only see his/her grades, attendance, submissions, financial data and support tickets; a parent may only see information about his/her linked students; a lecturer may only manage the attendance, assignments and grading of the classes he/she is assigned to; a staff member may only manage an academic structure and students within the scope of their institution; and an administrator may only change the global configuration or assign roles. Any attempt to access prohibited URLs, APIs, or resources has to be blocked and display a relevant error (e.g. HTTP 403/401 or a no permission page). Checks on security cannot be spread throughout the code (e.g. by configuring Spring Security or other similar measures) but instead should be implemented centrally, to minimize the chances of bypasses. Third, the system should facilitate the data confidentiality during transit and rest. Every interaction between the clients (web browsers) and the server in the production should

be encrypted with the latest version of TLS settings, where credentials, personal information, and grades can not be deciphered. SIDs should be properly generated and only placed in cookies of HTTP-only and should be invalidated when a user logs out. Sensitive data should never be displayed in URLs or client logs-- passwords, reset tokens, internal identifiers, etc. At the data-storage level, only application-level accounts that require limited privileges to execute operations should be allowed to access the database, and only the administrators should have direct access to the database. In case the deployment environment allows it, it can be secured with encryption at rest of particularly sensitive fields (such as financial records or face data) or of the entire storage volume of the databases. Fourth, integrity of data input and validation of data is compulsory to curb corruption and exploitation. Any user entry, including input fields, URL parameters, uploaded files, etc., should be checked on the server side based on the type, length, range and format. Business rules should be implemented so that the illegal states are not able to be created: e.g., a student cannot be registered in two classes and in the same one, the grades cannot be scored in a class a student is not taking, and financial records have to be balanced. To defend against typical web vulnerabilities, the application should be based on sound patterns of database access (ORM parameter binding) to prevent SQL injection, consistently escape/ sanitize untrusted content when rendering HTML to prevent cross-site scripting (XSS), and have CSRF protection against state-changing HTTP requests. Each upload of a file, including assignment submissions and support ticket files, should have a size and file type limit, as well as should be stored in a manner that it cannot be directly executed on the server. Fifth, the system should offer auditing, recording, and tracking of actions that are of security interest. Every sensitive operation, including attempts to log in, reset a password, switch to a different role, add or edit grades and transcripts, change enrollments, conduct financial transactions, or support ticket statuses, should be logged in an audit log with no less than the following information: the time of the operation, name of the user, and type of operation and id of the target object. To ensure that such logs are not tampered with (such as by blocking write permissions and by using append-only read-only mechanisms where feasible), these logs should be stored over a specified duration in order to aid investigations, dispute resolution, or compliance audits. Meanwhile, the logs should not include confidential information, including full passwords, reset tokens, or entire financial information. Along with functional logging the system must make health and status information visible to monitoring tools so that administrators can identify signs of abnormal patterns in things like repeated failed logins, unusual error rates or spikes in certain operations which could be indicative of an attack or abuse. Lastly, the design and operation of the system should observe privacy and regulatory compliance requirements. The personal information of students, staff, and parents should also be used in academic and administrative purposes and only by authorized positions. The system should pay as little attention as possible to unnecessary exposure, such as not displaying all the personal information where it is not required, by enabling export only to trusted users, and by setting up multi-campus/ multi-institution deployments to ensure that the data of one institution does not get mixed in with those of another. The system should facilitate the processes of fixing any erroneous data, disabling or anonymizing accounts (as with graduating students), and managing data access requests in a controlled way through staff interfaces as opposed to direct database manipulation as mandated by institutional or legal policies. In general, the security requirements pose both the assurance that the deployed system in the repository is not only functional but also fulfills the confidentiality, integrity and availability of academic data in a practical deployment scenario.

3.4.3 Scalability and Reliability

The system is designed to serve a large number of users (students, lecturers, staff, parents) and store data accumulated over many years of study, so scalability and reliability are two inseparable requirements. The application uses Java as the main platform combining a three-tier web model (presentation – service – data access), packaging with Maven and supporting containerization through Dockerfile and separate deployment configuration. This allows the system to be deployed as a monolithic application but can be replicated horizontally (multiple instances running in parallel) and upgraded vertically (increasing CPU, RAM, I/O) without having to change the source code – just adjust the deployment configuration. In terms of scalability, the system must meet two aspects: scaling according to the number of concurrent users and scaling according to data capacity. First, the application must be implemented as stateless as possible at the web/service layer: session state is stored as a secure cookie or a centralized sharing mechanism (e.g., session store), shared data is stored entirely in a relational database. This allows the system to run multiple application containers behind a load balancer; when the number of users increases (especially during peak periods such as class registration, grade announcement), additional instances can be added to share the load without having to modify the business logic. Second, the database must be able to handle data growth over the academic year: each year, add students, classes, subjects, submissions, grades, financial history, support tickets, etc. The system is required to be designed with an indexing strategy for frequently queried fields (student ID, class ID, subject, campus, year of admission, creation date, status, etc.) and a pagination mechanism on all large list screens. As a result, queries such as “view a class’s student list”, “view a student’s financial transaction history”, “view a subject’s assignment slots” must still respond within acceptable time even when the total number of records has reached tens/hundreds of thousands. In addition, the system must be flexible to scale according to the deployment scope. In the scenario of using for multiple campuses or expanding to multiple training facilities, the deployment architecture must allow: (1) running multiple application instances accessing the same database or database cluster; (2) separating the database connection configuration (URL, driver, user, password) from the source code for easy switching between environments (local, staging, production); (3) supporting database separation by tenant/campus if needed in the future (e.g., each campus has a schema or a database), without completely breaking the design. These requirements are not necessarily fully implemented in the initial version, but are set as design constraints so that the system can be upgraded, scaled up/scaled out in a real-world context. In terms of reliability, the system must ensure that the academic and financial functions operate stably, without losing data and without creating inconsistent states. Important business operations – such as enrolling students in classes, creating or updating transcripts, entering grades, updating tuition fees, recording financial transactions, processing support requests – must be executed in atomically transactions: either all steps are committed successfully, or all are rolled back when there is an error. This avoids “half-baked” situations, such as deducting money but not recording payment for a specific subject, or creating a transcript but not saving detailed grades. At the same time, the system must use foreign key constraints and integrity constraints to prevent deletion or modification of “parent” records when there is still dependent “child” data, ensuring that data related to students, classes, subjects, scores, and finances are not “orphaned”. Another important requirement for reliability is safe and stable error handling. When an unusual error occurs (e.g., service-layer exception, database connection error, file upload processing error), the system must:

(1) record detailed logs for developer/devops analysis; (2) return a friendly message to the user, without revealing stack traces or sensitive information; (3) ensure that the current transaction has been rolled back and does not leave an unfinished data state. Non-critical components such as sending email notifications, synchronizing with external systems if any, should be designed to fail without stopping the entire system: if the email sending function fails, the class creation, assignment creation, grade entry, etc. must still be completed, and the email sending error is logged for later processing. Operationally, the system needs to support recovery in case of problems. Thanks to Docker packaging and isolated deployment configurations, recovering a crashed application instance or redeploying a stable version is relatively fast: just restart the container or rollback the image, the data is still kept at the database layer. In a real deployment environment, the requirement is to have a periodic database backup process (e.g. daily) and periodic recovery testing to ensure that in case of a serious problem (hard drive failure, severe configuration error, attack), the system can be restored to an operational state with an acceptable level of data loss (e.g. no more than 24 hours of data). Additionally, the system should incorporate basic health-checks/monitoring (at a minimum, application live/dead checks, database connectivity, error rates) so that the operations team can detect problems early (e.g., spikes in 500 errors, increased response times) and intervene promptly. Finally, the scalability and reliability requirements also include graceful degradation. In situations of extremely high load or an overloaded backend component, the system must prioritize keeping core functions (viewing class schedules, submitting assignments, checking attendance, viewing grades, enrolling...) still running, even if some secondary functions (e.g., downloading large reports, exporting data in bulk) must be temporarily disabled or limited. This helps ensure that from the end user perspective, the system is “not perfect but not broken”: it may be slower or lack some features, but the important academic operations are still performed, maintaining an acceptable level of reliability in a real-world operating environment.

3.4.4 Usability and Accessibility

Since the system is developed as a web application using Java, HTML and CSS, all user interactions are through a browser with a multi-screen interface: dashboard, class management, student management, schedule, grades, finance, ticket support... Therefore, usability and accessibility requirements are set to ensure that students, trainees, staff and parents – with very different technological understanding – can use the system effectively, with few errors and without too much training. In terms of usability, the system must ensure an intuitive and minimal interface between screens. The main navigation elements (role-based menus, navigation bars, links...) should be placed in a fixed position, using clear naming (e.g. “Classes”, “Timetable”, “Assignments”, “Finance”, “Support Tickets”) and not change meaning between pages. Every CRUD (create, view, edit, delete) operation screen must follow the same pattern: the list has a page and a search box; the “Create”/“Add” button is in a visible position; the detailed form always has the “Save” and “Cancel” buttons in a familiar position. This helps users, after being familiar with one module (e.g., class management), to quickly deduce how to operate in another module (e.g., support request management) without having to learn it all over again. The system must also provide clear feedback for every operation. When the user submits a form (create a class, enroll students, create an assignment, delegate, send a ticket, etc.), the system will display a success message (success) with easy-to-understand content and immediately update the

corresponding list/status so that the user can receive the results. If an operation fails (validation error, missing data, incorrect format, system error), an immediate error message should be displayed in a friendly, non-technical format and clearly associated with the field causing the error (e.g., red mark the field, display a message below the input). Messages such as “Data saved successfully”, “Registration failed because class is full” or “Deadline must be after current time” help users know what to edit, minimizing the number of trial-and-error attempts. Another important aspect is organizing information by context and role. Each user type should have its own dashboard, focusing on the main actions of the role: students see today’s class schedule, restricted assignment arrangements, new announcements; lecturers have seen the class teaching list, award slots to be graded, high absentee alerts; staff have found ticket types to be processed, enrollment requests, scholarship requests; The administrator has found the index configuration, number of users, log system... Gathering important tasks in one place helps reduce the number of clicks and reduces the “cognitive load” for the user. At the same time, each screen should only display the amount of information that is sufficient for the purpose, avoiding being intoxicated by too many tables/forms/buttons that create a confusing interface. The system needs to be paid attention to at a minimum level to streamline user errors through information design forms. Important fields such as student code, class code, date/time, number, amount... must be cleared of formatting and strictly validated; complex choices should use dropdown, date-picker, time-picker instead of input, to reduce errors. Operations that can be high risk (delete records, register names, edit transcripts, complete/cancel account transactions). Attach a confirmation dialog (confirmation) before performing. For multi-step operations (e.g., creating a scorecard, creating a complex curriculum), it is possible to divide it into small steps (wizards) or arrange groups of fields into clear sections so that users are not “overwhelmed”. Regarding Accessibility, the system must be designed to be usable by many different groups of users and on many popular devices. First of all, the interface must use easy-to-read fonts, large enough font sizes, reasonable line spacing and white space, avoid writing all capital letters or using too long sentences in labels. Colors must have high enough compatibility between text and background to be easy to read on laptop screens, PCs and all types of average quality screens. Important information (e.g., errors, warnings, status) should not be distinguished by color only (e.g., only red/green), but still use icons or descriptive text so that users with color problems can still receive it.

Since the system is a web application, keyboard accessibility is also a requirement. Interactive elements such as buttons, links, inputs, dropdowns must be able to be focused with the Tab key in a logical order on the form; Enter or Space must activate the button when focused; and focus state must be clearly displayed (e.g. border, shadow) so that the user knows where they are. This not only supports users who have difficulty using a mouse, but also helps those who are used to keyboard shortcuts to operate faster. In addition, using correct semantic HTML (heading, list, table, form) instead of just pure div will better support support tools (screen readers) if the system needs to expand accessibility in the future. In addition, the system should be cross-browser compatible and adaptable to different resolutions. Users in the real environment can access using Chrome, Firefox, Edge on Windows or macOS laptops; some may use tablets or smaller screens. The interface should be designed to display well on common resolutions, limit horizontal scrolling and avoid important buttons being “lost” from the screen. For large data tables, pagination and internal scroll bars can be used instead of forcing users to zoom in or scroll too much. Finally, for both usability and accessibility, the system should support language and style appropriate to the context: labels, notifications, and titles should be in English/Vietnamese

consistently according to the requirements of the school and the project. If multilingualism is required later, all displayed text should be separated from the code and managed in a configuration file/resource bundle, so that translation and language adjustment do not affect the program logic. Thus, 3.4.4 not only sets the goal of “beautiful interface”, but also specifies the requirements for structure, feedback, navigation, contrast, keyboard and multi-device, so that this learning management system can be used sustainably, easily understood and friendly to many different users.

3.4.5 Compliance Requirements

Table 13 Compliance Requirements

ID	Category	Requirement Name	Detailed Description	Measurement / Acceptance Criteria
CR-01	Data Protection & Privacy	Personal Data Protection Compliance	The system shall handle all personal data (students, lecturers, staff, parents) in accordance with applicable data protection and privacy regulations (e.g., national data protection law, GDPR-equivalent, institutional privacy policy). Personal data must only be collected and processed for legitimate academic and administrative purposes.	System documentation clearly identifies categories of personal data and purposes of processing. Access control enforces that only authorized roles can view or modify personal data. Data processing activities align with institutional privacy policies and are approved by the institution. No features exist that expose personal data to unauthorized third parties.
CR-02	Data Protection & Privacy	Data Minimization & Purpose Limitation	The system shall implement data minimization by collecting only the data fields required for academic and administrative processes and shall avoid storing unnecessary sensitive information.	Database and UI field definitions are reviewed against business processes to ensure each stored field has a documented purpose. No unused or redundant fields remain in production schema. Any new collection of sensitive data (e.g. face data, financial

			Each type of data must be linked to a clear purpose (e.g., enrollment, grading, finance, communication).	identifiers) requires explicit justification and approval.
CR-03	Data Protection & Privacy	Parental Access & Student Privacy Compliance	The system shall enforce access to student data by parents/guardians strictly according to institutional rules and local regulations about minors and higher-education students. Parents may only see information for students they are officially linked to and only in the scope allowed by policy.	Parent accounts are technically linked to specific student accounts via an explicit relationship. UI views for parents only expose permitted data (e.g., summary performance and financial status, if allowed). Attempts by a parent to access another student's profile must be blocked and logged. Policies on what parents can see are documented and reflected in role-based access control.
CR-04	Academic Regulations	Grading & Transcript Policy Compliance	The system shall support grading and transcript management in line with institutional academic regulations (grading scales, pass/fail thresholds, retake policies, grade change procedures).	Grading scales and grade types implemented in the system match official academic regulations. Retake and temporary retake mechanisms reflect documented policies (e.g., when a retake is allowed and how it affects transcripts/GPA). Grade change operations are restricted to authorized roles and are logged, with procedures described in institutional documentation.
CR-05	Academic Regulations	Enrollment & Curriculum Compliance	The system shall enforce enrollment rules that comply with curricula definitions and institutional policies (prerequisites, maximum course	Enrollment logic prevents students from enrolling in subjects/classes that violate curricular rules (e.g., missing prerequisites, exceeding load). Test cases or documented scenarios

			load, compulsory vs elective subjects, admission year constraints).	show that invalid enrollments are rejected with clear messages. Curriculum definitions (required subjects per programme and admission year) in the system match official programme specifications.
CR-06	Financial & Accounting	Financial Record & Tuition Policy Compliance	The system shall manage tuition, scholarships, and financial transactions in accordance with institutional fee policies and applicable financial regulations. It must maintain accurate records of charges, discounts, payments, and balances for each student.	Tuition calculation and scholarship application logic are documented and consistent with official fee tables. Every financial transaction (deposit, payment, support ticket charge) is logged with correct amounts and timestamps. Internal or external audits can trace how a student's balance was computed over time. Any correction to financial records is traceable and authorized.
CR-07	Financial & Accounting	Record Retention for Financial Data	The system shall retain financial records (transactions, balances, scholarship allocations, support ticket charges) for at least the minimum period required by institutional and legal regulations for accounting and audit purposes.	Retention period for financial records is defined (e.g., X years) and documented. There is no automatic deletion of financial records before the retention period ends. Any archival or anonymization process preserves required information for audit. Compliance can be demonstrated during financial audits by retrieving historical records.
CR-08	Data Retention & Archiving	Academic Record Retention	The system shall retain core academic records (enrollments, grades, transcripts, attendance summaries) for the	A formal retention policy exists (e.g., academic records retained for the lifetime of the institution or a regulatory minimum). The system does not

			period required by institutional and regulatory requirements, including long-term archiving for graduates.	delete core academic records automatically unless an approved archival/anonymization process exists. Archived records remain retrievable for authorized staff for verification and certificate issuance.
CR-09	Data Retention & Archiving	Right to Rectification & Corrections	The system shall support institutional processes for correcting inaccurate personal or academic data in compliance with regulations (e.g., right to rectification). Corrections must be controlled, auditable, and performed only by authorized personnel.	Interfaces exist for authorized staff to correct personal information or academic entries (e.g., fixing mis-typed names, correcting grades after formal approval). All corrections are logged (who, when, what changed). Students can request corrections through defined channels (e.g., support tickets), and the system can reflect approved changes without direct DB manipulation.
CR-10	Security & Compliance	Security Best Practices (OWASP / Institutional Policy)	The system shall adhere to industry-standard security practices and institutional security policies. It must implement controls to mitigate common web vulnerabilities and protect confidential educational data.	Security checks (e.g., OWASP Top 10) are part of pre-release review. Code and configuration enforce secure authentication, authorization, input validation, CSRF protection, and logging of security events. The institution's IT or security office can review the system and confirm alignment with internal security guidelines.
CR-11	Security & Compliance	Audit & Regulatory Inspection Readiness	The system shall maintain sufficient logs and structured data to support internal and external audits related to	The system can produce reports and logs that show: who changed a grade, who updated a financial transaction, who processed a support ticket,

			academic integrity, grading, financial operations, and data protection compliance.	and when these actions occurred. Logs are retained for a policy-defined period and are accessible to authorized auditors. No critical decision-making process (e.g., passing a student, releasing a scholarship) is completely opaque or untraceable.
CR-12	Accessibility & Legal Requirements	Accessibility Regulation Alignment (where applicable)	The system shall support basic accessibility requirements in line with institutional guidelines and, where relevant, national or international standards (e.g., WCAG-inspired principles) so that users with certain disabilities are not unfairly excluded.	UI follows key accessibility practices: sufficient color contrast, keyboard navigation, clear labels, and understandable error messages. Institutional accessibility reviewers can navigate core use cases (view timetable, submit assignment, view grades) without encountering blocking accessibility issues. Any formal accessibility requirement by the institution is documented and mapped to system behaviour.
CR-13	Communication & Email	Anti-Spam and Email Policy Compliance	The system shall comply with institutional and legal rules on email communication (e.g., anti-spam rules, consent management, institutional branding requirements) when using email templates to contact students and parents.	Email templates use institutional-approved branding and content structure. Bulk notifications are only sent to relevant recipients (e.g., enrolled students for a class). If regulations require consent for certain types of emails (marketing vs academic notifications), the system supports respecting these preferences. No uncontrolled mass emailing is possible from general user accounts.
CR-14	Intellectual Property	Intellectual Property &	The system shall respect intellectual	Access to teaching materials and documents

		Content Usage Compliance	property rights for stored and distributed content (syllabuses, documents, assignments, learning materials) and prevent unauthorized sharing beyond institutional policies.	is role-based (students in a class only see materials for that class, etc.). There is no public, unauthenticated access to internal documents. Institutional policies on copyright and content usage are documented; system features (such as download or sharing links) do not contradict those policies.
CR-15	Disaster Recovery & Business Continuity	Compliance with Institutional DR/BC Policies	The system shall align with the institution's disaster recovery (DR) and business continuity (BC) policies to ensure continuity of critical academic operations in case of system failures or disasters.	Backup and restore procedures are documented and tested according to institutional DR/BC policy. Recovery Time Objective (RTO) and Recovery Point Objective (RPO) for academic and financial data are defined and achievable with the implemented backup strategy. Evidence of backup tests (e.g., restoration drills) can be shown if requested by auditors or management.
NFR-16	Usability	Error Messages & Validation Feedback	The system shall provide clear feedback when user input is invalid or when operations fail.	Validation errors must be displayed near the corresponding input fields with meaningful messages (e.g., "Deadline must be in the future"). For system errors, a generic error page must appear without exposing stack traces, and logs must contain technical details for developers.
NFR-17	Maintainability	Code Modularity & Layered Architecture	The system shall follow a modular layered architecture (presentation, service, persistence) to simplify	The application structure separates controllers, services, repositories, and domain entities. Adding a new feature (e.g., a new type of academic report)

			maintenance and enhancements.	should primarily affect a small number of well-defined modules without requiring pervasive code changes.
NFR-18	Maintainability	Coding Standards & Documentation	The system codebase shall follow consistent coding standards and include documentation for key modules and APIs.	Source code adheres to a defined style guide (e.g., standard Java/Spring conventions). Core business logic methods and public APIs include comments or JavaDoc. A basic developer guide or README exists describing build steps, environment variables, and deployment configuration.
NFR-19	Maintainability	Testability & Automated Tests	The system shall be testable at unit and integration levels, enabling automated regression testing.	Critical business logic (e.g., enrollment rules, grade calculation, financial transactions) is covered by unit and/or integration tests. A minimal automated test suite can be executed via standard build tools (e.g., Maven/Gradle) and must pass before release.
NFR-20	Portability	Environment & Database Independence	The system shall be deployable on standard Java application servers and support common relational databases with minimal configuration changes.	The application runs on any environment supporting Java/Spring (e.g., local machine, on-prem server, cloud container). Database connection details (URL, credentials, dialect) are externalized in configuration files; switching from one SQL database to another requires configuration changes but no core code change.
NFR-21	Portability	Browser Compatibility	The system's web interface shall support modern browsers commonly	The UI must be usable and visually correct in current versions of major browsers (e.g., Chrome, Firefox,

			used by students and staff.	Edge). Minor visual differences are acceptable, but no major functional feature (forms, navigation, uploads) may break on supported browsers.
NFR-22	Localization	Multi-language Support (if applicable)	The system shall support localization mechanisms to allow multiple languages (e.g., English and Vietnamese) for UI labels and messages.	All user-facing text is externalized into message bundles or a similar mechanism. Switching the language in configuration or user settings updates labels and messages without modifying source code.
NFR-23	Data Integrity	Transactional Consistency	The system shall ensure that multi-step operations (e.g., enrollment + financial updates, grading + transcript updates) are executed atomically.	Database transactions must be used so that either all steps of an operation succeed or none are committed. In case of errors during processing, the system must roll back partial writes and return a clear error. No user should observe partially updated academic or financial records.
NFR-24	Data Integrity	Referential Integrity & Constraints	The system shall enforce referential integrity for key relationships (students–classes, subjects–curricula, financial histories–account balances, etc.).	Database schemas must define primary and foreign keys appropriately. It must be impossible to delete parent records when child records still exist, unless referential actions (e.g., cascade, restrict) are explicitly designed. Integrity must be maintained during imports, updates, and deletion operations.
NFR-25	Logging & Monitoring	Application Logging	The system shall log important application events, warnings, and errors to support debugging and monitoring.	Logs must include timestamps, log levels (INFO/WARN/ERROR), and contextual information (e.g., user, request ID) where possible. Log size

				and rotation should be configurable. Logs must not contain confidential data (passwords, full credit card numbers, etc.).
NFR-26	Logging & Monitoring	Operational Monitoring	The system shall expose basic health and status information for monitoring tools (e.g., uptime, database connectivity, error rates).	Health check endpoints or integration with application monitoring (e.g., Spring Actuator, or equivalent) must provide status (UP/DOWN) and basic metrics. Operations team (or developers) can quickly detect if the system is down or experiencing abnormal error rates.
NFR-27	Compliance	Academic & Privacy Regulations	The system shall support compliance with institutional policies and relevant privacy laws regarding student data.	Access to personal and academic data is restricted to authorized roles only. Data exports for reporting are controlled and logged. Procedures for data correction and removal (where required by policy or law) can be carried out by authorized staff without code changes.
NFR-28	Disaster Recovery	Recovery Objectives	The system shall support recovery from catastrophic failures (e.g., hardware failure, data corruption) within defined recovery targets.	Recovery Time Objective (RTO): system can be restored to an operational state within 4–8 hours after a major incident. Recovery Point Objective (RPO): data loss is limited to the time between the latest backup and incident (≤ 24 hours). Backup and recovery procedures must be documented and tested periodically.

3.5 Competitive Analysis

3.5.1 Features of Competing Solutions

The proposed software must compete directly with many widely used training management and learning support solutions, typically popular LMS (Learning Management System) systems: Moodle, Canvas, Google Classroom, Microsoft Teams for Education, along with SIS (Student Information System) systems that some universities build themselves or buy from external suppliers. In general, this platform supports core functions such as course and class management (class creation, member resolution, student addition, schedule setting), learning content management (downloading documents, slides, lectures, videos), online assignment and utilization (homework, deadlines, submission of grades, online grading) and learning outcome tracking (storing component grades, final grades, transcripts). In addition, most competing systems provide communication tools between students and learners such as discussion forums, comment areas under assignments or posts, internal mailboxes or email integration, which helps reduce the need to use external communication channels. In addition to functional facilities, many solutions are also developed with additional advanced features to enhance user experience and support large-scale training management. Pharmaceuticals, some systems allow electronic attendance by class, integrate class schedules - class schedules into calendars, send automatic notifications to students when new assignments are available, when assignments expire or when updates are received. A platform that supports very detailed authorization for administrators, students, teaching assistants, students and parents, provides statistical reports, intuitive dashboards on class participation rates, on-time assignment rates, score analysis, thereby helping schools and lecturers easily evaluate course quality. In addition, commercial solutions often have additional mobile applications, the ability to integrate with email systems, Single Sign-On (SSO) or external services such as Zoom, Microsoft Teams, Google Meet, etc. to make the development, declaration and operation process more convenient, especially in the context of blended learning (blended learning) and online learning. However, although these systems have relatively rich features and have been proven to be stable, they are often designed in the direction of "one model for many schools", lacking specialized functions for each organizational model or training program tool. Most competing solutions focus heavily on learning content and scores in each course, but pay attention to the tight connection between main classes - sub-classes, displaying student upgrade progress through program stages, or detailed management with major financial enterprises (with limitations such as account balance, deposit history, tuition payment by subject, tuition by year and school). This creates space for a new system that can synchronize multiple service areas within the same platform, optimized for the school's toolkit context and training process.

3.5.2 SWOT Analysis

In that context, a SWOT analysis can be conducted for the proposed GUMS system as follows. In terms of strengths, the system has a clear advantage in the ability to consolidate many operations that are currently scattered across AP, CMS, and FLM systems into a unified platform, helping to reduce data duplication, limit information errors, and optimize the workflow of

stakeholders. The data model is designed to be rich in semantics, reflecting in detail the specific training characteristics of Greenwich Vietnam: clear separation of major/minor, student upgrade stages, classroom management, timetables, attendance, assignments, learning outcomes, lecturer evaluation, parent-student relations, along with an internal financial module (account balance, deposit history, tuition fees by subject, by year, and by campus). Deployment on the Spring Boot platform, JPA/Hibernate and domain-oriented architecture helps the system have a relatively clear code structure, easy to expand and convenient for maintenance, integrating new functions. In addition, the authorization for many user groups (students, major/minor lecturers, staff, admin, parents) is closely linked to the data model, allowing the construction of precise functions according to roles instead of just stopping at the general level like many pure LMS solutions. However, besides the strengths, the system also has many weaknesses when compared to mature commercial or open source solutions. GUMS is still a project-scale product, has not undergone a large-scale deployment and practical testing for many years like Moodle or Canvas, so there are still potential risks in terms of performance, load capacity and long-term stability. Some of the convenient features that users are familiar with on large platforms – for example, native mobile applications, rich plugin repositories, advanced statistical and reporting tools, big data-based learning analytics – are not fully realized in the current version. In addition, the rich domain design, many tables, many inheritance relationships and compound keys, although highly expressive, can also make onboarding new development teams more complicated, requiring careful documentation and a strict development process to avoid creating additional technical debt. In terms of opportunities, GUMS is built with the orientation of directly serving Greenwich Vietnam's specific training model, so it has the ability to deeply customize the internal process, thereby improving the user experience compared to "adjusting to fit" on a system shared by many schools. Simultaneously possessing both academic data (classes, scores, majors) and financial data (tuition, payments, account balances) and behavioral data (attendance, assignment submission, class interaction, lecturer evaluation) creates a very good premise for the future integration of additional analysis modules, study suggestions, early warnings for students at risk of low scores or dropping out. The system can also be expanded to other campuses, joint programs, or even move towards a customizable product model deployed for universities with similar characteristics in the region. In addition, the choice of popular technology (Java Spring, RDBMS) helps GUMS easily integrate with existing services such as SSO, email systems, online meeting platforms, as well as take advantage of available programmer human resources on the market. Finally, the system also faces a number of challenges and risks from the external environment. The market for training management solutions and LMS is currently very competitive with the presence of many platforms that have affirmed their brands, have large communities, rich documentation and an extensive plugin ecosystem, making it necessary to convince schools to switch or invest deeply in a new system to demonstrate clear benefits in terms of cost, management efficiency and user experience. The risk of user resistance to change is also a significant barrier if training, user manuals and technical support are not properly prepared. In addition, increasingly strict requirements on information security, personal data protection, and compliance with current legal regulations can increase the cost and complexity of system operations. Ultimately, without a long-term strategy for maintenance, updating and development, GUMS risks becoming obsolete in the face of rapid technological change and new school needs, thereby losing its competitive edge over solutions developed and supported by more resourced vendors.

3.5.3 Market Differentiation Strategy

The strategy to differentiate GUMS is primarily based on positioning the system as a specialized solution for the Greenwich Vietnam context, rather than trying to become a “one-size-fits-all” platform like many existing LMS or SIS. The teacher only focuses on managing learning content and scores for each section, GUMS is designed around the entire learning lifecycle of students in a specific training model: clearly separating main classes - small classes, managing the upgrade phase (upgrade) between classes, closely linking classes - timetables - attendance - assignments - transcripts, and integrating internal financial modules with learning by subject, by year and by school, account balance, deposit and payment history. This vertical connection helps GUMS create a comprehensive picture of each student's learning and financial actions, while competing systems often only provide a few discrete modules or through different systems. Second, GUMS chooses to differentiate itself in terms of game integration and interaction channels. The system not only serves students and learners like conventional LMSs, but also includes user groups such as staff, administrators, major/minor students, parents, along with appropriate interaction mechanisms: class posts, comments, internal messages, public blogs, assessment prices, ticket support. Instead of parties having to use multiple separate channels (email, social networks, private chat applications), GUMS aims to collect interactions related to learning and training management in one platform, thereby creating a “one-stop” (one-stop) experience for users. At home, this not only improves operational efficiency but also opens up the ability to track, analyze student user behavior and engagement on a central data source. Third, at the technical level, GUMS’s other chemistry strategy lies in using a rich semantic data model and domain-oriented architecture as a long-term competitive advantage. The system applies inheritance to core concepts (attendance, transcripts, class posts, comments, timetables, classes), uses compound keys and intermediate entities to accurately describe many-to-many relationships in an academic environment, and normalizes state through an enum system. This approach not only makes the codebase highly expressive, but also lays a foundation for adding advanced analytics features in the future (e.g., progress tracking, early detection of at-risk students, multidimensional statistics on teaching quality and class participation). While many other systems choose to minimize the data model to easily develop and exploit for the majority, GUMS accepts a higher level of complexity to change the reasonable depth with the operating characteristics of a training unit. Finally, GUMS is oriented differently by using a product strategy that is closely tied to the development and continuous improvement strategy. Since the system is a fixed software package, the document sets the goal of using GUMS as an “evolving platform”, capable of expanding the module in stages: first focusing on core services (classes, timetables, attendance, assignments, finance), then gradually adding advanced features such as administrative dashboards, campaign reports, learning analytics tools, deep analytics with SSO, email, online platforms and existing school systems. In parallel, collecting regular feedback from user groups (students, trainees, staff, administrators) is seen as part of the differentiation strategy, helping GUMS adapt quickly to changing practical requirements, thereby forming a competitive advantage based on the rationality and high acceptance of end users - adjusting “boxed” solutions from third parties that can be in a multi-lobed environment like Greenwich Vietnam.

3.6 System Architecture Design

3.6.1 High-Level Architecture

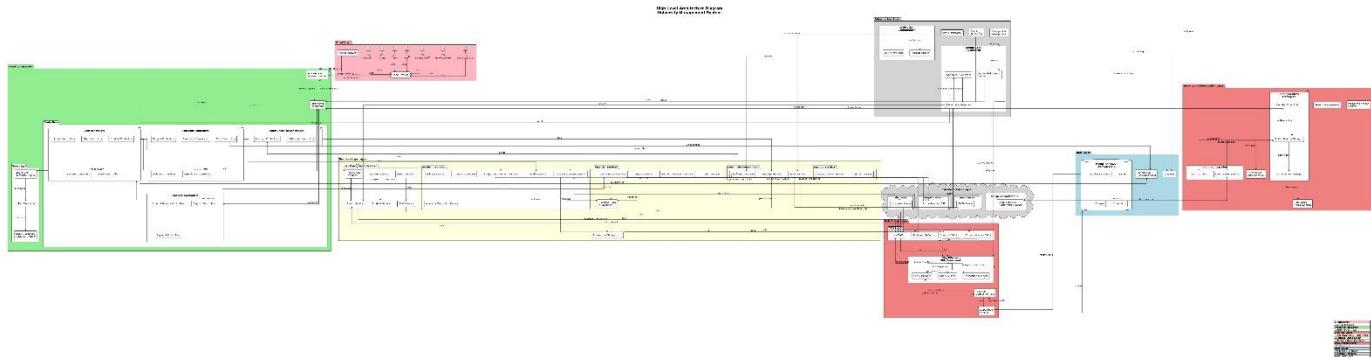


Diagram 1 University Management System – High-Level Logical Architecture

This logical architecture diagram is a high level presentation of the University Management System in the form of a layered, modular platform whose responsibilities lie distinctly between the browser of the user and the database as well as the external services. Client Layer, at the edge, brings together all user roles (admin, staff, lecturers, students, parents, deputy staff) and their access channels (web and mobile browsers) making it evident that all people interact with the system via HTTP/HTTPS. The initial hits were made on the Presentation Layer and processed by an HTTP/HTTPS handler and Spring MVC, which then routes to the relevant controller groups: user controllers (account and role specific flows), academic controllers (classes, subjects, timetables, syllabuses, and attendance), financial controllers (account balances, deposits and payments), and communication controllers (messages and notifications). Views are compiled using a Thymeleaf based view layer (template engine, html templates, static resources), that converts the output of the service into html responses to be sent to the browser. Security is concentrated on the Security and Authentication Layer whereby Spring Security has a filter chain, which filters out all requests, and hands over to authentication and authorization managers. This layer can use both form-based authentication and OAuth2 Authentication, and an OAuth2 client and user service are integrated with Google OAuth2 and session management, password encoding (BCrypt), and role-based access control are used to make sure that only authorized users can access the controllers and services. The Business Logic Layer in the background provides all of the core use cases: user management services (admin, staff, student, lecturer, parent), academic services (class, subject, timetable, syllabus, attendance, transcripts), financial services (account balance, deposit, payment processing, tuition), communication services (message, email, notifications), and support services (ticket, comments). Business rules/validation and transaction management have common components that make sure that there is consistency among domain logic and that changes are applied atomically. The Data Access Layer is responsible to data-centric concerns and utilizes JPA/Hibernate (entity manager, query builder, transaction manager) and domain-specific Data Access Objects (user, academic, financial, communication) to decouple services and persistence issues. The HikariCP connection pooling and MySQL JDBC driver are used to provide efficient and reliable access to the Data Layer, which

is a MySQL 8.4 database holding schemas, tables, indexes, and constraints, and other components are used to model file storage of uploaded content and operational records on the log files. The Layer of External Services includes the integrations with Google OAuth2 (authentication), Stripe (payments), Gmail SMTP (email), and Jitsi Meet (online rooms), which is why the openness regarding the fact that some of the capabilities are delegated to a third-party service is apparent. Lastly, there is the Infrastructure Layer, which demonstrates the technical runtime: Spring Boot application with embedded Tomcat server, managed by the Spring IoC container and dependency injection, WebSocket configuration (STOMP + message broker) of real time message delivery, caching (EhCache), asynchronous processing, and centralized configuration management. Combined, these layers depict an archetypal multi-layer architecture: requests are sent out to clients through security and presentation to services, to DAOs and the database, and responses and notifications are received by the same back upwards, and infrastructure and external services offer the cross-cutting functionality that renders the system secure, scalable and maintainable.

3.6.2 Component Diagram

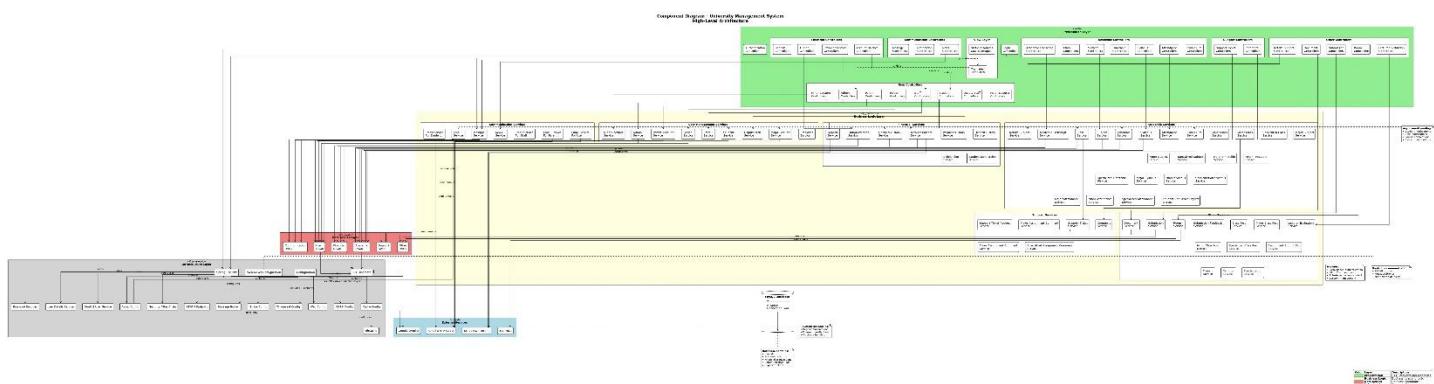


Diagram 2 University Management System - High-Level Component Architecture

The University Management System - High-Level Component Architecture proposes the whole platform as a layered, modular system with responsibilities distinctly divided among presentation, business logic, data access, infrastructure and external services. On the external services, we have the External Services package which includes integrations that are third-party with the system: Google OAuth2 has federated authentication, Stripe has payment APIs via which the system can perform financial transactions, and the SMTP server of Gmail is used to deliver outbound email messages, and Jitsi Meet has online meeting rooms. A common MySQL Database is located at the persistence boundary and is accessed using a connection pool (HikariCP) to effectively handle JDBC connections and enhance performance and use of resources. The Presentation Layer has a rich collection of controllers which is domain-oriented and centered internally to make HTTP concerns focused and cohesive. Endpoints are exposed through the User Controllers package (Admin, Staff, Deputy Staff, Major/ Minor Lecturer, Student, Parent, Person Controllers) and the Academic Controllers package (classes, subjects, timetables, syllabuses, attendance, transcripts and curricula) through which all the user roles are exposed, along with

their workflows, and the Financial Controllers package (account balances, deposits, tuition, and payment history) which exposes the workflows of those roles. The package Academic Controllers exposes the user workflows of all of Cross-cutting presentation characteristics, e.g. Login Controller and Authentication Controller, are specifically structured to communicate with the security infrastructure. The View Layer deals with view rendering, with Thymeleaf templates and statical assets (CSS/JS/images) rendering the final UI to the end user, templates rendering responses returned by controllers and statical resources serving the frontend. Below that, the Business Logic Layer is a reflection of the controller structure and is based on domain services instead of HTTP endpoints. The User Management Services (Admin, Staff, Deputy Staff, Major/Minor Lecturer, Student, Parent Account, Person, Employee Services) contains the user and role rules; Academic Services deals with class type (major/minor/specialized), subject and subject variant, timetable (including major/minor/specialized timetables), syllabus, attendance, academic transcripts, curriculum, and subject requirement; Support Services deals with support tickets and various flavors of assignment comments (major, minor, specialized); and Other Services deals with rooms, documents, submissions, submission feedback, All these services delegate persistence operations to the Data Access Layer which groups domain-specific DAO groups (UserDAOs, AcademicDAOs, FinancialDAOs, CommunicationDAOs, SupportDAOs, OtherDAOs). Such DAOs, in turn, are based on JPA/Hibernate to project the entities to the MySQL database by means of the HikariCP Connection Pool, which creates a clean chain of responsibility: controllers - services - DAOs - ORM - database. Capabilities that are cross-cutting are captivated in the Infrastructure Layer accomplishing technical plumbing and system-wide issues. The spring security cluster (Security Filter Chain, OAuth2 User Service, User Details Service, Password Encoder) performs authentication and authorization, connects to Google OAuth2, and links to the user domain to load and authenticate credentials. STOMP endpoints and a real-time communication message broker (e.g., notifications, live updates) are configured via the WebSocket Configuration, whereas all technical settings are in a single location (e.g., Thymeleaf, web, CORS, cache, async, and Stripe settings) specified via the general Configuration component. EhCache caches (as mentioned in cache config) and has non-blocking configurations such as email sending. The arrows in the relationships across the diagram indicate the desired dependency flow: controllers rely on services; services rely on DAOs; DAOs rely on JPA/Hibernate and the connection pool; security and configuration components support authentication, HTTP behavior, caching, asynchronous execution, and integration of external services including Google OAuth2, Stripe, Gmail SMTP and Jitsi meet. In total, the diagram presents a system that is horizontally (domain) (user, academic, financial, communication, support, other) and vertically (UI to database) layered which allows the architecture to be easy to understand at a glance and closely aligned with best practices in enterprise.

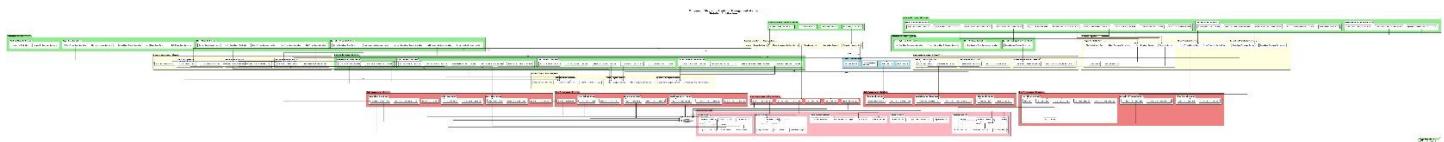


Diagram 3 Academic Management Module - Detailed Component Architecture

The Academic Management Module - Detailed Component Architecture divides the academic domain into a highly structured component model in layers that divide concerns away both implements UI types of concerns down to persistence and domain models. The top level of the presentation is divided into specialist controller groups of the various academic concepts, each controlling classroom operations (such as generic Classroom Controller and Member In Classroom Controller), and concrete flows relating to each kind of class (major, minor, specialization) such as adding, editing, listing, and searching classes, subject-related operations such as subject approval and subject restudy lists, and other academic operations such as attendance, student academic transcripts, curriculum and retake subject. All these controllers push business rules down to the business logic level, where services are packaged together by domain: class services are centralized by an AbstractClassesService, which contains the common behavior, extended by MajorClassesService, MinorClassesService, and SpecializedClassesService (and their Impl implementations) to support type-specific logic, and subject services are analogous with abstract subjects service and concrete major/minor/specialized services; timetable services are separated into a generic TimetableService and major, minor, and specialized timetable services, each with an implementation and SlotsService to manage individual time Below that, each domain has its data access layer, using a DAO (data access object), abstract ones (abstract data access object) indicating shared patterns of persistence and concrete ones indicating major, minor and specialized variants and timetable ones (DAOImpl) providing the actual database interaction and the DAOs visible to the services defining the repository interfaces. The model/entity layer contains the domain structures on which the DAOs are operating: the class models distinguish between AbstractClasses and concretely implemented MajorClasses, MinorClasses, SpecializedClasses, and concatenated association objects, including StudentsClasses and LecturersClasses; the subject models give the same pattern between abstract and concrete subjects; the timetable models define a AbstractTimetable base with major/minor/specialized timetables and slot objects; the syllabus models with major, minor and specialization syllabuses; and the other academic models with academic transcripts, curriculum, retake Relationships are regular and directional: controllers call services; services are dependent upon DAOs; DAOs operate on models; and all DAOs finally go to a common MySQL database. Cross-module dependencies are represented explicitly: class services need subject services to authenticate and map classes to subjects; timetable services need class services and room management (through external RoomService) to compose valid schedules, syllabus services need subject services to resolve syllabus content; retake subject services are combined with non-financial and user management services to obtain payment and eligibility, and academic transcript services are combined with class services to consolidate outcomes across the various class types. This layered architecture is supported by inheritance relationships at the model layer (e.g., AbstractClassesModel extended by major/minor/specialized models, and similarly subjects and timetables) which forms a polymorphic domain foundation, and makes the entire module extensible (new academic types or flows can be plugged in by adding parallel controllers/services/DAOs/models) and maintainable (as each concern (presentation, business logic, data access, domain modeling) is well isolated but also well linked).

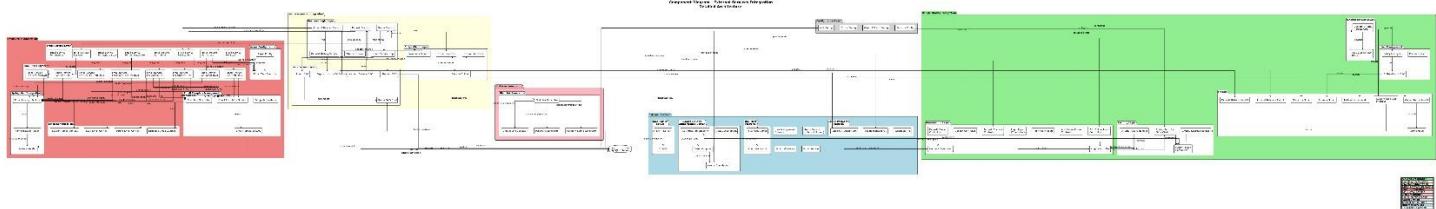


Diagram 4 External Services Integration - OAuth2, Payments, Email, Meetings

The External Services Integration - OAuth2, Payments, Email, Meetings diagram illustrates the integration of the university system with four significant external platforms (Google OAuth2, Stripe, Gmail SMTP, and Jitsi Meet) with a vivid layering of configuration, presentation, business logic, data access and specialist client/utility components. On the surface, it has External Services package containing the Google OAuth2 Authorization Server, which provides authorization endpoint, token endpoint, user info endpoint, and JWK set endpoint to implement the entire OAuth2/OIDC authentication flow; the Stripe Payment Platform, which offers a checkout session endpoint, payment intent endpoint, and webhook for online payments and confirmation; the Gmail SMTP Server, which is an SMTP server with TLS/SSL security to transmit emails; and the Jitsi Meet Platform, which exposes a Jitsi server and room generator to create online meeting rooms. It has a common Configuration Layer, containing Security Config, Stripe Config, Mail Config, and OAuth2 Client Config, which is used to initialize and wire up the security filters, OAuth2 client properties, Stripe API keys and mail sender settings around which the rest of the system is built. There is a sub-architecture within each of the integration domains. In the case of Google OAuth2 Integration, the presentation layer comprise of a LoginController and a Thymeleaf-based login page, which initiates the process of the login when a user clicks Sign in with Google. Security layer represents the Spring Security pipeline: a SecurityFilterChain is forwarded to an OAuth2LoginConfigurer, and the success/failure handlers handle the redirection logic to call after the login. The OAuth2 service layer adds a CustomOAuth2UserService which is based on an underlying OidcUserService, which makes a call to the user info endpoint of Google, converting the claims received to a domain-specific CustomOidcUserPrincipal and delegating to the user management components. The user management package consists of an EntityManager, PersonService, and AuthenticatorDAO that query PersonsModel and AuthenticatorsModel to find a match between accounts, identify roles, and store data relating to authentication. The flow connecting the relationships is as follows: login page, login controller, security filter chain, OAuth2 login configurer, Google endpoints, custom user service, DAO and models, custom principal, success/failure handlers, controller, which reflects the whole authentication cycle by SecurityConfig, OAuth2ClientConfig. In the case of Stripe Payment Integration, the architecture begins with the presentation layer and has controllers specific to payment scenarios: DepositController is used to start payment, and success and cancel controllers are responsible to respond to the return URL and display feedback to the user. These controllers are delegated to the business logic layer, which consists of DepositService, AccountBalanceService, DepositHistoryService, and StudentService, that coordinate the generation of payments, updating of balances and recording of the history and retrieving a student. StripeSDK is used directly to describe the implementation of the StripeClient Layer: the StripeConfig is used to configure the StripeSDK with API keys, and a checkout session is built with the StripeCheckoutAPI; the payment is verified with the StripePaymentAPI by the StripeSessionRetriever and StripePaymentValidator.

Stripe-related data Data Access Layer takes the form of DAOs (AccountBalanceDAO, DepositHistoryDAO, StudentDAO), and they store themselves to models (AccountBalancesModel, DepositHistoriesModel, StudentsModel), and save to the common MySQL database. The relationships encode a realistic flow of payments: the user initiates a deposit, the system opens a Stripe checkout session and redirects them to Stripe, and on successful completion it checks the session, updates the account balance, records the deposit history, and links all this to the appropriate record of the student. The Email SMTP Integration is organised on the basis of an enriched email service chain. The Email Service Layer determines individual services to each audience: students, staff, parents, admins, lecturers, deputy staff, minor lecturers and generic users, in order that business logic can select a semantically appropriate entry point (such as EmailServiceForStudent) without consideration of low-level implementation particulars. Each of these services invokes its own DAO in the Email DAO Layer which communicates with the Email Template Management package (EmailTemplatesService and EmailTemplatesDAO), to fetch templates out of the database and makes use of aTemplateGenerator and aTemplateValidator to construct final email bodies. StudentEmailContext, StaffEmailContext, ParentEmailContext, ScheduleEmailContext context-specific DTOs can influence data sent to templates to make each email properly personalized and fitted with schedule information. Technically, the Spring Mail Integration package (JavaMailSender, MimeMessage helper and Email message builder) is used to construct MIME messages and actually send them to the SMTPserver through the SMTPServer secured by TLS. JavaMailSender is configured by MailConfig and the package AsyncConfiguration contains AsyncConfig and EmailTaskExecutor to ensure that email sending is asynchronous, and thus more responsive. The dependencies have a pipeline structure: email services - email DAOs - template service and generator - context DTOs - message builder - mail sender - SMTP/TLS with the AsyncConfig allowing non-blocking execution. Lastly, the Jitsi Meet is a model that is used to describe how the system establishes and maintains online meeting rooms. RoomControllers and online room adding and editing controllers are in the presentation layer and make calls to the RoomService in the business logic layer (implemented by RoomServiceImpl). That service implements the data access layer of RoomDAO/RoomDAOImpl which interacts with the Jitsi Link Generation package- JitsiLinkGenerator, UniqueLinkValidator, PasswordGenerator and RandomStringGenerator to create unique and secure meeting URLs and passwords. JitsiLinkGenerator liaises with the external JitsiServer and JitsiRoomGenerator to generate real rooms on the Jitsi platform, which is then saved in OnlineRoomsModel and RoomsModel in the models package and RoomDAO in MySQL. All this is bound together by cross-integration dependencies: the DepositService and AccountBalanceService access the email services to send the confirmations and balance changes, and the RoomService accesses the email to send the meeting links to the users and this illustrates how all the payments, communications and meetings are all coordinated around these external integrations. In general, the diagram displays a clean and layered integration architecture; configuration is at the core, auth, payments, email, and meetings are distinct integration modules, client and DAO layers are explicit, and there is a clear flow of controllers to services to integration clients/DAOs to external services and database, which makes the system both extensible and operationally understandable.

3.6.3 Deployment Diagram

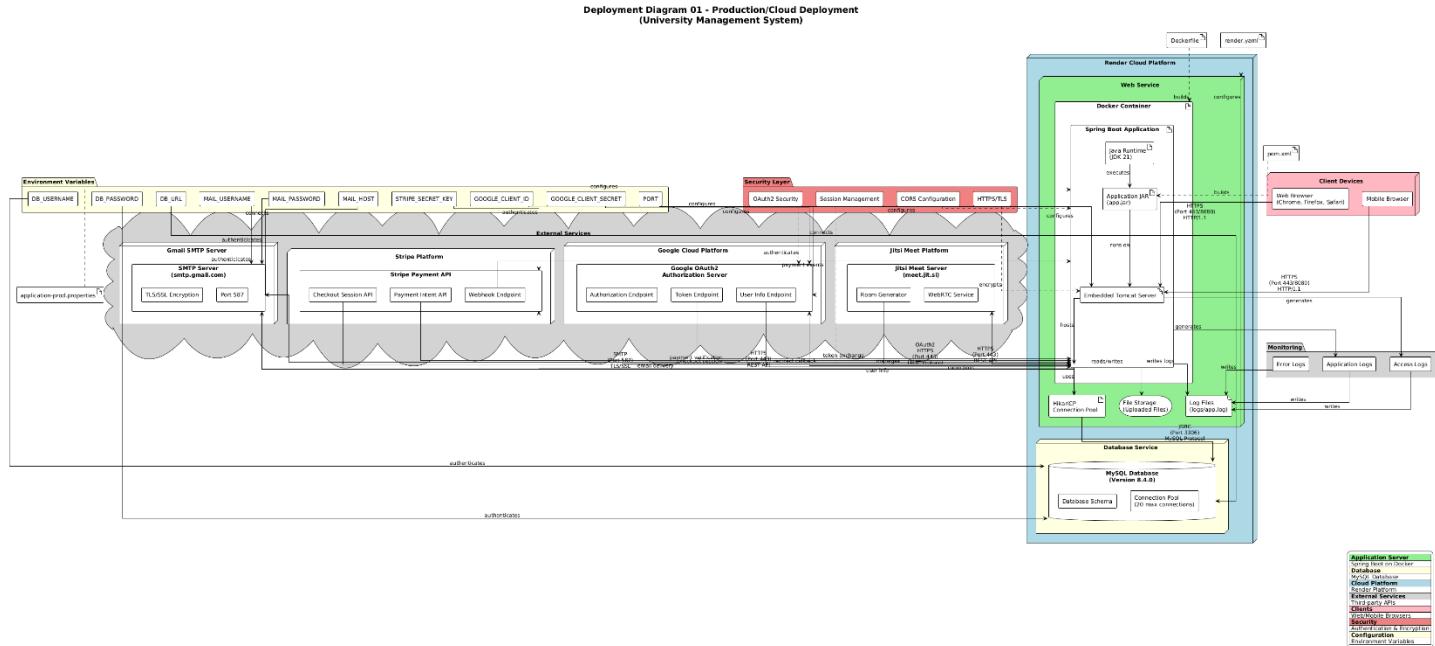


Diagram 5 University Management System – Deployment 01: Production / Cloud Architecture

The University Management System - Deployment 01: Production / Cloud Architecture diagram outlines the way the app would be run in a real production cloud by illustrating all the key runtime nodes, the artifacts deployed to them and the external services with which it will interact. The core is composed of the Render Cloud Platform that includes a Web Service node that is the primary application server. The system is installed in this node as a Docker container containing a Spring boot application; the same application involves a Tomcat server, the compiled application JAR (app.jar), and the Java runtime (JDK 21), which is why HTTP traffic, business logic and runtime are all installed in the same containerized unit. The app relies on HikariCP to access the database efficiently and with a high level of performance, using JDBC connection pool, and to write and read user-generated files into file storage space and records both operational and access data into specific logs files in the server. An independent node of Database Service is a representation of the production MySQL 8.4.0 instance where the schema of the database is stored and it has its own connection pool, and it is defined as the long-term storage of all the academic data, user data, financial data, and communication data. The diagram around this core depicts a group of external services that the application integrates with on the internet: Google OAuth2 on Google Cloud to authenticate and identify the user (using explicit authorization, token, and user-info endpoints) and integrate with Stripe to process payments (checkout sessions, payment intents, and payment event webhook), Gmail SMTP server to deliver outbound email over TLS/SSL on port 587, and the Jitsi Meet platform to create and manage online meeting rooms and WebRTC sessions. At the client, a Client Devices node has Web and mobile browsers which can access the system on any port using HTTPS (usually port 443 mapped to the internal Tomcat port 8080), which points out the fact that users can access the system using normal modern browsers. The connections in the diagram are apparent: browsers access the built-in Tomcat in the containerized version of the Spring Boot app; the app accesses MySQL via JDBC and HikariCP; it is logged in with Google OAuth2 and Stripe payment system as well as Gmail SMTP email and Jitsi video call; and the app stores data to file and log files. There are also deployment

artifacts like the Dockerfile, render.yaml, application-prod.properties, and pom.xml, which are connected to the runtime objects that they construct or define and which explicitly convert source and configuration into a running production system. Further packages of Security Layer (HTTPS/TLS, OAuth2 security, session management, CORS) and Monitoring and Logging (application, access, and error logs) point to the fact that production deployment is not merely about running the code, but also about traffic security, session management, and telemetry, and environment variables (DB credentials, Google/Stripe/SMTP secrets, port configuration) are next to outside-in configuration, injected into the runtime to customize the application to this particular production/cloud environment.

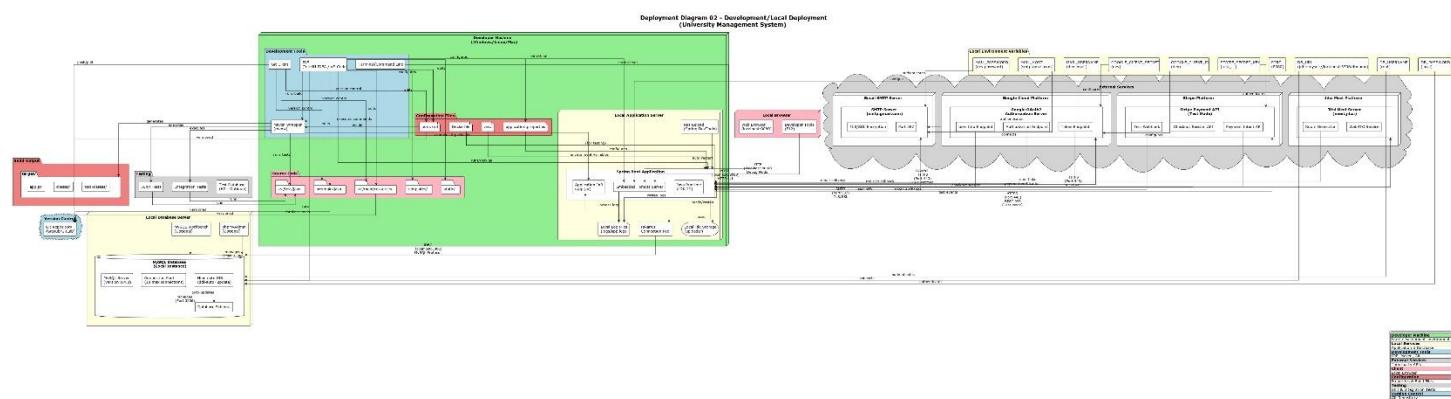


Diagram 6 University Management System – Deployment 02: Development / Local Environment

The “**University Management System – Deployment 02: Development / Local Environment**” diagram describes how the system runs on a developer’s own machine during everyday coding, debugging, and testing, showing both the runtime topology and the surrounding toolchain. At the center is the **Developer Machine** node (Windows/Linux/Mac), which hosts a **Local Application Server** running the **Spring Boot application** directly, with an **embedded Tomcat server**, a locally built **app.jar**, and a **Java runtime (JDK 21)**, mirroring the production stack but without containerization. This local application uses **HikariCP** as its JDBC connection pool, reading and writing user uploads into a **local file storage** directory and writing application/access logs to **local log files**, while **Spring DevTools** provides hot reload and automatic restarts to speed up the development feedback loop. Alongside the runtime, the diagram shows the main **development tools** package: an **IDE** (IntelliJ IDEA / VS Code) used to edit source code, manage configuration, and launch/debug the application; a **Maven wrapper** to compile and package the app and run unit/integration tests; a **Git client** for version control; and a **terminal/command line** for running build and run commands manually. The **source code** section captures the project layout (Java sources, resources, tests, templates, static assets), while a **configuration files** package includes application.properties, pom.xml, Dockerfile, and .env, all of which are edited and executed from the IDE and terminal. A separate **Local Database Server** node runs a local **MySQL** instance with its schema, pool settings, and Hibernate DDL auto-update enabled, and can be managed via tools like **MySQL Workbench** or **phpMyAdmin**, which connect to it on localhost:3306. The local application connects to this MySQL instance via **JDBC** through HikariCP, and a **local browser** node (with localhost:8080 and browser dev tools) invokes the embedded Tomcat over plain **HTTP**, giving the developer instant feedback while debugging front-

end behavior and HTTP flows. Despite being a local setup, the diagram also shows that the application still talks to real **external services**—Google OAuth2, Stripe (in **test mode**), Gmail SMTP, and Jitsi Meet—using development/test credentials and test webhooks, so that authentication, payments, email, and meeting link generation can be exercised in realistic conditions without real financial impact. A **testing** package models **JUnit tests**, **integration tests**, and an optional in-memory **H2 test database**, all driven by Maven; a **build output** package (target/ folder, compiled classes, test-classes, and the built app.jar) shows how build artifacts flow from the Maven wrapper into the local runtime; and a **version control** cloud represents the remote **Git repository** (GitHub/GitLab) that the Git client syncs with, tying local changes back to the team's codebase. Finally, a dedicated **Local Environment Variables** package documents key configuration values (DB URL, credentials, Google/Stripe keys in dev/test, SMTP settings, server port), which are injected into the local Spring Boot process via .env or IDE run configurations, ensuring the local environment behaves consistently but remains clearly separated and safer than the production deployment.

3.6.4 System Interfaces

Communication with various external platforms and infrastructure services Greenwich University Management System (GUMS) is developed as a center of integration, which not only unites internal academic and administrative processes but also communicates with various external platforms as well. This sub chapter explains the key system interfaces in an architectural sense, explaining the ways GUMS communicates information and organises behaviour with user browsers, institutional infrastructure and third-party providers. This is aimed at ensuring interface boundaries, protocols and responsibilities are made explicit hence ensuring extensions or replacement of individual services in future is handled with minimal effects on core system. Web interface (client-server interaction). GUMS presents itself mainly through the web interface that is made available to end users through normal browsers, over HTTPS. Any communication between the user and the system is done by way of HTML views rendered via Thymeleaf, form submissions and very few asynchronous requests. The server side controls the sessions and Spring Security performs the authentication, the authorisation and CSRF. In the eyes of the client, no extra software needs to be installed and any current browser with support of secure HTTP, cookies and simple JavaScript can be used. This design will provide wide access to the students, the lecturers, the staff and the parents and will centralise the security and logic to the server. Identity provider and authentication interfaces. GUMS works with external identify providers with flows based on the OAuth 2.0 protocol, with Google OAuth2 being the most standard example using it. Posting single sign-on, the system redirects the browser to the authorisation endpoint of the provider, obtains an authorisation code or a token when the user successfully logs in and replaces it with user identity information at the token/userinfo endpoint. The backend identifies both internal user accounts and roles based on the identity returned (e.g. email address or unique subject identifier), and then a normal Spring Security session is created. Sensitive credentials do not go through GUMS, and only the minimum identity attributes needed to perform authorisation are kept, all communication with the identity provider goes through TLS. Payment gateway interface. GUMS will interface with Stripe payment platform to manage tuition related transactions or any other type of payments. By initiating a payment, the system will either generate a Stripe Checkout session or a payment intent by calling the Stripe REST API and redirect the user to the hosted payment

page. Stripe processes the card data and, not a single card data is stored or processed by GUMS. When finished, Stripe informs the system about success, failure or cancellation with the help of secure callback URLs (webhooks). GUMS then updates the respective financial records, transaction history and student account status depending on the transaction identifiers and statuses experienced by Stripe. Email interface and notification interface. GUMS maintains an email interface which utilizes SMTP to transmit transactional and informational messages to various groups of stakeholders. The system sends the emails to an authorized SMTP over TLS that is connected to a configured mail server and builds emails in situations like accounts notifications, timetable updates, assessment notifications, payment confirmations and system alerts. The email service layer hides the specifics of the underlying mail infrastructure in such a way that, in the event that a replacement of the SMTP server or provider is required, higher level modules can be changed with minimal alterations to the email service layer. This interface is important in enabling crucial activities in the system to be transferred to users in a timely and dependable manner. Internet-based meeting and cooperation platform. GUMS is built on Jitsi Meet to provide real-time virtual classes to facilitate online or blended learning activities. Upon a lecturer making an online session, or updating one, the system creates and stores a Jitsi meeting room URL and links it to the related entry in the timetable and class. These links in GUMS allow students and lecturers to access meetings and the audio/video data is transmitted directly to the Jitsi service, with GUMS controlling the metadata only (meeting identifiers, URLs, meeting time and accessibility). This isolation guarantees that the core system is lightweight and at the same time can interact synchronously. All these interfaces make GUMS capable of functioning as a secure, extensible and interoperable platform. The web client interface offers a single point of access to users and the external integrations of identity, payments, email and meetings enable the system to utilise specialised services without redundancy of their services. Protocols, roles and data packet flow definition at both ends of every interface boundary are also a foundation of future extensions, including the incorporation of further identity providers, messaging or institutional services.

3.7 Database Design

3.7.1 Entity-Relationship Diagram

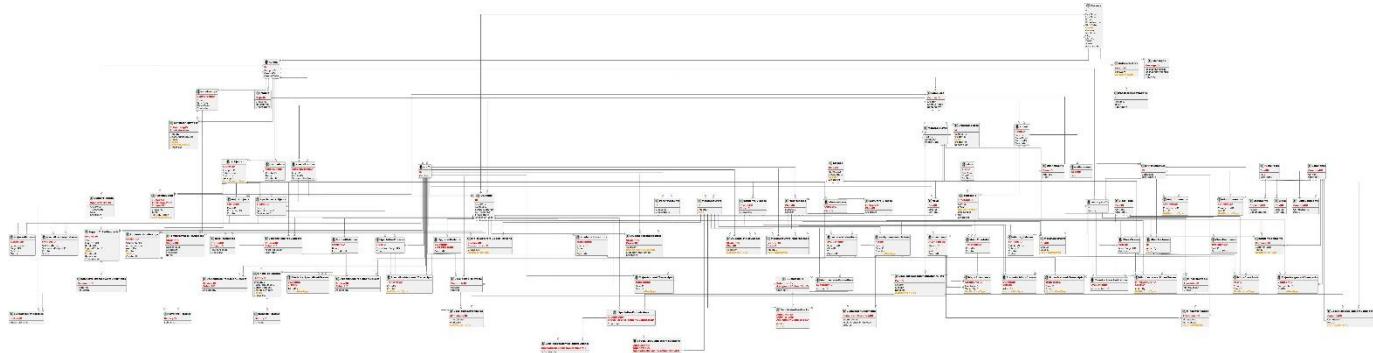


Diagram 7 Logical Entity-Relationship Diagram (Logical Data Model) in Crow's Foot notation



Diagram 8 Entity-Relationship Diagram Chen form (conceptual)

Two ERD represents a very large and detailed university information system where almost everything revolves around people, roles, academic structures, learning activities, finances, and communication. At the core, the Persons entity stores generic personal information (name, contact, birth date, gender, full address), and from this base entity multiple role-specific entities branch out: Admins, MajorEmployees, MinorEmployees, Students, ParentAccounts, plus authentication-related data in Authenticators and direct person-to-person Messages. Each of these role tables has its own attributes and responsibilities. Admins represent high-level administrators who create and manage institutional structures: campuses (Campuses), academic offerings (Majors, Specializations, Curriculums, Subjects), financial rules (TuitionByYear, Scholarships, ScholarshipByYear), infrastructure (Rooms), system templates (EmailTemplates), and service definitions (SupportTickets). Campuses represent separate physical or organizational branches; each campus holds its own Students, MajorEmployees, MinorEmployees, Rooms, TuitionByYear records, and EmailTemplates, reflecting that each campus can operate semi-independently with different staff, students, tuition, and communication templates. Academic staff are split into MajorEmployees and MinorEmployees, which then fan out into more specific roles: MajorEmployees link to Staffs (primary academic/administrative staff), MajorLecturers, and several “major or specialized” operational tables such as MajorAttendance, MajorComments, MajorClassPosts, MajorAssignmentComments, SpecializedAssignmentComments, and SpecializedAssignmentSubmitSlots. Similarly, MinorEmployees are associated with DeputyStaffs, MinorLecturers, and minor-related operations like MinorAttendance, MinorComments, and MinorClassPosts. Staffs act as the main academic coordinators for the major and specialized side: they manage MajorClasses and SpecializedClasses, configure MajorTimetable and SpecializedTimetable, assign students to classes (Students_MajorClasses, Students_SpecializedClasses), assign lecturers (MajorLecturers_MajorClasses), manage student

and parent records (Students, ParentAccounts, Student_ParentAccounts), control academic structures (MajorSubjects, SpecializedSubjects, MajorSyllabuses, SpecializationSyllabuses), handle special obligations like StudentRequiredMajorSubjects, maintain grade-related entities (MajorAcademicTranscripts, SpecializedAcademicTranscripts), handle scholarship assignment (Students_Scholarships), and process SupportTicketRequests. DeputyStaffs play a similar but narrower role on the minor side: they manage MinorClasses, MinorLecturers, MinorTimetable, MinorAcademicTranscripts, Students_MinorClasses, MinorLecturers_MinorClasses, StudentRequiredMinorSubjects, and MinorSyllabuses. MajorLecturers and MinorLecturers are teaching roles; they are linked to classes via the generic Lecturers_Classes join table, which is then refined by MajorLecturers_MajorClasses and MinorLecturers_MinorClasses to capture who teaches which class and who added the assignment, with notification metadata. Major lecturers are also tied to AssignmentSubmitSlots, SubmissionFeedbacks, SpecializedSubmissionFeedbacks, MajorLecturerEvaluations, and MajorLecturers_Specializations, reflecting their role in evaluating students and contributing to specialized programs. MinorLecturers are similarly evaluated via MinorLecturerEvaluations. On the academic structure side, Majors define degree programs and are linked to MajorEmployees, Specializations, and MajorSubjects. Specializations refine majors and connect to SpecializedSubjects, Students (each student has a specialization), and MajorLecturers_Specializations (to indicate which lecturers cover which specialization). Curriculums model study programs over time and relate to Students, MajorSubjects, and SpecializedSubjects, so that each student is tied to a particular curriculum version. Subjects represent abstract courses with attributes like SubjectName, Semester, approval status (isAccepted), and RequirementType, and are specialized into MajorSubjects, SpecializedSubjects, and MinorSubjects depending on their role in the program. These subjects connect to tuition (TuitionByYear with combinations of subject, admission year, and campus), payments (PaymentHistories), student requirements (StudentRequiredSubjects, and through it to StudentRequiredMajorSubjects and StudentRequiredMinorSubjects), and retakes (RetakeSubjects, TemporaryRetakeSubjects). Teaching is organized into Classes (generic class shells) that are specialized into MajorClasses, MinorClasses, and SpecializedClasses, each connecting to appropriate timetables (MajorTimetable, MinorTimetable, SpecializedTimetable), student-class mappings (Students_MajorClasses, Students_MinorClasses, Students_SpecializedClasses), lecturer-class mappings, class posts (MajorClassPosts, MinorClassPosts, SpecializedAssignmentSubmitSlots), and grade records (MajorAcademicTranscripts, MinorAcademicTranscripts, SpecializedAcademicTranscripts). Physical and virtual teaching spaces are modeled by Rooms (linked to Campuses and created by Admins), which are further specialized into OfflineRooms (address, floor) and OnlineRooms (link). Scheduling is handled by Slots (time ranges) and Timetable (room, slot, day of week, week, year), then further partitioned into MajorTimetable, MinorTimetable, and SpecializedTimetable to attach particular timetables to specific types of classes, as well as to drive attendance. Attendance captures a student's attendance record and is specialized into MajorAttendance, MinorAttendance, and SpecializedAttendance, which add timetable references, marker IDs, and notification details. Assessment and academic records are handled via AcademicTranscripts as a generic transcript record (with number, score, grade, timestamps), then specialized into MajorAcademicTranscripts, MinorAcademicTranscripts, and SpecializedAcademicTranscripts that attach these records to specific classes and creators. Assignments and grading workflows are represented by AssignmentSubmitSlots and SpecializedAssignmentSubmitSlots (assignment

postings tied to class posts and specialized classes respectively, with deadlines and notifications), Submissions and SpecializedSubmissions (student submissions), SubmissionDocuments and SpecializedSubmissionDocuments (files attached to submissions), and SubmissionFeedbacks and SpecializedSubmissionFeedbacks (feedback and grades from announcers/lecturers). Student-facing requirements and exceptions are modeled by StudentRequiredSubjects (a generic mapping of student to required subject with reason), which branches into StudentRequiredMajorSubjects and StudentRequiredMinorSubjects (recording who assigned the requirement), plus RetakeSubjects and TemporaryRetakeSubjects for students needing to repeat courses, including additional flags for processing and notes. On the communication/content side, there is a clear separation between public and class-level content. PublicPosts represent public-facing posts and are specialized into Blogs, News, PublicComments associations, and Documents attached to public posts. ClassPosts represent posts within classes and are related to MajorClassPosts, MinorClassPosts, AssignmentSubmitSlots, StudentComments, ClassDocuments, and SpecializedAssignmentSubmitSlots. Comments are normalized via a generic Comments table (storing content and timestamps) with multiple specialized linking tables for context: PublicComments, MajorComments, MinorComments, StudentComments, MajorAssignmentComments, and SpecializedAssignmentComments, each connecting comments to various post types, commenters, and notification modes. The financial subsystem centers around AccountBalances (per-student wallet or account), FinancialHistories (history of balance changes with amounts and statuses), and specializations of financial events: DepositHistories (pure deposits), PaymentHistories (payments mapped to specific subjects), and SupportTicketHistories (charges/transactions related to support services). Scholarships are modeled by Scholarships (base definition of scholarship type, creator, award date) and ScholarshipByYear (scholarship rules per admission year with amount, discount, status and contracts), while the assignment of scholarships to students is captured in Students_Scholarships (linking student, scholarship, admission year, award date, creator, and status). Student-parent relationships are captured via Student_ParentAccounts (linking Students to ParentAccounts, with relationship type, emergency contact phone, who added it, and when). Support and administrative services are represented by SupportTickets (ticket types with cost) and SupportTicketRequests (individual student requests for these services, including requester, handler, status workflow, and timestamps), SupportTicketRequestsDocuments for associated files, and ties back into financial and history tracking via SupportTicketHistories. System-level communication and security are handled by Authenticators (per-person login info with password and account status) and PasswordResetTokens (reset tokens linked to persons with expiry), as well as EmailTemplates (per campus and creator, for different email types) and direct Messages between persons (sender, recipient, text, datetime). At the evaluation layer, LecturerEvaluations store students' evaluations of classes and lecturers, then are specialized into MajorLecturerEvaluations and MinorLecturerEvaluations to distinguish which lecturer or minor lecturer is being reviewed. Finally, UpgradeStudents tracks student upgrade status over time, and StudentRequiredSubjects plus its specialized child tables, Students_Classes and its specialized mappings, along with many notification-type fields, all indicate that the design systematically uses base tables plus specialized child tables to separate generic data (core attributes and relationships) from context-specific semantics (major/minor/specialized, notification rules, who created or assigned what), resulting in a highly normalized and role-driven ERD that supports detailed academic, administrative, financial, and communication workflows across multiple campuses and program types.

3.7.2 Data Dictionary

The data dictionary describes in a structured way all data elements used in the logical database design derived from the ERD. For each table, it specifies the attribute name, data type (and length/precision), whether it participates in the primary key (PK) or foreign key (FK), and a concise semantic description. This section ensures that developers, DBAs and analysts share the same understanding of what each column means and how it relates to other entities, thereby minimizing ambiguity when implementing, maintaining or querying the database. Below is a representative excerpt of the data dictionary for the core entities of the system (persons and roles, academic structure, scheduling, learning results, finance, communication, and support). The same pattern is applied consistently to all remaining tables in the schema.

Table 14 Data Dictionary

Table	Attribute	Data type	Length / Precision	PK	FK	Nullable	Description
Persons	ID	varchar	255	Yes	No	No	Unique identifier of a person in the system.
Persons	FirstName	varchar	100	No	No	Yes	Person's first name.
Persons	LastName	varchar	100	No	No	Yes	Person's last name.
Persons	Email	varchar	255	No	No	Yes	Email address of the person.
Persons	PhoneNumber	varchar	20	No	No	Yes	Phone number of the person.
Persons	BirthDate	date	–	No	No	Yes	Date of birth of the person.
Persons	Gender	enum	–	No	No	Yes	Gender classification of the person.
Persons	Country	varchar	100	No	No	Yes	Country of residence.
Persons	Province	varchar	100	No	No	Yes	Province/state of residence.
Persons	City	varchar	100	No	No	Yes	City of residence.
Persons	District	varchar	100	No	No	Yes	District of the address.
Persons	Ward	varchar	100	No	No	Yes	Ward or sub-district of the address.

Persons	Street	varchar	255	No	No	Yes	Street and house number information.
Persons	PostalCode	varchar	20	No	No	Yes	Postal / ZIP code.
Persons	Avatar	longblob	–	No	No	Yes	Binary avatar image of the person.
Admins	ID	varchar	255	Yes	Yes	No	FK to Persons.ID; person who has administrator role.
Admins	FaceData	longtext	–	No	No	Yes	Biometric or face recognition data for the admin.
Admins	CreatedDate	datetime	-6	No	No	Yes	Timestamp when the admin record was created.
Admins	CampusID	varchar	255	No	Yes	Yes	FK to Campuses.CampusID; campus this admin is linked to.
Admins	CreatorID	varchar	255	No	Yes	Yes	FK to Persons.ID/Admins.ID; user who created this admin.
MajorEmployees	ID	varchar	255	Yes	Yes	No	FK to Persons.ID; person who is a major employee.
MajorEmployees	MajorID	varchar	255	No	Yes	Yes	FK to Majors.MajorID; major this employee belongs to.
MajorEmployees	CampusID	varchar	255	No	Yes	Yes	FK to Campuses.CampusID; campus where the employee works.
MajorEmployees	createdDate	date	–	No	No	No	Date when the major employee record was created.
MinorEmployees	ID	varchar	255	Yes	Yes	No	FK to Persons.ID; person who is a minor employee.

MinorEmployees	CampusID	varchar	255	No	Yes	Yes	FK to Campuses.CampusID; campus where the minor employee works.
MinorEmployees	createdDate	date	—	No	No	No	Date when the minor employee record was created.
Staffs	ID	varchar	255	Yes	Yes	No	FK to MajorEmployees.ID; major employee who is a staff member.
Staffs	Creator	varchar	255	No	Yes	Yes	FK to Admins.ID; admin who created this staff record.
DeputyStaffs	ID	varchar	255	Yes	Yes	No	FK to MinorEmployees.ID; minor employee who is a deputy staff member.
DeputyStaffs	Creator	varchar	255	No	Yes	Yes	FK to Admins.ID; admin who created this deputy staff record.
MajorLecturers	ID	varchar	255	Yes	Yes	No	FK to MajorEmployees.ID; employee who is a major lecturer.
MajorLecturers	AddedBy	varchar	255	No	Yes	Yes	FK to Staffs.ID/Admins.ID; user who added this major lecturer.
MajorLecturers	Type	enum	—	No	No	Yes	Type of major lecturer (e.g. full-time, visiting).
MinorLecturers	ID	varchar	255	Yes	Yes	No	FK to MinorEmployees.ID; employee who

								is a minor lecturer.
MinorLecturers	AddedBy	varchar	255	No	Yes	Yes		FK to DeputyStaffs.ID/Admins.ID; user who added this minor lecturer.
MinorLecturers	Type	enum	—	No	No	Yes		Type of minor lecturer.
Students	ID	varchar	255	Yes	Yes	No		FK to Persons.ID; person who is a student.
Students	CreatorID	varchar	255	No	Yes	Yes		FK to Staffs.ID; staff who created the student record.
Students	CampusID	varchar	255	No	Yes	Yes		FK to Campuses.CampusID; campus where the student studies.
Students	SpecializationID	varchar	255	No	Yes	Yes		FK to Specializations.SpecializationID; student specialization.
Students	CurriculumID	varchar	255	No	Yes	Yes		FK to Curriculums.CurriculumID; curriculum assigned to the student.
Students	createdDate	date	—	No	No	No		Date when the student record was created.
Students	Admission_Year	int	—	No	No	No		Year when the student was admitted.
ParentAccounts	ID	varchar	255	Yes	Yes	No		FK to Persons.ID; person who is a parent/guardian account.
ParentAccounts	CreatorID	varchar	255	No	Yes	Yes		FK to Staffs.ID; staff who created the parent account.

ParentAccounts	CreatedDate	date	–	No	No	No	Date when the parent account was created.
Authenticators	PersonID	varchar	255	Yes	Yes	No	FK to Persons.ID; person whose credentials are stored.
Authenticators	Password	varchar	255	No	No	Yes	Hashed password used for authentication.
Authenticators	AccountStatus	enum	–	No	No	Yes	Status of the account (e.g. active, locked).
Campuses	CampusID	varchar	255	Yes	No	No	Unique identifier of a campus.
Campuses	Creator	varchar	255	No	Yes	Yes	FK to Admins.ID; admin who created the campus.
Campuses	CampusName	varchar	255	No	No	Yes	Name of the campus.
Campuses	OpeningDay	date	–	No	No	Yes	Official opening date of the campus.
Campuses	Description	varchar	255	No	No	Yes	Description of the campus.
Campuses	Avatar	longblob	–	No	No	Yes	Image/logo of the campus.
Majors	MajorID	varchar	255	Yes	No	No	Unique identifier of a major.
Majors	CreatorID	varchar	255	No	Yes	Yes	FK to Admins.ID; admin who created the major.
Majors	MajorName	varchar	255	No	No	No	Name of the major.
Majors	CreatedDate	date	–	No	No	No	Creation date of the major.
Majors	Avatar	longblob	–	No	No	Yes	Image/logo of the major.
Specializations	SpecializationID	varchar	255	Yes	No	No	Unique identifier of a specialization.
Specializations	MajorID	varchar	255	No	Yes	No	FK to Majors.MajorID; major that owns

							this specialization.
Specializations	CreatorID	varchar	255	No	Yes	No	FK to Admins.ID; admin who created the specialization.
Specializations	Specialization Name	varchar	255	No	No	No	Name of the specialization.
Specializations	CreatedAt	datetime	-6	No	No	Yes	Timestamp when the specialization was created.
Curriculums	CurriculumID	varchar	255	Yes	No	No	Unique identifier of a curriculum.
Curriculums	CreatorID	varchar	255	No	Yes	No	FK to Admins.ID; admin who created the curriculum.
Curriculums	Name	varchar	255	No	No	No	Name of the curriculum.
Curriculums	Description	varchar	1000	No	No	Yes	Description of the curriculum.
Curriculums	CreatedAt	datetime	-6	No	No	No	Timestamp when the curriculum was created.
Classes	ClassID	varchar	255	Yes	No	No	Unique identifier of a class.
Classes	NameClass	varchar	255	No	No	No	Name/label of the class.
Classes	SlotQuantity	int	—	No	No	Yes	Number of available slots (capacity) in the class.
Classes	Session	enum	—	No	No	Yes	Session type (e.g. morning, evening).
Classes	CreatedAt	datetime	-6	No	No	Yes	Timestamp when the class was created.
MajorClasses	ClassID	varchar	255	Yes	Yes	No	FK to Classes.ClassID; class that is a major class.
MajorClasses	SubjectID	varchar	255	No	Yes	Yes	FK to Subjects.SubjectID; major subject

								assigned to this class.
MajorClasses	Creator	varchar	255	No	Yes	Yes		FK to Staffs.ID; staff who created the major class record.
MinorClasses	ClassID	varchar	255	Yes	Yes	No		FK to Classes.ClassID; class that is a minor class.
MinorClasses	MinorSubjectID	varchar	255	No	Yes	Yes		FK to MinorSubjects.SubjectID; minor subject assigned to this class.
MinorClasses	Creator	varchar	255	No	Yes	Yes		FK to DeputyStaffs.ID; deputy staff who created the minor class.
SpecializedClasses	ClassID	varchar	255	Yes	Yes	No		FK to Classes.ClassID; class that is a specialized class.
SpecializedClasses	SpecializedSubjectID	varchar	255	No	Yes	Yes		FK to SpecializedSubjects.SubjectID; specialized subject for this class.
SpecializedClasses	Creator	varchar	255	No	Yes	Yes		FK to Staffs.ID; staff who created the specialized class.
Subjects	SubjectID	varchar	255	Yes	No	No		Unique identifier of a subject.
Subjects	AcceptorID	varchar	255	No	Yes	Yes		FK to Admins.ID/Staffs.ID; user who approved the subject.
Subjects	SubjectName	varchar	255	No	No	No		Name/title of the subject.
Subjects	Semester	int	—	No	No	Yes		Typical semester when the subject is offered.

Subjects	isAccepted	bit	-1	No	No	No	Flag indicating whether the subject is accepted.
Subjects	RequirementType	enum	—	No	No	Yes	Requirement category (e.g. compulsory, elective).
MajorSubjects	SubjectID	varchar	255	Yes	Yes	No	FK to Subjects.SubjectID; subject that is a major subject.
MajorSubjects	MajorID	varchar	255	No	Yes	Yes	FK to Majors.MajorID; major related to the subject.
MajorSubjects	CurriculumID	varchar	255	No	Yes	Yes	FK to Curriculums.CurriculumID; curriculum context of the subject.
MajorSubjects	Creator	varchar	255	No	Yes	No	FK to Staffs.ID; staff who created this major subject mapping.
SpecializedSubjects	SubjectID	varchar	255	Yes	Yes	No	FK to Subjects.SubjectID; subject that is a specialized subject.
SpecializedSubjects	Creator	varchar	255	No	Yes	No	FK to Staffs.ID; staff who created the specialized subject mapping.
SpecializedSubjects	SpecializationID	varchar	255	No	Yes	No	FK to Specializations.SpecializationID; specialization this subject belongs to.
SpecializedSubjects	CurriculumID	varchar	255	No	Yes	Yes	FK to Curriculums.CurriculumID; curriculum for this

								specialized subject.
MinorSubjects	SubjectID	varchar	255	Yes	Yes	No		FK to Subjects.SubjectID; subject that is a minor subject.
MinorSubjects	Creator	varchar	255	No	Yes	No		FK to Staffs.ID/DeputyStaffs.ID; user who created the minor subject.
TuitionByYear	SubjectID	varchar	255	Yes	Yes	No		FK to Subjects.SubjectID; subject whose tuition is defined.
TuitionByYear	Admission_Year	int	–	Yes	No	No		Admission year that this tuition applies to.
TuitionByYear	CampusID	varchar	255	Yes	Yes	No		FK to Campuses.CampusID; campus where tuition is applied.
TuitionByYear	CreatorID	varchar	255	No	Yes	Yes		FK to Admins.ID; admin who created the tuition configuration.
TuitionByYear	Tuition	double	–	No	No	Yes		Tuition fee for the subject.
TuitionByYear	ReStudyTuition	double	–	No	No	Yes		Tuition fee for retaking the subject.
TuitionByYear	ContractStatus	enum	–	No	No	Yes		Contract status associated with the tuition.
Rooms	RoomID	varchar	255	Yes	No	No		Unique identifier of a room (physical or virtual).
Rooms	Creator	varchar	255	No	Yes	Yes		FK to Admins.ID; admin who created the room.
Rooms	RoomName	varchar	255	No	No	Yes		Name or code of the room.
Rooms	CampusID	varchar	255	No	Yes	Yes		FK to Campuses.Camp

							usID; campus where the room is located.
Rooms	CreatedAt	datetime	-6	No	No	No	Timestamp when the room record was created.
Rooms	Avatar	longblob	–	No	No	Yes	Image or plan of the room.
OfflineRooms	RoomID	varchar	255	Yes	Yes	No	FK to Rooms.RoomID; room that is a physical offline room.
OfflineRooms	Address	varchar	500	No	No	Yes	Physical address of the room.
OfflineRooms	Floor	int	–	No	No	Yes	Floor number of the room.
OnlineRooms	RoomID	varchar	255	Yes	Yes	No	FK to Rooms.RoomID; room that is an online room.
OnlineRooms	Link	varchar	500	No	No	Yes	URL link to access the online room.
Slots	SlotID	varchar	255	Yes	No	No	Unique identifier of a time slot.
Slots	SlotName	varchar	100	No	No	Yes	Name or label of the time slot.
Slots	StartTime	time	-6	No	No	No	Start time of the time slot.
Slots	EndTime	time	-6	No	No	No	End time of the time slot.
Timetable	TimetableID	varchar	255	Yes	No	No	Unique identifier of a timetable entry.
Timetable	RoomID	varchar	255	No	Yes	No	FK to Rooms.RoomID; room used for this timetable entry.
Timetable	SlotID	varchar	255	No	Yes	No	FK to Slots.SlotID; slot used for this timetable entry.
Timetable	DayOfTheWeek	enum	–	No	No	No	Day of the week for the timetable entry.

Timetable	WeekOfYear	int	–	No	No	No	Week number within the year.
Timetable	Year	int	–	No	No	No	Calendar year for the timetable entry.
Timetable	CreatedAt	datetime	-6	No	No	Yes	Timestamp when the timetable entry was created.
MajorTimetable	TimetableID	varchar	255	Yes	Yes	No	FK to Timetable.TimetableID; timetable for a major class.
MajorTimetable	ClassID	varchar	255	No	Yes	No	FK to MajorClasses.ClassID; major class scheduled.
MajorTimetable	Creator	varchar	255	No	Yes	No	FK to Staffs.ID; staff who created the major timetable entry.
MinorTimetable	TimetableID	varchar	255	Yes	Yes	No	FK to Timetable.TimetableID; timetable for a minor class.
MinorTimetable	MinorClassID	varchar	255	No	Yes	No	FK to MinorClasses.ClassID; minor class scheduled.
MinorTimetable	Creator	varchar	255	No	Yes	No	FK to DeputyStaffs.ID; deputy staff who created the minor timetable.
SpecializedTime table	TimetableID	varchar	255	Yes	Yes	No	FK to Timetable.TimetableID; timetable for a specialized class.
SpecializedTime table	ClassID	varchar	255	No	Yes	No	FK to SpecializedClasses.ClassID; specialized class scheduled.
SpecializedTime table	Creator	varchar	255	No	Yes	No	FK to Staffs.ID; staff who created

							the specialized timetable.
PublicPosts	PostID	varchar	255	Yes	No	No	Unique identifier for a public post.
PublicPosts	Title	varchar	255	No	No	Yes	Title of the public post.
PublicPosts	CreatedAt	datetime	-6	No	No	No	Timestamp when the public post was created.
ClassPosts	PostID	varchar	255	Yes	No	No	Unique identifier for a class post.
ClassPosts	Content	varchar	1000	No	No	Yes	Text content of the class post.
ClassPosts	CreatedAt	datetime	-6	No	No	No	Timestamp when the class post was created.
Blogs	PostID	varchar	255	Yes	Yes	No	FK to PublicPosts.PostID; public post that is a blog.
Blogs	Creator	varchar	255	No	Yes	No	FK to Staffs.ID/Admins.ID; creator of the blog.
Blogs	Content	varchar	1000	No	No	Yes	Content of the blog post.
News	PostID	varchar	255	Yes	Yes	No	FK to PublicPosts.PostID; public post that is a news item.
News	Creator	varchar	255	No	Yes	No	FK to Staffs.ID; staff who created the news.
News	Content	varchar	1000	No	No	Yes	Content of the news post.
MajorClassPosts	PostID	varchar	255	Yes	Yes	No	FK to ClassPosts.PostID; class post for a major class.
MajorClassPosts	Creator	varchar	255	No	Yes	No	FK to MajorEmployees.ID; major employee who created the post.
MajorClassPosts	ClassID	varchar	255	No	Yes	No	FK to MajorClasses.Cla

							ssID; major class the post belongs to.
MajorClassPosts	NotificationType	enum	–	No	No	No	Notification type of this major class post.
MinorClassPosts	PostID	varchar	255	Yes	Yes	No	FK to ClassPosts.PostID; class post for a minor class.
MinorClassPosts	Creator	varchar	255	No	Yes	No	FK to MinorEmployees.ID; minor employee who created the post.
MinorClassPosts	ClassID	varchar	255	No	Yes	No	FK to MinorClasses.ClassID; minor class the post belongs to.
MinorClassPosts	NotificationType	enum	–	No	No	No	Notification type of this minor class post.
AssignmentSubmitSlots	PostID	varchar	255	Yes	Yes	No	FK to ClassPosts.PostID; class post that is an assignment submit slot.
AssignmentSubmitSlots	Creator	varchar	255	No	Yes	No	FK to MajorLecturers.ID ; lecturer who created the slot.
AssignmentSubmitSlots	ClassID	varchar	255	No	Yes	No	FK to MajorClasses.ClassID; major class of the assignment.
AssignmentSubmitSlots	Deadline	datetime	-6	No	No	Yes	Deadline for submitting the assignment.
AssignmentSubmitSlots	NotificationType	enum	–	No	No	No	Notification type for the assignment slot.
Comments	CommentID	varchar	255	Yes	No	No	Unique identifier for a comment.

Comments	Content	varchar	1000	No	No	Yes	Text content of the comment.
Comments	CreatedAt	datetime	-6	No	No	No	Timestamp when the comment was created.
PublicComments	CommentID	varchar	255	Yes	Yes	No	FK to Comments.CommentID; comment that is a public comment.
PublicComments	CommenterID	varchar	255	No	Yes	No	FK to Persons.ID; person who made the public comment.
PublicComments	PostID	varchar	255	No	Yes	No	FK to PublicPosts.PostID; public post commented on.
MajorComments	CommentID	varchar	255	Yes	Yes	No	FK to Comments.CommentID; comment related to a major class/post.
MajorComments	CommenterID	varchar	255	No	Yes	No	FK to MajorEmployees.ID; major employee who commented.
MajorComments	PostID	varchar	255	No	Yes	No	FK to ClassPosts.PostID; class post commented on.
MajorComments	NotificationType	enum	—	No	No	No	Notification type for this major comment.
MinorComments	CommentID	varchar	255	Yes	Yes	No	FK to Comments.CommentID; comment related to a minor class/post.
MinorComments	CommenterID	varchar	255	No	Yes	No	FK to MinorEmployees.ID; minor employee who commented.

MinorComments	PostID	varchar	255	No	Yes	No	FK to ClassPosts.PostID; class post commented on.
MinorComments	NotificationType	enum	—	No	No	No	Notification type for this minor comment.
StudentComments	CommentID	varchar	255	Yes	Yes	No	FK to Comments.CommentID; comment made by a student.
StudentComments	CommenterID	varchar	255	No	Yes	No	FK to Students.ID; student who commented.
StudentComments	PostID	varchar	255	No	Yes	No	FK to ClassPosts.PostID; class post commented on.
StudentComments	NotificationType	enum	—	No	No	No	Notification type for this student comment.
ClassDocuments	DocumentID	varchar	255	Yes	No	No	Unique identifier for a class document.
ClassDocuments	PostID	varchar	255	No	Yes	No	FK to ClassPosts.PostID; class post this document is attached to.
ClassDocuments	DocumentTitle	varchar	255	No	No	Yes	Title of the class document.
ClassDocuments	FilePath	varchar	255	No	No	Yes	Path to the stored class document file.
ClassDocuments	FileData	longblob	—	No	No	Yes	Binary data of the class document.
Documents	DocumentID	varchar	255	Yes	No	No	Unique identifier for a generic document.
Documents	PostID	varchar	255	No	Yes	Yes	FK to PublicPosts.PostID; public post this document is linked to.

Documents	DocumentTitle	varchar	255	No	No	Yes	Title of the document.
Documents	FilePath	varchar	500	No	No	Yes	File path of the stored document.
Documents	FileData	longblob	–	No	No	Yes	Binary file data of the document.
AcademicTranscripts	TranscriptID	varchar	255	Yes	No	No	Unique identifier for an academic transcript.
AcademicTranscripts	NumberID	varchar	255	No	Yes	No	Logical link to specific enrollment/numbered item.
AcademicTranscripts	Score	double	–	No	No	Yes	Final total score.
AcademicTranscripts	ScoreComponent1	double	–	No	No	Yes	Score of component 1 (e.g. midterm).
AcademicTranscripts	ScoreComponent2	double	–	No	No	Yes	Score of component 2.
AcademicTranscripts	ScoreComponent3	double	–	No	No	Yes	Score of component 3.
AcademicTranscripts	Grade	enum	–	No	No	Yes	Grade label derived from score.
AcademicTranscripts	CreatedAt	datetime	-6	No	No	No	Timestamp when the transcript was created.
MajorAcademicTranscripts	TranscriptID	varchar	255	Yes	Yes	No	FK to AcademicTranscripts.TranscriptID; transcript of a major class.
MajorAcademicTranscripts	ClassID	varchar	255	No	Yes	Yes	FK to MajorClasses.ClassID; related major class.
MajorAcademicTranscripts	Creator	varchar	255	No	Yes	No	FK to Staffs.ID; staff who created the major transcript.
MajorAcademicTranscripts	NotificationType	enum	–	No	No	No	Notification type for this major transcript.
MinorAcademicTranscripts	TranscriptID	varchar	255	Yes	Yes	No	FK to AcademicTranscri

								pts.TranscriptID; transcript of a minor class.
MinorAcademicT ranscripts	ClassID	varchar	255	No	Yes	Yes		FK to MinorClasses.Cla ssID; related minor class.
MinorAcademicT ranscripts	Creator	varchar	255	No	Yes	No		FK to DeputyStaffs.ID; deputy staff who created the minor transcript.
MinorAcademicT ranscripts	NotificationTy pe	enum	—	No	No	No		Notification type for this minor transcript.
SpecializedAcad emicTranscripts	TranscriptID	varchar	255	Yes	Yes	No		FK to AcademicTranscri pts.TranscriptID; transcript of a specialized class.
SpecializedAcad emicTranscripts	ClassID	varchar	255	No	Yes	Yes		FK to SpecializedClass es.ClassID; related specialized class.
SpecializedAcad emicTranscripts	Creator	varchar	255	No	Yes	No		FK to Staffs.ID; staff who created the specialized transcript.
SpecializedAcad emicTranscripts	NotificationTy pe	enum	—	No	No	No		Notification type for this specialized transcript.
AccountBalanc es	StudentID	varchar	255	Yes	Yes	No		FK to Students.ID; student whose balance is tracked.
AccountBalanc es	Balance	double	—	No	No	No		Current monetary balance of the student account.
AccountBalanc es	LastUpdated	datetime	-6	No	No	No		Timestamp when the balance was last updated.

FinancialHistories	HistoryID	varchar	255	Yes	No	No	Unique identifier for a financial history record.
FinancialHistories	StudentID	varchar	255	No	Yes	No	FK to Students.ID; student concerned by this record.
FinancialHistories	AccountBalanceID	varchar	255	No	Yes	No	FK to AccountBalances.StudentID; related account balance.
FinancialHistories	CurrentAmount	decimal	15,2	No	No	No	Amount after the transaction.
FinancialHistories	ChangedAmount	decimal	15,2	No	No	No	Amount of change applied by the transaction.
FinancialHistories	CreatedAt	datetime	-6	No	No	No	Timestamp when the financial history was created.
FinancialHistories	Status	enum	—	No	No	No	Status of the financial transaction.
FinancialHistories	version	bigint	—	No	No	Yes	Version for concurrency/history control.
DepositHistories	HistoryID	varchar	255	Yes	Yes	No	FK to FinancialHistories.HistoryID; deposit transaction.
DepositHistories	Amount	double	—	No	No	No	Amount of money deposited.
PaymentHistories	HistoryID	varchar	255	Yes	Yes	No	FK to FinancialHistories.HistoryID; payment transaction.
PaymentHistories	SubjectID	varchar	255	No	Yes	No	FK to Subjects.SubjectID; subject the payment relates to.

SupportTicketHistories	HistoryID	varchar	255	Yes	Yes	No	FK to FinancialHistories.HistoryID; ticket-related transaction.
SupportTicketHistories	SupportTicketID	varchar	255	No	Yes	No	FK to SupportTickets.SupportTicketID; support ticket referenced.
Attendance	AttendanceID	varchar	255	Yes	No	No	Unique identifier for an attendance record.
Attendance	StudentID	varchar	255	No	Yes	No	FK to Students.ID; student whose attendance is recorded.
Attendance	Status	enum	—	No	No	Yes	Attendance status (present, absent, etc.).
Attendance	Note	varchar	500	No	No	Yes	Extra note explaining the attendance record.
Attendance	CreatedAt	datetime	-6	No	No	No	Timestamp when the attendance was created.
MajorAttendance	AttendanceID	varchar	255	Yes	Yes	No	FK to Attendance.AttendanceID; attendance for a major class.
MajorAttendance	TimetableID	varchar	255	No	Yes	Yes	FK to MajorTimetable.TimetableID; timetable entry for this record.
MajorAttendance	MarkedByID	varchar	255	No	Yes	Yes	FK to MajorEmployees.ID; person who marked attendance.
MajorAttendance	NotificationType	enum	—	No	No	No	Notification type for the major attendance.

MinorAttendance	AttendanceID	varchar	255	Yes	Yes	No	FK to Attendance.AttendanceID; attendance for a minor class.
MinorAttendance	TimetableID	varchar	255	No	Yes	Yes	FK to MinorTimetable.TimetableID; timetable entry for this record.
MinorAttendance	MarkedByID	varchar	255	No	Yes	Yes	FK to MinorEmployees.ID; person who marked attendance.
MinorAttendance	NotificationType	enum	—	No	No	No	Notification type for the minor attendance.
SpecializedAttendance	AttendanceID	varchar	255	Yes	Yes	No	FK to Attendance.AttendanceID; attendance for a specialized class.
SpecializedAttendance	TimetableID	varchar	255	No	Yes	No	FK to SpecializedTimetable.TimetableID; timetable entry.
SpecializedAttendance	MarkedBy	varchar	255	No	Yes	No	FK to Staffs.ID; staff who marked the attendance.
SpecializedAttendance	NotificationType	enum	—	No	No	No	Notification type for the specialized attendance.
LecturerEvaluations	EvaluationID	varchar	255	Yes	No	No	Unique identifier of a lecturer evaluation.
LecturerEvaluations	ReviewerID	varchar	255	No	Yes	No	FK to Students.ID; student who reviewed the lecturer.
LecturerEvaluations	ClassID	varchar	255	No	Yes	No	FK to Classes.ClassID; class being evaluated.

LecturerEvaluations	Text	varchar	1000	No	No	Yes	Text feedback given by the reviewer.
LecturerEvaluations	CreatedAt	datetime	-6	No	No	No	Timestamp when the evaluation was created.
MajorLecturerEvaluations	EvaluationID	varchar	255	Yes	Yes	No	FK to LecturerEvaluations.EvaluationID; evaluation for a major lecturer.
MajorLecturerEvaluations	LecturerID	varchar	255	No	Yes	No	FK to MajorLecturers.ID ; lecturer evaluated.
MinorLecturerEvaluations	EvaluationID	varchar	255	Yes	Yes	No	FK to LecturerEvaluations.EvaluationID; evaluation for a minor lecturer.
MinorLecturerEvaluations	MinorLecturerID	varchar	255	No	Yes	No	FK to MinorLecturers.ID ; minor lecturer evaluated.
Lecturers_Classes	LecturerID	varchar	255	Yes	Yes	No	FK to Persons.ID (lecturer); lecturer assigned to a class.
Lecturers_Classes	ClassID	varchar	255	Yes	Yes	No	FK to Classes.ClassID; class taught by the lecturer.
Lecturers_Classes	CreatedAt	datetime	-6	No	No	No	Timestamp when the lecturer-class mapping was created.
MajorLecturers_MajorClasses	LecturerID	varchar	255	Yes	Yes	No	FK to MajorLecturers.ID ; major lecturer assigned.
MajorLecturers_MajorClasses	ClassID	varchar	255	Yes	Yes	No	FK to MajorClasses.ClassID; major class taught.

MajorLecturers_MajorClasses	AddedBy	varchar	255	No	Yes	Yes	FK to Staffs.ID; staff who created the mapping.
MajorLecturers_MajorClasses	NotificationType	enum	—	No	No	Yes	Notification type for the mapping.
MinorLecturers_MinorClasses	MinorLecturerID	varchar	255	Yes	Yes	No	FK to MinorLecturers.ID ; minor lecturer assigned.
MinorLecturers_MinorClasses	ClassID	varchar	255	Yes	Yes	No	FK to MinorClasses.ClassID; minor class taught.
MinorLecturers_MinorClasses	AddedBy	varchar	255	No	Yes	Yes	FK to DeputyStaffs.ID; deputy staff who created mapping.
MinorLecturers_MinorClasses	NotificationType	enum	—	No	No	Yes	Notification type for the mapping.
Students_Classes	StudentID	varchar	255	Yes	Yes	No	FK to Students.ID; student enrolled in a class.
Students_Classes	ClassID	varchar	255	Yes	Yes	No	FK to Classes.ClassID; class the student attends.
Students_Classes	CreatedAt	datetime	-6	No	No	No	Timestamp when the student-class link was created.
Students_MajorClasses	StudentID	varchar	255	Yes	Yes	No	FK to Students.ID; student in a major class.
Students_MajorClasses	ClassID	varchar	255	Yes	Yes	No	FK to MajorClasses.ClassID; major class.
Students_MajorClasses	AddedBy	varchar	255	No	Yes	Yes	FK to Staffs.ID; staff who added the student.
Students_MajorClasses	NotificationType	enum	—	No	No	No	Notification type for this major enrollment.
Students_MinorClasses	StudentID	varchar	255	Yes	Yes	No	FK to Students.ID;

								student in a minor class.
Students_MinorClasses	ClassID	varchar	255	Yes	Yes	No		FK to MinorClasses.ClassID; minor class.
Students_MinorClasses	AddedBy	varchar	255	No	Yes	Yes		FK to DeputyStaffs.ID; deputy staff who added the student.
Students_MinorClasses	NotificationType	enum	—	No	No	No		Notification type for this minor enrollment.
Students_SpecializedClasses	StudentID	varchar	255	Yes	Yes	No		FK to Students.ID; student in a specialized class.
Students_SpecializedClasses	ClassID	varchar	255	Yes	Yes	No		FK to SpecializedClasses.ClassID; specialized class.
Students_SpecializedClasses	AddedBy	varchar	255	No	Yes	Yes		FK to Staffs.ID; staff who added the student.
Students_SpecializedClasses	NotificationType	enum	—	No	No	No		Notification type for this specialized enrollment.
UpgradeStudents	StudentID	varchar	255	Yes	Yes	No		FK to Students.ID; student whose upgrade is tracked.
UpgradeStudents	UpgradeStatus	enum	—	Yes	No	No		Upgrade status (e.g. promoted, repeated).
UpgradeStudents	CreatedAt	datetime	-6	No	No	No		Timestamp when this upgrade status was recorded.
Student_ParentAccounts	StudentID	varchar	255	Yes	Yes	No		FK to Students.ID; student in student-parent link.

Student_Parent Accounts	ParentID	varchar	255	Yes	Yes	No	FK to ParentAccounts.ID; parent in the relationship.
Student_Parent Accounts	AddedBy	varchar	255	No	Yes	Yes	FK to Staffs.ID; staff who created the relationship.
Student_Parent Accounts	RelationshipToStudent	enum	–	No	No	Yes	Relationship type (father, mother, guardian).
Student_Parent Accounts	SupportPhoneNumber	varchar	15	No	No	Yes	Support phone number of the parent.
Student_Parent Accounts	CreatedAt	datetime	-6	No	No	No	Timestamp when the relationship was created.
Submissions	SubmittedBy	varchar	255	Yes	Yes	No	FK to Students.ID; student who submitted.
Submissions	AssignmentSubmitSlotID	varchar	255	Yes	Yes	No	FK to AssignmentSubmitSlots.PostID; assignment slot.
Submissions	CreatedAt	datetime	-6	No	No	No	Timestamp when the submission was created.
SubmissionDocuments	SubmissionDocumentID	varchar	255	Yes	No	No	Unique identifier for a submission document.
SubmissionDocuments	SubmissionID	varchar	255	No	Yes	No	Composite reference to Submissions; submission this file belongs to.
SubmissionDocuments	AssignmentSubmitSlotID	varchar	255	No	Yes	No	FK to AssignmentSubmitSlots.PostID; related assignment slot.
SubmissionDocuments	Creator	varchar	255	No	Yes	No	FK to MajorLecturers.ID /Staffs.ID; user who uploaded the document.

SubmissionDocuments	FilePath	varchar	500	No	No	Yes	Path of the submission document file.
SubmissionDocuments	FileData	longblob	–	No	No	Yes	Binary file data of the submission document.
SpecializedSubmissionDocuments	SpecializedSubmissionDocumentID	varchar	255	Yes	No	No	Unique identifier for specialized submission document.
SpecializedSubmissionDocuments	SubmittedBy	varchar	255	No	Yes	Yes	FK to Students.ID; student who submitted.
SpecializedSubmissionDocuments	SpecializedAssignmentSubmitSlotID	varchar	255	No	Yes	Yes	FK to SpecializedAssignmentSubmitSlots.PostID; slot associated.
SpecializedSubmissionDocuments	Creator	varchar	255	No	Yes	Yes	FK to MajorLecturers.ID /Staffs.ID; user who uploaded the document.
SpecializedSubmissionDocuments	FilePath	varchar	500	No	No	Yes	File path of the specialized document.
SpecializedSubmissionDocuments	FileData	longblob	–	No	No	Yes	Binary data of the specialized submission document.
SubmissionFeedbacks	AnnouncerID	varchar	255	Yes	Yes	No	FK to MajorLecturers.ID ; lecturer giving feedback.
SubmissionFeedbacks	SubmissionID	varchar	255	Yes	Yes	No	Composite FK to Submissions; submission being evaluated.
SubmissionFeedbacks	AssignmentSubmitSlotID	varchar	255	Yes	Yes	No	FK to AssignmentSubmitSlots.PostID; assignment slot.
SubmissionFeedbacks	Content	varchar	1000	No	No	Yes	Text content of the feedback.

SubmissionFeedbacks	Grade	enum	–	No	No	Yes	Grade assigned to the submission.
MajorSyllabuses	SyllabusID	varchar	255	Yes	No	No	Unique identifier of a major syllabus.
MajorSyllabuses	SyllabusName	varchar	255	No	No	No	Name of the major syllabus.
MajorSyllabuses	SubjectID	varchar	255	No	Yes	No	FK to Subjects.SubjectID; subject described by this syllabus.
MajorSyllabuses	Creator	varchar	255	No	Yes	No	FK to Staffs.ID; staff who created the syllabus.
MajorSyllabuses	file_type	varchar	255	No	No	Yes	File type (e.g. PDF, DOCX).
MajorSyllabuses	FilePath	varchar	500	No	No	Yes	File path of the syllabus document.
MajorSyllabuses	FileData	longblob	–	No	No	Yes	Binary data of the syllabus document.
MajorSyllabuses	Status	varchar	50	No	No	Yes	Status of the syllabus (e.g. active, draft).
MinorSyllabuses	SyllabusID	varchar	255	Yes	No	No	Unique identifier of a minor syllabus.
MinorSyllabuses	SyllabusName	varchar	255	No	No	No	Name of the minor syllabus.
MinorSyllabuses	SubjectID	varchar	255	No	Yes	No	FK to Subjects.SubjectID; minor subject described.
MinorSyllabuses	Creator	varchar	255	No	Yes	No	FK to DeputyStaffs.ID; deputy staff who created the syllabus.
MinorSyllabuses	file_type	varchar	255	No	No	Yes	File type of the minor syllabus file.
MinorSyllabuses	FilePath	varchar	500	No	No	Yes	File path of the minor syllabus document.

MinorSyllabuses	FileData	longblob	–	No	No	Yes	Binary data of the minor syllabus document.
MinorSyllabuses	Status	varchar	50	No	No	Yes	Status of the minor syllabus.
SpecializationSyllabuses	SyllabusID	varchar	255	Yes	No	No	Unique identifier of a specialization syllabus.
SpecializationSyllabuses	SyllabusName	varchar	255	No	No	No	Name of the specialization syllabus.
SpecializationSyllabuses	SpecializedSubjectID	varchar	255	No	Yes	No	FK to SpecializedSubjects.SubjectID; specialized subject.
SpecializationSyllabuses	Creator	varchar	255	No	Yes	No	FK to Staffs.ID; staff who created the specialization syllabus.
SpecializationSyllabuses	file_type	varchar	255	No	No	Yes	File type of the specialization syllabus file.
SpecializationSyllabuses	FilePath	varchar	500	No	No	Yes	File path of the specialization syllabus document.
SpecializationSyllabuses	FileData	longblob	–	No	No	Yes	Binary data of the specialization syllabus document.
SpecializationSyllabuses	Status	varchar	50	No	No	Yes	Status of the specialization syllabus.
PasswordResetTokens	id	bigint	–	Yes	No	No	Unique identifier for a password reset token.
PasswordResetTokens	PersonID	varchar	255	No	Yes	No	FK to Persons.ID; person who requested reset.
PasswordResetTokens	Token	varchar	255	No	No	No	Generated token string for password reset.
PasswordResetTokens	ExpiryDate	datetime	-6	No	No	No	Expiration datetime of the reset token.

Messages	MessageID	varchar	255	Yes	No	No	Unique identifier for a private message.
Messages	MessageSenderId	varchar	255	No	Yes	No	FK to Persons.ID; sender of the message.
Messages	MessageRecipientID	varchar	255	No	Yes	No	FK to Persons.ID; recipient of the message.
Messages	Text	varchar	1000	No	No	Yes	Text content of the private message.
Messages	Datetime	datetime	-6	No	No	No	Timestamp when the message was sent.
StudentRequired Subjects	StudentID	varchar	255	Yes	Yes	No	FK to Students.ID; student for whom the subject is required.
StudentRequired Subjects	SubjectID	varchar	255	Yes	Yes	No	FK to Subjects.SubjectID; subject that is required.
StudentRequired Subjects	RequiredReason	varchar	255	No	No	Yes	Reason why the subject is required.
StudentRequired Subjects	CreatedAt	datetime	-6	No	No	No	Timestamp when the requirement was created.
StudentRequired MajorSubjects	StudentID	varchar	255	Yes	Yes	No	FK to Students.ID; student with required major subject.
StudentRequired MajorSubjects	SubjectID	varchar	255	Yes	Yes	No	FK to Subjects.SubjectID; required major subject.
StudentRequired MajorSubjects	AssignedBy	varchar	255	No	Yes	Yes	FK to Staffs.ID; staff who assigned the requirement.
StudentRequired MinorSubjects	StudentID	varchar	255	Yes	Yes	No	FK to Students.ID; student with

								required minor subject.
StudentRequiredMinorSubjects	SubjectID	varchar	255	Yes	Yes	No		FK to Subjects.SubjectID; required minor subject.
StudentRequiredMinorSubjects	AssignedBy	varchar	255	No	Yes	Yes		FK to DeputyStaffs.ID; deputy staff who assigned the requirement.
Scholarships	ScholarshipID	varchar	255	Yes	No	No		Unique identifier for a scholarship type.
Scholarships	Creator	varchar	255	No	Yes	Yes		FK to Admins.ID; admin who created the scholarship.
Scholarships	TypeName	varchar	255	No	No	Yes		Name/type of the scholarship.
Scholarships	AwardDate	date	–	No	No	Yes		Default award date or reference date of the scholarship.
Scholarships	CreatedAt	datetime	-6	No	No	Yes		Timestamp when the scholarship type was created.
ScholarshipByYear	ScholarshipID	varchar	255	Yes	Yes	No		FK to Scholarships.ScholarshipID; scholarship configured for a year.
ScholarshipByYear	AdmissionYear	int	–	Yes	No	No		Admission year for which the scholarship applies.
ScholarshipByYear	Amount	double	–	No	No	No		Amount of the scholarship.
ScholarshipByYear	DiscountPercentage	double	–	No	No	Yes		Discount percentage offered.
ScholarshipByYear	Creator	varchar	255	No	Yes	No		FK to Admins.ID; admin who created this configuration.

ScholarshipByYear	Status	enum	–	No	No	No	Status of the scholarship configuration.
ScholarshipByYear	ContractStatus	enum	–	No	No	Yes	Contract status of the scholarship.
ScholarshipByYear	CreatedAt	datetime	-6	No	No	No	Timestamp when the scholarship-by-year record was created.
Students_Scholarships	StudentID	varchar	255	Yes	Yes	No	FK to Students.ID; student receiving scholarship.
Students_Scholarships	ScholarshipID	varchar	255	No	Yes	No	FK to Scholarships.ScholarshipID; scholarship assigned.
Students_Scholarships	AdmissionYear	int	–	No	No	No	Admission year of the student associated with the scholarship.
Students_Scholarships	AwardDate	date	–	No	No	No	Date when the scholarship was awarded to the student.
Students_Scholarships	Creator	varchar	255	No	Yes	No	FK to Staffs.ID; staff who assigned the scholarship.
Students_Scholarships	Status	enum	–	No	No	No	Status of the student's scholarship (active, revoked, etc.).
Students_Scholarships	CreatedAt	datetime	-6	No	No	No	Timestamp when the student-scholarship record was created.
EmailTemplates	id	int	–	Yes	No	No	Unique identifier for an email template.
EmailTemplates	campus_id	varchar	255	No	Yes	Yes	FK to Campuses.Camp

							usID; campus this template is for.
EmailTemplates	creator_id	varchar	255	No	Yes	Yes	FK to Admins.ID; admin who created the template.
EmailTemplates	type	enum	–	No	No	No	Template type (e.g. welcome, notification).
EmailTemplates	header_image	longblob	–	No	No	Yes	Binary data for the header image.
EmailTemplates	greeting	varchar	255	No	No	Yes	Greeting text.
EmailTemplates	salutation	varchar	255	No	No	Yes	Salutation/closing phrase.
EmailTemplates	body	text	–	No	No	Yes	Main body content of the email.
EmailTemplates	link_cta	varchar	255	No	No	Yes	Call-to-action link.
EmailTemplates	support	varchar	255	No	No	Yes	Support contact text.
EmailTemplates	copyright_notice	varchar	255	No	No	Yes	Copyright notice text.
EmailTemplates	link_facebook	varchar	255	No	No	Yes	Facebook link.
EmailTemplates	link_youtube	varchar	255	No	No	Yes	YouTube link.
EmailTemplates	link_tiktok	varchar	255	No	No	Yes	TikTok link.
EmailTemplates	banner_image	longblob	–	No	No	Yes	Binary data for banner image.
EmailTemplates	created_at	datetime	-6	No	No	No	Timestamp when the template was created.
EmailTemplates	updated_at	datetime	-6	No	No	Yes	Timestamp when the template was last updated.
SupportTickets	SupportTicketID	varchar	255	Yes	No	No	Unique identifier for a support ticket type.
SupportTickets	CreatorID	varchar	255	No	Yes	Yes	FK to Admins.ID; admin who created the ticket type.
SupportTickets	TicketName	varchar	255	No	No	No	Name of the support ticket type.
SupportTickets	Description	varchar	1000	No	No	Yes	Description of the support ticket type.

SupportTickets	Cost	double	–	No	No	No	Cost associated with this ticket type.
SupportTickets	CreatedAt	datetime	-6	No	No	No	Timestamp when the support ticket type was created.
SupportTicketRequests	RequestID	varchar	255	Yes	No	No	Unique identifier for a support ticket request.
SupportTicketRequests	Title	varchar	255	No	No	No	Title of the support request.
SupportTicketRequests	Description	varchar	2000	No	No	Yes	Detailed description of the request.
SupportTicketRequests	RequesterID	varchar	255	No	Yes	No	FK to Students.ID; student who submitted the request.
SupportTicketRequests	HandlerID	varchar	255	No	Yes	Yes	FK to Staffs.ID/Admins.ID; user handling the request.
SupportTicketRequests	SupportTicketID	varchar	50	No	Yes	Yes	FK to SupportTickets.SupportTicketID; requested ticket type.
SupportTicketRequests	Status	enum	–	No	No	No	Current status of the support request.
SupportTicketRequests	CreatedAt	datetime	-6	No	No	No	Timestamp when the request was created.
SupportTicketRequests	UpdatedAt	datetime	-6	No	No	Yes	Timestamp when the request was last updated.
SupportTicketRequests	CompletedAt	datetime	-6	No	No	Yes	Timestamp when the request was completed.
SupportTicketRequestsDocuments	DocumentID	varchar	255	Yes	No	No	Unique identifier for a support ticket request document.

SupportTicketRequestsDocuments	FileName	varchar	255	No	No	No	Name of the uploaded file.
SupportTicketRequestsDocuments	FileType	varchar	100	No	No	Yes	Type/MIME of the file.
SupportTicketRequestsDocuments	FileSize	bigint	—	No	No	Yes	Size of the file in bytes.
SupportTicketRequestsDocuments	FileData	longblob	—	No	No	No	Binary data of the uploaded file.
SupportTicketRequestsDocuments	UploadedAt	datetime	-6	No	No	No	Timestamp when the file was uploaded.
SupportTicketRequestsDocuments	RequestID	varchar	255	No	Yes	No	FK to SupportTicketRequests.RequestID; related request.
RetakeSubjects	StudentID	varchar	255	Yes	Yes	No	FK to Students.ID; student retaking the subject.
RetakeSubjects	SubjectID	varchar	255	Yes	Yes	No	FK to Subjects.SubjectID; subject to be retaken.
RetakeSubjects	RetakeReason	varchar	500	No	No	Yes	Reason for retaking the subject.
RetakeSubjects	CreatedAt	datetime	-6	No	No	No	Timestamp when retake was recorded.
TemporaryRetakeSubjects	StudentID	varchar	255	Yes	Yes	No	FK to Students.ID; student requesting temporary retake.
TemporaryRetakeSubjects	SubjectID	varchar	255	Yes	Yes	No	FK to Subjects.SubjectID; subject to be temporarily retaken.
TemporaryRetakeSubjects	RetakeReason	varchar	500	No	No	Yes	Reason for the temporary retake.

TemporaryRetakeSubjects	CreatedAt	datetime	-6	No	No	No	Timestamp when temporary retake was recorded.
TemporaryRetakeSubjects	Processed	tinyint	-1	No	No	No	Flag indicating whether the request has been processed.
TemporaryRetakeSubjects	Notes	varchar	1000	No	No	Yes	Additional processing notes.
MajorLecturers_Specializations	LecturerID	varchar	255	Yes	Yes	No	FK to MajorLecturers.ID ; major lecturer linked.
MajorLecturers_Specializations	SpecializationID	varchar	255	Yes	Yes	No	FK to Specializations.SpecializationID; specialization taught.
MajorLecturers_Specializations	CreatedAt	datetime	-6	No	No	No	Timestamp when the mapping was created.
MajorLecturers_Specializations	NotificationType	enum	—	No	No	Yes	Notification type for this mapping.
MajorAssignmentComments	CommentID	varchar	255	Yes	Yes	No	FK to Comments.CommentID; comment about a major assignment.
MajorAssignmentComments	CommenterID	varchar	255	No	Yes	No	FK to MajorEmployees.ID; major employee commenter.
MajorAssignmentComments	PostID	varchar	255	No	Yes	No	FK to AssignmentSubmitSlots.PostID; assignment slot commented on.
MajorAssignmentComments	NotificationType	enum	—	No	No	No	Notification type for this major assignment comment.
SpecializedAssignmentComments	CommentID	varchar	255	Yes	Yes	No	FK to Comments.CommentID; comment about a

								specialized assignment.
SpecializedAssignmentComments	CommenterID	varchar	255	No	Yes	No		FK to MajorEmployees.ID; employee who commented.
SpecializedAssignmentComments	PostID	varchar	255	No	Yes	No		FK to SpecializedAssignmentSubmitSlots.PostID; specialized slot.
SpecializedAssignmentComments	NotificationType	enum	—	No	No	No		Notification type for this specialized assignment comment.
SpecializedAssignmentSubmitSlots	PostID	varchar	255	Yes	Yes	No		FK to ClassPosts.PostID; class post that is a specialized assignment slot.
SpecializedAssignmentSubmitSlots	Creator	varchar	255	No	Yes	No		FK to MajorEmployees.ID; employee who created the slot.
SpecializedAssignmentSubmitSlots	ClassID	varchar	255	No	Yes	No		FK to SpecializedClasses.ClassID; specialized class of the assignment.
SpecializedAssignmentSubmitSlots	Deadline	datetime	-6	No	No	Yes		Deadline for specialized assignment submission.
SpecializedAssignmentSubmitSlots	NotificationType	enum	—	No	No	No		Notification type for the specialized assignment slot.
SpecializedSubmissions	SubmittedBy	varchar	255	Yes	Yes	No		FK to Students.ID; student who submitted the specialized assignment.

SpecializedSubmissions	SpecializedAssignmentSubmitSlotID	varchar	255	Yes	Yes	No	FK to SpecializedAssignmentSubmitSlots.PostID; slot submitted to.
SpecializedSubmissions	CreatedAt	datetime	-6	No	No	No	Timestamp when the specialized submission was created.
SpecializedSubmissionFeedbacks	AnnouncerID	varchar	255	Yes	Yes	No	FK to MajorLecturers.ID ; lecturer who gives feedback.
SpecializedSubmissionFeedbacks	SubmittedBy	varchar	255	Yes	Yes	No	FK to Students.ID; student whose submission is graded.
SpecializedSubmissionFeedbacks	SpecializedAssignmentSubmitSlotID	varchar	255	Yes	Yes	No	FK to SpecializedAssignmentSubmitSlots.PostID; related slot.
SpecializedSubmissionFeedbacks	Content	varchar	1000	No	No	Yes	Text content of the feedback.
SpecializedSubmissionFeedbacks	Grade	enum	—	No	No	Yes	Grade assigned to the specialized submission.
SpecializedSubmissionFeedbacks	CreatedAt	datetime	-6	No	No	No	Timestamp when the specialized feedback was created.

3.7.3 Database Schema

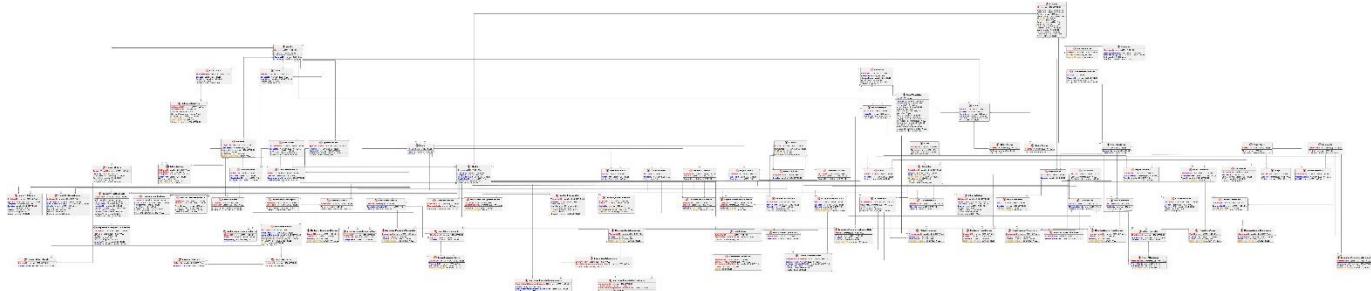


Diagram 9 Physical database schema diagram

The database schema defines the physical implementation of the logical data model and data dictionary on a concrete database management system. In this project, the system is implemented on **MySQL 8.x**, using the **InnoDB** storage engine and the **utf8mb4** character set to support multilingual data and special characters. Each entity defined in the ERD (e.g. Persons, Students, Admins, MajorEmployes, MinorEmployes, Majors, Specializations, Curriculums, Subjects, Classes, Timetable, Attendance, AcademicTranscripts, AccountBalances, FinancialHistories, SupportTickets, Submissions, Comments, etc.) is mapped directly to a physical table in MySQL. Primary keys, foreign keys, data types, and integrity constraints are explicitly specified using **CREATE TABLE** statements so that the physical schema exactly matches the semantics described in the ERD and data dictionary.

At the physical level, several global design conventions are applied. Table and column names are in English and follow consistent naming, closely aligned with the ERD. Primary keys are predominantly implemented as **VARCHAR(255)** so that they can store UUIDs or institution-specific identifiers without being constrained to numeric sequences. One-to-one “person–role” relationships (for example: Persons–Students, Persons–Admins, Persons–MajorEmployes, Persons–MinorEmployes, Persons–ParentAccounts, Persons–Authenticators) are implemented using the **table-per-role** pattern, where the role table uses the same ID value as both **PRIMARY KEY** and **FOREIGN KEY** referencing Persons(ID). Many-to-many relationships (such as student–class, lecturer–class, student–scholarship, and student–required-subject) are implemented via dedicated junction tables with composite primary keys over the participating identifiers (e.g. Students_Classes(StudentID, ClassID), Lecturers_Classes(LecturerID, ClassID), StudentRequiredSubjects(StudentID, SubjectID), RetakeSubjects(StudentID, SubjectID)). Status and classification attributes (e.g. gender, account status, requirement type, notification type, grade, scholarship status, contract status) are implemented using MySQL ENUM types to restrict values to a predefined set consistent with the business rules. Temporal attributes that require high precision use **DATETIME(6)** and **TIME(6)**; monetary values use **DECIMAL(15,2)** or **DOUBLE** as appropriate; large binary data such as images and documents are stored in **LONGBLOB**, and long textual content uses **TEXT** or **LONGTEXT**. Foreign key constraints are used extensively to enforce referential integrity; in general, deletion of parent records is restricted to prevent orphan records (e.g. **ON DELETE RESTRICT** or default behavior), while updates of key values are either restricted or cascaded depending on the nature of the relationship.

The **person and role layer** is organized around the Persons table, which stores core identity and contact information for every human in the system (students, staff, lecturers, parents, administrators). It includes attributes such as FirstName, LastName, Email, PhoneNumber, BirthDate, Gender, full address fields, and an optional Avatar. Specialized roles are implemented as separate tables whose primary key is also a foreign key to Persons(ID). For example, Students(ID) references Persons(ID) and adds student-specific attributes such as CampusID, SpecializationID, CurriculumID, Admission_Year, and createdDate. Admins(ID) references Persons(ID) and stores administrative detail, including CampusID, CreatorID, and FaceData. MajorEmployees and MinorEmployees identify academic and operational employees associated with specific campuses (and majors, for major employees). Staffs and DeputyStaffs specialize employees into staff roles for major and minor tracks respectively. MajorLecturers and MinorLecturers specialize employees into teaching roles, with additional attributes such as AddedBy and Type. Parent accounts are represented by ParentAccounts, which also use the person ID and can be linked to students through the Student_ParentAccounts junction table. Authentication data is separated into Authenticators, which links PersonID to a hashed Password and an AccountStatus enum, and PasswordResetTokens, which stores reset Token values and ExpiryDate tied back to PersonID.

The **academic structure layer** covers campuses, majors, specializations, curriculums, subjects, and tuition configurations. Campuses defines each campus with a CampusID primary key, an optional Creator referencing Admins(ID), descriptive attributes such as CampusName, OpeningDay, Description, and an Avatar. Majors defines high-level fields of study, with MajorID as primary key and a CreatorID referencing the admin who defined the major. Specializations further refine majors, with SpecializationID as primary key, MajorID as foreign key to Majors, and CreatorID referencing the responsible admin. Curriculums defines study programs with CurriculumID as primary key, descriptive attributes (Name, Description), and CreatorID referencing the admin who created the curriculum. Subjects is the central subject catalog, with SubjectID as primary key, AcceptorID referencing the user who approved the subject, SubjectName, Semester, an isAccepted flag, and a RequirementType enum specifying whether the subject is compulsory or elective. The relationships between subjects, majors, specializations, and curriculums are implemented in mapping tables: MajorSubjects links subjects to majors and curriculums; SpecializedSubjects links subjects to specializations and curriculums; MinorSubjects marks which subjects are taught as minor subjects. Tuition settings are stored in TuitionByYear, which uses a composite primary key (SubjectID, Admission_Year, CampusID) and foreign keys to Subjects, Campuses, and Admins (CreatorID), and stores Tuition, ReStudyTuition, and a ContractStatus enum.

The **class and scheduling layer** includes Classes, MajorClasses, MinorClasses, SpecializedClasses, Rooms, OfflineRooms, OnlineRooms, Slots, Timetable, MajorTimetable, MinorTimetable, and SpecializedTimetable. Classes defines generic classes with ClassID as primary key and attributes such as NameClass, SlotQuantity, Session (enum) and CreatedAt. The three specialized class tables each use ClassID as both primary key and foreign key to Classes(ClassID): MajorClasses links to a major SubjectID and a Creator (staff), MinorClasses links to a MinorSubjectID and a Creator (deputy staff), and SpecializedClasses links to a SpecializedSubjectID and a Creator (staff). Physical and virtual rooms are modeled via Rooms (with RoomID, RoomName, CampusID, Creator, CreatedAt, and Avatar), OfflineRooms (extending rooms with Address and Floor), and OnlineRooms (extending rooms with a Link URL). Time slots are defined in Slots with SlotID, a SlotName, and the StartTime and EndTime fields.

The Timetable table ties rooms and slots to specific days and weeks, with TimetableID as primary key, foreign keys to Rooms(RoomID) and Slots(SlotID), and attributes DayOfTheWeek, WeekOfYear, Year, and CreatedAt. Timetable entries are then specialized by class type: MajorTimetable, MinorTimetable, and SpecializedTimetable each use TimetableID as primary key and foreign key to Timetable, and add foreign keys to the appropriate class table (MajorClasses, MinorClasses, SpecializedClasses) together with Creator to record who created or scheduled the timetable entry.

The **learning, enrollment and evaluation layer** encompasses registrations, attendance, transcripts, and lecturer-class and lecturer-evaluation relationships. Student enrollment is represented via Students_Classes, which uses composite primary key (StudentID, ClassID) and foreign keys to Students and Classes. For additional semantics and notification metadata, Students_MajorClasses, Students_MinorClasses, and Students_SpecializedClasses refine these relationships for major, minor and specialized classes, and add attributes such as AddedBy and NotificationType. Lecturer-class assignments follow a similar pattern: Lecturers_Classes uses composite primary key (LecturerID, ClassID) and foreign keys to lecturer person IDs and Classes, while MajorLecturers_MajorClasses and MinorLecturers_MinorClasses further specialize these connections for major and minor lecturers and classes, storing AddedBy and NotificationType. Attendance is modeled by Attendance, which holds an AttendanceID primary key, StudentID, a Status enum, an optional Note, and CreatedAt. This base record is then classified by context through MajorAttendance, MinorAttendance, and SpecializedAttendance, each using AttendanceID as primary key and foreign key to Attendance, and adding foreign keys to the relevant timetable (MajorTimetable, MinorTimetable, SpecializedTimetable) and marker user (MarkedByID or MarkedBy), plus a NotificationType. Academic performance is stored in AcademicTranscripts (with TranscriptID, NumberID, Score, ScoreComponent1, ScoreComponent2, ScoreComponent3, Grade, and CreatedAt), and then specialized into MajorAcademicTranscripts, MinorAcademicTranscripts, and SpecializedAcademicTranscripts, each using TranscriptID as primary key and foreign key to AcademicTranscripts and adding ClassID, Creator, and NotificationType to tie results to the correct class type and responsible staff or deputy staff. Lecturer evaluation is implemented through LecturerEvaluations (linking ReviewerID to ClassID with feedback Text and CreatedAt) and specialized through MajorLecturerEvaluations and MinorLecturerEvaluations, which attach evaluations to MajorLecturerID or MinorLecturerID.

The **financial layer** is composed of AccountBalances, FinancialHistories, DepositHistories, PaymentHistories, SupportTicketHistories, Scholarships, ScholarshipByYear, and Students_Scholarships. AccountBalances uses StudentID as both primary key and foreign key to Students(ID) to record the current Balance and LastUpdated timestamp for each student account. All financial events affecting student balances are recorded in FinancialHistories, which has HistoryID as primary key, foreign keys StudentID and AccountBalanceID, and fields CurrentAmount, ChangedAmount, CreatedAt, Status, and version. Specialized history tables, DepositHistories, PaymentHistories, and SupportTicketHistories, use HistoryID as primary key and foreign key to FinancialHistories, and add context-specific attributes such as Amount or SubjectID or SupportTicketID. Scholarships are represented by Scholarships (defining base scholarship types, with ScholarshipID, TypeName, AwardDate, Creator, CreatedAt) and ScholarshipByYear (defining configuration for each AdmissionYear, with a composite primary key (ScholarshipID, AdmissionYear) and attributes such as Amount, DiscountPercentage, Status, ContractStatus, Creator, CreatedAt). Scholarships assigned to students are recorded in

Students_Scholarships, which links StudentID and ScholarshipID and stores AdmissionYear, AwardDate, Status, Creator, and CreatedAt. Additional control tables such as StudentRequiredSubjects, StudentRequiredMajorSubjects, StudentRequiredMinorSubjects, RetakeSubjects, TemporaryRetakeSubjects, and UpgradeStudents implement academic requirements, retake processes, and upgrade statuses, using composite keys over StudentID and SubjectID where needed and foreign keys back to students, subjects, and the staff or deputy staff who assigned them.

The **communication, content and support layer** contains all posts, comments, attachments, submissions, support tickets and internal messages. Public posts are modeled by PublicPosts (with PostID, Title, CreatedAt), which are specialized into Blogs and News via tables that use PostID as primary key and foreign key to PublicPosts and add Creator and Content. Class-level posts are stored in ClassPosts and specialized into MajorClassPosts, MinorClassPosts, AssignmentSubmitSlots, and SpecializedAssignmentSubmitSlots. These derived tables use PostID as primary key and foreign key to ClassPosts, and link to the appropriate class type (ClassID referencing MajorClasses, MinorClasses, SpecializedClasses) and creator role, while holding NotificationType and optional Deadline for assignment slots. All comments are stored in a base Comments table (with CommentID, Content, CreatedAt) and specialized in context-specific tables PublicComments, MajorComments, MinorComments, StudentComments, MajorAssignmentComments, and SpecializedAssignmentComments, which use CommentID as primary key and foreign key to Comments and add foreign keys to the target post, assignment slot, and commenter role, plus a NotificationType where needed. Generic documents attached to public posts are stored in Documents, whereas class-specific documents are stored in ClassDocuments. Assignment submissions and feedback are handled through Submissions and SpecializedSubmissions (composite primary keys over submitter and slot), with attached files in SubmissionDocuments and SpecializedSubmissionDocuments, and textual feedback and grading in SubmissionFeedbacks and SpecializedSubmissionFeedbacks, each linking back to lecturers, students, and assignment slots. Support processes are implemented by SupportTickets (ticket definitions), SupportTicketRequests (individual requests submitted by students), and SupportTicketRequestsDocuments (files associated with requests). Each support request is linked to its creator (RequesterID), handler (HandlerID), and SupportTicketID type, with Status, CreatedAt, UpdatedAt, and CompletedAt tracking the lifecycle. Additional financial links are made via SupportTicketHistories into FinancialHistories. Internal person-to-person communication is handled by the Messages table, which stores MessageID, MessageSenderId, MessageRecipientID, Text, and Datetime. Email layout and branding is configured in EmailTemplates, which includes a template type enum, links to the campus and creator, optional header and banner images, textual sections (greeting, salutation, body, etc.), social links, and timestamps.

To illustrate how this schema is implemented in MySQL, the following excerpts show representative CREATE TABLE statements. The full set of DDL statements for all tables follows the same pattern and can be provided in an appendix:

```

1   -- Core person table
2   CREATE TABLE Persons (
3       ID          VARCHAR(255) NOT NULL,
4       FirstName   VARCHAR(100),
5       LastName    VARCHAR(100),
6       Email       VARCHAR(255),
7       PhoneNumber VARCHAR(20),
8       BirthDate   DATE,
9       Gender      ENUM('MALE', 'FEMALE', 'OTHER'),
10      Country     VARCHAR(100),
11      Province    VARCHAR(100),
12      City        VARCHAR(100),
13      District    VARCHAR(100),
14      Ward        VARCHAR(100),
15      Street      VARCHAR(255),
16      PostalCode  VARCHAR(20),
17      Avatar      LONGBLOB,
18      PRIMARY KEY (ID)
19  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
20
21  -- Student role (1-1 with Persons)
22 • CREATE TABLE Students (
23      ID          VARCHAR(255) NOT NULL,
24      CreatorID   VARCHAR(255),
25      CampusID    VARCHAR(255),
26      SpecializationID VARCHAR(255),
27      CurriculumID VARCHAR(255),
28      createdDate  DATE      NOT NULL,
29      Admission_Year INT      NOT NULL,

```

Screenshot 1 Code database Persons and Students

SQL File 7* |        | Limit to 1000 rows |    |  

```

87     CONSTRAINT fk_majorclasses_subject
88         FOREIGN KEY (SubjectID) REFERENCES Subjects(SubjectID),
89     CONSTRAINT fk_majorclasses_creator
90         FOREIGN KEY (Creator) REFERENCES Staffs(ID)
91     ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
92
93     -- Time slot and timetable
94 • CREATE TABLE Slots (
95     SlotID      VARCHAR(255) NOT NULL,
96     SlotName    VARCHAR(100),
97     StartTime   TIME(6)      NOT NULL,
98     EndTime    TIME(6)      NOT NULL,
99     PRIMARY KEY (SlotID)
100    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
101
102 • CREATE TABLE Timetable (
103     TimetableID  VARCHAR(255) NOT NULL,
104     RoomID       VARCHAR(255) NOT NULL,
105     SlotID       VARCHAR(255) NOT NULL,
106     DayOfTheWeek ENUM('MON','TUE','WED','THU','FRI','SAT','SUN') NOT NULL,
107     WeekOfYear   INT          NOT NULL,
108     Year         INT          NOT NULL,
109     CreatedAt    DATETIME(6),
110     PRIMARY KEY (TimetableID),
111     CONSTRAINT fk_timetable_room
112         FOREIGN KEY (RoomID) REFERENCES Rooms(RoomID),
113     CONSTRAINT fk_timetable_slot
114         FOREIGN KEY (SlotID) REFERENCES Slots(SlotID)
115    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

```

Screenshot 2 Code database Timetable and Slot

```

SQL File 7* 
[File] [New] [Save] [Print] [Copy] [Find] [Replace] [X] Limit to 1000 rows | [Star] [Help] [Search] [?] [Close]

42
43      -- Campus definition
44 • Ⓜ CREATE TABLE Campuses (
45         CampusID      VARCHAR(255) NOT NULL,
46         Creator       VARCHAR(255),
47         CampusName    VARCHAR(255),
48         OpeningDay    DATE,
49         Description   VARCHAR(255),
50         Avatar        LONGBLOB,
51         PRIMARY KEY  (CampusID),
52         CONSTRAINT fk_campuses_creator
53             FOREIGN KEY (Creator) REFERENCES Admins(ID)
54     ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
55
56      -- Subject catalog
57 • Ⓜ CREATE TABLE Subjects (
58         SubjectID     VARCHAR(255) NOT NULL,
59         AcceptorID   VARCHAR(255),
60         SubjectName   VARCHAR(255) NOT NULL,
61         Semester      INT,
62         isAccepted    BIT(1)      NOT NULL,
63         RequirementType ENUM('COMPULSORY', 'ELECTIVE', 'OPTIONAL'),
64         PRIMARY KEY  (SubjectID),
65         CONSTRAINT fk_subjects_acceptor
66             FOREIGN KEY (AcceptorID) REFERENCES Admins(ID)
67     ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
68
69      -- Generic class
70 • Ⓜ CREATE TABLE Classes (

```

Screenshot 3 Code database subjects

3.7.4 Data Migration Strategy

Data Migration Strategy The data migration strategy defines how data from existing legacy systems will be identified, cleaned, transformed and loaded into the new, fully normalized database schema designed in this project, while preserving data consistency, referential integrity and minimizing disruption to users. Because the new model centralizes all human entities into the Persons table and then specializes them into role-specific tables such as Admins, MajorEmployees, MinorEmployees, Staffs, DeputyStaffs, MajorLecturers, MinorLecturers, Students and ParentAccounts, and further normalizes academic, financial, attendance and support processes into separate entities with many relationships, the migration must be executed in well-defined stages and in a strict dependency order. First, a source data analysis phase is carried out to inventory all legacy sources such as separate student, staff, lecturer and parent tables, user account tables, course and class section tables, grade and transcript tables, attendance logs, fee and payment records, scholarship spreadsheets, support ticket tools and any auxiliary CSV/Excel files. For each source dataset, the migration team identifies which target tables it maps to in the new schema, how existing identifiers can be translated into the varchar(255) primary keys used

across the new model, what data quality issues exist (missing values, inconsistent formats, duplicate persons, invalid foreign keys), and what implicit business rules are currently encoded (for example how admission year was calculated, how majors versus minors were differentiated, or how different grading components were stored). From this analysis, a formal source-to-target mapping document and a set of transformation and cleansing rules are defined. The second stage focuses on the person and role model. All legacy people records (students, staff, lecturers, admins, parents) are first consolidated and deduplicated into the Persons table by assigning each unique real person a single Persons.ID (possibly using generated UUIDs and keeping legacy IDs in a mapping table for traceability), while standardizing textual data (trimming spaces, unifying case, splitting full names into FirstName and LastName) and normalizing codes such as gender into the Gender enum. Once the common identity layer is populated, the role-specific tables are derived: students are inserted into Students with references to CampusID, SpecializationID and CurriculumID; administrative users are inserted into Admins; employees are distributed into MajorEmployees and MinorEmployees depending on organizational structure; operating staff into Staffs and DeputyStaffs; teaching roles into MajorLecturers and MinorLecturers; parents and guardians into ParentAccounts; and authentication data into Authenticators with AccountStatus mapped from legacy account states. At the same time, initial AccountBalances can be created for existing students, and, where possible, Avatar and FaceData fields can be populated from legacy profile storage. The third stage deals with institutional and academic structure. Legacy records for campuses, faculties, departments and programs are transformed into Campuses, Majors, Specializations and Curriculums, with clear rules for generating stable IDs and for mapping old codes and names to the new attributes such as CampusName, MajorName, SpecializationName and curriculum descriptions. The subject catalog from the legacy system is mapped into Subjects, with conversion rules for SubjectID, SubjectName, Semester and the isAccepted and RequirementType flags, then distributed into MajorSubjects, SpecializedSubjects and MinorSubjects based on the previous classification of courses in the old system, linking them correctly to Majors, Specializations and Curriculums. Historical tuition and fee settings are normalized into TuitionByYear by computing the combination of SubjectID, Admission_Year and CampusID and assigning Tuition, ReStudyTuition and ContractStatus. In parallel, scholarship types are mapped into Scholarships and ScholarshipByYear, and individual student awards are loaded into Students_Scholarships, ensuring that admission years and status flags correctly reflect the legacy data. The next stage targets classes, timetables, enrollment and attendance. Legacy class sections or course offerings are mapped to Classes, where each section is assigned a ClassID and basic attributes such as NameClass and SlotQuantity, and then specialized into MajorClasses, MinorClasses or SpecializedClasses depending on academic rules and whether the class is considered major, minor or specialized. Room definitions from the old system (physical rooms and online links) are normalized into Rooms, OfflineRooms and OnlineRooms, and time schedules into Slots and Timetable, with transformation logic to convert legacy time ranges and day-of-week formats into StartTime, EndTime, DayOfTheWeek, WeekOfYear and Year fields. From there, MajorTimetable, MinorTimetable and SpecializedTimetable records are derived by linking Timetable entries to their corresponding class types. Historical enrollment data are mapped into Students_Classes, then further specialized into Students_MajorClasses, Students_MinorClasses and Students_SpecializedClasses, while ensuring that StudentID and ClassID are resolved through the previously imported Persons, Students and Classes data. Attendance logs from the legacy system are converted into Attendance records (per student and date/time) and then classified into MajorAttendance, MinorAttendance or SpecializedAttendance

depending on which timetable and class the attendance event corresponds to, also resolving who marked the attendance into the MarkedByID/MarkedBy fields and setting NotificationType appropriately. In the grading and academic outcome stage, legacy grade book data and transcript tables are transformed into AcademicTranscripts, where business rules define how many components exist and how to map them into Score, ScoreComponent1, ScoreComponent2 and ScoreComponent3, and how to derive Grade according to institutional grading policies. These base transcripts are then linked to their appropriate academic context through MajorAcademicTranscripts, MinorAcademicTranscripts and SpecializedAcademicTranscripts, referencing the corresponding ClassID and Creator (staff or deputy staff). Additional constraints are applied to avoid creating transcripts for non-existent students or classes; any records that fail validation are logged into a migration error report for manual review. Financial information is migrated in a dedicated stage to ensure consistency of balances. Historical balances or aggregate fee/credit information is used to initialize AccountBalances for each student, while detailed transaction logs are mapped into FinancialHistories with clear semantics for CurrentAmount, ChangedAmount, Status and version. Transactions identified as deposits are inserted into DepositHistories, subject-related payments into PaymentHistories with SubjectID, and support-ticket-related financial operations into SupportTicketHistories linking to SupportTickets. During this phase, reconciliation scripts compare the sum of ChangedAmount over all histories with the final Balance for each student to detect and correct discrepancies. The system also migrates any retake information into RetakeSubjects and TemporaryRetakeSubjects, and required courses into StudentRequiredSubjects and its major/minor variants, ensuring that reasons and assignment users (AssignedBy) are preserved. Communication, support and content data form another migration block. Legacy emails/news/blogs/announcements are mapped into PublicPosts, Blogs and News, and class-level announcements into ClassPosts, MajorClassPosts and MinorClassPosts, with associated attachments moved into Documents or ClassDocuments, preserving file paths or embedding binary FileData where feasible. Comments are transformed into Comments and then into PublicComments, MajorComments, MinorComments, StudentComments, MajorAssignmentComments and SpecializedAssignmentComments based on who commented and what the target entity is. Historical assignment structures and submissions (if available) are mapped to AssignmentSubmitSlots and SpecializedAssignmentSubmitSlots (per class), Submissions and SpecializedSubmissions (per student and slot), submission files into SubmissionDocuments and SpecializedSubmissionDocuments, and grading feedback into SubmissionFeedbacks and SpecializedSubmissionFeedbacks, ensuring grading data remains linked to both the student and the lecturer. Additionally, any internal messaging data is migrated into Messages, ticketing data into SupportTickets and SupportTicketRequests (with related files in SupportTicketRequestsDocuments), and existing email layout configurations into EmailTemplates. After all transformation rules are defined, the physical migration is executed using an ETL (Extract–Transform–Load) pipeline, typically implemented with scripts, integration tools or database procedures. The loading order strictly respects referential integrity: first lookup tables and structural entities (Campuses, Majors, Specializations, Curriculums, Subjects, Rooms, Slots), then Persons and all role tables, then Classes and timetables, then enrollments, attendance, transcripts, finances, scholarships, support data, submissions and comments. Each stage writes migration logs, counts inserted/updated rows and records any rejected records with detailed error descriptions. Before going live, a series of validation and reconciliation tests are executed, including sample record comparison between old and new systems, aggregate checks

(total number of students, classes, credits, balances, scholarships), referential integrity checks (no orphaned foreign keys), functional tests of critical flows (login via Authenticators, student timetable and grade visibility, fee and payment history, support tickets) and performance checks on key queries. Finally, the cut-over plan defines how to switch from the legacy system to the new one with minimal downtime: a pilot migration is run in a staging environment, user acceptance tests are performed, then a final delta migration is executed during a planned maintenance window, after which the legacy system is frozen or moved to read-only mode. A rollback strategy (database backups and the ability to re-point applications to the old system) is prepared in case of critical issues. This end-to-end data migration strategy ensures that the complex ERD of the new platform—covering people, roles, academic structure, classes, timetables, attendance, grading, finance, communication and support—is populated in a controlled, auditable and reliable way, providing a consistent and trusted foundation for all subsequent system features.

3.8 User Interface Design

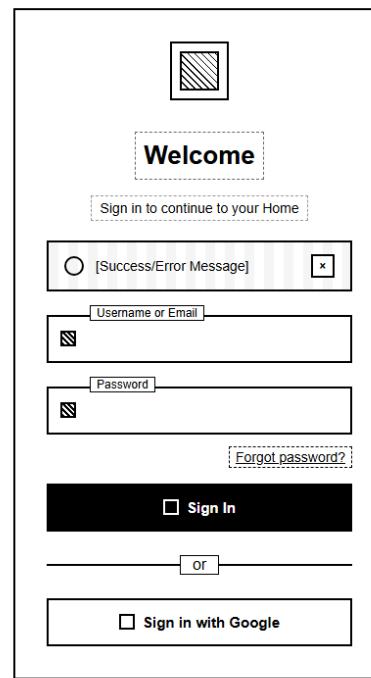
3.8.1 Design Philosophy and Principles

The Greenwich University Management System (GUMS) has a user centred design philosophy that is guided by the need to focus on clarity, simplicity and consistency in all modules and user roles. Since the system is being used by students on a daily basis, lecturers, academic staff and parents with varying degrees of digital literacy, the interface is available in such a way that cognitive load is kept at the lowest possible level and that the key information needed in academia can be seen and acted upon immediately. As opposed to being focused on visual novelty, GUMS is more functional and task-oriented: each screen is designed to focus on the most common workflows of each position, like checking out schedules, monitoring assessment, maintaining class lists, or approval requests or financial obligations. This philosophy is consistent with technology acceptance models that show that perception of usefulness and perceived ease of use are two determinants of user adoption, and constructivist perspectives of learning that focus on easy access to resources, feedback and interaction. In order to put this philosophy into practice, there are a few fundamental design principles that are used in the system all the time: Consistency and familiarity - The patterns of layouts, colour use, typography and navigation styles are repeated in all modules in such a way that, once a user has mastered the way to perform an activity in one part, the learning is transferred to another. Primary actions are put at predictable positions; icons and labels are in line with the most common web conventions to lessen the learning curve. Simplicity and visual hierarchy - Every page is logically organized with clear headings and the related contents are grouped together in logical blocks and the number of options given is minimized. Priority information like future classes, deadlines, messages and outstanding payments is displayed at the top page of the view whereas minor ones can be viewed either as collapsible panels or on additional pages. Role-conscious design - Interfaces are designed to meet the requirements of each role such that only the information and actions are displayed to users that are pertinent to the requirements of their role. As an example, students are preoccupied with schedules, assignments and grades; lecturers with class lists and marking software; employees with administrative processes; parents with a condensed overview of progress, attendance and finances. This minimizes clutter and assists to avoid errors. Feedback,

transparency and error prevention Communication - The system offers strong feedback to users on all user activity (submission of forms, updating records or making payments) through status messages, highlight states and confirmation dialogs. Validation rules and helpful error messages assist in preventing suboptimal or inconsistent data entry by forms, which helps maintain data quality and user trust. Being accessible and responsive - The interface is designed to be accessible to numerous devices and screen sizes to make sure that the key functionality is available on laptops, tablets and smartphones that the Greenwich Vietnam stakeholders use regularly. The font sizes, colors and interactive features are selected to facilitate readability and easy usability even in a low bandwidth or less powerful device environment. These principles ensure that the GUMS interface is coherent, efficient and inclusive to facilitate the frequent academic and administrative activities, in addition to supporting the overall objectives of the system relating to its usability, adoption and integrated information management.

3.8.2 Wireframes and Prototypes

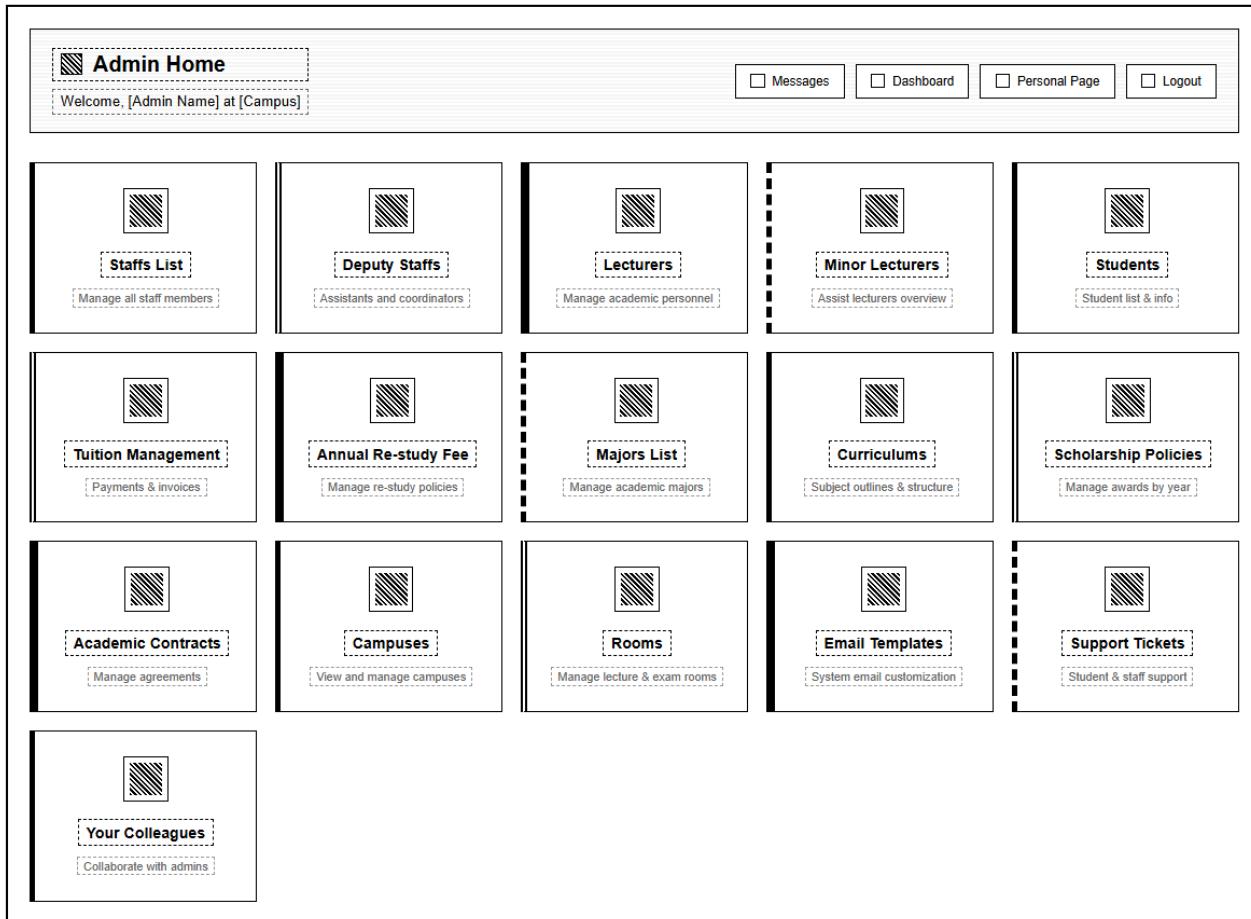
Wireframe and Prototype of the login page



The wireframe diagram illustrates the layout of a login page. At the top center is a placeholder for a logo, followed by the word "Welcome" in a box. Below this is a button labeled "Sign in to continue to your Home". A success/error message box contains a close button and the text "[Success/Error Message]". Below the message box are two input fields: one for "Username or Email" and one for "Password", both with placeholder text and a small icon. To the right of the password field is a link "[Forgot password?]". Below these fields is a large black rectangular button with the text "Sign In" in white. A horizontal line with the word "or" in the center separates the sign-in section from another button below. This second button is labeled "Sign in with Google" and also has a small icon.

Wireframe 1 Wireframe of login page

Wireframes and Prototype of Admin Home page



Wireframe 2 Wireframes and Prototype of Admin Home page

Wireframe and Prototype of staff list page

 Home
 Your Staffs List

Show:
 Total Staffs: 45
 Add Staff

ID	Avatar	Full Name	Email	Phone	Birth Date	Gender	Major	Campus	Creator	Actions
001		[Staff Name]	[email@example.com]	[Phone Number]	[YYYY-MM-DD]	[Gender]	[Major Name]	[Campus Name]	[Creator ID]	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
002		[Staff Name]	[email@example.com]	[Phone Number]	[YYYY-MM-DD]	[Gender]	[Major Name]	[Campus Name]	[Creator ID]	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
003		[Staff Name]	[email@example.com]	[Phone Number]	[YYYY-MM-DD]	[Gender]	[Major Name]	[Campus Name]	[Creator ID]	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
004		[Staff Name]	[email@example.com]	[Phone Number]	[YYYY-MM-DD]	[Gender]	[Major Name]	[Campus Name]	[Creator ID]	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
005		[Staff Name]	[email@example.com]	[Phone Number]	[YYYY-MM-DD]	[Gender]	[Major Name]	[Campus Name]	[Creator ID]	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
006		[Staff Name]	[email@example.com]	[Phone Number]	[YYYY-MM-DD]	[Gender]	[Major Name]	[Campus Name]	[Creator ID]	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
007		[Staff Name]	[email@example.com]	[Phone Number]	[YYYY-MM-DD]	[Gender]	[Major Name]	[Campus Name]	[Creator ID]	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
008		[Staff Name]	[email@example.com]	[Phone Number]	[YYYY-MM-DD]	[Gender]	[Major Name]	[Campus Name]	[Creator ID]	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
009		[Staff Name]	[email@example.com]	[Phone Number]	[YYYY-MM-DD]	[Gender]	[Major Name]	[Campus Name]	[Creator ID]	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

1

Go to page:

Wireframe 3 Wireframe and Prototype of staff list page

ID	Avatar	Name	Action
001		John Doe	Edit Delete
002		Jane Smith	Edit Delete
003		David Johnson	Edit Delete
004		Emily Williams	Edit Delete
005		Michael Brown	Edit Delete
006		Sarah Davis	Edit Delete
007		Matthew Wilson	Edit Delete
008		Amy Green	Edit Delete
009		Christopher Black	Edit Delete

Add New Staff

Avatar

Select Avatar
 Chon tệp | Không có tệp nào được chọn
Previously selected: [filename.jpg]

Staff Information

First Name	Last Name
<input type="text"/>	<input type="text"/>
Email	Phone Number
<input type="text"/>	<input type="text"/>
Account details will be emailed	
Birth Date	Gender
<input type="text"/> dd/mm/yyyy	<input type="button" value="Select gender"/>

Workplace Information

Major	Campus
<input type="button" value="Select major"/>	<input type="button" value="Select campus"/>

Address Information

Country	Province
<input type="text"/>	<input type="text"/>
City	District
<input type="text"/>	<input type="text"/>

[First](#) [Prev](#) [1](#) [2](#) [3](#) [...](#) [10](#) [Next](#) [Last](#) Go to page: [▼](#)

Wireframe 4 Wireframe and Prototype of staff list page

Wireframe and Prototype of staff edit page


 Upload New Avatar (Optional):

Không có tệp nào được chọn

 [Error: Invalid file format or size]

Staff Information

First Name * <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [First Name] "/> <div style="border: 1px dashed #ccc; padding: 2px; margin-top: 5px;">  [Error: First name is required] </div>	Last Name * <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [Last Name] "/>
<div style="border: 1px dashed #ccc; padding: 2px; margin-bottom: 5px;">  [Error: Invalid email format] </div>	
Email * <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [email@example.com] "/> <div style="border: 1px dashed #ccc; padding: 2px; margin-top: 5px;">  [Error: Invalid email format] </div>	Phone Number * <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [0123456789] "/>
Birth Date <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" 01 / 01 / 2000 "/>	Gender * <div style="display: flex; align-items: center;"> <input style="border: 1px solid #ccc; padding: 2px 10px; margin-right: 10px;" type="button" value="Male"/> <div style="flex-grow: 1;"></div> </div>

Workplace Information

Major * <div style="display: flex; align-items: center;"> <input style="border: 1px solid #ccc; padding: 2px 10px; margin-right: 10px;" type="button" value="Computer Science"/> <div style="flex-grow: 1;"></div> </div> <div style="border: 1px dashed #ccc; padding: 2px; margin-top: 5px;">  [Error: Major is required] </div>	Campus * <div style="display: flex; align-items: center;"> <input style="border: 1px solid #ccc; padding: 2px 10px; margin-right: 10px;" type="button" value="Main Campus"/> <div style="flex-grow: 1;"></div> </div>
---	---

Address Information

Country <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [Country] "/>	Province <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [Province] "/>
City <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [City] "/>	District <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [District] "/>
Ward <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [Ward] "/>	Street/House <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [Street Address] "/>
Postal Code <input style="width: 100%; border: 1px solid #ccc; height: 25px; margin-bottom: 5px;" type="text" value=" [10000] "/> <div style="font-size: small; color: #777;"> Example: 10000 </div>	

Wireframe 5 Wireframe and Prototype of staff edit page

Wireframe and Prototype of Manage Tuition Fees page

<input type="button" value="Home"/>	<input type="button" value="Approve Subjects"/>	<input type="button" value="Year: 2025"/>	<input type="button" value="Total Subjects: 45"/>	<input type="button" value="Reference Tuition Fees"/>																																													
Subjects already updated with fee - 2025																																																	
<table border="1"> <thead> <tr> <th>Subject ID</th> <th>Subject Name</th> <th>Subject Major</th> <th>Subject Type</th> <th>Admission Year</th> <th>Creator</th> <th>Creator Email</th> <th>Campus</th> <th>Tuition Fee</th> <th>Contract Status</th> </tr> </thead> <tbody> <tr> <td>SUB001</td> <td>[Subject Name]</td> <td>[Major]</td> <td>[Type]</td> <td>2025</td> <td>[Creator Name]</td> <td>[email@example.com]</td> <td>[Campus Name]</td> <td><input type="text" value="5000000"/></td> <td>PENDING</td> </tr> <tr> <td>SUB002</td> <td>[Subject Name]</td> <td>[Major]</td> <td>[Type]</td> <td>2025</td> <td>[Creator Name]</td> <td>[email@example.com]</td> <td>[Campus Name]</td> <td><input type="text" value="4500000"/></td> <td>ACTIVE</td> </tr> <tr> <td>SUB003</td> <td>[Subject Name]</td> <td>[Major]</td> <td>[Type]</td> <td>2025</td> <td>[Creator Name]</td> <td>[email@example.com]</td> <td>[Campus Name]</td> <td><input type="text" value="6000000"/></td> <td>PENDING</td> </tr> </tbody> </table> <p>Note: Fields with ACTIVE status are disabled</p>										Subject ID	Subject Name	Subject Major	Subject Type	Admission Year	Creator	Creator Email	Campus	Tuition Fee	Contract Status	SUB001	[Subject Name]	[Major]	[Type]	2025	[Creator Name]	[email@example.com]	[Campus Name]	<input type="text" value="5000000"/>	PENDING	SUB002	[Subject Name]	[Major]	[Type]	2025	[Creator Name]	[email@example.com]	[Campus Name]	<input type="text" value="4500000"/>	ACTIVE	SUB003	[Subject Name]	[Major]	[Type]	2025	[Creator Name]	[email@example.com]	[Campus Name]	<input type="text" value="6000000"/>	PENDING
Subject ID	Subject Name	Subject Major	Subject Type	Admission Year	Creator	Creator Email	Campus	Tuition Fee	Contract Status																																								
SUB001	[Subject Name]	[Major]	[Type]	2025	[Creator Name]	[email@example.com]	[Campus Name]	<input type="text" value="5000000"/>	PENDING																																								
SUB002	[Subject Name]	[Major]	[Type]	2025	[Creator Name]	[email@example.com]	[Campus Name]	<input type="text" value="4500000"/>	ACTIVE																																								
SUB003	[Subject Name]	[Major]	[Type]	2025	[Creator Name]	[email@example.com]	[Campus Name]	<input type="text" value="6000000"/>	PENDING																																								
<input type="button" value="Save Tuition Fees"/>																																																	
Subjects not updated with fee - 2025																																																	
<table border="1"> <thead> <tr> <th>Subject ID</th> <th>Subject Name</th> <th>Subject Major</th> <th>Subject Type</th> <th>Admission Year</th> <th>Creator</th> <th>Creator Email</th> <th>Campus</th> <th>Tuition Fee</th> <th>Contract Status</th> </tr> </thead> <tbody> <tr> <td>SUB004</td> <td>[Subject Name]</td> <td>[Major]</td> <td>[Type]</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td><input type="text" value="Enter fee"/></td> <td>N/A</td> </tr> <tr> <td>SUB005</td> <td>[Subject Name]</td> <td>[Major]</td> <td>[Type]</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td><input type="text" value="Enter fee"/></td> <td>N/A</td> </tr> <tr> <td>SUB006</td> <td>[Subject Name]</td> <td>[Major]</td> <td>[Type]</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td><input type="text" value="Enter fee"/></td> <td>N/A</td> </tr> </tbody> </table> <p>Note: Enter tuition fees for subjects without data</p>										Subject ID	Subject Name	Subject Major	Subject Type	Admission Year	Creator	Creator Email	Campus	Tuition Fee	Contract Status	SUB004	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A	SUB005	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A	SUB006	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A
Subject ID	Subject Name	Subject Major	Subject Type	Admission Year	Creator	Creator Email	Campus	Tuition Fee	Contract Status																																								
SUB004	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A																																								
SUB005	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A																																								
SUB006	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A																																								
<input type="button" value="Save Tuition Fees"/>																																																	
Reference Tuition Fees for Campus - [Campus Name] - 2025																																																	
<table border="1"> <thead> <tr> <th>Subject ID</th> <th>Subject Name</th> <th>Admission Year</th> <th>Creator</th> <th>Creator Email</th> <th>Tuition Fee</th> </tr> </thead> <tbody> <tr> <td>REF001</td> <td>[Reference Subject 1]</td> <td>2025</td> <td>[Creator Name]</td> <td>[email@example.com]</td> <td>5,000,000</td> </tr> <tr> <td>REF002</td> <td>[Reference Subject 2]</td> <td>2025</td> <td>[Creator Name]</td> <td>[email@example.com]</td> <td>4,800,000</td> </tr> <tr> <td>REF003</td> <td>[Reference Subject 3]</td> <td>2025</td> <td>[Creator Name]</td> <td>[email@example.com]</td> <td>6,200,000</td> </tr> </tbody> </table>										Subject ID	Subject Name	Admission Year	Creator	Creator Email	Tuition Fee	REF001	[Reference Subject 1]	2025	[Creator Name]	[email@example.com]	5,000,000	REF002	[Reference Subject 2]	2025	[Creator Name]	[email@example.com]	4,800,000	REF003	[Reference Subject 3]	2025	[Creator Name]	[email@example.com]	6,200,000																
Subject ID	Subject Name	Admission Year	Creator	Creator Email	Tuition Fee																																												
REF001	[Reference Subject 1]	2025	[Creator Name]	[email@example.com]	5,000,000																																												
REF002	[Reference Subject 2]	2025	[Creator Name]	[email@example.com]	4,800,000																																												
REF003	[Reference Subject 3]	2025	[Creator Name]	[email@example.com]	6,200,000																																												

Wireframe 6 Wireframe and Prototype of Manage Tuition Fees page

<input type="button" value="Home"/> <input type="button" value="Approve Subjects"/>	Select Campus for Reference <input type="checkbox"/> Campus: Choose a campus		Total Subjects: 45 <input type="checkbox"/> Reference Tuition Fees																																										
Subjects already updated with fee - 2025 <table border="1" style="width: 100%; border-collapse: collapse; margin-top: 5px;"> <thead> <tr> <th>Subject ID</th> <th>Subject Name</th> <th>Subject Major</th> </tr> </thead> <tbody> <tr> <td>SUB001</td> <td>[Subject Name]</td> <td>[Major]</td> </tr> <tr> <td>SUB002</td> <td>[Subject Name]</td> <td>[Major]</td> </tr> <tr> <td>SUB003</td> <td>[Subject Name]</td> <td>[Major]</td> <td>[Type]</td> <td>2025</td> <td>[Creator Name]</td> <td>[email@example.com]</td> <td>[Campus Name]</td> </tr> </tbody> </table> <p>Note: Fields with ACTIVE status are disabled</p>		Subject ID	Subject Name	Subject Major	SUB001	[Subject Name]	[Major]	SUB002	[Subject Name]	[Major]	SUB003	[Subject Name]	[Major]	[Type]	2025	[Creator Name]	[email@example.com]	[Campus Name]	<input type="button" value="Reference"/> <input type="button" value="Cancel"/> <table border="1" style="width: 100%; border-collapse: collapse; margin-top: 5px;"> <thead> <tr> <th>Subject Name</th> <th>Tuition Fee</th> <th>Contract Status</th> </tr> </thead> <tbody> <tr> <td>Subject Name 1</td> <td>5000000</td> <td>PENDING</td> </tr> <tr> <td>Subject Name 2</td> <td>4500000</td> <td>ACTIVE</td> </tr> <tr> <td>Subject Name 3</td> <td>6000000</td> <td>PENDING</td> </tr> </tbody> </table>			Subject Name	Tuition Fee	Contract Status	Subject Name 1	5000000	PENDING	Subject Name 2	4500000	ACTIVE	Subject Name 3	6000000	PENDING												
Subject ID	Subject Name	Subject Major																																											
SUB001	[Subject Name]	[Major]																																											
SUB002	[Subject Name]	[Major]																																											
SUB003	[Subject Name]	[Major]	[Type]	2025	[Creator Name]	[email@example.com]	[Campus Name]																																						
Subject Name	Tuition Fee	Contract Status																																											
Subject Name 1	5000000	PENDING																																											
Subject Name 2	4500000	ACTIVE																																											
Subject Name 3	6000000	PENDING																																											
Subjects not updated with fee - 2025 <table border="1" style="width: 100%; border-collapse: collapse; margin-top: 5px;"> <thead> <tr> <th>Subject ID</th> <th>Subject Name</th> <th>Subject Major</th> <th>Subject Type</th> <th>Admission Year</th> <th>Creator</th> <th>Creator Email</th> <th>Campus</th> <th>Tuition Fee</th> <th>Contract Status</th> </tr> </thead> <tbody> <tr> <td>SUB004</td> <td>[Subject Name]</td> <td>[Major]</td> <td>[Type]</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td><input type="text" value="Enter fee"/></td> <td>N/A</td> </tr> <tr> <td>SUB005</td> <td>[Subject Name]</td> <td>[Major]</td> <td>[Type]</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td><input type="text" value="Enter fee"/></td> <td>N/A</td> </tr> <tr> <td>SUB006</td> <td>[Subject Name]</td> <td>[Major]</td> <td>[Type]</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td>N/A</td> <td><input type="text" value="Enter fee"/></td> <td>N/A</td> </tr> </tbody> </table> <p>Note: Enter tuition fees for subjects without data</p>					Subject ID	Subject Name	Subject Major	Subject Type	Admission Year	Creator	Creator Email	Campus	Tuition Fee	Contract Status	SUB004	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A	SUB005	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A	SUB006	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A	<input type="button" value="Save Tuition Fees"/>
Subject ID	Subject Name	Subject Major	Subject Type	Admission Year	Creator	Creator Email	Campus	Tuition Fee	Contract Status																																				
SUB004	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A																																				
SUB005	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A																																				
SUB006	[Subject Name]	[Major]	[Type]	N/A	N/A	N/A	N/A	<input type="text" value="Enter fee"/>	N/A																																				
Reference Tuition Fees for Campus - [Campus Name] - 2025 <table border="1" style="width: 100%; border-collapse: collapse; margin-top: 5px;"> <thead> <tr> <th>Subject ID</th> <th>Subject Name</th> <th>Admission Year</th> <th>Creator</th> <th>Creator Email</th> <th>Tuition Fee</th> </tr> </thead> <tbody> <tr> <td>REF001</td> <td>[Reference Subject 1]</td> <td>2025</td> <td>[Creator Name]</td> <td>[email@example.com]</td> <td>5,000,000</td> </tr> <tr> <td>REF002</td> <td>[Reference Subject 2]</td> <td>2025</td> <td>[Creator Name]</td> <td>[email@example.com]</td> <td>4,800,000</td> </tr> <tr> <td>REF003</td> <td>[Reference Subject 3]</td> <td>2025</td> <td>[Creator Name]</td> <td>[email@example.com]</td> <td>6,200,000</td> </tr> </tbody> </table>					Subject ID	Subject Name	Admission Year	Creator	Creator Email	Tuition Fee	REF001	[Reference Subject 1]	2025	[Creator Name]	[email@example.com]	5,000,000	REF002	[Reference Subject 2]	2025	[Creator Name]	[email@example.com]	4,800,000	REF003	[Reference Subject 3]	2025	[Creator Name]	[email@example.com]	6,200,000	<input type="button" value="Save Tuition Fees"/>																
Subject ID	Subject Name	Admission Year	Creator	Creator Email	Tuition Fee																																								
REF001	[Reference Subject 1]	2025	[Creator Name]	[email@example.com]	5,000,000																																								
REF002	[Reference Subject 2]	2025	[Creator Name]	[email@example.com]	4,800,000																																								
REF003	[Reference Subject 3]	2025	[Creator Name]	[email@example.com]	6,200,000																																								

Wireframe 7 Wireframe and Prototype of Manage Tuition Fees page

Wireframe and Prototype of Approve Subjects page

		 Home	 Back to List	 Approve Subjects	 Total Unaccepted Subjects: 12
 ID	 Subject Name	 Semester	 Creator	<input type="checkbox"/> All	
SUB001	[Introduction to Programming]	Semester 1	[Creator Name]	<input type="checkbox"/>	
SUB002	[Data Structures]	Semester 2	[Creator Name]	<input type="checkbox"/>	
SUB003	[Database Management]	Semester 3	[Creator Name]	<input type="checkbox"/>	
SUB004	[Web Development]	N/A	[Creator Name]	<input type="checkbox"/>	
SUB005	[Software Engineering]	Semester 4	N/A	<input type="checkbox"/>	
SUB006	[Machine Learning]	Semester 5	[Creator Name]	<input type="checkbox"/>	

Note: Select subjects by checking the boxes, then click "Approve Selected" to approve them. Use "All" checkbox in header to select/deselect all subjects at once.

 **Approve Selected**

Wireframe 8 Wireframe and Prototype of Approve Subjects page

Wireframe and Prototype of Reference Tuition Fees page

Home
 Manage Tuition Fees
 Reference Tuition Fees for [Campus Name]

Campus: North Campus
 Year:

2025
 Reference
Total Subjects: 28

Reference Tuition Fees for Campus - [North Campus] - 2025

Subject ID	Subject Name	Subject Major	Subject Type	Admission Year	Creator	Creator Email	Tuition Fee
SUB001	[Introduction to Programming]	[Computer Science]	[Core]	2025	[Creator Name]	[email@example.com]	5,000,000
SUB002	[Data Structures]	[Computer Science]	[Core]	2025	[Creator Name]	[email@example.com]	4,800,000
SUB003	[Database Management]	[Information Systems]	[Elective]	2025	[Creator Name]	[email@example.com]	6,200,000
SUB004	[Web Development]	[Computer Science]	[Elective]	2025	N/A	N/A	5,500,000
SUB005	[Software Engineering]	[Computer Science]	[Core]	2025	[Creator Name]	[email@example.com]	7,000,000
SUB006	[Machine Learning]	[Artificial Intelligence]	[Advanced]	2025	[Creator Name]	[email@example.com]	8,500,000
SUB007	[Business Analytics]	[Business Administration]	[Core]	2025	[Creator Name]	[email@example.com]	5,200,000

Note: This table displays reference tuition fees from the selected campus and year. Use these values as a reference when setting tuition fees for your campus.

Wireframe 9 Wireframe and Prototype of Reference Tuition Fees page

Wireframe and Prototype of Reference Annual Re-study Fees page

[!\[\]\(0d63a65cf533297d7f95515ebb3ccd37_img.jpg\) Home](#)
[!\[\]\(7e9cd24a7a74c26f5ff3663c528413dc_img.jpg\) Manage Re-study Fees](#)
Reference Annual Re-study Fees

Reference Annual Re-study Fees for Campus - [North Campus] - 2025

Subject ID	Subject Name	Subject Major	Subject Type	Admission Year	Creator	Creator Email	Standard Tuition Fee	Re-study Fee
SUB001	[Introduction to Programming]	[Computer Science]	[Core]	2025	[Creator Name]	[email@example.com]	5,000,000	6,000,000
SUB002	[Data Structures & Algorithms]	[Computer Science]	[Core]	2025	[Creator Name]	[email@example.com]	4,800,000	5,760,000
SUB003	[Database Management Systems]	[Information Systems]	[Elective]	2025	[Creator Name]	[email@example.com]	6,200,000	7,440,000
SUB004	[Web Development]	[Computer Science]	[Elective]	2025	N/A	N/A	5,500,000	6,600,000
SUB005	[Software Engineering]	[Computer Science]	[Core]	2025	[Creator Name]	[email@example.com]	7,000,000	8,400,000
SUB006	[Machine Learning]	[Artificial Intelligence]	[Advanced]	2025	[Creator Name]	[email@example.com]	8,500,000	10,200,000
SUB007	[Business Analytics]	[Business Administration]	[Core]	2025	[Creator Name]	[email@example.com]	5,200,000	6,240,000
SUB008	[Operating Systems]	[Computer Science]	[Core]	2025	[Creator Name]	[email@example.com]	5,800,000	6,960,000

Note: Re-study fees are typically 120% of the standard tuition fee. This table displays reference values from the selected campus and year. Use these as a guide when setting re-study fees for your campus.

Wireframe 10 Wireframe and Prototype of Reference Annual Re-study Fees page

Wireframe and Prototype of Majors Management page

 Home
 Majors Management
 Add Major

 [Success Message: Major added/updated/deleted successfully]

Majors

<p>[Computer Science]</p> <p>ID: MAJ001 Created Date: 2024-01-15 Creator: ADMIN001</p> <p> Edit  Delete  View Specs</p>	<p>[Business Administration]</p> <p>ID: MAJ002 Created Date: 2024-01-20 Creator: ADMIN001</p> <p> Edit  Delete  View Specs</p>	<p>[Mechanical Engineering]</p> <p>ID: MAJ003 Created Date: 2024-02-01 Creator: ADMIN002</p> <p> Edit  Delete  View Specs</p>
<p>[Graphic Design]</p> <p>ID: MAJ004 Created Date: 2024-02-10 Creator: N/A</p> <p> Edit  Delete  View Specs</p>	<p>[Chemistry]</p> <p>ID: MAJ005 Created Date: 2024-02-15 Creator: ADMIN001</p> <p> Edit  Delete  View Specs</p>	<p>[Data Science]</p> <p>ID: MAJ006 Created Date: 2024-02-20 Creator: ADMIN002</p> <p> Edit  Delete  View Specs</p>

Wireframe 11 Wireframe and Prototype of Majors Management page

[Home](#)
[Majors Management](#)
[Add Major](#)

(Success Message: Major added/updated/deleted successfully)

Majors

[Computer Science]		
ID: MAJ001	Created Date: 2024-01-15	Creator: ADMIN001
Edit	Delete	View Specs
[Graphic Design]		
ID: MAJ004	Created Date: 2024-02-10	Creator: N/A
Edit	Delete	View Specs
[Chemistry]		
ID: MAJ005	Created Date: 2024-02-15	Creator: ADMIN001
Edit	Delete	View Specs
[Data Science]		
ID: MAJ006	Created Date: 2024-02-20	Creator: ADMIN002
Edit	Delete	View Specs

Add New Major

Major Name *

[Error: Major name is required]

Avatar

Không có tệp nào được chọn

[Error: Invalid file format]

[Save Major](#)
[Cancel](#)

[Mechanical Engineering]

[Mechanical Engineering]		
ID: MAJ002	Created Date: 2024-02-01	Creator: ADMIN002
Edit	Delete	View Specs

Wireframe 12 Wireframe and Prototype of Majors Management page

Wireframe and Prototype of Majors Edit page

Edit Major

Edit Major Form

Major Name *

[Enter major name]

[Error: Major name is required]

Avatar

Chon tệp Không có tệp nào được chọn

[Error: Invalid file format]

Save **Cancel**

Wireframe 13 Wireframe and Prototype of Majors Edit page

Wireframe and Prototype of Specializations Management page

Home
 Back
Specializations

Specializations List

ID	Name	Creator	Created At	Actions
SP001	[Software Engineering]	ADMIN001	2024-01-20	<input type="button" value="Edit"/> <input style="border: 1px dashed black; border-radius: 5px; padding: 2px 10px;" type="button" value="Delete"/>
[No data found]				

First
Prev
1
2
3
Next
Last

Wireframe 14 Wireframe and Prototype of Specializations Management page

Home
 Back
Specializations

Specializations List

ID	Name	Creator	Created At	Actions
SP001	[Software Engineering]			<input type="button" value="Edit"/> <input style="border: 1px dashed black; border-radius: 5px; padding: 2px 10px;" type="button" value="Delete"/>

Add Specialization

Wireframe 15 Wireframe and Prototype of Specializations Management page

Wireframe and Prototype of Manage Curriculums page

 Home
Manage Curriculums
Add Curriculum

 **Curriculum List**

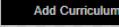
ID	Name	Description	Creator	Created At	Actions
CUR001	[Computer Science Program]	[Intro to algorithms & programming fundamentals]	ADMIN001	12/03/2024	 
CUR002	[Business Administration]	[Basics of management & enterprise operations]	ADMIN002	19/03/2024	 
CUR003	[Mechanical Engineering]	[Dynamics, materials, and manufacturing fundamentals]	ADMIN003	22/03/2024	 
CUR004	[Graphic Design]	[Visual communication & digital tools foundation]	ADMIN001	25/03/2024	 
CUR005	[Data Science]	[Statistics, machine learning & data processing basics]	ADMIN002	28/03/2024	 
CUR006	[Chemistry & Lab Foundations]	[Organic chemistry & laboratory practice basics]	ADMIN004	01/04/2024	 
CUR007	[Information Systems]	[Systems analysis and database fundamentals]	ADMIN002	05/04/2024	 
CUR008	[Electrical Engineering]	[Circuits, electromagnetism & electronics fundamentals]	ADMIN003	07/04/2024	 
CUR009	[AI Engineering]	[Deep learning basics & implementation overview]	ADMIN005	10/04/2024	 
CUR010	[Cybersecurity Fundamentals]	[Network security & vulnerability analysis basics]	ADMIN006	12/04/2024	 
[No curriculums found]					

First Prev 1 2 Next Last

Wireframe 16 Wireframe and Prototype of Manage Curriculums page

 Home

Manage Curriculums



Curriculum List

ID	Name	Description	Creator	Created At	Actions
CUR001	[Computer Science Program]	[Intro to algorithms & programming fundamentals]	ADMIN001	12/03/2024	 
CUR002	[Business Administration]			03/2024	 
CUR003	[Mechanical Engineering]			03/2024	 
CUR004	[Graphic Design]			03/2024	 
CUR005	[Data Science]			03/2024	 
CUR006	[Chemistry & Lab Foundations]			04/2024	 
CUR007	[Information Systems]			04/2024	 
CUR008	[Electrical Engineering]			04/2024	 
CUR009	[AI Engineering]	[Deep learning basics & implementation overview]	ADMIN005	10/04/2024	 
CUR010	[Cybersecurity Fundamentals]	[Network security & vulnerability analysis basics]	ADMIN006	12/04/2024	 
[No curriculums found]					

First Prev 1 2 Next Last

 Add Curriculum

Curriculum Name *

[Required]

Description

Save Cancel

Wireframe 17 Wireframe and Prototype of Manage Curriculums page

Wireframe and Prototype of Manage Scholarships by Year page

		<input type="button" value="Home"/>	<input type="button" value="Add Scholarship"/>	<input type="button" value="Year"/>	<input type="button" value="2023"/>	<input type="button" value="Total: [12]"/>
Scholarships With Amounts [Year]						
ID	Type Name	Award Date	Creator	Amount	Discount %	
SC001	[Merit Scholarship]	[12-04-2024]	[ADMIN001]	<input type="text"/>	<input type="text"/>	
SC002	[Excellent Performance]	[10-04-2024]	[ADMIN002]	<input type="text"/>	<input type="text"/>	
SC003	[Financial Aid]	[09-04-2024]	[ADMIN004]	<input type="text"/>	<input type="text"/>	
[No Data]						
<input type="button" value="Save Amounts"/>						
Scholarships Without Amount						
ID	Type Name	Award Date	Creator	Amount	Discount %	
SC010	[Volunteer Scholarship]	[-]	[ADMIN003]	<input type="text"/>	<input type="text"/>	
[No Data]						
<input type="button" value="Save Amounts"/>						

Wireframe 18 Wireframe and Prototype of Manage Scholarships by Year page

 Home
 Add Scholarship
Year: [2023] 
Total: [12]

 Scholarships With Amounts [Year]					
ID	Type Name	Award Date	Creator	Amount	Discount %
SC001	[Merit Scholarship]	[12-04-2024]	[ADMIN001]	<input type="text"/>	<input type="text"/>
SC002	[Excellent Performance]	[10-04-2024]	[ADMIN002]	<input type="text"/>	<input type="text"/>
SC003	[Financial Aid]			<input type="text"/>	<input type="text"/>

 Add Scholarship

Scholarship Type Name *
 [Enter scholarship type]
[Error: Required]

 Save  Cancel

 Scholarships Without Amount					
ID	Type Name	Award Date	Creator	Amount	Discount %
SC010	[Volunteer Scholarship]	[-]	[ADMIN003]	<input type="text"/>	<input type="text"/>

[No Data]  Save Amounts

Wireframe 19 Wireframe and Prototype of Manage Scholarships by Year page

Wireframe and Prototype of Manage Academic Contracts page

 Home
 Manage Academic Contracts
Year:
Total Subjects: 28 Total Scholarships: 10

 **Subjects with Tuition & Re-study Fee – 2024**

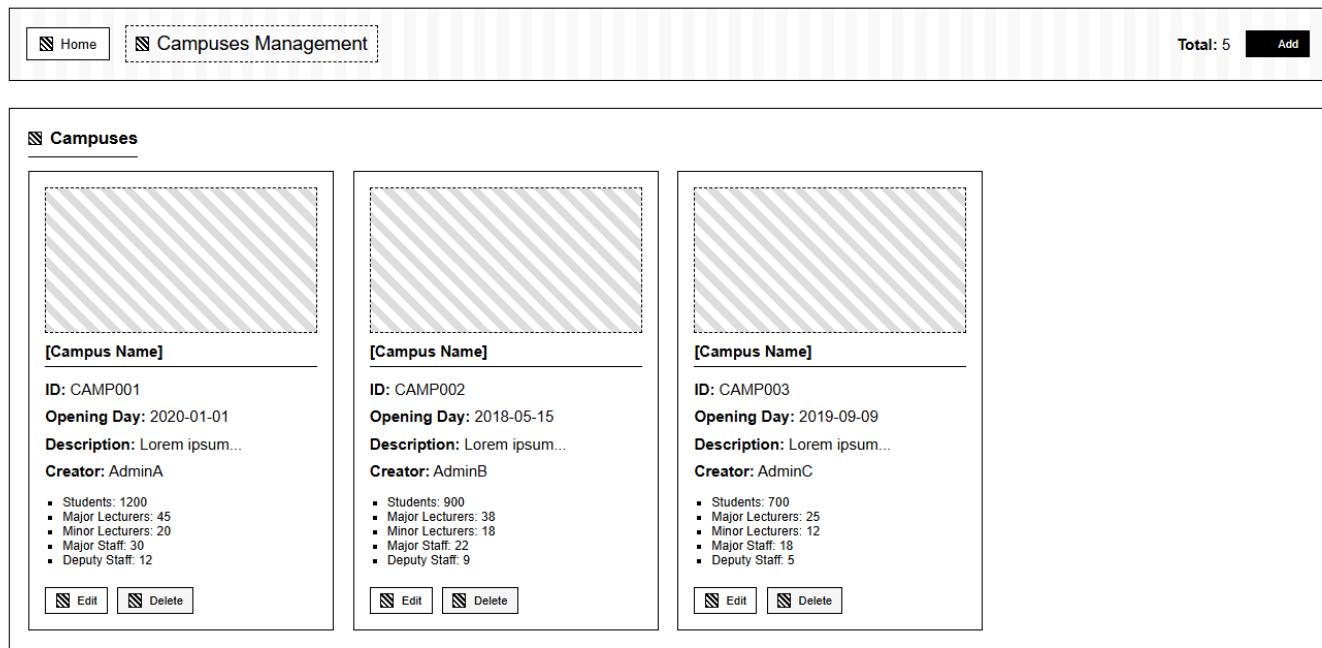
ID	Subject Name	Major	Type	Year	Creator	Email	Campus	Tuition	Re-study Fee	Contract Status
SUB001	[Intro to Programming]	[Computer Science]	[Compulsory]	2024	[Admin A]	[admin@school.com]	[Main]	1,200	600	[ACTIVE]
SUB002	[Data Structures]	[Software Engineering]	[Compulsory]	2024	[Admin B]	[staff@school.com]	[Main]	1,400	700	[PENDING]
SUB003	[Calculus I]	[General]	[General]	2024	[Admin A]	[admin@school.com]	[North]	900	450	[ACTIVE]
SUB004	[Database Systems]	[IT]	[Compulsory]	2024	[Admin C]	[c@school.com]	[South]	1,350	680	[DRAFT]
SUB005	[Machine Learning]	[AI]	[Elective]	2024	[Admin D]	[d@school.com]	[Main]	1,600	800	[ACTIVE]

 **Scholarships – 2024**

ID	Type Name	Year	Creator	Email	Amount	Discount %	Status
SC001	[Merit Scholarship]	2024	[Admin A]	[admin@school.com]	2,000	20%	[ACTIVE]
SC002	[Excellence Award]	2024	[Admin B]	[b@school.com]	1,500	15%	[PENDING]
SC003	[Financial Aid A]	2024	[Admin D]	[d@school.com]	1,000	10%	[ACTIVE]

Wireframe 20 Wireframe and Prototype of Manage Academic Contracts page

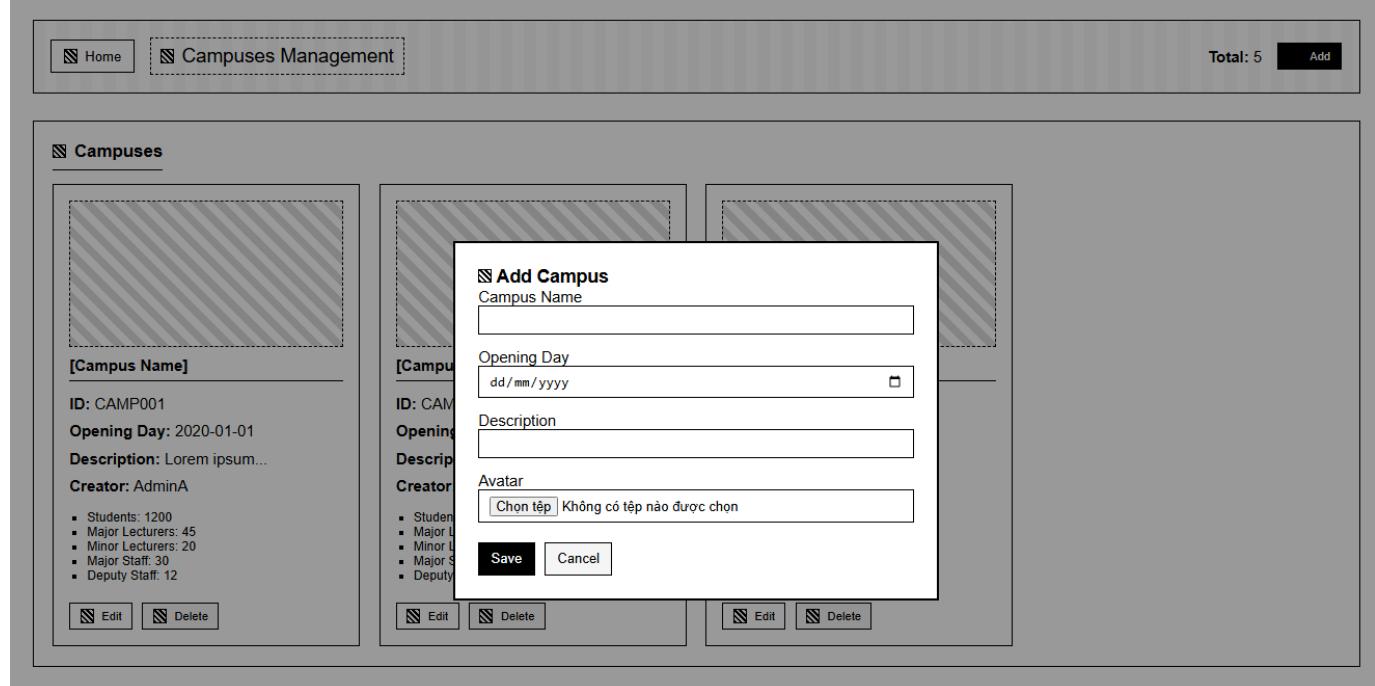
Wireframe and Prototype of Campuses Management page



This wireframe shows a list of campuses. Each entry includes a placeholder for the campus name, its ID, opening day, description, creator, and staff statistics. Below each entry are 'Edit' and 'Delete' buttons.

Campus Name	ID: CAMP001	Opening Day: 2020-01-01	Description: Lorem ipsum...	Creator: AdminA	Students: 1200 Major Lecturers: 45 Minor Lecturers: 20 Major Staff: 30 Deputy Staff: 12
[Campus Name]	ID: CAMP002	Opening Day: 2018-05-15	Description: Lorem ipsum...	Creator: AdminB	Students: 900 Major Lecturers: 38 Minor Lecturers: 18 Major Staff: 22 Deputy Staff: 9
[Campus Name]	ID: CAMP003	Opening Day: 2019-09-09	Description: Lorem ipsum...	Creator: AdminC	Students: 700 Major Lecturers: 25 Minor Lecturers: 12 Major Staff: 18 Deputy Staff: 5

Wireframe 21 Wireframe of Campuses Management page



This wireframe shows the same list of campuses as above, but with an 'Add Campus' modal window overlaid. The modal contains fields for Campus Name, Opening Day (dd/mm/yyyy), Description, and Avatar selection. It also includes 'Save' and 'Cancel' buttons.

Wireframe 22 Wireframe of Campuses Management page

Wireframe and Prototype of Email Templates Management page

<input type="button" value="Home"/> <input type="button" value="Email Templates Management"/> <input type="text" value="Page size"/> <input type="button" value="Apply"/> Total: 12 <input type="button" value="Add Template"/>					
ID	Type	Campus	Header	Banner	Actions
001	Welcome Email	HCM			<input type="button" value="Edit"/> <input type="button" value="Delete"/>
002	Password Reset	Hanoi		No Image	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
003	Enrollment Notice	N/A	No Image		<input type="button" value="Edit"/> <input type="button" value="Delete"/>
004	Event Reminder	Da Nang			<input type="button" value="Edit"/> <input type="button" value="Delete"/>
005	Newsletter	HCM	No Image	No Image	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
006	Application Confirmation	N/A		No Image	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
007	Offer Letter	Hanoi	No Image		<input type="button" value="Edit"/> <input type="button" value="Delete"/>
008	Payment Reminder	Da Nang		No Image	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

Wireframe 23 Wireframe and Prototype of Email Templates Management page

ID	Name	Type	Greeting	Header Image	Banner Image	Body	Actions
001	Welcome Email						Edit Delete
002	Password Reset				<input type="text" value="Chọn tệp"/> Không có tệp nào được chọn		Edit Delete
003	Enrollment Notice						Edit Delete
004	Event Reminder						Edit Delete
005	Newsletter	HCM	No Image	No Image			Edit Delete
006	Application Confirmation	N/A		No Image			Edit Delete
007	Offer Letter	Hanoi	No Image				Edit Delete
008	Payment Reminder	Da Nang		No Image			Edit Delete

[First](#) [Prev](#) [1](#) [2](#) [Next](#) [Last](#)

Wireframe 24 Wireframe and Prototype of Email Templates Management page

Wireframe and Prototype of Email Templates Edit page

- Error message example line 1
- Error message example line 2

Template Info

<input type="text" value="Type"/>	<input type="text" value="Greeting"/>
<input type="text" value="Welcome Email"/>	<input type="text" value="Dear [Name]"/>
<input type="text" value="Salutation"/>	
<input type="text" value="Best regards"/>	

Images

<input type="text" value="Current Header Image"/>	<input type="text" value="Current Banner Image"/>
	
<input type="text" value="Replace Header Image"/>	<input type="text" value="Replace Banner Image"/>
<input type="button" value="Chọn tệp"/> Không có tệp nào được chọn	<input type="button" value="Chọn tệp"/> Không có tệp nào được chọn

Content

<input type="text" value="Body"/>	
<input type="text" value=""/>	
<input type="text" value="CTA Link"/>	
<input type="text" value="https://example.com"/>	
<input type="text" value="Support"/>	
<input type="text" value="support@example.com"/>	
<input type="text" value="Copyright"/>	
<input type="text" value="Facebook"/>	
<input type="text" value="YouTube"/>	
<input type="text" value="TikTok"/>	
<input type="text" value=""/>	

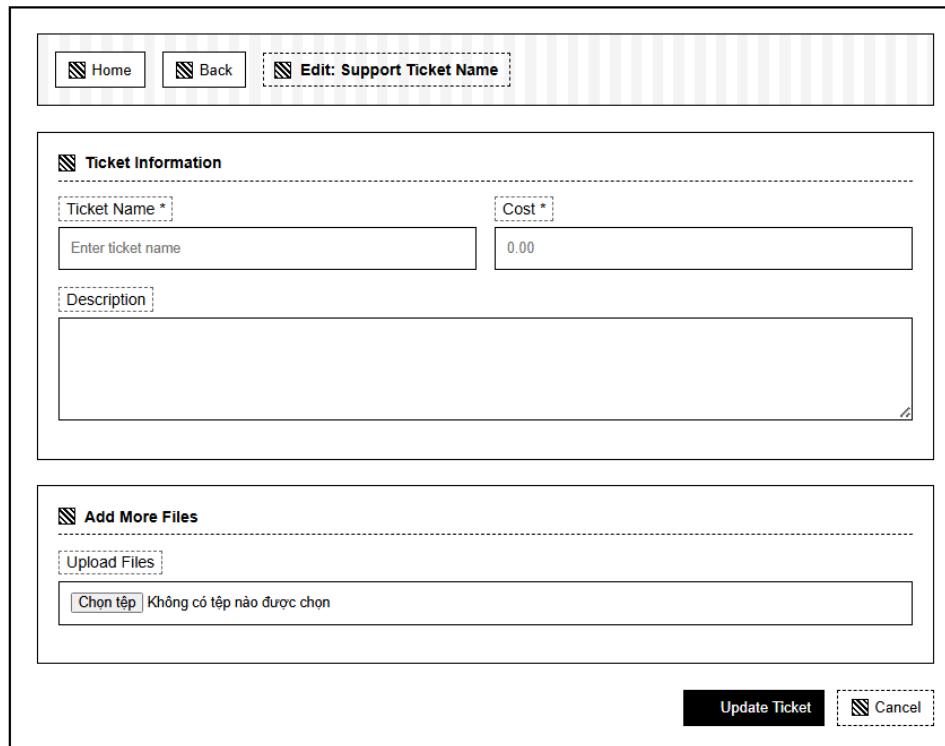
Wireframe 25 Wireframe and Prototype of Email Templates Edit page

Wireframe and Prototype of Support Tickets List page

 Home		 Support Tickets		Name <input type="button" value="▼"/>	Search...	 	Show	Apply	Total: 12	 Add Ticket
Success / Error messages placeholder										
ID	Name	Description	Cost	Creator	Created	Docs	Actions			
001	Transcript Request	Student requests official transcript...	\$10.00	Admin	12/01/2024					
002	ID Card Replacement	Lost ID, student needs new card...	\$5.00	System	10/01/2024					
002	ID Card Replacement	Lost ID, student needs new card...	\$5.00	System	10/01/2024					
002	ID Card Replacement	Lost ID, student needs new card...	\$5.00	System	10/01/2024					
002	ID Card Replacement	Lost ID, student needs new card...	\$5.00	System	10/01/2024					
<input type="button" value="First"/> <input type="button" value="Prev"/> <input type="button" value="1"/> <input type="button" value="2"/> <input type="button" value="Next"/> <input type="button" value="Last"/>										

Wireframe 26 Wireframe and Prototype of Support Tickets List page

Wireframe and Prototype of Support Tickets Edit page



The wireframe illustrates the layout of the 'Edit Support Ticket' page. At the top, there are navigation links: 'Home', 'Back', and a dashed box labeled 'Edit: Support Ticket Name'. The main content area is divided into two sections: 'Ticket Information' and 'Add More Files'. The 'Ticket Information' section contains fields for 'Ticket Name *' (with placeholder 'Enter ticket name') and 'Cost *' (with value '0.00'). Below these is a large 'Description' text area. The 'Add More Files' section includes a 'Upload Files' button and a text input field containing the Vietnamese message 'Chon tệp | Không có tệp nào được chọn'. At the bottom right are 'Update Ticket' and 'Cancel' buttons.

Wireframe 27 Wireframe and Prototype of Support Tickets Edit page

Wireframe and Prototype of Colleagues List page

		 Home		 Colleagues List		Search name/id		 Search	 Clear	Show	 Apply	Total: 108	Add Colleague
ID	Avatar	Full Name	Email	Phone	Birth Date	Gender	Campus	Actions					
CL001		John Doe	john@example.com	0123456789	12/03/1990	Male	Hanoi	 Msg	 Del				
CL002		Emily Carter	emily@example.com	0981234567	20/07/1993	Female	Saigon	 Msg	 Del				
CL003		Michael Brown	mike@example.com	0901231111	14/09/1988	Male	Danang	 Msg	 Del				
CL004		Sarah Johnson	sarah@example.com	0977774444	03/11/1995	Female	Hanoi	 Msg	 Del				
CL005		Daniel Smith	daniel@example.com	0832229999	29/01/1992	Male	Hue	 Msg	 Del				
CL006		Linda Nguyen	linda@example.com	0905558888	10/04/1994	Female	Saigon	 Msg	 Del				
CL007		Alex Turner	alex@example.com	0921113333	22/08/1985	Male	Danang	 Msg	 Del				
CL008		Jessica Lee	jessica@example.com	0966667777	16/12/1991	Female	Hanoi	 Msg	 Del				

Wireframe 28 Wireframe and Prototype of Colleagues List page

[Home](#)

[Your Colleagues](#)

ID	Avatar	Name
CL001		John
CL002		Emily
CL003		Michael
CL004		Sarah
CL005		Daniel
CL006		Linda
CL007		Alex
CL008		Jessica

Add New Colleague

Avatar

Select Avatar :

Colleague Information

Address Information

Total: 108
Add Colleague

Actions

Msg
Del

Wireframe 29 Wireframe and Prototype of Colleagues List page

Wireframe and Prototype of Admin Personal page

Avatar



[Error: Invalid file format / File too large]

Personal Information

First Name * <input type="text" value="Enter first name"/>	Last Name * <input type="text" value="Enter last name"/>
Email * <input type="text" value="admin@example.com"/>	Phone <input type="text" value="+84 ..."/>
Birth Date <input type="text" value="dd/mm/yyyy"/> <input type="button" value=""/>	Gender <input type="button" value="Select gender"/>

Address Information

Country <input type="text"/>	Province <input type="text"/>
City <input type="text"/>	District <input type="text"/>
Ward <input type="text"/>	Street/House <input type="text"/>
Postal Code <input type="text"/>	

Wireframe 30 Wireframe and Prototype of Admin Personal page

Avatar



[Error: Invalid file format / File too large]

Personal Information

First Name *	<input type="text" value="Enter first name"/>
Email *	<input type="text" value="admin@example.com"/>
Birth Date	<input type="text" value="dd/mm/yyyy"/>

Address Information

Country	<input type="text"/>
City	<input type="text"/>
Ward	<input type="text"/>
Postal Code	<input type="text"/>

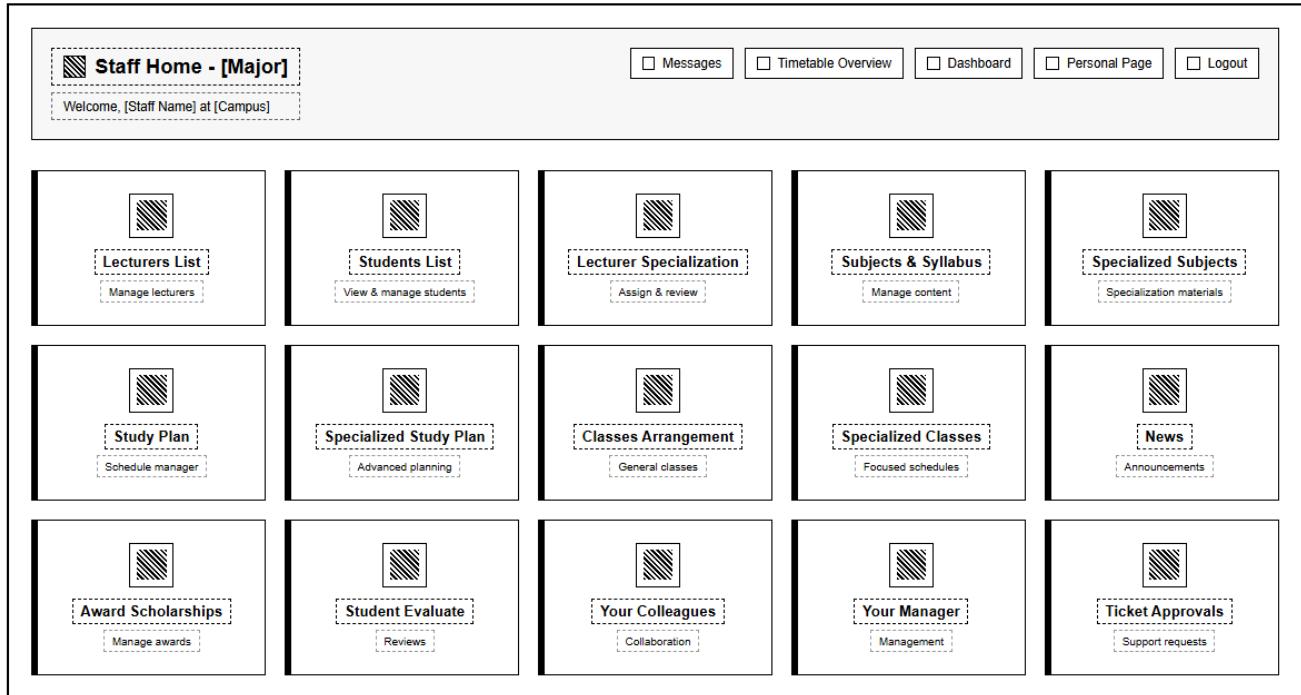
Change Password

[Error: Password must be at least 8 characters]

[Error: Passwords do not match]

Wireframe 31 Wireframe and Prototype of Admin Personal page

Wireframe and Prototype of Staff Home page



Wireframe 32 Wireframe and Prototype of Staff Home page

Wireframe and Prototype of Lecturer List page

Home		Your Lecturers List		Name	Search...	Search	10	Apply	Total: ###	+ Add Lecturer
ID	Avatar	Full Name	Email	Timetable		Specializations		Actions		
001		John Doe	john@example.com	View	View	Edit	Delete			
001		John Doe	john@example.com	View	View	Edit	Delete			
001		John Doe	john@example.com	View	View	Edit	Delete			
001		John Doe	john@example.com	View	View	Edit	Delete			
001		John Doe	john@example.com	View	View	Edit	Delete			
More rows...										
First Prev 1 2 ... Next Last										

Wireframe 33 Wireframe and Prototype of Lecturer List page

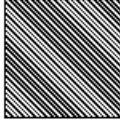
Home		Your Lecturers		Add New Lecturer								Total: ###		+ Add Lecturer	
ID	Avatar	Full Name	Email	First Name	Last Name	Email	Phone	Birth Date	Gender	Country	City	District	Street	Save	Cancel
001		John Doe	john@example.com	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	Save	Cancel
001		John Doe	john@example.com	View	View	Edit	Delete	View	View	Edit	Delete	View	Edit	Delete	
001		John Doe	john@example.com	View	View	Edit	Delete	View	View	Edit	Delete	View	Edit	Delete	
001		John Doe	john@example.com	View	View	Edit	Delete	View	View	Edit	Delete	View	Edit	Delete	
001		John Doe	john@example.com	View	View	Edit	Delete	View	View	Edit	Delete	View	Edit	Delete	
More rows...															
First Prev 1 2 ... Next Last															

Wireframe 34 Wireframe and Prototype of Lecturer List page

Wireframe and Prototype of Lecturer Edit page

Edit Lecturer

Avatar



Upload New Avatar (Optional)

Không có tệp nào được chọn

Lecturer Information

First Name	Last Name
<input type="text"/>	<input type="text"/>
Email	Phone Number
<input type="text"/>	<input type="text"/>
Birth Date	Gender
<input type="text"/> dd/mm/yyyy	<input type="button"/>

Address Information

Country	Province
<input type="text"/>	<input type="text"/>
City	District
<input type="text"/>	<input type="text"/>
Ward	Street
<input type="text"/>	<input type="text"/>
Postal Code	<input type="text"/>

Wireframe 35 Wireframe and Prototype of Lecturer Edit page

Wireframe and Prototype of Lecturer Specializations page

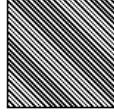
[] ID	[] Specialization Name	[] Created At
SP001	Software Engineering	2025-01-12 09:30
SP002	Artificial Intelligence	2025-02-10 10:21
SP003	Information Security	2025-01-28 14:05
SP004	Data Science	2025-02-02 11:10
SP005	Networking Technologies	2025-01-15 16:22
SP006	Embedded Systems	2025-02-20 09:45
SP007	Cloud Computing	2025-03-01 08:33
SP008	Blockchain Development	2025-03-08 13:14

Wireframe 36 Wireframe and Prototype of Lecturer Specializations page

Wireframe and Prototype of Students List page

Home		Your Students List		Name	Search...	Search	10	Apply	Total: ###	+ Add Student
ID	Avatar	Full Name	Email	Timetable	Learning	Transcript	Actions			
001		John Doe	john@example.com	View	View	View	Edit	Delete		
001		John Doe	john@example.com	View	View	View	Edit	Delete		
001		John Doe	john@example.com	View	View	View	Edit	Delete		
001		John Doe	john@example.com	View	View	View	Edit	Delete		
001		John Doe	john@example.com	View	View	View	Edit	Delete		
001		John Doe	john@example.com	View	View	View	Edit	Delete		
More rows...										
First Prev 1 2 ... Next Last										

Wireframe 37 Wireframe and Prototype of Students List page

Home	Your Students	
Add New Student		
Avatar (Optional)		
		
<input style="width: 100px; height: 20px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 10px; margin-bottom: 5px;" type="button" value="Chon tệp"/> Không có tệp nào được chọn		
Student Information		
First Name	Last Name	
<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px; margin-right: 5px;" type="button" value="dit"/> <input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px;" type="button" value="Delete"/>
Email	Phone Number	
<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px; margin-right: 5px;" type="button" value="dit"/> <input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px;" type="button" value="Delete"/>
Birth Date	Admission Year	
<input style="width: 100px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 100px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px; margin-right: 5px;" type="button" value="dit"/> <input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px;" type="button" value="Delete"/>
Gender	Curriculum	
<input style="width: 100px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="button"/>	<input style="width: 100px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="button"/>	<input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px; margin-right: 5px;" type="button" value="dit"/> <input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px;" type="button" value="Delete"/>
Narrow Specialization		
<input style="width: 100px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="button"/>		
Address (Optional)		
Country	Province	
<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px; margin-right: 5px;" type="button" value="dit"/> <input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px;" type="button" value="Delete"/>
City	District	
<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px; margin-right: 5px;" type="button" value="dit"/> <input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px;" type="button" value="Delete"/>
Ward	Street	
<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>	<input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px; margin-right: 5px;" type="button" value="dit"/> <input style="width: 20px; height: 20px; border: 1px solid #ccc; border-radius: 5px;" type="button" value="Delete"/>
Postal Code		
<input style="width: 100%; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="text"/>		
<input style="width: 100px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px; margin-right: 10px;" type="button" value="Save Student"/> <input style="width: 100px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 2px 5px;" type="button" value="Cancel"/>		

Wireframe 38 Wireframe and Prototype of Students List page

Wireframe of Students Edit page

Edit Student

[Alert message area]

Avatar



[] Upload New Avatar (Optional)

Student Information

First Name:	Last Name:
<input type="text"/>	<input type="text"/>
Email:	Phone Number:
<input type="text"/>	<input type="text"/>
Birth Date:	Admission Year:
<input type="text"/>	<input type="text"/>
Gender:	Curriculum:
<input type="text"/>	<input type="text"/>
Major Specialization:	
<input type="text"/>	

Address Information

Country:	Province:
<input type="text"/>	<input type="text"/>
City:	District:
<input type="text"/>	<input type="text"/>
Ward:	Street:
<input type="text"/>	<input type="text"/>
Postal Code:	
<input type="text"/>	

Linked Parents

Avatar	Full Name	Email	Relationship	Support Phone	Action
<input alt="Placeholder for parent avatar" type="image"/>	[Name]	[Email]	[Relationship]	[Phone]	[Remove]

Wireframe 39 Wireframe of Students Edit page

Edit Student

[Alert message area]

Avatar

Upload New Avatar (Optional)

Student Information

First Name:	Last Name:
Email:	Phone Number:
Birth Date:	Admission Year:
Gender:	
Narrow Specialization:	

Address Information

Country:	
City:	District:
Ward:	Street:
Postal Code:	

Linked Parents

Avatar	Full Name	Email	Relationship	Support Phone	Action
	[Name]	[Email]	[Relationship]	[Phone]	[Remove]

Wireframe 40 Wireframe of Students Edit page

Wireframe of Students View Timetable page

Timetable							
Year	Week	Today					
[Slot 1] [07:30 – 09:30]	Room: [R101] Class: [OOP123] Creator: [John Doe] Created: [12/01] Detail ATTENDED	Empty	Class: [PRU211] Detail Not Marked	Empty	Empty	Empty	Empty
[Slot 2] [09:45 – 11:45]	Empty	Class: [DBI202] Detail	Empty	Empty	Room: [LAB03] Detail	Empty	Empty
[Slot 3] [13:00 – 15:00]	Class: [MATH301] Detail	Empty	Empty	Room: [R205] Detail	Empty	Empty	Empty
[Slot 4] [15:15 – 17:15]	Empty	Empty	Class: [PRM391] Detail	Empty	Empty	Room: [R104] Detail	Empty
[Slot 5] [18:00 – 20:00]	Class: [ENGLISH01] Detail	Empty	Empty	Empty	Room: [R303] Detail	Empty	Empty

Wireframe 41 Wireframe of Students View Timetable page

Wireframe of Students View Academic Transcript page

		Academic Transcript						
<input type="checkbox"/> Major Subjects								
ID	Name	Semester	Score	C1	C2	C3	Grade	
SE101	Intro to Software	1	8.2	7.5	8.0	9.0	A	
PRF192	Programming Fundamentals	1	7.9	7.0	8.5	7.5	B+	
MAE101	Applied Math	1	8.4	8.0	8.5	7.0	C+	
CSI101	Computer Systems	2	8.7	9.0	8.5	8.0	A	
OOP212	Object-Oriented Programming	2	9.0	8.5	9.0	9.5	A+	
WED201	Web Development	2	8.1	7.5	8.0	9.0	B+	
DBI202	Databases	3	7.2	7.0	7.5	7.0	B	
OSG202	Operating Systems	3	6.8	6.0	7.0	7.5	C+	
NWC203	Computer Networks	3	7.7	7.0	8.0	8.0	B	
SWE300	Software Engineering	4	8.9	9.0	8.5	9.0	A	

		Minor Subjects						
<input type="checkbox"/> Minor Subjects								
ID	Name	Semester	Score	C1	C2	C3	Grade	
MIN101	Introduction to Marketing	2	7.5	7.0	8.0	7.5	B	
MIN102	Basic Economics	2	6.8	6.5	7.0	7.0	C+	
MIN103	Teamwork Skills	1	8.4	8.0	8.5	9.0	A	
MIN201	Public Speaking	3	7.9	8.0	7.5	8.0	B+	
MIN202	Creative Thinking	3	8.0	7.5	8.5	8.5	B+	
MIN301	Business Basics	4	7.2	7.0	7.0	7.5	B	
MIN302	Economics II	4	6.5	6.0	7.0	6.5	C	

		Specialized Subjects						
<input type="checkbox"/> Specialized Subjects								
ID	Name	Semester	Specialization	Score	C1	C2	C3	Grade
SPC301	Machine Learning	5	AI	8.8	9.0	8.5	9.0	A
SPC302	Deep Learning	5	AI	8.4	8.0	8.5	8.5	A-
SPC303	Data Mining	5	AI	7.8	7.5	8.0	8.0	B+
SPC321	Cloud Architecture	6	Cloud	7.9	8.0	7.5	8.0	B+
SPC322	Cyber Security	6	Security	7.1	7.0	7.0	7.5	B
SPC323	***** Testing	6	Security	6.9	6.0	7.0	7.5	C+

Wireframe 42 Wireframe of Students View Academic Transcript page

Wireframe of Students View Academic Transcript page

		[Home]	[Students List]	Learning Process			Show: <input type="text" value="10"/>	Apply	Total Classes: [42]
Class ID	Class Name	Subject	Session	Created At	Actions				
C101	Class Alpha	Math	10	03/01/2025	[View Scores]				
C102	Class Beta	Physics	15	05/01/2025	[View Scores]				
C103	Class Gamma	Chemistry	12	10/01/2025	[View Scores]				
C104	Class Delta	OOP	14	12/01/2025	[View Scores]				
C105	Class Omega	SE	16	14/01/2025	[View Scores]				
C106	Class Tiger	DSA	12	15/01/2025	[View Scores]				
C107	Class Eagle	Network	13	18/01/2025	[View Scores]				
C108	Class Lion	Database	14	20/01/2025	[View Scores]				
C109	Class Hawk	Testing	11	21/01/2025	[View Scores]				
C110	Class Storm	Cloud	15	22/01/2025	[View Scores]				
C111	Class Wind	Security	9	23/01/2025	[View Scores]				
C112	Class Water	AI Basics	8	24/01/2025	[View Scores]				
C113	Class Fire	ML	10	25/01/2025	[View Scores]				
C114	Class Stone	OS	12	27/01/2025	[View Scores]				
C115	Class Sky	Web Dev	14	28/01/2025	[View Scores]				

First Last Go to:

Wireframe 43 Wireframe of Students View Academic Transcript page

Wireframe of Students View Detail Scores page

		[Home]	[Students List]	[Back to Classes]	Scores — [Class ABC123]			
Subject Name	P1	Mid	Prac	Final	Grade	Creator	Created At	Status
Mathematics	8.50	7.80	9.20	8.70	B+	U001	10/02/2025 09:32	<input type="button" value="Pass"/>
Data Structures	7.00	6.50	8.00	7.30	C	U003	11/02/2025 10:14	<input type="button" value="Pass"/>
Operating Systems	5.50	4.20	6.00	5.10	D	U004	12/02/2025 13:40	<input type="button" value="Pass"/>
Artificial Intelligence	3.00	4.00	3.80	3.50	F	U002	12/02/2025 14:10	<input type="button" value="Fail"/>
Software Engineering	9.00	8.70	9.50	9.20	A	U005	13/02/2025 08:45	<input type="button" value="Pass"/>
Database Systems	6.80	7.10	7.50	7.20	C+	U001	13/02/2025 15:22	<input type="button" value="Pass"/>
Machine Learning	4.00	5.00	4.20	4.60	F	U003	14/02/2025 09:12	<input type="button" value="Fail"/>

Wireframe 44 Wireframe of Students View Detail Scores page

Wireframe of Subjects List page

<input type="button" value="Home"/> <input type="text"/> <input type="button" value="Name"/> <input type="text" value="Search..."/> <input type="button" value="Search"/> <input type="button" value="Clear"/> Show: <input type="text"/> <input type="button" value="Apply"/> <input type="button" value="Add Subject"/>						
ID	Name	Semester	Curriculum	Creator	Actions	Syllabus
SUB001	Subject Name	1	Curriculum A	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
<input type="button" value="First"/> <input type="button" value="Prev"/> <input type="button" value="1"/> <input type="button" value="2"/> <input type="button" value="Next"/> <input type="button" value="Last"/>						

Wireframe 45 Wireframe of Subjects List page

<input type="button" value="Home"/> <input type="text"/> <input type="button" value="Add Subject"/>						
ID	Name	Add Subject			Actions	Syllabus
SUB001	Subject Name	<input type="text" value="Subject Name"/> <input type="text" value="Semester"/> <input type="text" value="Curriculum"/> <input type="button" value="Curriculum A"/>			<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject				<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject				<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	<input type="button" value="Save"/> <input type="button" value="Cancel"/>			<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
SUB002	Another Subject	2	Curriculum B	Creator Name	<input type="button" value="View"/>	<input type="button" value="View"/>
<input type="button" value="First"/> <input type="button" value="Prev"/> <input type="button" value="1"/> <input type="button" value="2"/> <input type="button" value="Next"/> <input type="button" value="Last"/>						

Wireframe 46 Wireframe of Subjects List page

Wireframe of Syllabuses List page

		Home	Back to Subjects	Show:	Apply			Add Syllabus
ID	Syllabus Name	File Type	Creator	Actions		View File		
SYL001	Syllabus File 1	PDF	User 1			<input type="button" value="View"/>		
SYL002	Syllabus File 2	PDF	User 2			<input type="button" value="View"/>		
SYL003	Syllabus File 3	PDF	User 3			<input type="button" value="View"/>		
SYL004	Syllabus File 4	PDF	User 4			<input type="button" value="View"/>		
SYL005	Syllabus File 5	PDF	User 5			<input type="button" value="View"/>		
SYL006	Syllabus File 6	PDF	User 6			<input type="button" value="View"/>		
SYL007	Syllabus File 7	PDF	User 7			<input type="button" value="View"/>		
SYL008	Syllabus File 8	PDF	User 8			<input type="button" value="View"/>		
SYL009	Syllabus File 9	PDF	User 9			<input type="button" value="View"/>		
SYL010	Syllabus File 10	PDF	User 10			<input type="button" value="View"/>		
SYL011	Syllabus File 11	PDF	User 11			<input type="button" value="View"/>		
SYL012	Syllabus File 12	PDF	User 12			<input type="button" value="View"/>		
SYL013	Syllabus File 13	PDF	User 13			<input type="button" value="View"/>		
SYL014	Syllabus File 14	PDF	User 14			<input type="button" value="View"/>		
SYL015	Syllabus File 15	PDF	User 15			<input type="button" value="View"/>		
SYL016	Syllabus File 16	PDF	User 16			<input type="button" value="View"/>		
SYL017	Syllabus File 17	PDF	User 17			<input type="button" value="View"/>		
SYL018	Syllabus File 18	PDF	User 18			<input type="button" value="View"/>		
SYL019	Syllabus File 19	PDF	User 19			<input type="button" value="View"/>		
SYL020	Syllabus File 20	PDF	User 20			<input type="button" value="View"/>		

Wireframe 47 Wireframe of Syllabuses List page

		Home	Back to Subjects	Add Syllabus		View File		
ID	Syllabus Name							
SYL001	Syllabus Fil							
SYL002	Syllabus Fil							
SYL003	Syllabus Fil							
SYL004	Syllabus Fil							
SYL005	Syllabus Fil							
SYL006	Syllabus Fil							

Wireframe 48 Wireframe of Syllabuses List page

Wireframe of Study Plan page

Home All Admisor All Curricular Filter					
Success or Error Message Placeholder					
ID	Subject Name	Semester	Creator	Major	Action
SUB001	Subject Name 1	2	Creator 1	Major 2	Assign Members
SUB002	Subject Name 2	3	Creator 2	Major 3	Assign Members
SUB003	Subject Name 3	4	Creator 3	Major 4	Assign Members
SUB004	Subject Name 4	5	Creator 4	Major 5	Assign Members
SUB005	Subject Name 5	6	Creator 5	Major 1	Assign Members
SUB006	Subject Name 6	7	Creator 6	Major 2	Assign Members
SUB007	Subject Name 7	8	Creator 7	Major 3	Assign Members
SUB008	Subject Name 8	1	Creator 8	Major 4	Assign Members
SUB009	Subject Name 9	2	Creator 9	Major 5	Assign Members
SUB010	Subject Name 10	3	Creator 10	Major 1	Assign Members
No subjects available for the selected criteria.					

Wireframe 49 Wireframe of Study Plan page

Wireframe of Assign Members page

		Assign Members to: SUBJECT NAME																																							
<input type="button" value="Home"/>	<input type="button" value="Back"/>																																								
Assigned Students <table border="1"> <thead> <tr> <th><input type="checkbox"/></th> <th>Avatar</th> <th>Student ID</th> <th>Name</th> <th>Reason</th> <th>Assigned By</th> </tr> </thead> <tbody> <tr> <td><input type="checkbox"/></td> <td></td> <td>S0001</td> <td>Student Name 1</td> <td>Reason 1</td> <td>Staff 1</td> </tr> <tr> <td><input type="checkbox"/></td> <td></td> <td>S0002</td> <td>Student Name 2</td> <td>Reason 2</td> <td>Staff 2</td> </tr> <tr> <td><input type="checkbox"/></td> <td></td> <td>S0003</td> <td>Student Name 3</td> <td>Reason 3</td> <td>Staff 3</td> </tr> <tr> <td><input type="checkbox"/></td> <td></td> <td>S0004</td> <td>Student Name 4</td> <td>Reason 4</td> <td>Staff 4</td> </tr> <tr> <td><input type="checkbox"/></td> <td></td> <td>S0005</td> <td>Student Name 5</td> <td>Reason 5</td> <td>Staff 5</td> </tr> </tbody> </table> <input type="button" value="Remove Selected"/>						<input type="checkbox"/>	Avatar	Student ID	Name	Reason	Assigned By	<input type="checkbox"/>		S0001	Student Name 1	Reason 1	Staff 1	<input type="checkbox"/>		S0002	Student Name 2	Reason 2	Staff 2	<input type="checkbox"/>		S0003	Student Name 3	Reason 3	Staff 3	<input type="checkbox"/>		S0004	Student Name 4	Reason 4	Staff 4	<input type="checkbox"/>		S0005	Student Name 5	Reason 5	Staff 5
<input type="checkbox"/>	Avatar	Student ID	Name	Reason	Assigned By																																				
<input type="checkbox"/>		S0001	Student Name 1	Reason 1	Staff 1																																				
<input type="checkbox"/>		S0002	Student Name 2	Reason 2	Staff 2																																				
<input type="checkbox"/>		S0003	Student Name 3	Reason 3	Staff 3																																				
<input type="checkbox"/>		S0004	Student Name 4	Reason 4	Staff 4																																				
<input type="checkbox"/>		S0005	Student Name 5	Reason 5	Staff 5																																				
Students Not Assigned <table border="1"> <thead> <tr> <th><input type="checkbox"/></th> <th>Avatar</th> <th>Student ID</th> <th colspan="3">Name</th> </tr> </thead> <tbody> <tr> <td><input type="checkbox"/></td> <td></td> <td>U0001</td> <td colspan="3">Unassigned Student 1</td> </tr> <tr> <td><input type="checkbox"/></td> <td></td> <td>U0002</td> <td colspan="3">Unassigned Student 2</td> </tr> <tr> <td><input type="checkbox"/></td> <td></td> <td>U0003</td> <td colspan="3">Unassigned Student 3</td> </tr> <tr> <td><input type="checkbox"/></td> <td></td> <td>U0004</td> <td colspan="3">Unassigned Student 4</td> </tr> <tr> <td><input type="checkbox"/></td> <td></td> <td>U0005</td> <td colspan="3">Unassigned Student 5</td> </tr> </tbody> </table> <input type="button" value="Assign Selected"/>						<input type="checkbox"/>	Avatar	Student ID	Name			<input type="checkbox"/>		U0001	Unassigned Student 1			<input type="checkbox"/>		U0002	Unassigned Student 2			<input type="checkbox"/>		U0003	Unassigned Student 3			<input type="checkbox"/>		U0004	Unassigned Student 4			<input type="checkbox"/>		U0005	Unassigned Student 5		
<input type="checkbox"/>	Avatar	Student ID	Name																																						
<input type="checkbox"/>		U0001	Unassigned Student 1																																						
<input type="checkbox"/>		U0002	Unassigned Student 2																																						
<input type="checkbox"/>		U0003	Unassigned Student 3																																						
<input type="checkbox"/>		U0004	Unassigned Student 4																																						
<input type="checkbox"/>		U0005	Unassigned Student 5																																						

Wireframe 50 Wireframe of Assign Members page

Wireframe of Classes List page

Home Classes List – Campus ABC		Name	Search...	Search	Show: 10	Apply	Total Classes: 120	+ Add Class
ID	Class Name	Subject	Total Slots	Room	Timetable	Members	Transcript	Actions
CL001	Class Name 1	Subject 1	11		View	Open	Enter	Edit Delete
CL002	Class Name 2	Subject 2	12		View	Open	Enter	Edit Delete
CL003	Class Name 3	Subject 3	13		View	Open	Enter	Edit Delete
CL004	Class Name 4	Subject 4	14		View	Open	Enter	Edit Delete
CL005	Class Name 5	Subject 5	15		View	Open	Enter	Edit Delete
CL006	Class Name 6	Subject 6	16		View	Open	Enter	Edit Delete
CL007	Class Name 7	Subject 7	17		View	Open	Enter	Edit Delete
CL008	Class Name 8	Subject 8	18		View	Open	Enter	Edit Delete
CL009	Class Name 9	Subject 9	19		View	Open	Enter	Edit Delete
CL010	Class Name 10	Subject 10	20		View	Open	Enter	Edit Delete
CL011	Class Name 11	Subject 11	21		View	Open	Enter	Edit Delete
CL012	Class Name 12	Subject 12	22		View	Open	Enter	Edit Delete
CL013	Class Name 13	Subject 13	23		View	Open	Enter	Edit Delete
CL014	Class Name 14	Subject 14	24		View	Open	Enter	Edit Delete
CL015	Class Name 15	Subject 15	25		View	Open	Enter	Edit Delete
CL016	Class Name 16	Subject 16	26		View	Open	Enter	Edit Delete
CL017	Class Name 17	Subject 17	27		View	Open	Enter	Edit Delete
CL018	Class Name 18	Subject 18	28		View	Open	Enter	Edit Delete
CL019	Class Name 19	Subject 19	29		View	Open	Enter	Edit Delete
CL020	Class Name 20	Subject 20	30		View	Open	Enter	Edit Delete

[First](#) [Prev](#) [1](#) [2](#) [3](#) [Next](#) [Last](#)

Wireframe 51 Wireframe of Classes List page

Classes List – Campus ABC				Add New Class				Actions			
ID	Class Name	Subject	Total	Class Information				Members	Transcript	Actions	
CL001	Class Name 1	Subject 1	1	Class Name:	<input type="text"/>	<input type="button" value="Save"/>	<input type="button" value="Cancel"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
CL002	Class Name 2	Subject 2	1	Total Slots:	<input type="text"/>	<input type="button" value="Save"/>	<input type="button" value="Cancel"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
CL003	Class Name 3	Subject 3	1	Subject:	<input type="text" value="Select subject..."/>	<input type="button" value="Save"/>	<input type="button" value="Cancel"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
CL004	Class Name 4	Subject 4	1	Semester:	<input type="text" value="Semester 1"/>	<input type="button" value="Save"/>	<input type="button" value="Cancel"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
CL005	Class Name 5	Subject 5	1					<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
CL006	Class Name 6	Subject 6	16	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL007	Class Name 7	Subject 7	17	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL008	Class Name 8	Subject 8	18	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL009	Class Name 9	Subject 9	19	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL010	Class Name 10	Subject 10	20	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL011	Class Name 11	Subject 11	21	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL012	Class Name 12	Subject 12	22	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL013	Class Name 13	Subject 13	23	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL014	Class Name 14	Subject 14	24	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL015	Class Name 15	Subject 15	25	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL016	Class Name 16	Subject 16	26	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL017	Class Name 17	Subject 17	27	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL018	Class Name 18	Subject 18	28	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL019	Class Name 19	Subject 19	29	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
CL020	Class Name 20	Subject 20	30	<input type="button" value="View"/>	<input type="button" value="Open"/>	<input type="button" value="Enter"/>	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>			
<input type="button" value="First"/>	<input type="button" value="Prev"/>	<input type="button" value="1"/>	<input type="button" value="2"/>	<input type="button" value="3"/>	<input type="button" value="Next"/>	<input type="button" value="Last"/>	Show: <input type="text" value="10"/>	<input type="button" value="Apply"/>	Total Classes: 120	<input type="button" value="+ Add Class"/>	

Wireframe 52 Wireframe of Classes List page

Wireframe of Major Timetable page

Arrange Timetable — Class XYZ — Campus ABC

Year: 2023 Week: Week 1 Apply

Save Back

Required Slots: 20 Booked: 12 Remaining: 8 This Week Booked: 3

Success message (Wireframe)

Slot (Time)	Mon (01/01)	Tue (02/01)	Wed (03/01)	Thu (04/01)	Fri (05/01)	Sat (06/01)	Sun (07/01)
Slot 1 08:00 – 10:00	Room A101 Creator: John Doe Date: 01/01 Detail Delete	Select Room 2 rooms available	No room	Select	No room	Select	Past week
Slot 2 10:00 – 12:00	Select Available	No room	Select	Booked Detail Delete	Past	Select	No room
Slot 3 11:00 – 13:00	Select Available	No room	Select	Booked Detail Delete	Past	Select	No room
Slot 4 12:00 – 14:00	Select Available	No room	Select	Booked Detail Delete	Past	Select	No room
Slot 5 13:00 – 15:00	Select Available	No room	Select	Booked Detail Delete	Past	Select	No room
Slot 6 14:00 – 16:00	Select Available	No room	Select	Booked Detail Delete	Past	Select	No room

Wireframe 53 Wireframe of Major Timetable page

Wireframe of Major Timetable Detail page

Schedule Detail					Back
Review class info, lecturers & student attendance					
Success: Attendance saved					
CLASS INFORMATION					
CLASS [Class Name]	SUBJECT [Subject Name]	SLOT Slot 1 (08:00 - 10:00)	DATE 01/02/2025 (Mon)	WEEK Week 5 / 2025	
ROOM A101	CREATED BY Staff ABC	ATTENDANCE TIME Not taken			
LECTURERS					
Avatar [img]	ID L001	Full Name John Doe	Email johndoe@example.com	Phone 0123456789	
No lecturers assigned					
STUDENT ATTENDANCE					
#	Avatar [img]	Student ID S001	Full Name Nguyen Van A	Status <input type="radio"/> Present <input type="radio"/> Absent	Note Note...
2	[img]	S002	Tran Thi B	<input type="radio"/> Present <input type="radio"/> Absent	Note... Note...
Save Attendance All Present All Absent					

Wireframe 54 Wireframe of Major Timetable Detail page

Wireframe of Member Arrangement page

Member Arrangement – [Class Name]																			
STUDENTS IN CLASS																			
<table border="1"> <thead> <tr> <th>Select</th> <th>Avatar</th> <th>ID</th> <th>Name</th> <th>Email</th> </tr> </thead> <tbody> <tr> <td><input type="checkbox"/></td> <td></td> <td>S001</td> <td>Nguyen Van A</td> <td>a@example.com</td> </tr> <tr> <td colspan="5">No students in this class</td> </tr> </tbody> </table>					Select	Avatar	ID	Name	Email	<input type="checkbox"/>		S001	Nguyen Van A	a@example.com	No students in this class				
Select	Avatar	ID	Name	Email															
<input type="checkbox"/>		S001	Nguyen Van A	a@example.com															
No students in this class																			
<input type="button" value="Remove Selected"/>																			
LECTURERS IN CLASS																			
<table border="1"> <thead> <tr> <th>Select</th> <th>Avatar</th> <th>ID</th> <th>Name</th> <th>Email</th> </tr> </thead> <tbody> <tr> <td><input type="checkbox"/></td> <td></td> <td>L001</td> <td>Teacher A</td> <td>teacherA@example.com</td> </tr> <tr> <td colspan="5">No lecturers in this class</td> </tr> </tbody> </table>					Select	Avatar	ID	Name	Email	<input type="checkbox"/>		L001	Teacher A	teacherA@example.com	No lecturers in this class				
Select	Avatar	ID	Name	Email															
<input type="checkbox"/>		L001	Teacher A	teacherA@example.com															
No lecturers in this class																			
<input type="button" value="Remove Selected"/>																			
LECTURERS NOT IN CLASS																			
<table border="1"> <thead> <tr> <th>Select</th> <th>Avatar</th> <th>ID</th> <th>Name</th> <th>Email</th> </tr> </thead> <tbody> <tr> <td><input type="checkbox"/></td> <td></td> <td>L004</td> <td>Teacher B</td> <td>b@example.com</td> </tr> <tr> <td colspan="5">No lecturers available</td> </tr> </tbody> </table>					Select	Avatar	ID	Name	Email	<input type="checkbox"/>		L004	Teacher B	b@example.com	No lecturers available				
Select	Avatar	ID	Name	Email															
<input type="checkbox"/>		L004	Teacher B	b@example.com															
No lecturers available																			
<input type="button" value="Add Selected"/>																			
STUDENTS WITHOUT ENOUGH BALANCE																			
<table border="1"> <thead> <tr> <th>Avatar</th> <th>ID</th> <th>Name</th> <th>Email</th> </tr> </thead> <tbody> <tr> <td colspan="4">All students cleared</td> </tr> </tbody> </table>					Avatar	ID	Name	Email	All students cleared										
Avatar	ID	Name	Email																
All students cleared																			
STUDENTS WITH ENOUGH MONEY																			
<table border="1"> <thead> <tr> <th>Select</th> <th>Avatar</th> <th>ID</th> <th>Name</th> <th>Email</th> </tr> </thead> <tbody> <tr> <td><input type="checkbox"/></td> <td></td> <td>S021</td> <td>Tran B</td> <td>b@example.com</td> </tr> <tr> <td colspan="5">No students available</td> </tr> </tbody> </table>					Select	Avatar	ID	Name	Email	<input type="checkbox"/>		S021	Tran B	b@example.com	No students available				
Select	Avatar	ID	Name	Email															
<input type="checkbox"/>		S021	Tran B	b@example.com															
No students available																			

Wireframe 55 Wireframe of Member Arrangement page

Wireframe of Award Scholarships page

Home	Year: <select>2023</select>	Total Awarded: 12																																										
Award New Scholarship <hr/> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 45%; border-bottom: 1px dashed black;">Student ID</td> <td style="width: 5%;"></td> <td style="width: 50%; border-bottom: 1px dashed black;">Select Scholarship</td> </tr> <tr> <td><input type="text" value="Enter Student ID"/></td> <td></td> <td><input type="text" value="Scholarship A"/></td> </tr> <tr> <td colspan="3" style="text-align: center;"><input type="button" value="Submit"/></td> </tr> </table>			Student ID		Select Scholarship	<input type="text" value="Enter Student ID"/>		<input type="text" value="Scholarship A"/>	<input type="button" value="Submit"/>																																			
Student ID		Select Scholarship																																										
<input type="text" value="Enter Student ID"/>		<input type="text" value="Scholarship A"/>																																										
<input type="button" value="Submit"/>																																												
Students Awarded Scholarships <hr/> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 15%;">Student ID</th> <th style="width: 20%;">Full Name</th> <th style="width: 15%;">Scholarship ID</th> <th style="width: 20%;">Scholarship Name</th> <th style="width: 15%;">Award Date</th> <th style="width: 15%;">Percentage</th> </tr> </thead> <tbody> <tr><td>S001</td><td>Nguyen Van A</td><td>SC01</td><td>Top Scholarship</td><td>01/01/2024</td><td>50%</td></tr> <tr> <td colspan="6" style="text-align: center; font-size: small;">No scholarships awarded for this year</td> </tr> </tbody> </table>			Student ID	Full Name	Scholarship ID	Scholarship Name	Award Date	Percentage	S001	Nguyen Van A	SC01	Top Scholarship	01/01/2024	50%	S001	Nguyen Van A	SC01	Top Scholarship	01/01/2024	50%	S001	Nguyen Van A	SC01	Top Scholarship	01/01/2024	50%	S001	Nguyen Van A	SC01	Top Scholarship	01/01/2024	50%	S001	Nguyen Van A	SC01	Top Scholarship	01/01/2024	50%	No scholarships awarded for this year					
Student ID	Full Name	Scholarship ID	Scholarship Name	Award Date	Percentage																																							
S001	Nguyen Van A	SC01	Top Scholarship	01/01/2024	50%																																							
S001	Nguyen Van A	SC01	Top Scholarship	01/01/2024	50%																																							
S001	Nguyen Van A	SC01	Top Scholarship	01/01/2024	50%																																							
S001	Nguyen Van A	SC01	Top Scholarship	01/01/2024	50%																																							
S001	Nguyen Van A	SC01	Top Scholarship	01/01/2024	50%																																							
No scholarships awarded for this year																																												

Wireframe 56 Wireframe of Award Scholarships page

Wireframe of Student Evaluate Major List page

[Home](#) **Student Evaluations - Major Lecturers** Total: 23 evaluations

 **Nguyen Van A**
evaluated *Lecturer Name* • Class: SE1801 • 20/03/2024 09:45
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer tincidunt velit non urna gravida cursus.

 **Nguyen Van A**
evaluated *Lecturer Name* • Class: SE1801 • 20/03/2024 09:45
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer tincidunt velit non urna gravida cursus.

 **Nguyen Van A**
evaluated *Lecturer Name* • Class: SE1801 • 20/03/2024 09:45
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer tincidunt velit non urna gravida cursus.

 **Nguyen Van A**
evaluated *Lecturer Name* • Class: SE1801 • 20/03/2024 09:45
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer tincidunt velit non urna gravida cursus.

No student evaluations yet.

Wireframe 57 Wireframe of Student Evaluate Major List page

Wireframe and Prototype of News Edit page

[← Back to List](#) **EDIT NEWS (Wireframe)**

Flash message placeholder (success/error)

News Details

Title	Content
Error message placeholder	Error message placeholder

Add More Attachments:

Không có tệp nào được chọn

Existing files are kept.

Current Attachments

Document 1.pdf	<input type="button" value="Download"/>
Document 2.docx	<input type="button" value="Download"/>

Wireframe 58 Wireframe and Prototype of News Edit page

Wireframe and Prototype of Ticket Approval page

[Home](#) [Ticket Request Review \(Wireframe\)](#) [Back to List](#)

Request Information

Request ID	Requester Name
Ticket Package	Created At
Status	

Request Message / Description

Decision

Staff Notes

[Reject](#) [Approve](#)

Wireframe 59 Wireframe and Prototype of Ticket Approval page

Wireframe and Prototype of News List page

Home
NEWS LIST (Wireframe)
Search by Title
Search...
Search
Clear
Show:
Apply
Total:
+ Add News

ID	Title	Content (short)	Date	Author	Actions
001	News title example	Short content preview...	12/03/2024	John Doe	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
002	Another news item	Short content preview...	13/03/2024	Jane Doe	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
002	Another news item	Short content preview...	13/03/2024	Jane Doe	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
002	Another news item	Short content preview...	13/03/2024	Jane Doe	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
002	Another news item	Short content preview...	13/03/2024	Jane Doe	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

First Prev 1 2 3 Next Last Go to page:

Wireframe 60 Wireframe and Prototype of News List page

Home

ID

001

002

002

002

Add News (Wireframe)

Title

Content

Attachments Không có tệp nào được chọn

Wireframe 61 Wireframe and Prototype of News List page

Wireframe and Prototype of All Timetable page

ALL SCHEDULES — Class Name Placeholder						
No	Week / Year	Day	Room	Slot (Time)	Creator	Action
1	12 / 2025	Monday	Room A101	Slot 1 (08:00 - 10:00)	John Doe	<button>Delete</button>
1	12 / 2025	Monday	Room A101	Slot 1 (08:00 - 10:00)	John Doe	<button>Delete</button>
1	12 / 2025	Monday	Room A101	Slot 1 (08:00 - 10:00)	John Doe	<button>Delete</button>
1	12 / 2025	Monday	Room A101	Slot 1 (08:00 - 10:00)	John Doe	<button>Delete</button>
1	12 / 2025	Monday	Room A101	Slot 1 (08:00 - 10:00)	John Doe	<button>Delete</button>
1	12 / 2025	Monday	Room A101	Slot 1 (08:00 - 10:00)	John Doe	<button>Delete</button>
1	12 / 2025	Monday	Room A101	Slot 1 (08:00 - 10:00)	John Doe	<button>Delete</button>
No schedules found.						

Wireframe 62 Wireframe and Prototype of All Timetable page

Wireframe and Prototype of Staff Personal page

Staff Profile

Avatar



Personal Information

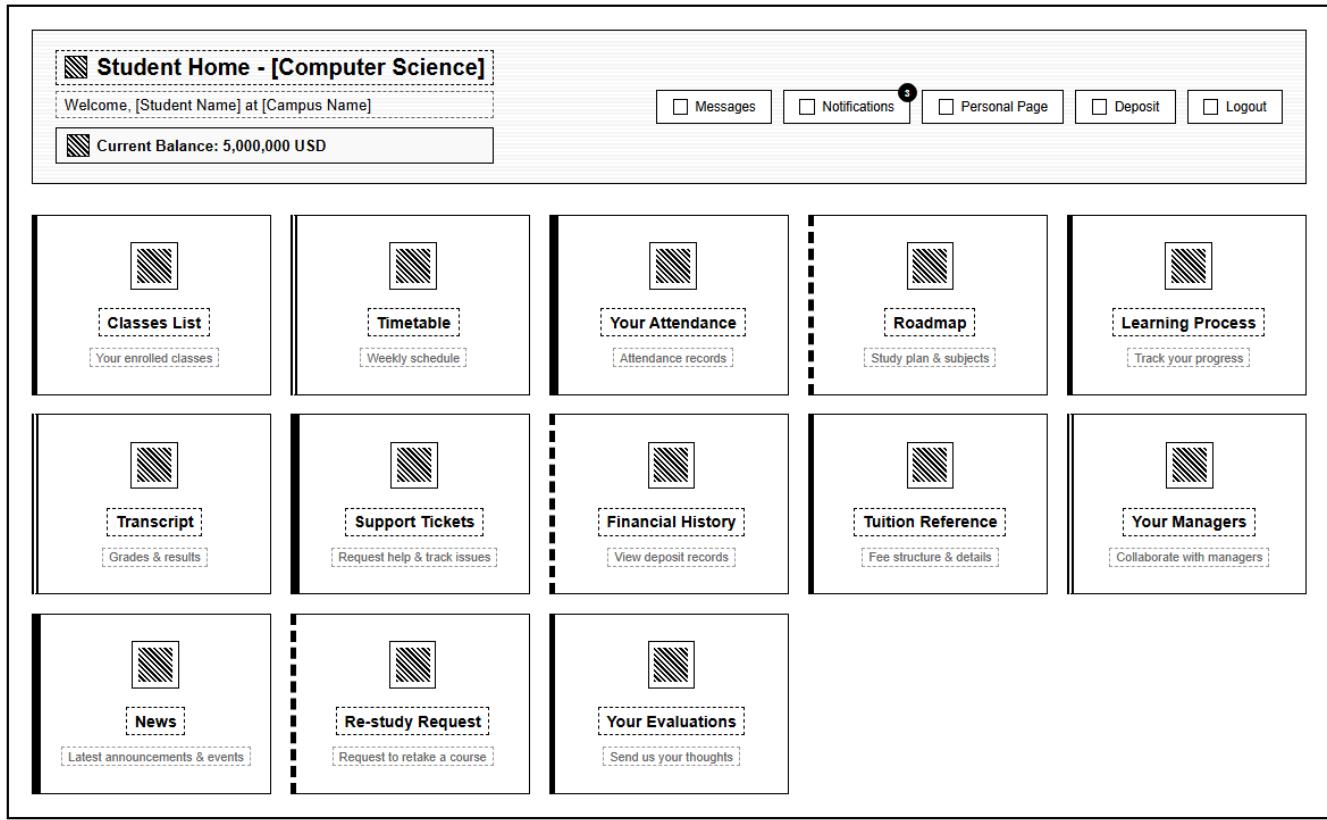
First Name	Last Name
<input type="text"/>	<input type="text"/>
Email	Phone
<input type="text"/>	<input type="text"/>
Birth Date	Gender
<input type="text"/>	<input type="text"/>

Address Information

Country	Province
<input type="text"/>	<input type="text"/>
City	District
<input type="text"/>	<input type="text"/>
Ward	Street
<input type="text"/>	<input type="text"/>
Postal Code	<input type="text"/>
<input type="button" value="Change Password"/>	<input type="button" value="Back"/>

Wireframe 63 Wireframe and Prototype of Staff Personal page

Wireframe and Prototype of Student Home page



Student Home - [Computer Science]

Welcome, [Student Name] at [Campus Name]

Messages Notifications (1) Personal Page Deposit Logout

Current Balance: 5,000,000 USD

 Classes List [Your enrolled classes]	 Timetable [Weekly schedule]	 Your Attendance [Attendance records]	 Roadmap [Study plan & subjects]	 Learning Process [Track your progress]
 Transcript [Grades & results]	 Support Tickets [Request help & track issues]	 Financial History [View deposit records]	 Tuition Reference [Fee structure & details]	 Your Managers [Collaborate with managers]
 News [Latest announcements & events]	 Re-study Request [Request to retake a course]	 Your Evaluations [Send us your thoughts]		

Wireframe 64 Wireframe and Prototype of Student Home page

Wireframe and Prototype of Student Classes List page

 Home
 Your Classes List
Total Classes: 8

 Class ID	 Class Name	 Subject / Spec	 Session	 Slot Quantity	 Created Date	 Action	 Syllabus
CS101-A	[Introduction to Programming]	[Computer Science / Core]	Morning	40	15 Jan 2025	 Access	 View
CS102-B	[Data Structures]	[Computer Science / Core]	Afternoon	35	18 Jan 2025	 Access	 View
CS201-A	[Database Management]	[Computer Science / Elective]	Evening	30	20 Jan 2025	 Access	 View
CS202-C	[Web Development]	[Computer Science / Elective]	Morning	45	22 Jan 2025	 Access	 View
CS301-A	[Software Engineering]	[Computer Science / Advanced]	Afternoon	25	25 Jan 2025	 Access	 View
CS302-B	[Machine Learning]	[AI / Advanced]	Evening	20	28 Jan 2025	 Access	 View

First Prev 1 2 3 4 Next Last | Go to page: 1 ▼

Wireframe 65 Wireframe and Prototype of Student Classes List page

Wireframe and Prototype of Classroom page

 Home
 Back to Class List
[CS101-A: Introduction to Programming]
 Upload Post
 Create Assignment
 Class Members

 Upcoming Deadlines

[Assignment 1: Variables & Data Types]
Due: 28 Nov 2025 23:59


[Assignment 2: Control Structures]
Due: 05 Dec 2025 23:59


[Final Project Proposal]
Due: 15 Dec 2025 17:00


 Assignment by [Lecturer Name]
25 Nov 2025 10:30 

Complete the programming exercises on variables, data types, and basic operations. Submit your code files and a brief report explaining your approach.

Due: 28 Nov 2025 23:59

Attached Files:

- [assignment1_instructions.pdf]
- [starter_code.zip]

Comments (3):

 [Student Name]
Can we use external libraries for this assignment?
10:45 25/11

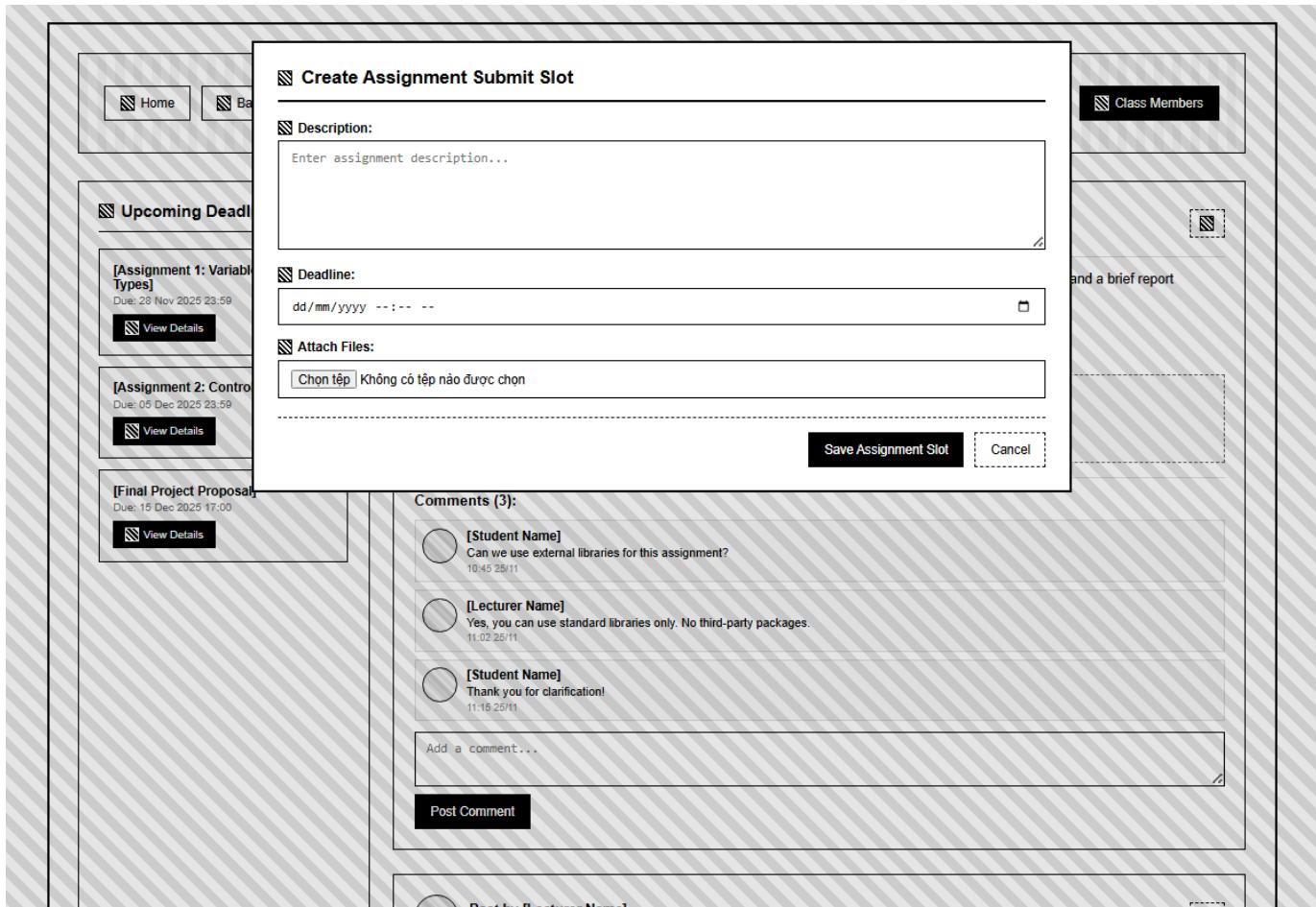
 [Lecturer Name]
Yes, you can use standard libraries only. No third-party packages.
11:02 25/11

 [Student Name]
Thank you for clarification!
11:15 25/11

Add a comment...

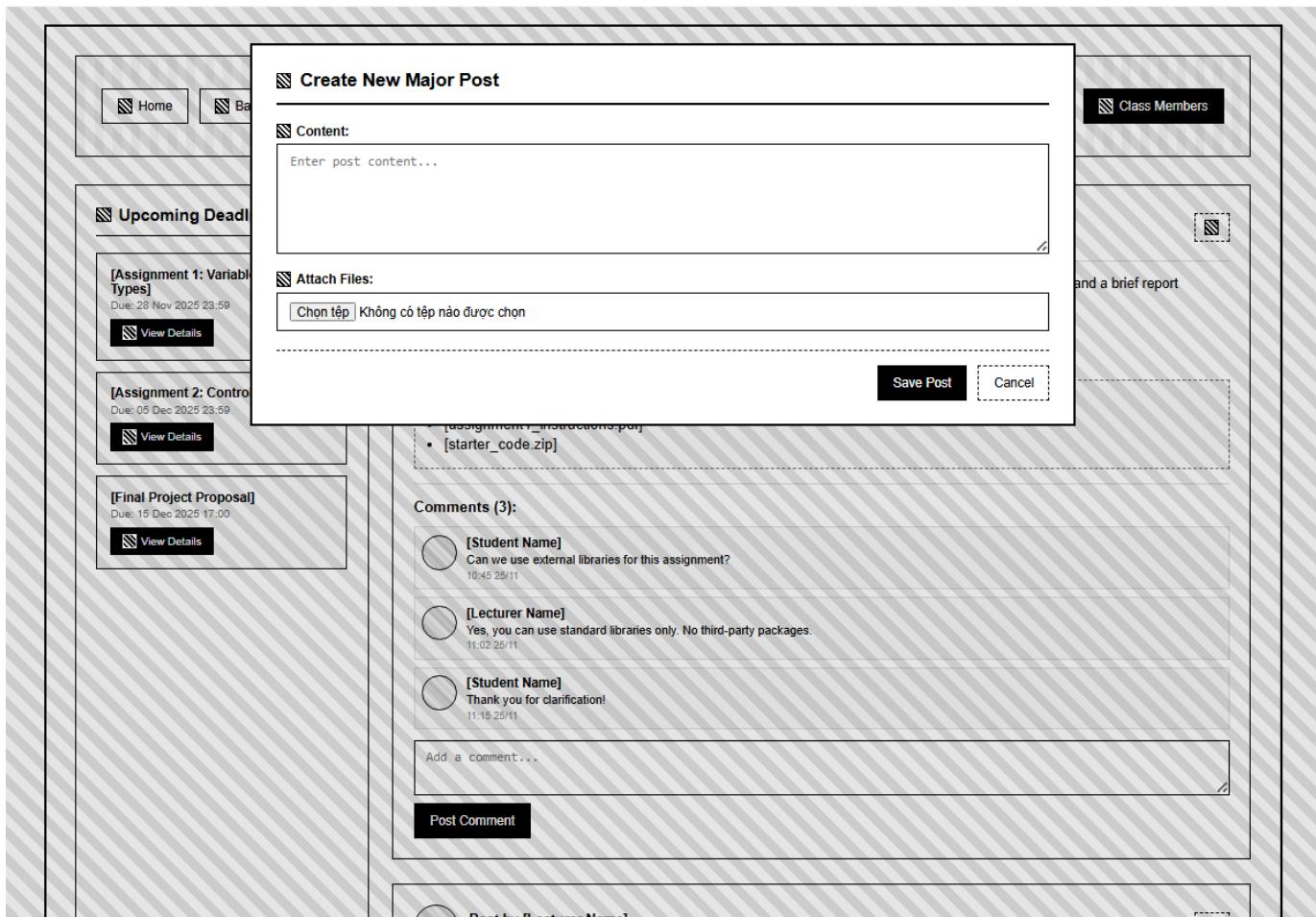


Wireframe 66 Wireframe and Prototype of Classroom page



The wireframe illustrates a classroom interface. On the left, a sidebar displays 'Upcoming Deadlines' with three items: '[Assignment 1: Variable Types]' due Nov 28, '[Assignment 2: Control]' due Dec 05, and '[Final Project Proposal]' due Dec 15. Each item has a 'View Details' button. The main area shows a 'Create Assignment Submit Slot' dialog. This dialog includes fields for 'Description' (with placeholder 'Enter assignment description...'), 'Deadline' (a date input field), and 'Attach Files' (a file upload input with placeholder 'Chon tệp | Không có tệp nào được chọn'). At the bottom are 'Save Assignment Slot' and 'Cancel' buttons. To the right of the dialog is a sidebar with 'Class Members' and a partially visible comment section starting with 'and a brief report'. Below the dialog is a 'Comments (3)' section with three entries from '[Student Name]', '[Lecturer Name]', and '[Student Name]'. Each entry includes a timestamp (e.g., 10:45 25/11). A text input field 'Add a comment...' and a 'Post Comment' button are at the bottom of the comments section.

Wireframe 67 Wireframe and Prototype of Classroom page



The wireframe illustrates a classroom interface. On the left, there's a sidebar with 'Upcoming Deadlines' listing three assignments:

- [Assignment 1: Variable Types]** Due: 28 Nov 2025 23:59 [View Details](#)
- [Assignment 2: Control Structures]** Due: 05 Dec 2025 17:00 [View Details](#)
- [Final Project Proposal]** Due: 15 Dec 2025 17:00 [View Details](#)

The main area features a 'Create New Major Post' dialog box. It includes fields for 'Content' (with placeholder 'Enter post content...'), 'Attach Files' (with a button 'Chọn tệp | Không có tệp nào được chọn'), and a file list showing '[assignment1_instructions.pdf]' and '[starter_code.zip]'. There are 'Save Post' and 'Cancel' buttons at the bottom.

On the right side of the main area, there are two boxes: 'Class Members' and a partially visible one starting with 'and a brief report'.

At the bottom of the dialog box, there's a section for 'Comments (3)' with three entries:

- [Student Name]** Can we use external libraries for this assignment?
10:45 25/11
- [Lecturer Name]** Yes, you can use standard libraries only. No third-party packages.
11:02 25/11
- [Student Name]** Thank you for clarification!
11:15 25/11

A text input field 'Add a comment...' and a 'Post Comment' button are also present.

Wireframe 68 Wireframe and Prototype of Classroom page

Wireframe and Prototype of Assignment Details page

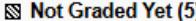
 Back to Class
 Assignment Details
Total Submissions: 15
Deadline: Nov 28, 2025 23:59

 Assignment Information 

Description:
Complete the programming exercises on variables, data types, and basic operations. Submit your code files and a brief report explaining your approach. Make sure to test your code thoroughly before submission.

 Graded (8)

Student	Submitted At	Status	Files	Grade / Feedback
 [Student Name 1] ID: ST001	Nov 27, 2025 18:30	On Time	 [assignment1.py] 	<div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> A  Good work! Code is well-structured.  </div> <div style="background-color: black; color: white; padding: 2px 10px; text-align: center;">  Update </div>
 [Student Name 2] ID: ST002	Nov 28, 2025 01:15	Late	 [solution.zip] 	<div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> B  Late submission. Good effort overall.  </div> <div style="background-color: black; color: white; padding: 2px 10px; text-align: center;">  Update </div>

 Not Graded Yet (5)

Student	Submitted At	Status	Files	Grade / Feedback
 [Student Name 3] ID: ST003	Nov 27, 2025 22:45	On Time	 [code.py] 	<div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> -- Select --  Feedback...  </div> <div style="background-color: black; color: white; padding: 2px 10px; text-align: center;">  Save </div>
 [Student Name 4] ID: ST004	Nov 28, 2025 23:50	On Time	 [assignment.zip] 	<div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> -- Select --  Feedback...  </div> <div style="background-color: black; color: white; padding: 2px 10px; text-align: center;">  Save </div>

Wireframe 69 Wireframe and Prototype of Assignment Details page

Wireframe and Prototype of Student Assignment Details page

[Back to Classroom](#)
Submit Assignment

[Success: Assignment submitted successfully!]

Assignment Details

Title: [Variables & Data Types Exercise]	Created by: [Lecturer Name]	Created at: 28 Nov 2025 10:30
Deadline: 28 Nov 2025 23:59 [(Deadline Passed)]		

Attached Files:

-  [assignment_instructions.pdf] 
-  [starter_code.zip] 

Submit Your Assignment

Upload Files (max 5, 10MB each):

Không có tệp nào được chọn

Supported: PDF, DOC, DOCX, TXT, PPT, PPTX, ZIP

Submit Assignment

Your Submission

Submitted at: 27 Nov 2025 18:45  Delete Submission

Your Files:

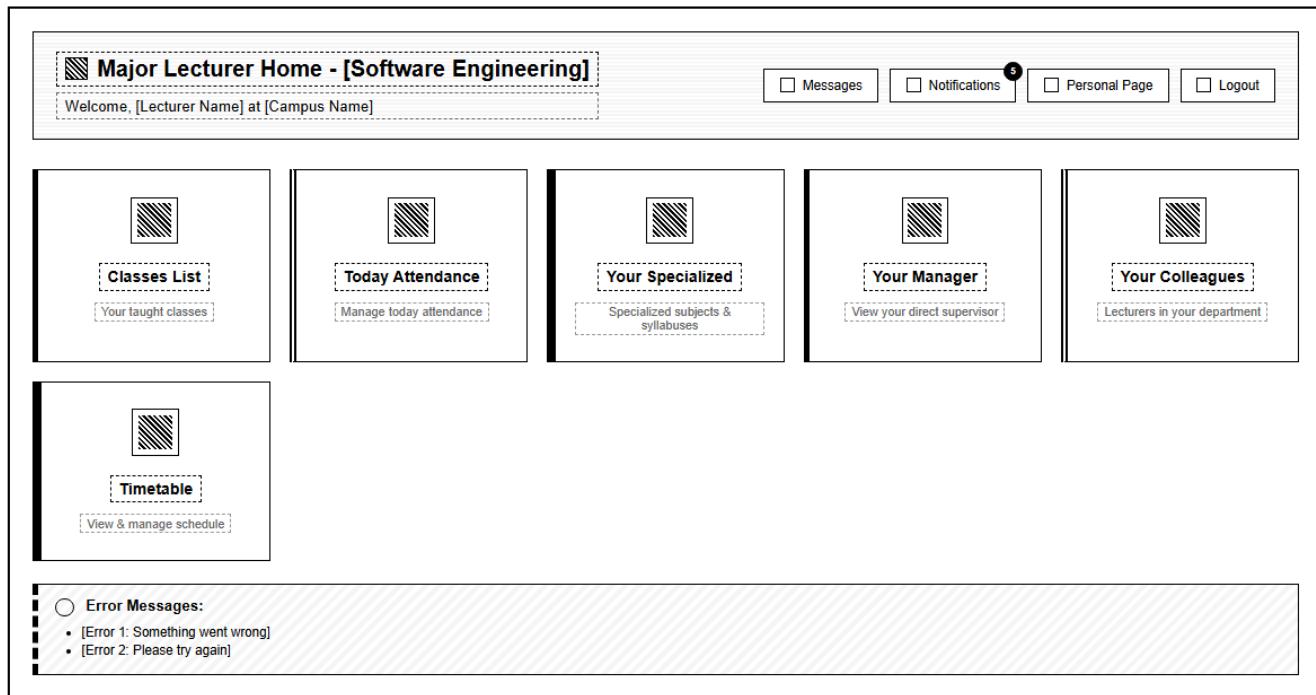
-  [solution.py] 
-  [report.pdf] 

Your Grade & Feedback

Grade: A

Wireframe 70 Wireframe and Prototype of Student Assignment Details page

Wireframe and Prototype of Major Lecturer Home page



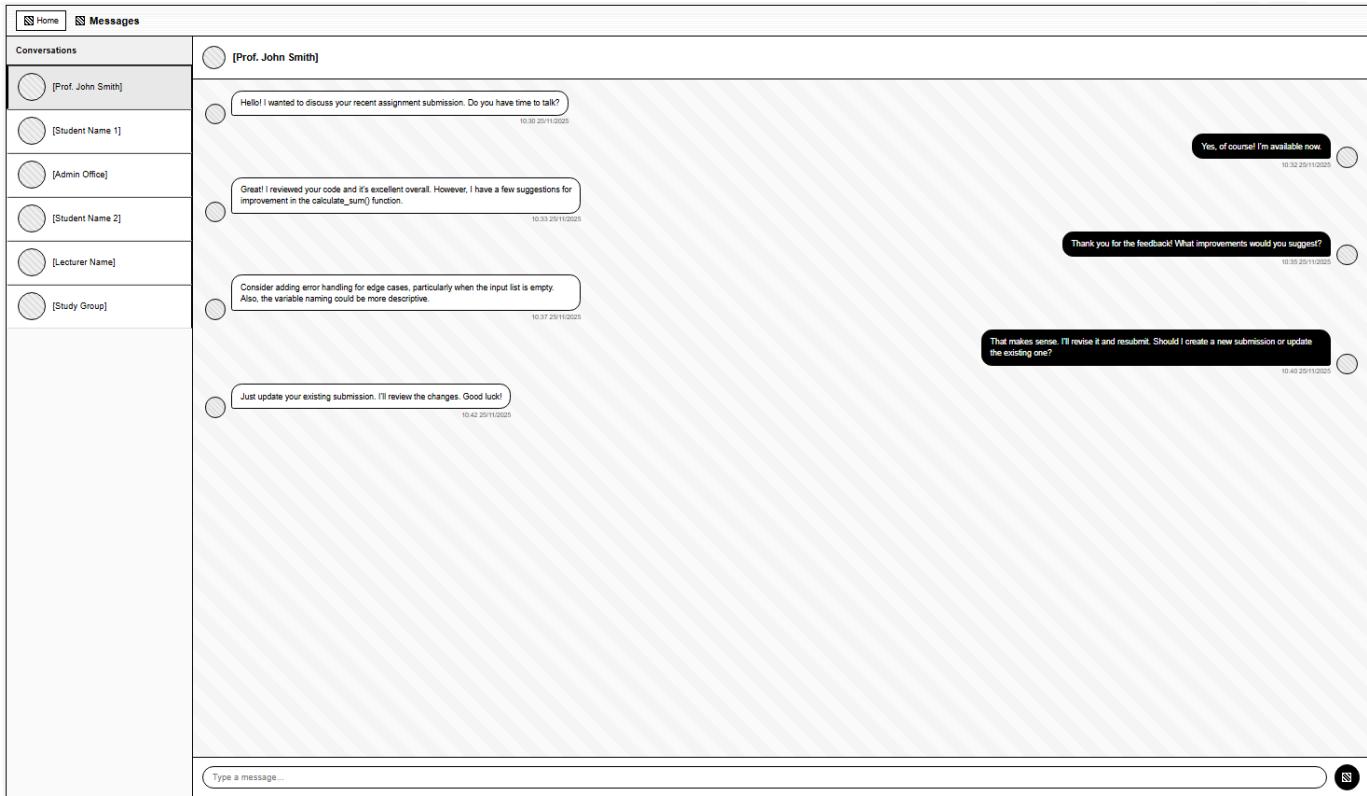
The wireframe shows the layout of the Major Lecturer Home page. At the top, there is a header bar with the title "Major Lecturer Home - [Software Engineering]" and a welcome message "Welcome, [Lecturer Name] at [Campus Name]". On the right side of the header are links for "Messages", "Notifications" (with a notification badge showing '5'), "Personal Page", and "Logout". Below the header are five cards arranged in two rows: the first row contains "Classes List" (with "Your taught classes"), "Today Attendance" (with "Manage today attendance"), "Your Specialized" (with "Specialized subjects & syllabuses"), "Your Manager" (with "View your direct supervisor"), and "Your Colleagues" (with "Lecturers in your department"); the second row contains a single card for "Timetable" (with "View & manage schedule"). At the bottom left, there is a section for "Error Messages" containing two items: "[Error 1: Something went wrong]" and "[Error 2: Please try again]".

Wireframe 71 Wireframe and Prototype of Major Lecturer Home page

Wireframe and Prototype of Notifications Student page

 Home  Notifications		 Total Notifications: 8
#	Notification	
1	<p>[IMPORTANT] Your assignment "Variables & Data Types" has been graded. Grade: A Feedback: Excellent work! Your code demonstrates clear understanding of the concepts.</p> <p>Graded by: Prof. John Smith Date: Nov 28, 2025</p>	
2	<p>New post in CS101-A: Introduction to Programming</p> <p>"Reminder: Next week's lecture will cover advanced topics. Please review Chapter 5 before class."</p> <p>Posted by: Prof. John Smith Date: Nov 27, 2025</p>	
3	<p>[DEADLINE REMINDER] Assignment "Control Structures" is due in 2 days</p> <p>Due date: Nov 30, 2025 23:59 Class: CS102-B</p>	
4	<p>Your attendance has been recorded for CS101-A</p> <p>Date: Nov 27, 2025 Session: Morning Status: Present</p>	
5	<p>[SCHEDULE CHANGE] CS201-A class on Nov 29 has been rescheduled</p> <p>Original time: 10:00 AM New time: 2:00 PM Room: A301 (changed from A201)</p> <p>Please take note of this change.</p>	
6	<p>New comment on your assignment submission</p> <p>Prof. John Smith commented: "Please clarify the logic in function calculate_sum()"</p> <p>Assignment: Variables & Data Types Class: CS101-A</p>	
7	<p>Wireframe 72 Wireframe and Prototype of Notifications Student page</p>	

Wireframe and Prototype of Messages page



The wireframe shows a 'Messages' interface. On the left, a sidebar lists 'Conversations' with icons and names: [Prof. John Smith], [Student Name 1], [Admin Office], [Student Name 2], [Lecturer Name], and [Study Group]. The main area displays a message thread with the following content:

- [Prof. John Smith]: Hello! I wanted to discuss your recent assignment submission. Do you have time to talk? (10:30 25/11/2025)
- [Student Name 1]: Yes, of course! I'm available now. (10:32 25/11/2025)
- [Prof. John Smith]: Great! I reviewed your code and it's excellent overall. However, I have a few suggestions for improvement in the calculate_sum() function. (10:33 25/11/2025)
- [Student Name 1]: Thank you for the feedback! What improvements would you suggest? (10:35 25/11/2025)
- [Prof. John Smith]: Consider adding error handling for edge cases, particularly when the input list is empty. Also, the variable naming could be more descriptive. (10:37 25/11/2025)
- [Student Name 1]: That makes sense. I'll revise it and resubmit. Should I create a new submission or update the existing one? (10:40 25/11/2025)
- [Prof. John Smith]: Just update your existing submission. I'll review the changes. Good luck! (10:42 25/11/2025)

A text input field at the bottom says 'Type a message...'. A small icon in the bottom right corner indicates a notification.

Wireframe 73 Wireframe and Prototype of Messages page

Wireframe and Prototype of Parent Account page

Your Children								 Personal Page	 Logout
ID	Avatar	Full Name	Email	Timetable	Learning Process	Transcript	Payment History		
ST001		[Student Name 1]	[student1@example.com]	 View Timetable	 View Learning Process	 View Transcript	 View Payment History		
ST002		[Student Name 2]	[student2@example.com]	 View Timetable	 View Learning Process	 View Transcript	 View Payment History		
ST003		[Student Name 3]	[student3@example.com]	 View Timetable	 View Learning Process	 View Transcript	 View Payment History		

Wireframe 74 Wireframe and Prototype of Parent Account page

3.8.3 User Journey Maps

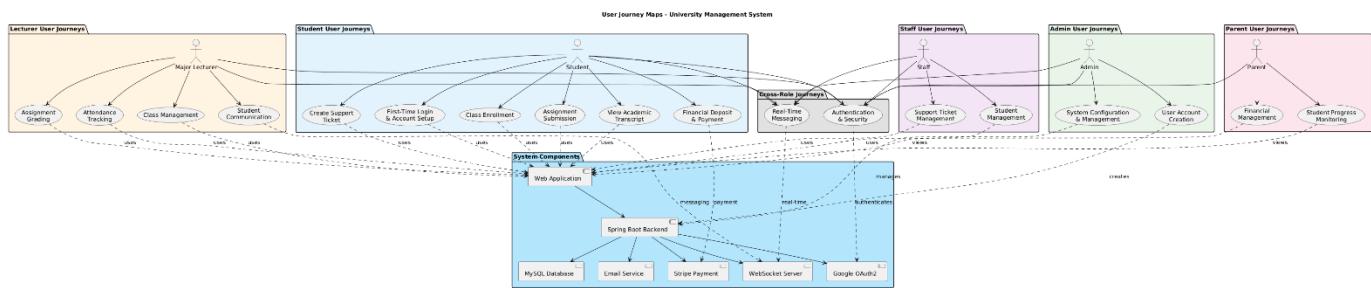


Diagram 10 User Journey Maps

The University Management System is detailed through distinct user journeys for each role, all tightly integrated on a unified platform. Students have a journey that starts with initial login and personal account setup, then performs core tasks including registering for courses, submitting assignments on time, making tuition and other payments through Stripe integration, viewing detailed academic transcripts, and creating support tickets for academic or technical issues. Major Lecturers have specialized functions such as managing classes, tracking and recording student attendance, grading and giving feedback on assignments, and communicating directly with students via a real-time messaging system using WebSocket. Administrative staff are responsible for managing all student information and handling and resolving support tickets from students. System administrators have the ultimate authority with the ability to configure the entire system, manage operational parameters, and directly create new user accounts through the Spring Boot backend. Parents are granted limited but essential access, allowing them to closely monitor their children's academic progress, grades, attendance, and review and manage related finances.

Common journeys across all roles include unified authentication and security via Google OAuth2 and real-time messaging and notifications via WebSocket Server. The entire user experience is provided through a modern web application, connected to a powerful Spring Boot backend, persistent data storage on a MySQL database, secure payment integration with Stripe, identity authentication via Google OAuth2, sending notifications via Email service and maintaining real-time connections via WebSocket, creating a comprehensive, secure and user-friendly university management ecosystem for all subjects from students, lecturers, staff, administrators to parents.

3.8.4 Responsive Design Considerations

The University Management System is built on a strict mobile-first responsive design philosophy, ensuring that every feature works flawlessly on the smallest screens first and is then progressively enhanced as screen real estate increases. Core functionality—whether a student checking assignment deadlines on a phone during a commute, a lecturer marking attendance on a tablet in the classroom, a parent monitoring grades from a smartphone, or an administrator managing user accounts from a large desktop monitor—remains fully accessible, intuitive, and performant across all device categories. The foundation begins with proper viewport meta tags that force the layout to match the device width at a 1:1 scale, enable user scaling when needed, and correctly respond to orientation changes between portrait and landscape. A clearly defined breakpoint strategy drives the adaptation: 480 px targets small phones with strict single-column layouts, maximized touch targets, and reduced padding; 500–600 px serves standard smartphones with slightly richer controls; 768 px marks the tablet threshold where two-column grids, expanded toolbars, and more visible table columns become feasible; 950 px and 1024 px+ unlock full desktop experiences featuring persistent sidebars, three-to-four-column dashboard grids, multi-panel administrative interfaces, and comprehensive weekly timetable views. Layout flexibility is achieved through modern CSS Grid and Flexbox techniques. Dashboard cards use grid-template-columns: repeat(auto-fit, minmax(250px, 1fr)) so columns automatically increase or decrease without explicit media queries, while forms, navigation bars, and toolbars rely on Flexbox wrap and reorder properties to stack gracefully on narrow screens and spread horizontally on wider ones. The unified title bar collapses from a horizontal three-zone layout (left title, center navigation, right actions) on desktop to a compact vertical stack on mobile. The real-time messaging interface transforms from a permanent left sidebar + main content view on large screens to a full-screen slide-over drawer on phones, preserving the entire conversation list while maximizing message reading space. Typography scales intelligently with the viewport: headings shrink proportionally (e.g., from 24 px to 20 px on small devices), body text maintains optimal line length through max-width constraints and generous line-height, and interactive elements respect minimum touch-target dimensions of 44×44 px (iOS) and 48×48 px (Android) with increased spacing to prevent accidental taps. Data tables employ multiple strategies—reduced font and padding, horizontal smooth scrolling when unavoidable, and card-based mobile alternatives that convert rows into vertically stacked, expandable blocks for better readability on narrow screens. Forms transition from multi-column desktop arrangements to full-width single-column flows on mobile, ensuring labels stay clearly associated, inputs are large enough for comfortable typing, and validation messages never overlap fields. Complex pages such as class management and timetables retain all functionality through prioritized content stacking: sidebars move above main content on small devices, timetable cells compress gracefully, and alternative list views are offered when grid

density becomes impractical. Performance is safeguarded by favoring pure CSS solutions, efficient media-query organization, GPU-accelerated properties, and responsive images served at appropriate resolutions. Accessibility remains intact across all breakpoints—color contrast, focus indicators, keyboard navigation, and screen-reader support function consistently, while relative units and proper semantic markup allow users to enlarge text without breaking layouts. Comprehensive testing across Chrome, Firefox, Edge, Safari, iOS and Android devices in both orientations, varying network conditions, and real user sessions confirms that the system delivers a fast, consistent, and inclusive experience regardless of how, when, or where faculty, students, staff, administrators, and parents choose to access it.

3.9 UML Modeling

3.9.1 System Architecture

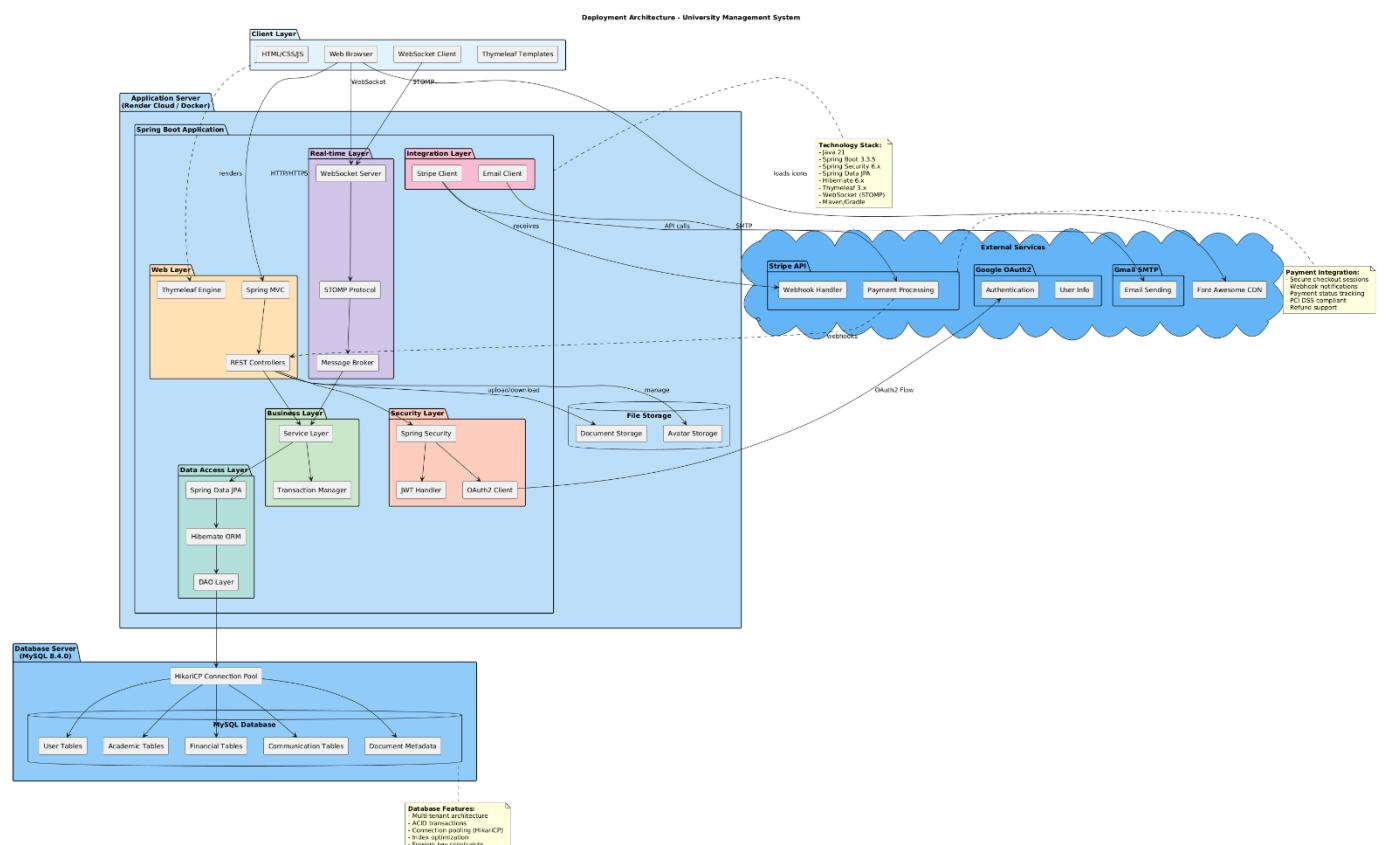


Diagram 11 System Architecture

The above Deployment Architecture diagram outlines a layer deployment architecture of University Management System whereby the Client Layer incorporates a web browser (Web Browser) loading HTML/CSS/JS rendered by Thymeleaf Templates, and also includes a WebSocket Client to achieve real time communication with the server; all communication with the Application Server through HTTP/HTTPS and WebSocket. Application Server contains spring boot application in cloud/Docker (e.g. Render) and internally is divided into numerous packages: Web Layer (Spring MVC, REST Controllers and Thymeleaf Engine) processes when browsers make requests, renders server-side views and provides API; Security Layer (Spring Security, OAuth2 Client, JWT Handler) is used to verify authentication and authorization, will support Google OAuth2 login and will generate/check JWT, the HikariCP connection pool; The Real-time Layer (WebSocket Server, STOMP Protocol, Furthermore, the Application Server is also linked with a File Storage block to store documents (Document Storage) and user avatars (Avatar Storage) as well as metadata of documents is stored in MySQL. The Database Server layer contains MySQL 8.4.0 that comprises numerous tables groups: User Tables (users, accounts) and Academic Tables (classes, subjects, scores, attendance) and Financial Tables (balances, transaction history) and Communication Tables (notifications, messages, tickets) and Document Metadata that are accessed by HikariCP Connection Pool which will maximize performance and guarantee ACID, foreign keys and indexes. Cloud External Services reveals third-party services: Stripe API to make payments (checkout sessions, webhooks, tracking) and Google OAuth2 to make one-time authentication and to send emails via Gmail SMTP and get webhooks delivered by controllers to update the payment status. The technology stack, i.e. Java 21, Spring Boot 3.3.5, Spring Security 6.x, Spring Data JPA etc., is clearly stated in the notes beside the Spring Boot Application and the features of DBs like ACID, connection pooling, index optimization, foreign key described in the notes beside MySQL, and the secure payment methods, webhooks, status monitoring, the compliance with the PCI DSS standards described in the notes beside Stripe API. In general, the figure presents a contemporary 3-level design: thin web client, logic-rich Spring Boot application server, stable MySQL database, and WebSocket to integrate in real-time and external services (payment, email, OAuth2) in a scalable and manageable and cloud-friendly model.

3.9.2 Use Case Diagrams

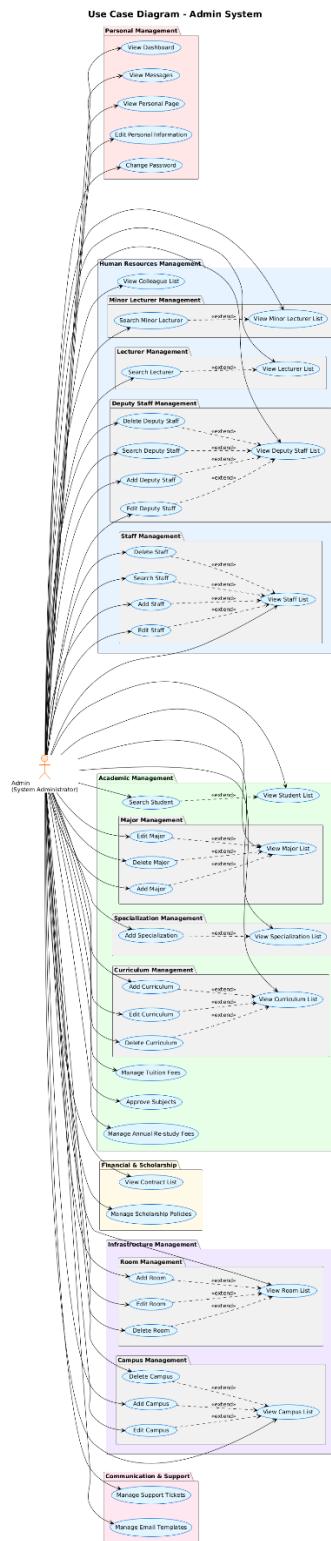


Diagram 12 Admin Management and Academic Administration System — Use Case Model

Admin System permits one main user, the System Administrator (Admin), to administer personal data, human resource, academic, financial, infrastructure, and communication in the institution. On the personal level, the Admin will be able to access their personal page, manage their personal data, change their password, access a dashboard, and read a message, where the dashboard will include statistics, user analytics, financial statistics, and health metrics of the system. The Admin in Human Resources Management can see the list of staff, and carry out such operations as adding, editing, deleting staff, and searching staff; the Admin can also add the deputy staff, which means looking at the deputy staff list, and adding and deleting, editing, and searching deputy staff. The Admin has access to the lecturer list and ability to search lecturers or view minor lecturer list and search minor lecturers and finally a general colleague list to look at the other staff in the system. Under Academic Management, the Admin will be able to see and search through the student list, administer the tuition fees, consent subjects, and the annual re-study fees. The Admin is also in charge of academic structures: in case of majors, the Admin can see the list of majors and create, modify and cancel majors; in case of specializations, the Admin can see the list of specializations and create, modify and cancel specializations; in case of curriculum, the Admin can see the list of curriculum and create, modify and cancel curriculum. The Admin can handle the policies on the scholarships and check the list of contracts associated with financial or academic agreements in the Financial & Scholarship area. In the case of Infrastructure Management, the Admin has the ability to see, add, edit, and delete campuses in the campus list, and to see, add, edit and delete rooms in the room list to be aware of the physical teaching environment. And lastly, Communication & Support, the Admin is able to manage email templates to be used in system notifications and emailing users, and manage support tickets, which ensure that user complaints and requests are recorded and processed in an organized manner. In general, the use case model explains the presence of an Admin who has complete system-wide access to user-related information, academic structures, finances, infrastructure, and communication flows in a centrally located administrative system.

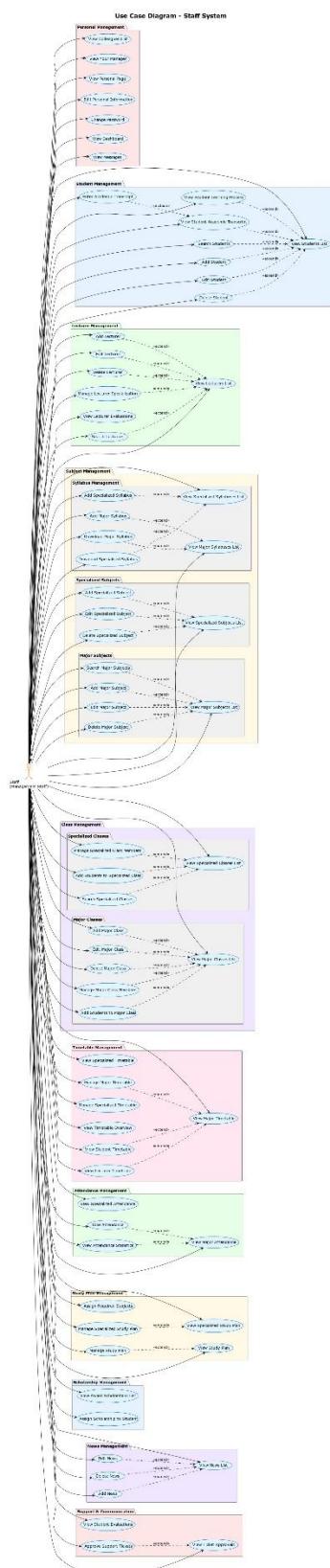


Diagram 13 Staff Academic and Administration Management System — Use Case Model

The Staff Academic and Administration Management System use case model defines the interaction of a member of the management staff with the system to manage personal, academic, and operational tasks of a learning institution. The staff user is able to look at his personal page, update the personal information, change the password, see a dashboard with the most important statistics, read message, list of colleagues, and see the list of his direct manager at the personal level. Regarding student management, the staff are able to view and search the student list, add, edit and delete student record, access the academic transcripts and learning process of the students and input the academic transcripts in the system. In the case of lecturer management, the staff will be able to see the list of lecturers, add, edit and delete lecturers as well as lecturer specialization, and lecturer evaluations. Under subject and syllabus management, the staff will be able to manage the major as well as specialized subjects, such as viewing subject lists, adding, editing, removing, and searching subjects and manage both the major and specialized syllabuses, which includes viewing lists, adding new syllabuses and downloading syllabuses. Under the class management, the staff are able to manage major and specialized classes by viewing class lists, adding, editing and deleting classes, managing class members and assigning students to both major and specialized classes. In managing timetables, the staff members will be able to have a general view of a timetable, each timetable of major, students, lecturers, and specialized programs, and also manage major and specialized timetables. The attendance management enables the staff to see major and specialized attendance records, attend to attendance taking and see attendance statistics. In managing the study plan, the staff will be able to access and manage both the general and specialized study plan and allocate the necessary subjects to have the pupils adhere to the appropriate curriculum map. The staff in scholarship management is able to access the list of awarded scholarships and award students with scholarships. In news management, the employees are allowed to access news published in the system, add, edit and delete them. Lastly, under support and communication, the staffs could check on ticket approvals, support ticket approvals and student evaluations, so that academic operations, communication and support process can be done with a centralized staff-based platform, both in a consistent and efficient manner.

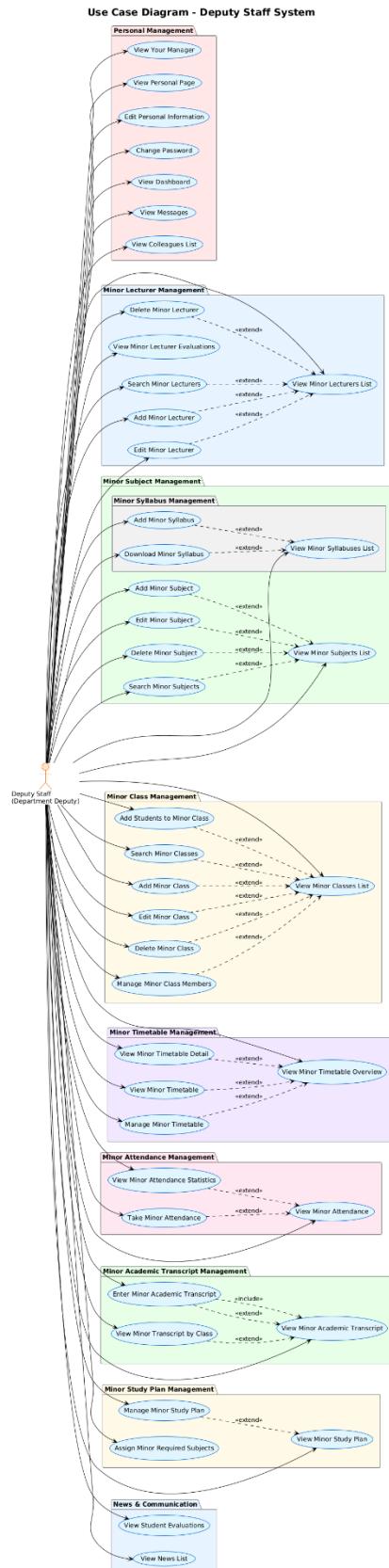


Diagram 14 Deputy Staff Academic and Minor Program Management System — Use Case Model

001479794

Page 237 | 635 total

The use case model of the Deputy Staff Academic and Minor Program Management System is a model of how a deputy-level academic administrator is able to oversee the entirety of minor academic programs in an educational institution. The deputy staff member starts with personal management abilities, including access to and updating personal data, changing his/her password, on a dashboard that displays operational insights, a message board, listing colleagues, and determining who his/her manager is. The essence of their work is to supervise the minor lecturers through viewing, searching, adding, editing, deleting and assessing minor teaching staff. They are also in charge of minor academic topics such as viewing and searching the subject list, adding new subjects, editing or deleting existing subjects and managing related minor syllabuses by viewing, adding or downloading syllabus files. Compared to minor classes, the deputy staff member will have an opportunity to see class lists, search them, create, edit, or delete minor classes, administer class membership, and place students into the right minor groups. The tasks that are performed under the timetable entail seeing a summary of the minor timetable, seeing the schedules of individual minor timetable and overseeing the minor timetable layout. To manage the attendance, the deputy staff member will be allowed to see the attendance records, record the attendance and discuss the attendance statistics of the minor courses. The system also facilitates the responsibility of academic transcripts, whereby the deputy staff user is able to view, input, and browse minor transcripts, including class based transcripts views. They also control minor plans of study by allowing students to see and make changes to structure of study plans and allocating minor subjects that one must take. There are also features related to communication that are included in the model like reading institutional news and looking at student reviews. On the whole, this use case model gives a detailed representation of the process that a Deputy Staff user manages all the academic, operational, scheduling, and evaluation issues of minor programs in the institution.

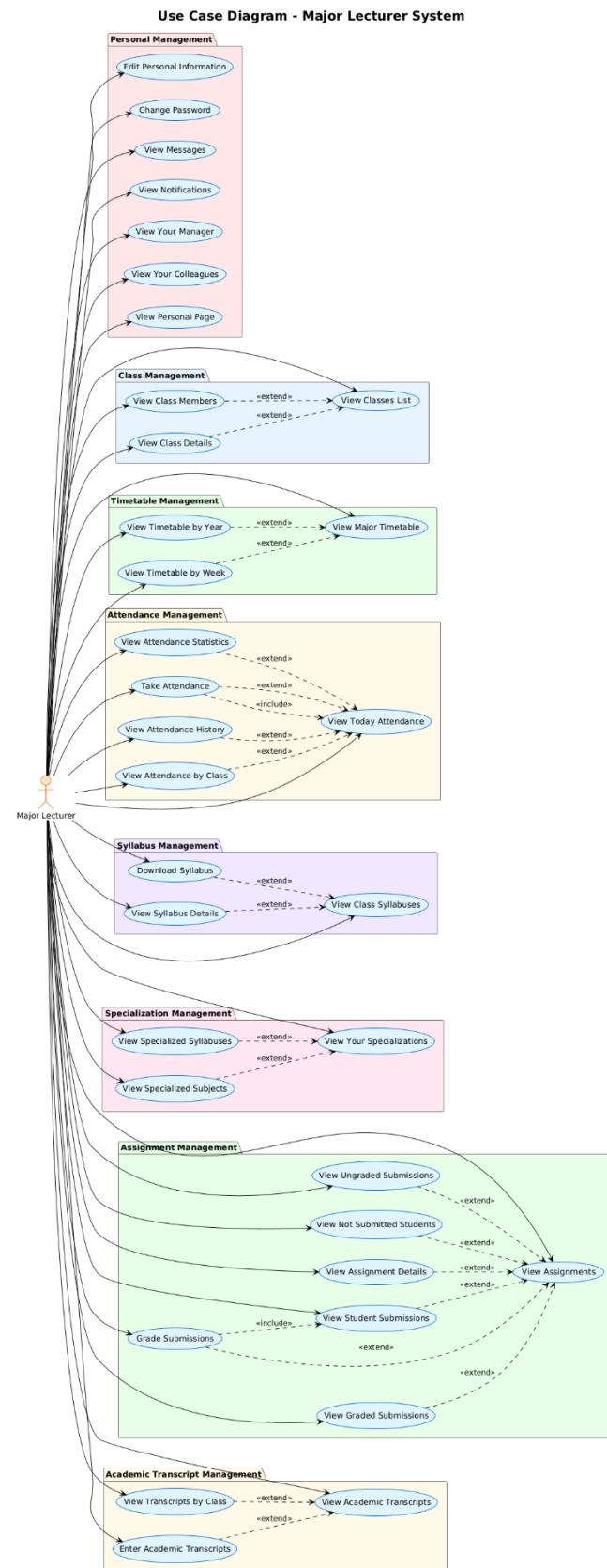


Diagram 15 Major Lecturer Academic and Classroom Management System — Use Case Model

001479794

Page 239 | 635 total

The use case model of the Major Lecturer Academic and Classroom Management System outlines the interactions of a major lecturer with the institutional academic platform to administer teaching activities, student interaction, scheduling, and assessment activities. The lecturer has access to personal information and can control it with the help of such features as looking at personal data and making amendments, getting accustomed to notification and messages, as well as having access to the information about their manager and colleagues. In classroom duties, the lecturer is able to see the classes that they are assigned, the classes, and see the class members. The system facilitates the operations of teaching schedule whereby the lecturer can see the large-scale schedule or the weekly and yearly schedule design. Attendance management allows the lecturer to see the daily attendance, record attendance per class session, see the attendance history, see the attendance by-class, and see analytical attendance statistics. There is also the interaction between the lecturer and the syllabus resources where they view the class syllabuses, check on the detailed syllabus contents and download syllabus files to be used later. The system also facilitates the specialization functions through displaying the specializations and the special subjects as well as the special syllabuses assigned to the lecturer. Assignment management involves checking assignments, assignment details, student submissions, grading submissions, assessing submitted assignments, and reading graded and ungraded submissions, as well as tracking the students who have not yet submitted. Lastly, management of academic transcripts offers the lecturer to see academic transcripts, input grades on the students and check transcripts at class level. In general, this use case model shows how a key lecturer oversees the academic delivery, classroom, student performance, and smooth instructional processes in a well-organized academic setup.

Use Case Diagram - Minor Lecturer System

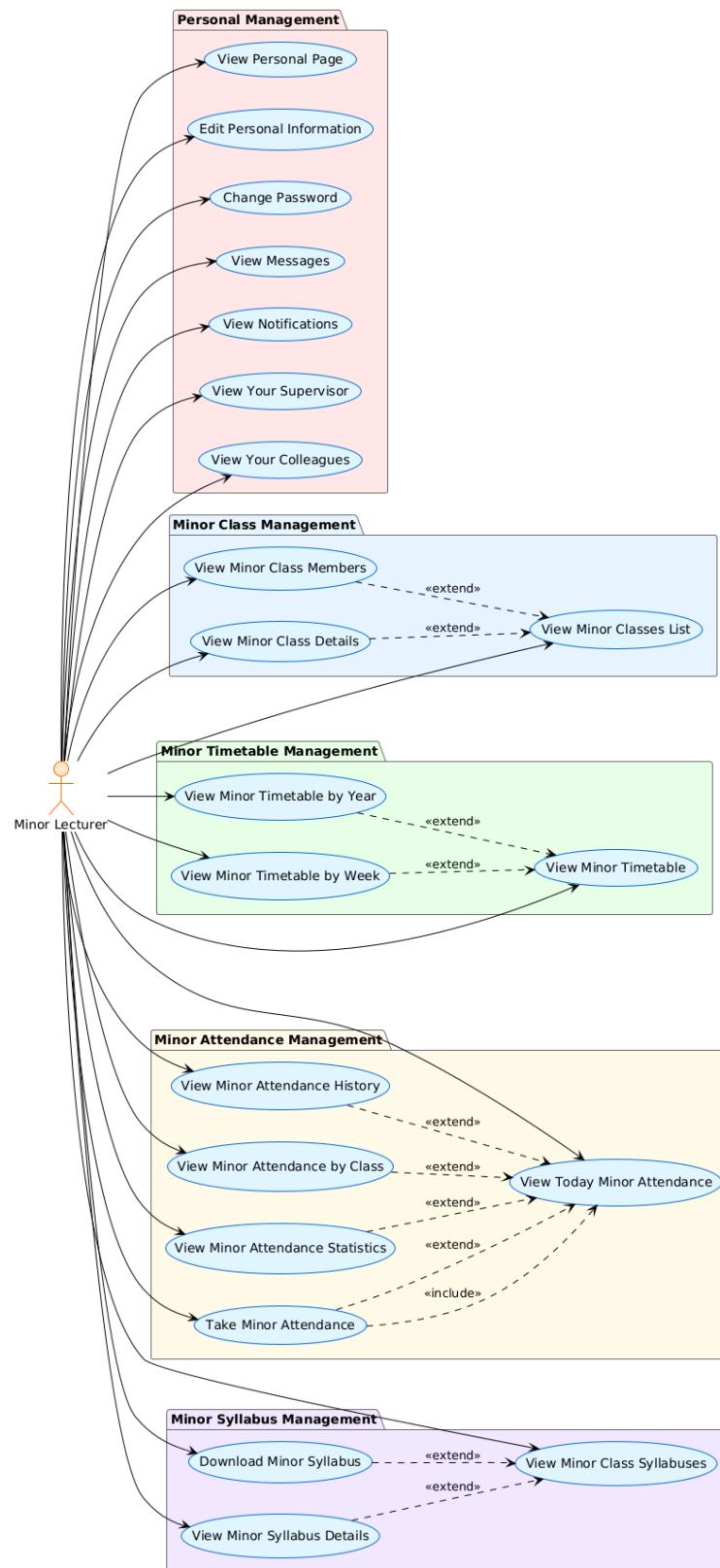


Diagram 16 Minor Lecturer Academic and Class Management System — Use Case Model

001479794

Page 241 | 635 total

The use case model of Minor Lecturer Academic and Class Management System shows the interaction of a minor lecturer with the academic platform to organize instructional tasks and daily teaching processes in the minor program. The lecturer starts with personal management skills, such as viewing and editing personal data, reviewing notifications and messages, and information about the supervisor and coworkers. Their teaching roles revolve around minor classes management, in which they can access minor classes list assignments, have access to class information and review classes members. The scheduling functions enable the lecturer to access the minor schedule and use it either by week or year, enabling him or her determine the hours of teaching, the room in which teaching takes place and the weekly schedule. Attendance management also entails viewing daily minor attendance, attending classes on a per-session basis, reviewing attendance history, checking the attendance by classes, and performing the attendance statistics to ensure student attendance. The lecturer also relates with minor syllabuses by looking at the syllabuses made available on assigned minor classes, establishing detailed syllabus information, and downloading syllabus material on teaching preparation. On the whole, the system allows minor lecturers to effectively organize the interactions in the classroom, monitor the work of students, prepare the lesson, and organize the academic works with the supervision system of the Deputy Staff.

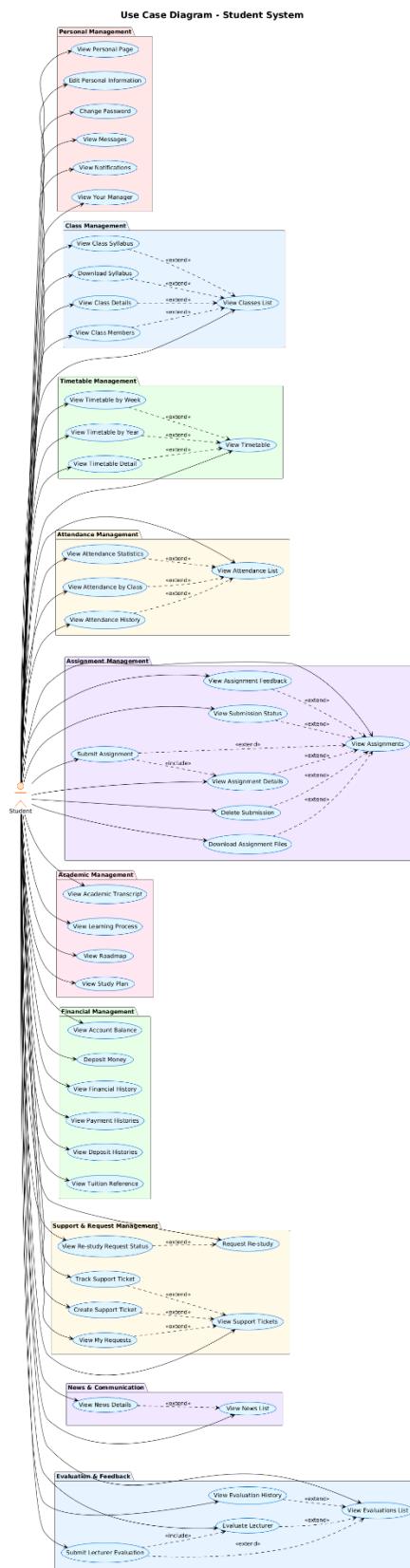


Diagram 17 Student Academic, Financial, and Support Management System — Use Case Model

Description (one paragraph): The Student Academic, Financial, and Support Management System use case model explains the interaction of a student with the institutional platform, whereby the student manages his or her whole lifecycle of studying, including academic actions as well as finances and support services. In personal management, the student will be able to see and make changes on personal information, change passwords, read messages, see notifications, and view whoever their assigned manager or advisor is. Under the management of classes, the student is able to see the list of classes he/she is enrolled in, the details of the classes, the members of the classes, the class syllabus and downloading of the syllabus documents. Timetable management will enable the student to see his or her timetable, search through it, by week or year, and view the schedule specifics in detail to discern the location and time of classes. The attendance management feature of the site will enable them keep track of their attendance by viewing the attendance lists, attendance by classes, the entire attendance history, and attendance statistics. The assignment management allows the students to see all assignments, review assignment information, submit assignment, monitor assignment submission status, review feedback and grades, as well as delete their submissions as may be permitted and download assignment files. The academic management features enable the student to have a look at his/her academic transcript, track the academic process as time progresses, see the roadmap, and look at the study plan that will take him/her through the program. Financial management aids in the viewing of account balance, money depositing, financial history review, payment and deposit history review, and tuition reference information. The student will be able to see support tickets, make new support requests, view the requests that he/she already made, check the status of the support requests, make re-study requests, and monitor the status of the re-study requests in the field of support and requests. The student can view evaluation lists, evaluate lecturers, submit lecturer evaluations, and see their evaluation history so that they can have the evaluation and feedback. Lastly, with news and communication, the student is able to see a list of news items and access detailed announcements, meaning that he/she is informed of any updates in the institution and on academic news.

Use Case Diagram - Parent System

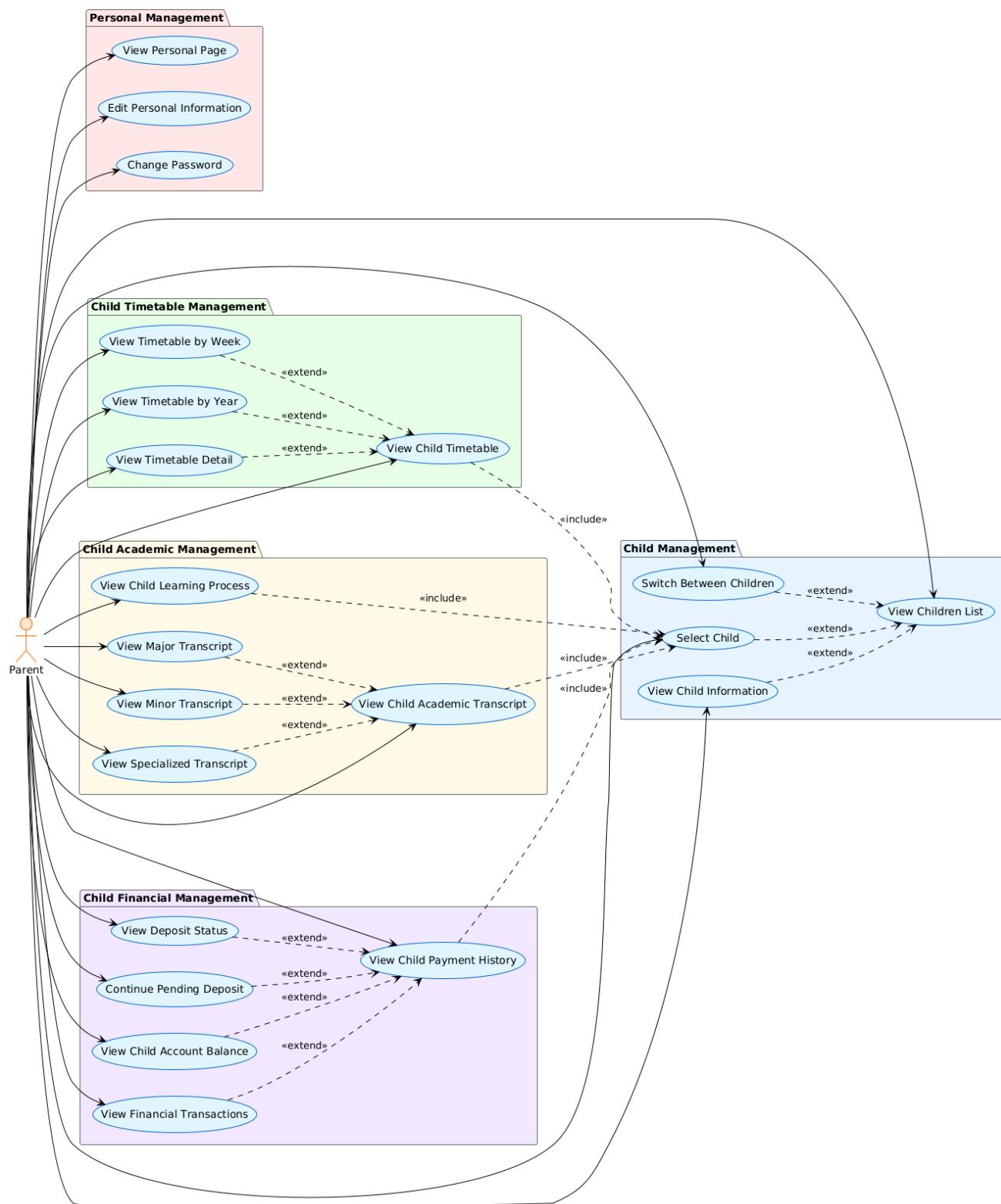


Diagram 18 Parent Academic Monitoring and Child Management System — Use Case Model

The Parent Academic Monitoring and Child Management System use case model describes how parents access and monitor their children's academic, scheduling, and financial information through the institutional platform. Parents can manage their personal information by viewing and updating their profile and changing their password. Through child management capabilities, they can view the list of all linked children, select a specific child, view detailed information, and switch between children seamlessly. The system also provides full visibility into each child's timetable, allowing parents to view daily, weekly, and yearly schedules, as well as detailed class information. Academic management features enable parents to monitor their child's learning progress, review their overall academic transcript, and explore major, minor, and specialized transcript categories in detail. Financial management allows parents to review their child's payment history, check account balance, view financial transactions, monitor deposit status, and continue any pending deposit operations. Overall, this use case model captures the core responsibilities of parents within the system, emphasizing transparency, multi-child support, academic oversight, and financial tracking in a structured and user-friendly environment.

3.9.3 Class Diagrams

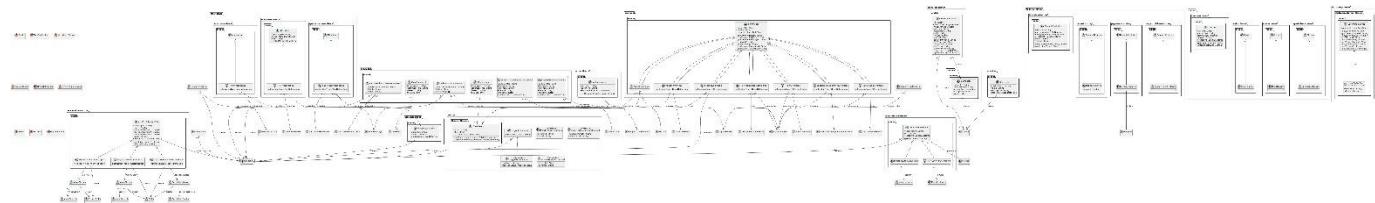


Diagram 19 Class Diagram

The given class diagram presents a highly detailed domain model of a large-scale university management system, which strictly isolates three parallel academic paths, i.e. Major (core curriculum), Minor (elective/supplementary), and Specialized (advanced/specialization) programs, where nearly all major features are replicated and specialized among the three parallel paths. The most basic of them are person-related entities: Persons is the base entity that is one-to-one related with Authenticators (handling passwords and AccountStatus), and the role-specific entities include Students, Staffs, DeputyStaffs, MajorLecturers, MinorLecturers, MajorEmployees, MinorEmployees, and Admins, which inherit or are related to it. Academic records are modelled using an abstract AcademicTranscripts class with common grading fields (score components, final score, Grades enum, timestamps) and abstract methods to identify the subject (3 concrete subclasses, each tied to their own type of class MajorClasses, MinorClasses, SpecializedClasses) and creator (Staffs or DeputyStaffs) with each being associated with a notification type on which to issue push alerts in the event of grade postings. Attendance is also a polymorphic class through an abstract Attendance base which has three specialized subclasses which are associated with three respective timetables (MajorTimetable, MinorTimetable, SpecializedTimetable) and characterized by the relevant type of employee. Classes are themselves based on an abstract Classes entity which has common properties (name, slot quantity, session, creator) which are then instantiated into MajorClasses (associated with MajorSubjects and created by Staffs), MinorClasses (associated with MinorSubjects and created by DeputyStaffs), and SpecializedClasses

(associated with SpecializedSubject and created by Staffs). The assignment and submission workflow is also divided into similar the AssignmentSubmitSlots - Submissions - SubmissionDocuments and SpecializedAssemblySubmitSlots - SpecializedSubmissions - SpecializedSubmissionDocuments with commenting layers (MajorAssignmentComments, SpecializedAssignmentComments, etc.) that generate OtherNotification types. Class discussion and announcement are provided on an abstract PublicPosts - PublicComments base, which is then specialized to MajorClassPosts/ MinorClassPosts/ SpecialisedClassPosts and their corresponding comment types (MajorComments, MinorComments, StudentComments, etc.), each with metadata of notification. Financial management comprises AccountBalances (one per student) and a rich polymorphic hierarchy of FinancialHistories (DepositHistories, PaymentHistories on a subject, and SupportTicketHistories) which has complete audit trails with versioning and status. The lecturer reviews are divided by MajorLecturerEvaluations and MinorLecturerEvaluations, which are associated with the student who is reviewing the lecture and the lecture that is being reviewed. Support ticket functionality has SupportTickets and SupportTicketRequests that contain support documents and financial history records. Other supporting classes are Campuses (managed by Admins), Curriculum frameworks, customizable EmailTemplates per campus, different document types attached to posts or submissions and several embeddable composite-id classes (e.g., LecturersClassesId, MajorLecturersClassesId, Studentparentaccountsid, Upgradestudentsid) to support many-to-many relationships and student upgrade processes. A mature domain-driven model with heavy use of abstract classes, inheritance, composition, and notification enums through the model is a deliberate reflection of the structure of the university of major, minor, and specialized courses at the same time with an elegant extensibility and a highly-readied notification and events system.

3.9.4 Sequence Diagrams

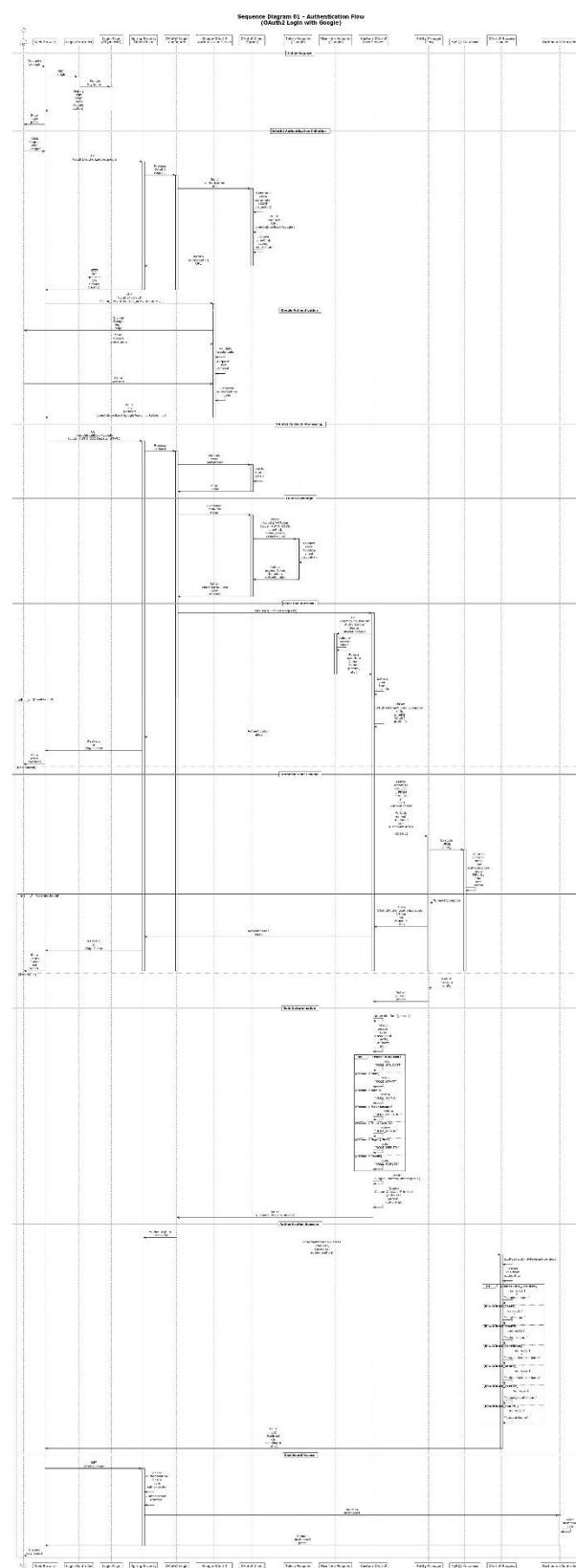


Diagram 20 Sequence Diagram 01 – Google OAuth2 Authentication and Role-Based Login Flow

This sequence diagram illustrates, in detail, the complete OAuth2 login flow when a user signs in to the University Management System using Google, from the moment they open the login page to the point they land on their role-specific dashboard. The process begins with the **User** accessing /login through the **Web Browser**, which sends a GET /login request to the **Login Controller**. The controller renders the **Login Page (Thymeleaf)**, which is returned to the browser and displayed to the user, including a “Sign in with Google” button. When the user clicks this button, the browser sends a GET /oauth2/authorization/google request that is intercepted by the **Spring Security Filter Chain**, which delegates to the **OAuth2 Login Configurer**. The configurer calls the **OAuth2 Client** to build the Google authorization URL: it generates a state parameter for CSRF protection, constructs the redirect URI (/oauth2/callback/google), and includes parameters such as client_id and scope. The security filter then responds with an HTTP 302 redirect to Google’s OAuth2 authorization endpoint, and the browser follows this redirect to the **Google OAuth2 Authorization Server**, where the user sees Google’s login page, enters their credentials, and grants consent. Google validates the credentials, generates an authorization code, and redirects the browser back to the application’s callback endpoint /oauth2/callback/google?code=...&state=....

On callback, the browser sends the request to the **Security Filter Chain**, which again routes it to the **OAuth2 Login Configurer**. The configurer first asks the **OAuth2 Client** to validate the state parameter, verifying the CSRF token. If valid, it proceeds to the token exchange phase by requesting the client to exchange the authorization code for tokens. The **OAuth2 Client** sends a POST /oauth2/v4/token call to Google’s **Token Endpoint**, which validates the code and client credentials, and returns an access_token, id_token, and optionally a refresh_token. These tokens are wrapped into an OidcUserRequest and handed back to the configurer, which then calls the **Custom OAuth2 User Service** (loadUser) to load the local user. The custom service first calls Google’s **User Info Endpoint** with the access_token to retrieve profile data such as email, name, and picture. When the response arrives, the service extracts the email. If the email is missing, it throws an OAuth2AuthenticationException, the security filter translates this into a failed authentication, and the user is redirected back to /login?error with an error message.

When a valid email is present, the **Custom OAuth2 User Service** queries the database via the **Entity Manager (JPA)** and **MySQL Database**, using a JPQL query that joins the Persons and Authenticators tables and filters by email and active account status. If no matching record is found, the database throws a NoResultException, which the user service translates into another OAuth2AuthenticationException (“User not found”), and the security filter again redirects the user to /login?error. If a matching person is found, the entity manager returns a Persons entity to the user service, which then performs role determination. It inspects the person object (e.g., whether it is a student, staff, admin, major lecturer, minor lecturer, deputy staff, or parent) and, based on this, selects the appropriate Spring Security role such as ROLE_STUDENT, ROLE_STAFF, ROLE_ADMIN, ROLE_LECTURER, ROLE_MINOR, ROLE_DEPUTY, or ROLE_PARENT. The service creates a SimpleGrantedAuthority for this role, builds a CustomOidcUserPrincipal combining the OIDC user info, the domain person, and the list of authorities, and returns it to the **OAuth2 Login Configurer** as the authenticated principal.

Once authentication succeeds, the configurer notifies the **Spring Security Filter Chain**, which invokes the **OAuth2 Success Handler**’s onAuthenticationSuccess method. The success handler reads the authorities from the authentication object, determines the user’s role, and maps that role to a specific dashboard URL—for example, /student-home for ROLE_STUDENT, /staff-home for ROLE_STAFF, /admin-home for ROLE_ADMIN, /major-lecturer-home for ROLE_LECTURER,

/minor-lecturer-home for ROLE_MINOR, /deputy-staff-home for ROLE_DEPUTY, or /parent-home for ROLE_PARENT. It then responds with an HTTP 302 redirect to that dashboard URL. The browser follows this redirect, triggering a new GET /{role}-home request that is again intercepted by the security filter, which now verifies that the user is authenticated and has the required role. Once authorization is granted, the request is dispatched to the **Dashboard Controller**, which loads the necessary dashboard data and returns the rendered page to the browser. Finally, the browser displays the dashboard to the user. Overall, the diagram captures both the happy path and main error branches, showing how Google OAuth2, Spring Security, database lookup, and role-based redirection work together to provide secure, role-aware single sign-on for the system.

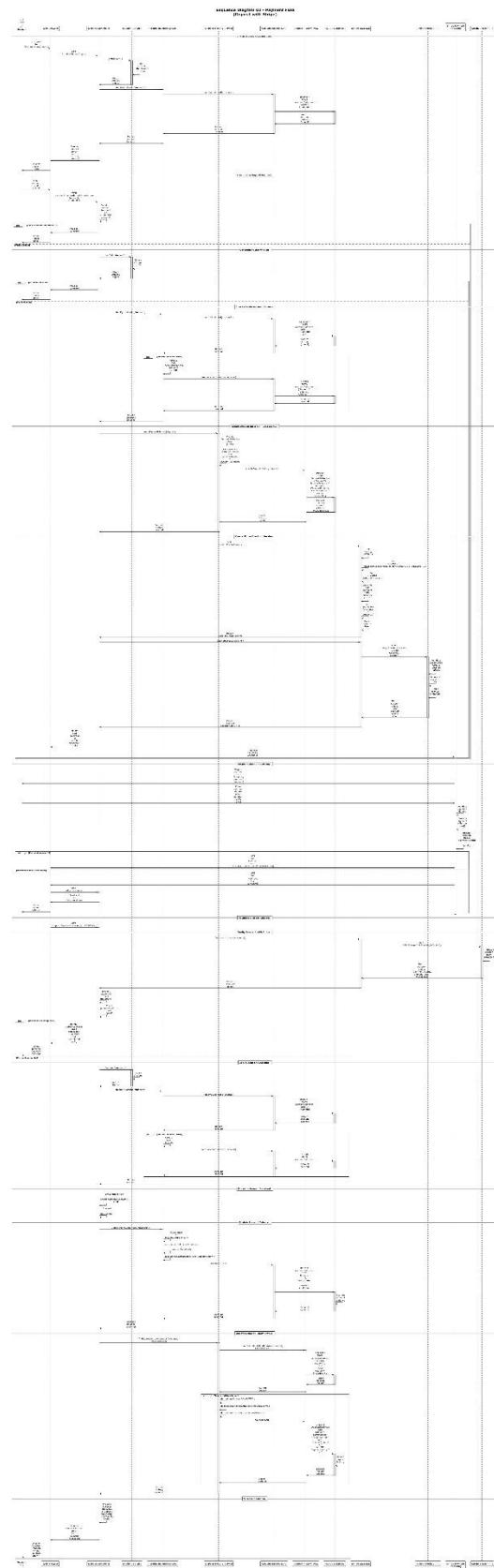


Diagram 21 Sequence Diagram 02 – Stripe Deposit Payment and Account Balance Update Flow

001479794

Page 251 | 635 total

This sequence diagram describes end-to-end how a student makes a deposit through Stripe and how the system updates both the account balance and the deposit history in the University Management System. The flow starts when the **Student** navigates to /student-home/deposit via the **Web Browser**, which calls the **Deposit Controller**. The controller retrieves the current authenticated student from the **Student Service**, then asks the **Account Balance Service** (through the **Account Balance DAO** and **MySQL Database**) for the student's current balance. This information is returned to the controller, which renders the deposit page with the current balance for the student to see. The student then enters a deposit amount and submits the form, triggering a POST /student-home/deposit/create-new. The **Deposit Controller** validates the input (amount > 0, studentId present); if invalid, it redirects to an error page. For a valid request, it loads the student by ID via the **Student Service**; if the student does not exist, the request is again redirected to an error page. When the student is found, the controller ensures there is an account: it calls **Account Balance Service** → **Account Balance DAO** → **Database** to fetch the account; if none exists, a new balance record is created and persisted.

Once the account is available, the controller creates a **deposit history** record in status PROCESSING through the **Deposit History Service**, which constructs a DepositHistories object with changedAmount = 0, currentAmount = account.balance, and passes it to the **Deposit History DAO** to be inserted into the database. With persistence done, the controller then moves to initiating payment with Stripe. It calls the **Stripe Java SDK** to build SessionCreateParams: setting mode to PAYMENT, specifying successUrl (/deposit/success?session_id={CHECKOUT_SESSION_ID}), cancelUrl (/deposit/cancel), adding a line item with amount converted to cents and currency (USD), and embedding metadata such as studentId. The SDK returns this parameter object; the controller then calls Session.create(params), which the SDK sends to the **Stripe Checkout API**. Stripe validates the parameters, creates a checkout session, stores metadata, and returns a **Session** object containing a session ID and checkout URL. The controller responds with an HTTP 302 redirect to that URL, and the browser redirects the student to the **Stripe Payment Processing** page.

On the Stripe side, the **Stripe Payment** participant displays the payment form, accepts card details from the student, validates them, and processes the charge. If the payment fails or is cancelled, Stripe redirects the browser to the cancel URL, which hits /deposit/cancel in the **Deposit Controller**; the controller then redirects back to /student-home, and the student returns to their dashboard without any balance change. If payment is successful, Stripe redirects the browser to the success URL with session_id. The browser calls GET /deposit/success?session_id=..., which the **Deposit Controller** uses to verify the payment. It asks the **Stripe Java SDK** to Session.retrieve(sessionId), which calls the **Stripe Session API** to fetch the session from Stripe's side, returning a Session object with payment_status, amount_total, and metadata. The controller reads the studentId from metadata and checks that payment_status == "paid". If payment is not completed, it renders a "payment not completed yet" page for the student.

When the payment is confirmed as completed, the controller reloads the student via the **Student Service** and retrieves the account via **Account Balance Service** → **DAO** → **Database**, creating a new account if one still does not exist. It then computes amountReceived from amount_total / 100 and converts it to a monetary type. To update the balance, the controller calls **Account Balance Service**, which reads the old balance, adds the received amount, updates the lastUpdated timestamp, and persists the new state through the **Account Balance DAO** with an UPDATE on the AccountBalances table. After the balance is updated, the controller synchronizes

the **deposit history**: it asks **Deposit History Service** to find the PROCESSING record for the student via **Deposit History DAO → Database**, and, if found, the service changes its status to COMPLETED, sets changedAmount to the actual amount received, and currentAmount to the new balance, then saves it back with an UPDATE statement. Finally, the deposit history update result is returned to the controller, which prepares a success message such as “Deposit successful! +{amount} USD”, renders the success page, and sends it to the browser. The browser shows the success message and updated balance to the student. Overall, the diagram captures input validation, account creation, deposit tracking (PROCESSING → COMPLETED), Stripe checkout session creation, external payment processing, secure payment verification, and the consistent update of both account balance and deposit history in the local database.

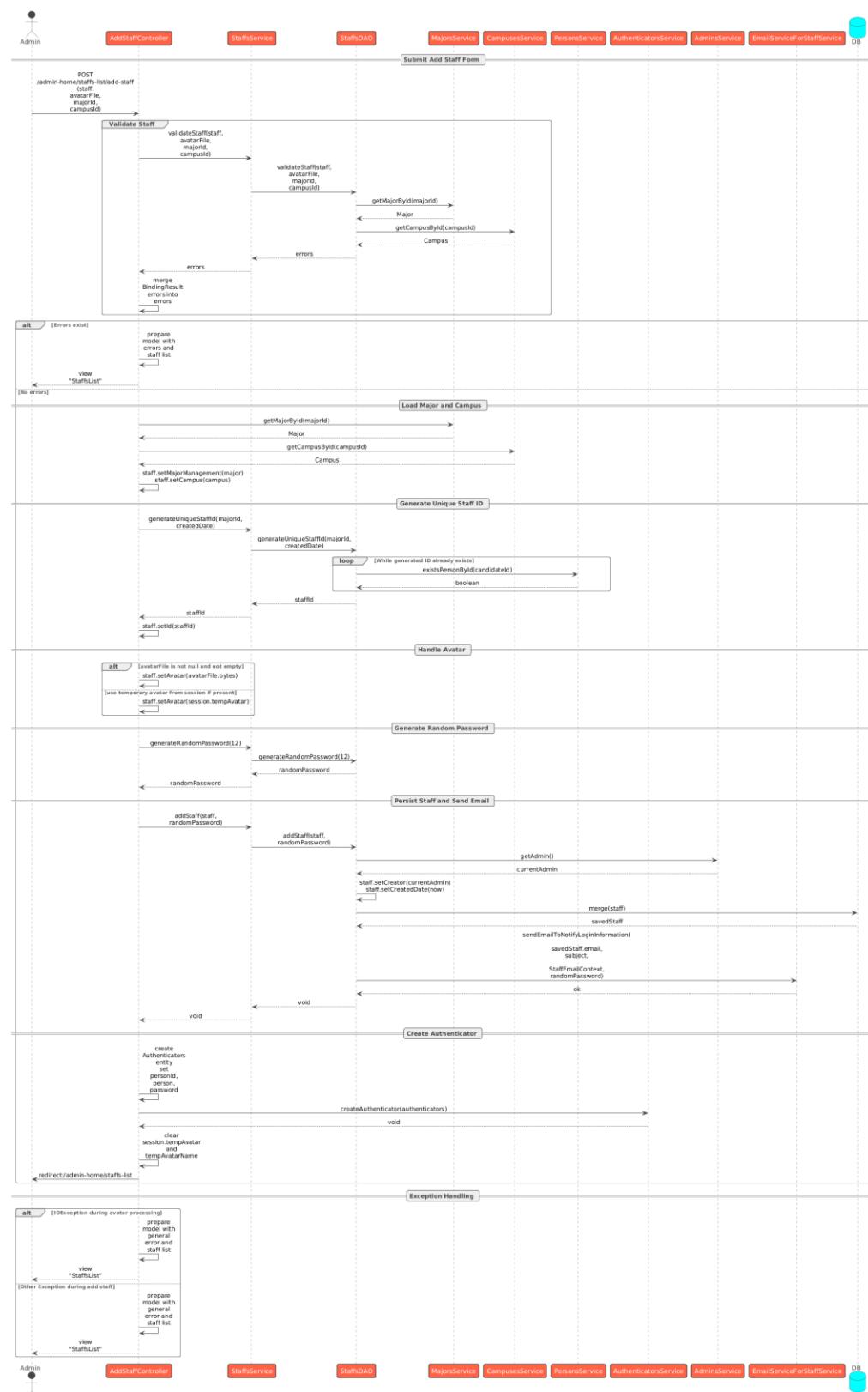


Diagram 22 Sequence Diagram 03 – Staff Onboarding, Validation, and Account Provisioning Flow

Adding new employee begins, when the Administrator sends POST request with the information about the employee, the avatar file, the department code and codebase to the AddStaffController. StaffsService is called by the Controller to do full validation of data and, as a result, transmitted to StaffsDAO, which first verifies the validity of MajorId and campusId by calling MajorsService and CampusesService, and then carries out other business checks such as duplicate email, phone number format. In case of any error detected, the list of errors is sent back in the reverse order DAO - Service - Controller, Controller throws the error to the model and gives the user edited frontend employee list. Otherwise, in the event of no errors, the Controller gets the Major and Campus objects to give the employee, requests the Service to create a unique employee code by searching to see if it has already been used via PersonsService until it gets a not-used one, and then processes the avatar (reads bytes out of a file uploaded) and the random 12-character bio password. The Controller then invokes StaffsService.addStaff() which forwards the service to the DAO to save the record to the database: marking the creator as the current administrator, giving the employee a creation date, storing the employee data, invoking the EmailService to send an email to the new employee with his/her account and temporary password. Having saved the employee successfully, the Controller adds an authentication record to the Authenticators table through AuthenticatorsService, deletes the data image in the session temporarily and redirects the user to the employee list page. In the process, any exception such as an IOException during the processing of images or any other crucial exception is handled by the Controller, logged and added with an error message and re-processed by the list of interface workers. Therefore, sequence diagram can offer a sufficient number of worker stream processors of authentication, bio-encoding, storage, sending notifications to account creation and handle login in all eventualities by defining and ranking.

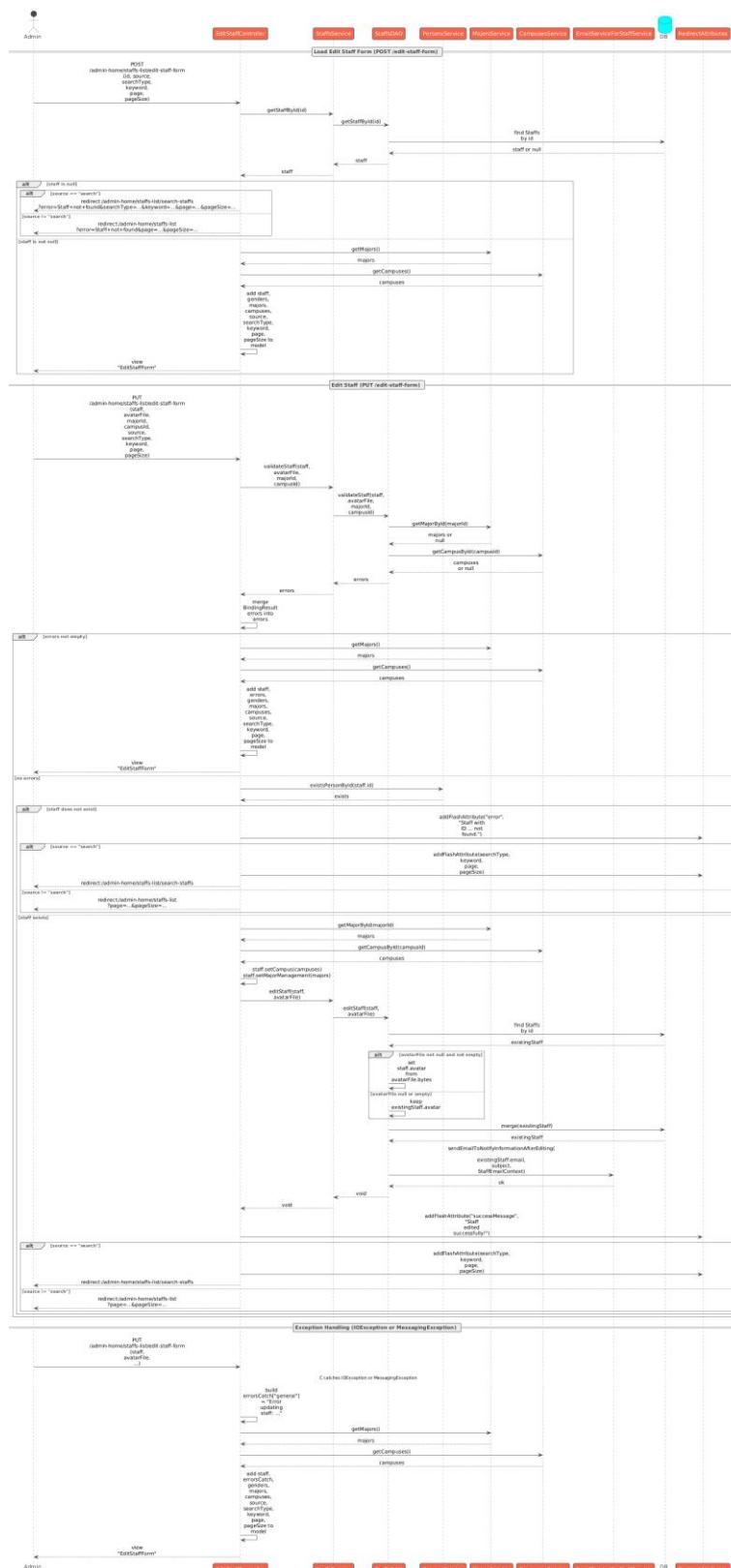


Diagram 23 Sequence Diagram 04 – Staff Editing, Validation, and Update Workflow

Managing the employee information begins with Admin clicking the "Edit" button on the list, which sends the system POST request to EditStaffController with the employee id and pagination parameters / search parameters (source, searchtype, keyword, page, page size). The Controller requests StaffsService to retrieve employee information by id which is then transferred to StaffsDAO to make a database query. In case the employee is not located, the system will automatically become redirected to the right page where Admin has already been standing in front (normal list or search results) and there is an error message saying, Staff not found. Another approach is whereby, on discovering such, the Controller will receive the entire listing of majors as provided by MajorsService and campuses as provided by CampusesService and this information is then inputted into the model with all the appropriate data (staff, gender list, majors, campuses and source parameters) and the interface returned to edit form is shown as EditStaffForm. Upon completion of Admin editing and submit button, the form generates a PUT request with the edited information (staff, new avatarFile in case of changes, majorId, campusId) and the old source parameters. The Controller calls StaffsService directly to do validation just like during the process of addition of new (checking duplicate email, format, majorId and campusId existence...). In case of any errors, the Controller re-loads the list of majors and campuses, then enters the error into the model and displays the user the form that they should edit, which is the EditStaffForm. In case no validation errors occur, the Controller verifies once again whether the employee remains in the Persons table (in case a person was deleted by another one at the same time). In case it is not available anymore, the system puts in a flash error and redirects to the right source page using the previous parameters. Assuming that it is still there, the Controller retrieves the Major and Campus object and assigns it to the staff, and invokes StaffsService.editStaff(). Move to DAO: the user uploads a new avatar: DAO gets the record of the current avatar in the database, overwrites the bytes of the new image, if the user does not, and then merges (updates) the record in the database and sends an email notification of the service to the employee. Once an update is made successfully, Controller inserts a flash message "Staff edited successfully!" and sends Admin back to the appropriate list page or search results they were on, of course with the pagination and search keywords. Throughout the PUT procedure, when an IOException (on processing an image file) or MessagingException (on sending an email fails), Controller traps the exception, recreates a general error "Error updating staff" and repopulates the list of majors and campuses, and reoffers the form "EditStaffForm" to Admin to attempt all over again without loss of the data typed. The sequence diagram, therefore, explains in detail, correctly and fully, two major flows, loading the edit form and updating the employee information, and manages effectively the occurrence of not finding the employees, validation errors, and the deleting halfway of the employees, technical errors, and always restoring the user with the initial position in the list after each action.

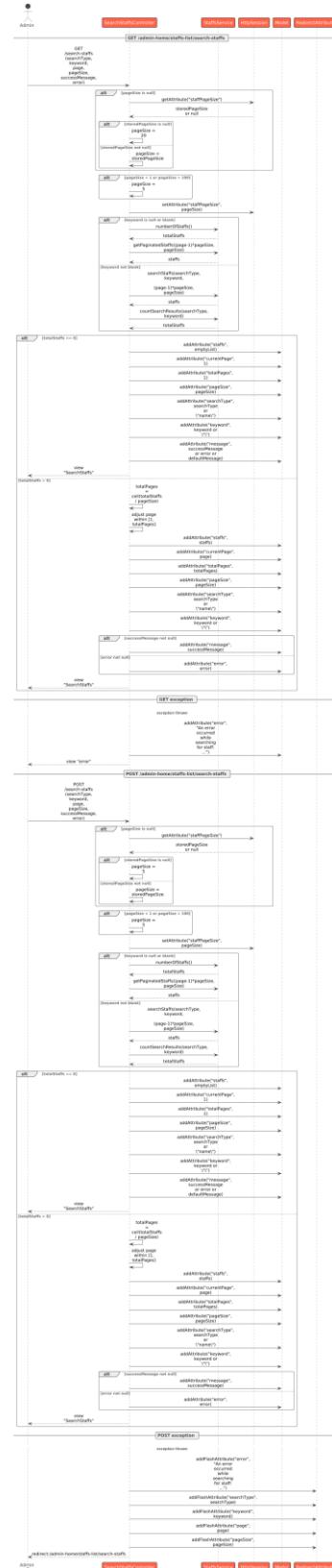


Diagram 24 Staff Search, Filtering, and Paginated Listing Flow

The process of searching and displaying the list of employees is handled uniformly through SearchStaffsController for both GET and POST requests to the path /admin-home/staffs-list/search-staffs to ensure a seamless experience whether the user changes pages or presses Enter on the search box. When the request arrives, the Controller prioritizes getting pageSize from HttpSession. If it does not exist or the value is invalid (less than 1 or greater than 100), it automatically assigns a default value (20 for GET, 5 for POST) and saves it to the session for subsequent uses without having to re-select. Then the Controller checks the keyword: if it is empty or null, it calls StaffsService to get the entire number of employees and the normal pagination list. Otherwise, if there is a keyword, it calls the search function by searchType (name, email, phone number...) and counts the number of resulting records. Whether there are results or not, the Controller always calculates the total number of pages, automatically adjusts the current page to be within the valid range, then puts all the necessary information into the Model including the employee list, current page, total number of pages, pageSize, searchType, current keyword and successMessage or error messages from the flash attribute. In case there are no records, the system still returns the SearchStaffs page with an empty list for the user to clearly recognize. Regarding exception handling, with GET, if an error occurs, add a message to the Model and return a general error page, and with POST, use RedirectAttributes to save all current search parameters and errors and then redirect back to the search page so that the user does not lose the entered conditions. Thanks to this mechanism, the user always keeps the search status, pagination and keywords after all add, edit, delete operations or when encountering errors, providing an extremely smooth and professional employee list management experience.

3.9.5 Activity Diagrams

Activity diagram related to student roles

AddStudentController:

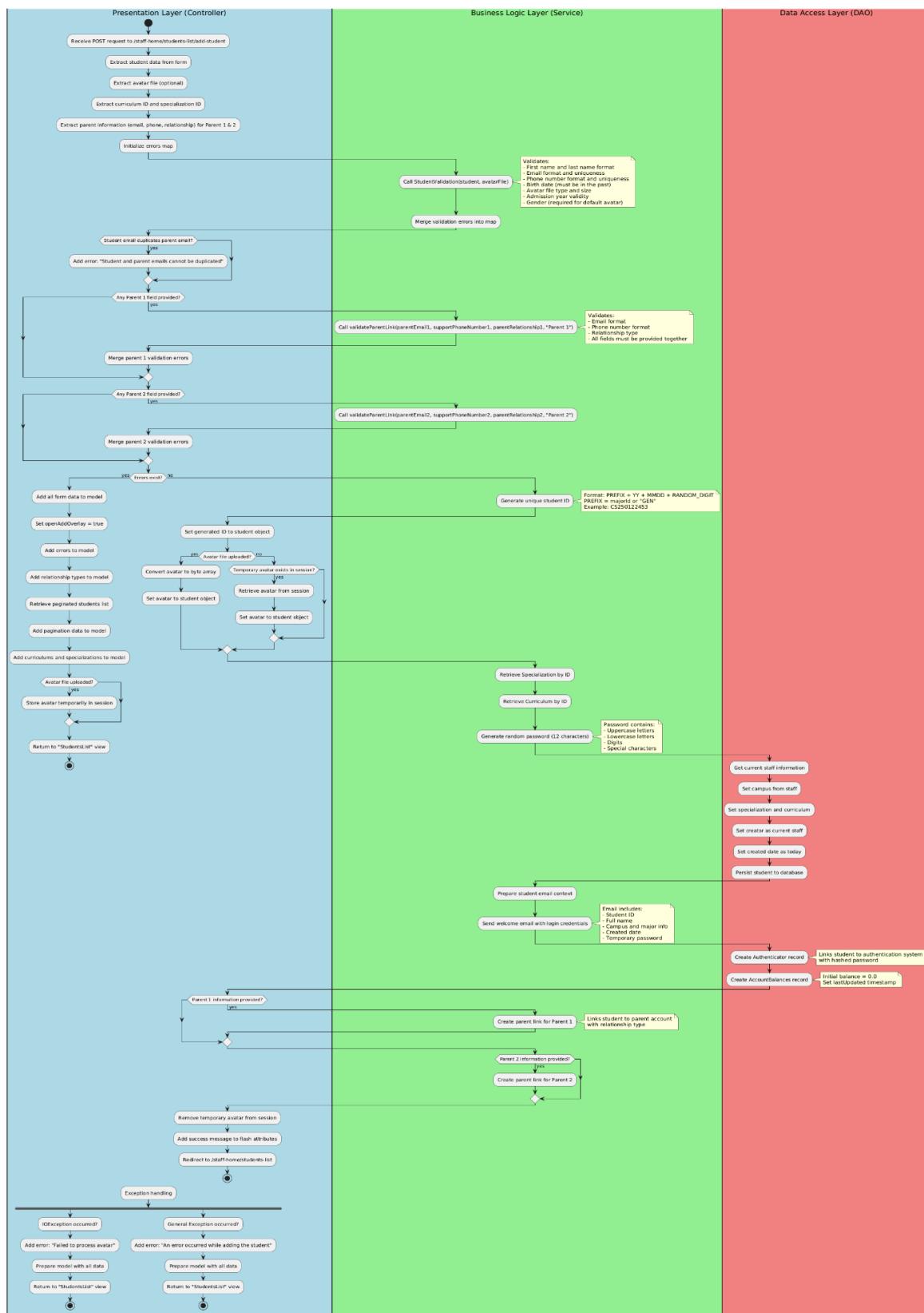


Diagram 25 Student Enrollment & Parent Registration Workflow – Three-Layer Architecture

Step 1: Receive a POST request from the browser to /staff-home/students-list/add-student.
Step 2: Extract the student's basic information from the submitted form.
Step 3: Extract the uploaded avatar file (if provided).
Step 4: Extract the curriculum ID from the form.
Step 5: Extract the specialization ID from the form.
Step 6: Extract Parent 1 information: email, phone number, and relationship.
Step 7: Extract Parent 2 information: email, phone number, and relationship.
Step 8: Initialize an empty errors map to store validation errors.

Step 9: Call the service method StudentValidation(student, avatarFile).

Step 10: Validate the student's first name format.
Step 11: Validate the student's last name format.
Step 12: Validate the student's email format.
Step 13: Check whether the student's email is unique.
Step 14: Validate the student's phone number format.
Step 15: Check whether the student's phone number is unique.
Step 16: Validate the student's date of birth (must be in the past).
Step 17: Validate the avatar upload's file type.
Step 18: Validate the avatar upload's file size.
Step 19: Validate the student's admission year.
Step 20: Validate that the student's gender is provided.
Step 21: Return all student validation errors from the service.
Step 22: Merge the student validation errors into the errors map in the controller.

Step 23: Check whether the student email matches Parent 1's email or Parent 2's email.

Step 24: If the student email duplicates any parent email, add a new error:
"Student and parent emails cannot be duplicated."

Step 25: Check whether any Parent 1 field is provided (email, phone, or relationship).
Step 26: If Parent 1 has any field provided, call the service method
validateParentLink(parentEmail1, supportPhoneNumber1, parentRelationship1, "Parent 1").
Step 27: Validate Parent 1 email format.
Step 28: Validate Parent 1 phone number format.
Step 29: Validate Parent 1 relationship type.
Step 30: Check whether all Parent 1 fields are provided together (no partial input).
Step 31: Return Parent 1 validation errors.
Step 32: Merge Parent 1 validation errors into the errors map.

Step 33: Check whether any Parent 2 field is provided (email, phone, or relationship).

Step 34: If Parent 2 has any field provided, call the service method
validateParentLink(parentEmail2, supportPhoneNumber2, parentRelationship2, "Parent 2").
Step 35: Validate Parent 2 email format.
Step 36: Validate Parent 2 phone number format.
Step 37: Validate Parent 2 relationship type.
Step 38: Check whether all Parent 2 fields are provided together.
Step 39: Return Parent 2 validation errors.

Step 40: Merge Parent 2 validation errors into the errors map.

Step 41: Check whether the errors map contains any validation errors.

Step 42: If errors exist, store all entered form data into the model.

Step 43: Set the model attribute openAddOverlay = true.

Step 44: Add all validation errors to the model.

Step 45: Add relationship types (Father/Mother/Guardian/etc.) to the model.

Step 46: Retrieve the paginated list of students.

Step 47: Add the students' pagination information to the model.

Step 48: Add curriculums and specializations to the model.

Step 49: Check whether an avatar file was uploaded.

Step 50: If avatar upload exists, temporarily store the avatar in session.

Step 51: Return the view "StudentsList" and stop the activity.

Step 52: If no errors exist, call the service method to generate a unique student ID.

Step 53: Format the ID: PREFIX + YY + MMDD + RANDOM_DIGIT.

Step 54: Assign the generated student ID to the student object.

Step 55: Check whether an avatar file was uploaded.

Step 56: If uploaded, convert the avatar file to a byte array.

Step 57: Attach the byte array as the student's avatar.

Step 58: If no avatar was uploaded, check whether a temporary avatar exists in session.

Step 59: If a temporary avatar exists, import it from the session.

Step 60: Attach the temporary avatar to the student object.

Step 61: Retrieve the Specialization by ID.

Step 62: Retrieve the Curriculum by ID.

Step 63: Generate a random 12-character password.

Step 64: Ensure the password includes uppercase letters, lowercase letters, digits, and special characters.

Step 65: Retrieve the current staff information from the DAO layer.

Step 66: Set the student's campus based on the staff's campus.

Step 67: Attach the specialization and curriculum to the student.

Step 68: Set the "created by" field to the current staff member.

Step 69: Set the student's creation date to today.

Step 70: Persist the student into the database.

Step 71: Prepare the email context for the student's welcome email.

Step 72: Send the welcome email containing ID, full name, campus, major info, creation date, and temporary password.

Step 73: Create an Authenticator record linking the student to the authentication system.

Step 74: Store the hashed password in the Authenticator entry.

Step 75: Create an AccountBalances record for the student.

Step 76: Set initial balance = 0.0.

Step 77: Set the lastUpdated timestamp.

Step 78: Check whether Parent 1 information was provided.

Step 79: If yes, call the service to create a parent link for Parent 1.

Step 80: Check whether Parent 2 information was provided.

Step 81: If yes, call the service to create a parent link for Parent 2.

Step 82: Remove any temporary avatar stored in the session.

Step 83: Add a success message to flash attributes.

Step 84: Redirect to /staff-home/students-list.

Step 85: Stop the activity.

EXCEPTION BRANCHES

Step 86: Detect whether an IOException occurred.

Step 87: Add error message: "Failed to process avatar."

Step 88: Prepare all model data needed for the StudentsList view.

Step 89: Return "StudentsList" and stop execution.

Step 90: Detect whether a general exception occurred.

Step 91: Add error message: "An error occurred while adding the student."

Step 92: Prepare all model data needed for the StudentsList view.

Step 93: Return "StudentsList" and stop execution.

EditStudentController

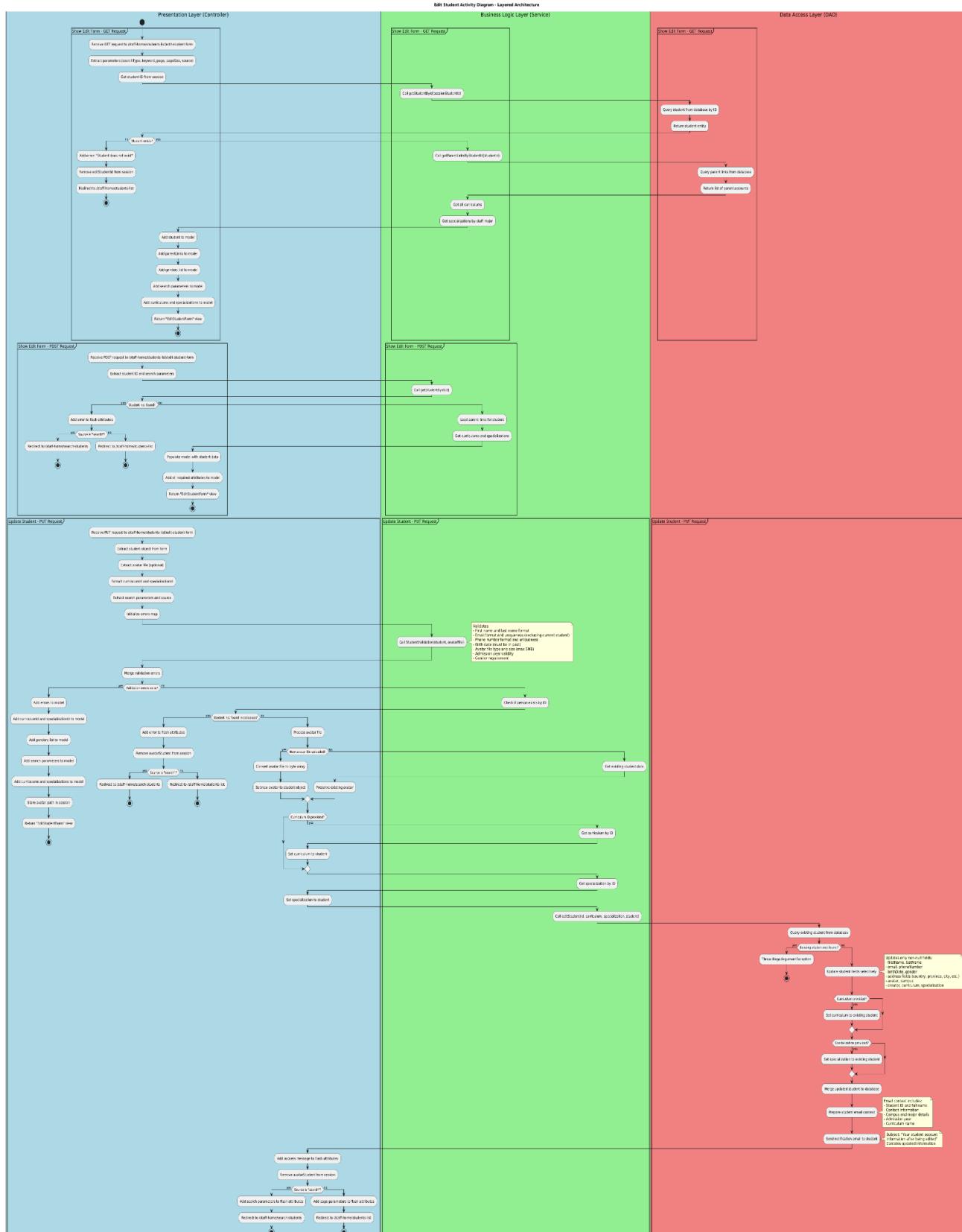


Diagram 26 Student Information Update Workflow – Three-Layer Architecture

SHOW EDIT FORM – GET REQUEST

- Step 1: Receive GET request to /staff-home/students-list/edit-student-form.
- Step 2: Extract parameters: searchType, keyword, page, pageSize, source.
- Step 3: Get student ID from session (editStudentId).
- Step 4: Call service method getStudentById(sessionStudentId).
- Step 5: DAO queries student from database by ID.
- Step 6: DAO returns student entity to service.
- Step 7: Controller checks whether the student exists.
- Step 8: If student does NOT exist, add error message: "Student does not exist".
- Step 9: Remove editStudentId from session.
- Step 10: Redirect to /staff-home/students-list.
- Step 11: Stop execution.
- Step 12: If student exists, call service method getParentLinksByStudentId(studentId).
- Step 13: DAO queries parent links from database.
- Step 14: DAO returns list of parent accounts.
- Step 15: Service retrieves all curriculums.
- Step 16: Service retrieves specializations based on staff major.
- Step 17: Controller adds student object to model.
- Step 18: Controller adds parent links to model.
- Step 19: Controller adds genders list to model.
- Step 20: Controller adds search parameters to model.
- Step 21: Controller adds curriculums and specializations to model.
- Step 22: Controller returns the "EditStudentForm" view.
- Step 23: Stop execution.

SHOW EDIT FORM – POST REQUEST

- Step 24: Receive POST request to /staff-home/students-list/edit-student-form.
- Step 25: Extract student ID and search parameters.
- Step 26: Call service method getStudentById(id).
- Step 27: Controller checks whether student exists.
- Step 28: If student does NOT exist, add error to flash attributes.
- Step 29: Check whether source == "search".
- Step 30: If source is "search", redirect to /staff-home/search-students.
- Step 31: Stop execution.
- Step 32: If source is NOT "search", redirect to /staff-home/students-list.
- Step 33: Stop execution.
- Step 34: If student exists, service loads parent links.
- Step 35: Service loads curriculums and specializations.
- Step 36: Controller populates model with student data.
- Step 37: Controller adds all required attributes to model.
- Step 38: Controller returns the "EditStudentForm" view.
- Step 39: Stop execution.

UPDATE STUDENT – PUT REQUEST

- Step 40: Receive PUT request to /staff-home/students-list/edit-student-form.
- Step 41: Extract student object from form.

Step 42: Extract avatar file (optional).
Step 43: Extract curriculumId and specializationId.
Step 44: Extract search parameters and source.
Step 45: Initialize an empty errors map.
Step 46: Call service method StudentValidation(student, avatarFile).
Step 47: Validate first name format.
Step 48: Validate last name format.
Step 49: Validate student email format.
Step 50: Check uniqueness of email (excluding current student).
Step 51: Validate phone number format.
Step 52: Check uniqueness of phone number.
Step 53: Validate date of birth (must be in the past).
Step 54: Validate avatar file type.
Step 55: Validate avatar file size (max 5MB).
Step 56: Validate admission year.
Step 57: Validate gender requirement.
Step 58: Service returns validation errors.
Step 59: Controller merges validation errors into errors.
Step 60: Controller checks if validation errors exist.
Step 61: If errors exist, add errors to model.
Step 62: Add curriculumId and specializationId to model.
Step 63: Add genders list to model.
Step 64: Add search parameters to model.
Step 65: Add curriculums and specializations to model.
Step 66: Store avatar path in session.
Step 67: Return "EditStudentForm" view.
Step 68: Stop execution.

UPDATE STUDENT WHEN NO VALIDATION ERRORS

Step 69: Call service method checkPersonExistsById(id).
Step 70: If student not found in database, add error to flash attributes.
Step 71: Remove avatarStudent from session.
Step 72: Check whether source == "search".
Step 73: If yes, redirect to /staff-home/search-students.
Step 74: Stop execution.
Step 75: If no, redirect to /staff-home/students-list.
Step 76: Stop execution.

PROCESS AVATAR

Step 77: Process avatar file.
Step 78: Check if a new avatar file was uploaded.
Step 79: If yes, convert avatar file to byte array.
Step 80: Set new avatar on student object.
Step 81: If no new avatar file was uploaded, call service to get existing student data.
Step 82: Preserve existing avatar from database.

APPLY CURRICULUM & SPECIALIZATION

- Step 83: Check if curriculum ID was provided.
 - Step 84: If yes, call service to get curriculum by ID.
 - Step 85: Set curriculum to student object.
 - Step 86: Call service to get specialization by ID.
 - Step 87: Set specialization to student object.
-

EXECUTE EDIT STUDENT SERVICE

- Step 88: Call service method editStudent(id, curriculum, specialization, student).
 - Step 89: DAO queries existing student by ID.
 - Step 90: If existing student not found, throw IllegalArgumentException.
 - Step 91: Stop execution.
 - Step 92: If existing student found, update fields selectively.
 - Step 93: Check if curriculum was provided; if yes, set curriculum.
 - Step 94: Check if specialization was provided; if yes, set specialization.
 - Step 95: Merge updated student into database.
 - Step 96: Prepare student email context.
 - Step 97: Send notification email containing updated student info.
-

FINAL CONTROLLER OUTPUT

- Step 98: Add success message to flash attributes.
 - Step 99: Remove avatarStudent from session.
 - Step 100: Check whether source == "search".
 - Step 101: If yes, add search parameters to flash attributes.
 - Step 102: Redirect to /staff-home/search-students.
 - Step 103: Stop execution.
 - Step 104: If no, add page parameters to flash attributes.
 - Step 105: Redirect to /staff-home/students-list.
 - Step 106: Stop execution.
-

EXCEPTION HANDLING

- Step 107: Catch all exceptions.
 - Step 108: Check if an IOException occurred.
 - Step 109: Add error: "Failed to process avatar".
 - Step 110: Check if a DataAccessException occurred.
 - Step 111: Add error: "Database error".
 - Step 112: Check if a general exception occurred.
 - Step 113: Add error: "Unexpected error".
 - Step 114: Add all errors to model.
 - Step 115: Add curriculumId and specializationId to model.
 - Step 116: Add genders list to model.
 - Step 117: Add curriculums and specializations to model.
 - Step 118: Store avatar path in session.
 - Step 119: Return "EditStudentForm" view.
 - Step 120: Stop execution.
- SearchStudentsController

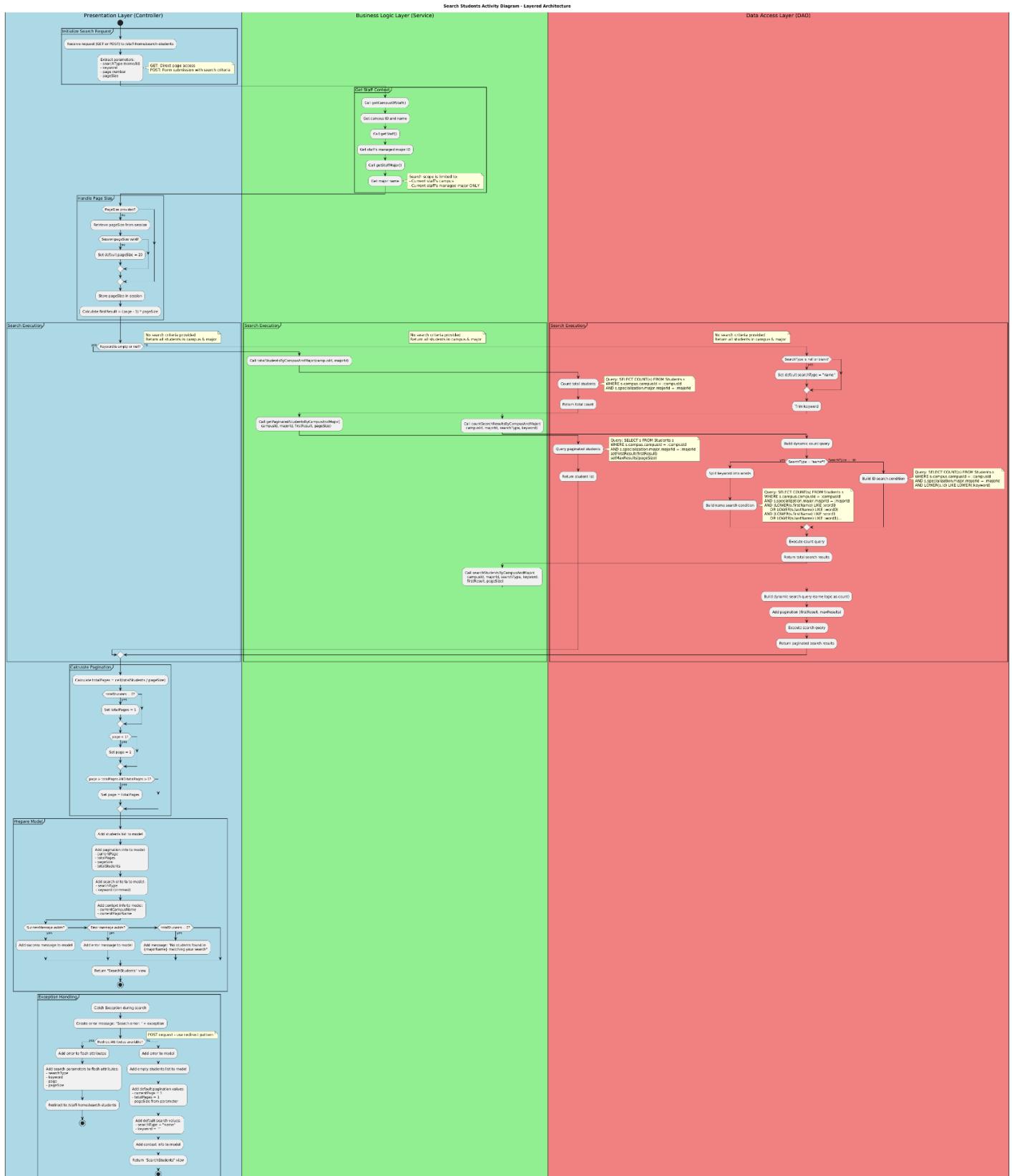


Diagram 27 Student Deletion Workflow – Three-Layer Architecture

INITIALIZE SEARCH REQUEST

Step 1: Receive GET or POST request to /staff-home/search-students.

Step 2: Extract parameter searchType (name/id).

Step 3: Extract parameter keyword.

Step 4: Extract parameter page.

Step 5: Extract parameter pageSize.

Step 6: Note request type:

GET = direct page access,

POST = form submission with search criteria.

GET STAFF CONTEXT

Step 7: Call getCampusOfStaff() in the service.

Step 8: Retrieve campus ID and campus name.

Step 9: Call getStaff().

Step 10: Retrieve the staff's managed major ID.

Step 11: Call getStaffMajor().

Step 12: Retrieve major name.

Step 13: Apply search restriction:

search is limited to this campus only and this major only.

HANDLE PAGE SIZE

Step 14: Check if pageSize is provided.

Step 15: If pageSize is NOT provided, retrieve pageSize from session.

Step 16: Check if session pageSize is valid.

Step 17: If session pageSize is not valid, set default pageSize = 20.

Step 18: Store pageSize in session.

Step 19: Calculate firstResult = (page - 1) * pageSize.

SEARCH EXECUTION – CASE 1: NO KEYWORD

Step 20: Check if keyword is empty or null.

Step 21: If keyword is empty/null, proceed with full list search for campus + major.

Step 22: Call service totalStudentsByCampusAndMajor(campusId, majorId).

Step 23: DAO counts all students matching campus and major.

Step 24: DAO returns total student count.

Step 25: Call service

getPaginatedStudentsByCampusAndMajor(campusId, majorId, firstResult, pageSize).

Step 26: DAO builds query for paginated students.

Step 27: DAO applies pagination:

setFirstResult(firstResult), setMaxResults(pageSize).

Step 28: DAO executes query for student list.

Step 29: DAO returns paginated student list.

SEARCH EXECUTION – CASE 2: WITH KEYWORD

Step 30: If keyword is NOT empty, execute search with criteria.

Step 31: Check if searchType is null or blank.

Step 32: If searchType is blank, set default searchType = "name".

Step 33: Trim keyword.

Step 34: Call service

countSearchResultsByCampusAndMajor(campusId, majorId, searchType, keyword).

Step 35: DAO builds dynamic COUNT query.

Step 36: Check if searchType = "name".

Step 37: If searchType = "name", split keyword into separate words.

Step 38: Build name search conditions using LIKE LOWER(:wordX).

Step 39: If searchType = "id", build ID search condition using
LOWER(s.id) LIKE LOWER(:keyword).

Step 40: Execute count query.

Step 41: DAO returns total search result count.

Step 42: Call service

searchStudentsByCampusAndMajor(campusId, majorId, searchType, keyword, firstResult, pageSize).

Step 43: DAO builds dynamic search query.

Step 44: DAO applies pagination values.

Step 45: DAO executes search query.

Step 46: DAO returns paginated search results.

CALCULATE PAGINATION

Step 47: Calculate totalPages = ceil(totalStudents / pageSize).

Step 48: Check if totalStudents = 0.

Step 49: If totalStudents = 0, set totalPages = 1.

Step 50: Check if page < 1.

Step 51: If page < 1, set page = 1.

Step 52: Check if page > totalPages AND totalPages > 0.

Step 53: If true, set page = totalPages.

PREPARE MODEL FOR VIEW

Step 54: Add students list to model.

Step 55: Add pagination info to model:

currentPage, totalPages, pageSize, totalStudents.

Step 56: Add search criteria to model:

searchType, trimmed keyword.

Step 57: Add campus name to model.

Step 58: Add major name to model.

Step 59: Check if success message exists.

Step 60: If success message exists, add success message to model.

Step 61: Else check if error message exists.

Step 62: If error message exists, add error message to model.

Step 63: Else check if totalStudents = 0.

Step 64: If totalStudents = 0, add message:

"No students found in {majorName} matching your search".

Step 65: Return "SearchStudents" view.

Step 66: Stop execution.

EXCEPTION HANDLING

Step 67: Catch any exception during search.
Step 68: Create error message: "Search error: " + exception.
Step 69: Check if RedirectAttributes are available.
Step 70: If redirect attributes available (POST request), add error to flash attributes.
Step 71: Add search parameters (searchType, keyword, page, pageSize) to flash attributes.
Step 72: Redirect to /staff-home/search-students.
Step 73: Stop execution.
Step 74: If redirect attributes NOT available (GET request), add error to model.
Step 75: Add empty students list to model.
Step 76: Add default pagination values:
currentPage = 1, totalPages = 1, pageSize = given pageSize.
Step 77: Add default search values:
searchType = "name", keyword = "".
Step 78: Add campus and major info to model.
Step 79: Return "SearchStudents" view.
Step 80: Stop execution.

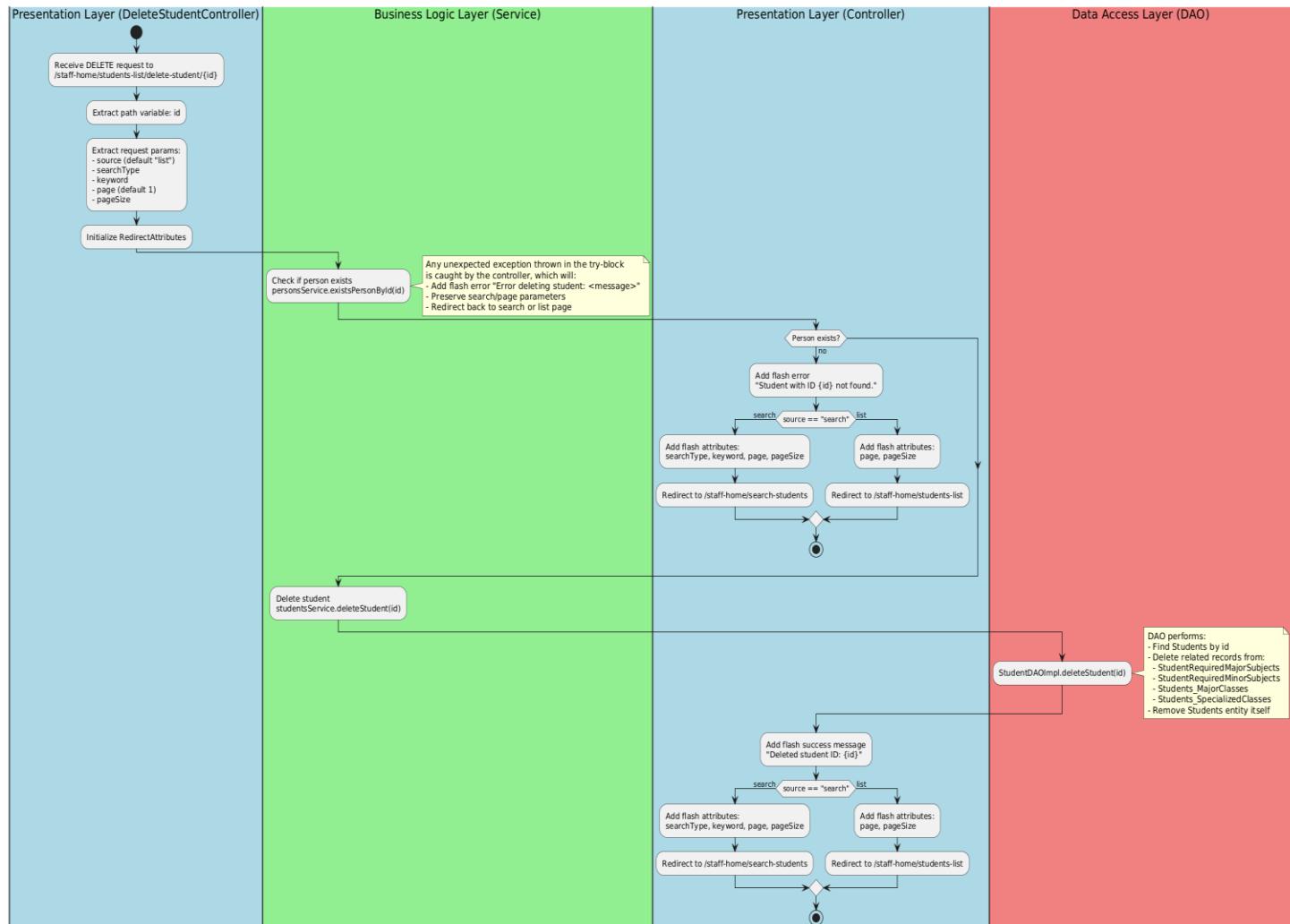


Diagram 28 Simple Student Deletion Flow – Layered Architecture

Step 1: Receive DELETE request to /staff-home/students-list/delete-student/{id}.

Step 2: Extract path variable id.

Step 3: Extract request parameter source (default "list").

Step 4: Extract request parameter searchType.

Step 5: Extract request parameter keyword.

Step 6: Extract request parameter page (default 1).

Step 7: Extract request parameter pageSize.

Step 8: Initialize RedirectAttributes for flash messages and parameter persistence.

VALIDATE STUDENT EXISTS

Step 9: Call service method personsService.existsPersonById(id).

Step 10: Controller checks whether the person exists.

BRANCH: STUDENT DOES NOT EXIST

Step 11: If person does NOT exist, add flash error message:

"Student with ID {id} not found."

Step 12: Check if source == "search".

Step 13: If source == "search", add flash attributes:
searchType, keyword, page, pageSize.

Step 14: Redirect to /staff-home/search-students.

Step 15: Stop execution.

Step 16: If source != "search", add flash attributes:
page, pageSize.

Step 17: Redirect to /staff-home/students-list.

Step 18: Stop execution.

BRANCH: STUDENT EXISTS — PERFORM DELETE

Step 19: Call service method studentsService.deleteStudent(id).

DAO DELETE TRANSACTION

Step 20: DAO method StudentDAOImpl.deleteStudent(id) is called.

Step 21: DAO finds student by ID.

Step 22: DAO deletes related records in table StudentRequiredMajorSubjects.

Step 23: DAO deletes related records in table StudentRequiredMinorSubjects.

Step 24: DAO deletes related records in table Students_MajorClasses.

Step 25: DAO deletes related records in table Students_SpecializedClasses.

Step 26: DAO removes the Students entity from database.

POST-DELETE SUCCESS FLOW

Step 27: Add flash success message:

"Deleted student ID: {id}".

Step 28: Check if source == "search".

Step 29: If source == "search", add flash attributes:
searchType, keyword, page, pageSize.

Step 30: Redirect to /staff-home/search-students.

Step 31: Stop execution.

Step 32: If source != "search", add flash attributes:
page, pageSize.

Step 33: Redirect to /staff-home/students-list.

Step 34: Stop execution.

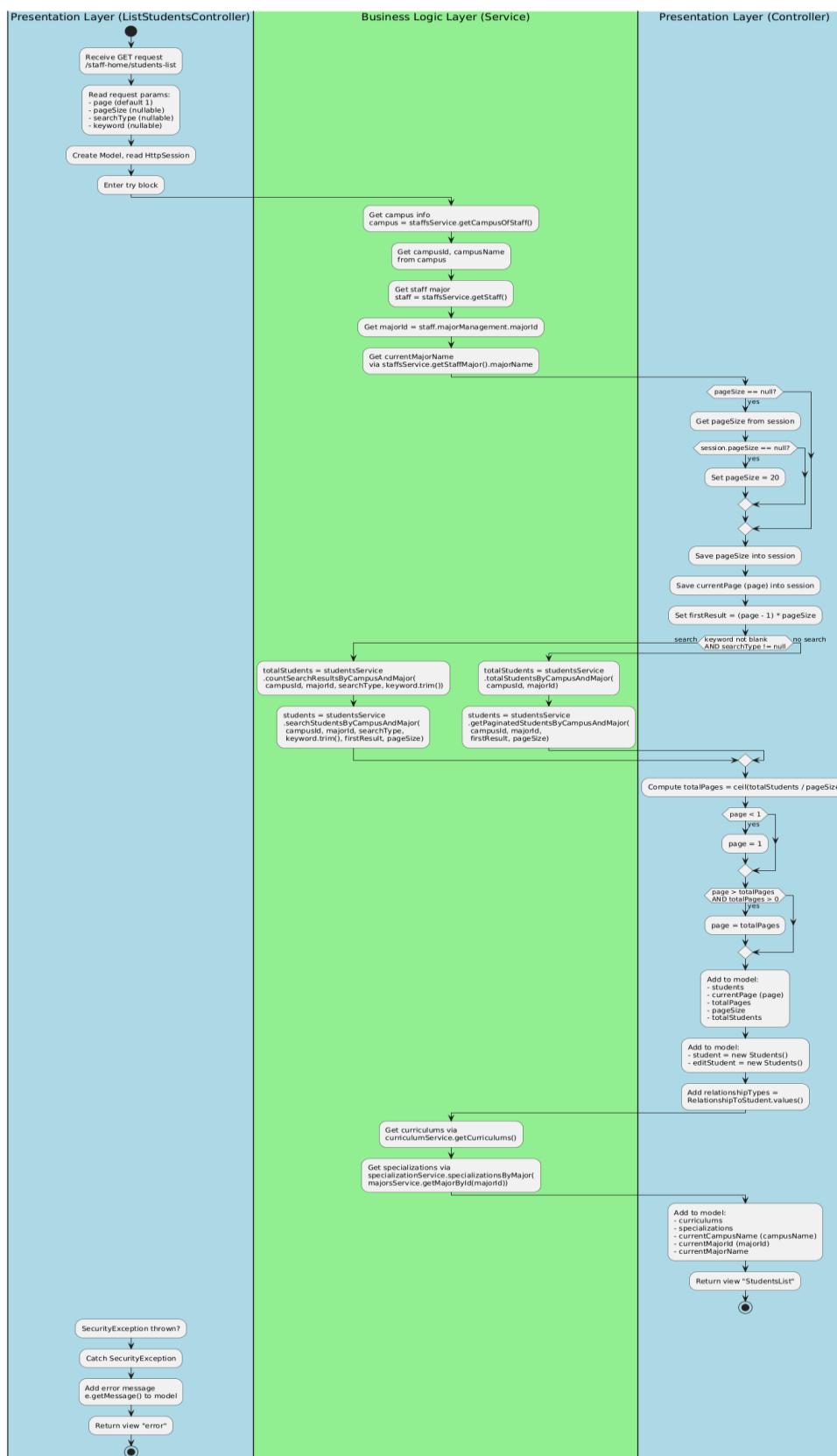


Diagram 29 Student Listing Workflow – Three-Layer Architecture

MAIN LIST-STUDENTS FLOW

- Step 1: Receive GET request to /staff-home/students-list.
 - Step 2: Read request parameter page (default = 1).
 - Step 3: Read request parameter pageSize (nullable).
 - Step 4: Read request parameter searchType (nullable).
 - Step 5: Read request parameter keyword (nullable).
 - Step 6: Create a new Model object.
 - Step 7: Read the current HttpSession.
 - Step 8: Enter try-block.
-

LOAD STAFF & CAMPUS CONTEXT

- Step 9: Call staffsService.getCampusOfStaff().
 - Step 10: Retrieve campus ID from campus.
 - Step 11: Retrieve campus name from campus.
 - Step 12: Call staffsService.getStaff().
 - Step 13: Retrieve staff's managed major from staff.
 - Step 14: Retrieve majorId from staff.majorManagement.
 - Step 15: Call staffsService.getStaffMajor().
 - Step 16: Retrieve current major name.
-

HANDLE PAGE SIZE

- Step 17: Check if pageSize == null.
 - Step 18: If pageSize == null, get pageSize from session.
 - Step 19: Check if session.pageSize == null.
 - Step 20: If session.pageSize == null, set pageSize = 20.
 - Step 21: Save pageSize into session.
 - Step 22: Save current page (page) into session.
 - Step 23: Compute firstResult = (page - 1) * pageSize.
-

CHECK SEARCH MODE

- Step 24: Check if keyword is not blank AND searchType != null.
-

BRANCH A – SEARCH MODE

- Step 25: Call
`studentsService.countSearchResultsByCampusAndMajor(campusId, majorId, searchType, keyword.trim())`.
 - Step 26: Save result into totalStudents.
 - Step 27: Call
`studentsService.searchStudentsByCampusAndMajor(campusId, majorId, searchType, keyword.trim(), firstResult, pageSize)`.
 - Step 28: Save returned student list into students.
-

BRANCH B – NO SEARCH MODE

- Step 29: Call
`studentsService.totalStudentsByCampusAndMajor(campusId, majorId)`.
 - Step 30: Save result into totalStudents.
-

Step 31: Call
studentsService.getPaginatedStudentsByCampusAndMajor(campusId, majorId, firstResult, pageSize).
Step 32: Save returned student list into students.

PAGINATION CALCULATIONS

Step 33: Compute totalPages = ceil(totalStudents / pageSize).
Step 34: Check if page < 1.
Step 35: If page < 1, set page = 1.
Step 36: Check if page > totalPages AND totalPages > 0.
Step 37: If true, set page = totalPages.

POPULATE MODEL

Step 38: Add students to model.
Step 39: Add currentPage = page to model.
Step 40: Add totalPages to model.
Step 41: Add pageSize to model.
Step 42: Add totalStudents to model.
Step 43: Add placeholder student = new Students() to model.
Step 44: Add placeholder editStudent = new Students() to model.
Step 45: Add relationshipTypes = RelationshipToStudent.values() to model.

LOAD CURRICULUM & SPECIALIZATION

Step 46: Call curriculumService.getCurriculums().
Step 47: Retrieve curriculums list.
Step 48: Call majorsService.getMajorById(majorId).
Step 49: Call
specializationService.specializationsByMajor(returnedMajor).
Step 50: Retrieve specialization list.

ADD CONTEXT TO MODEL

Step 51: Add curriculums to model.
Step 52: Add specializations to model.
Step 53: Add currentCampusName = campusName to model.
Step 54: Add currentMajorId = majorId to model.
Step 55: Add currentMajorName to model.

RETURN VIEW

Step 56: Return "StudentsList" view.
Step 57: Stop execution.

EXCEPTION PATH — SECURITY EXCEPTION

Step 58: A SecurityException is thrown.
Step 59: Catch the SecurityException.
Step 60: Add error message e.getMessage() to model.
Step 61: Return "error" view.

Step 62: Stop execution.

Activity diagram related to staff roles

AddStaffController

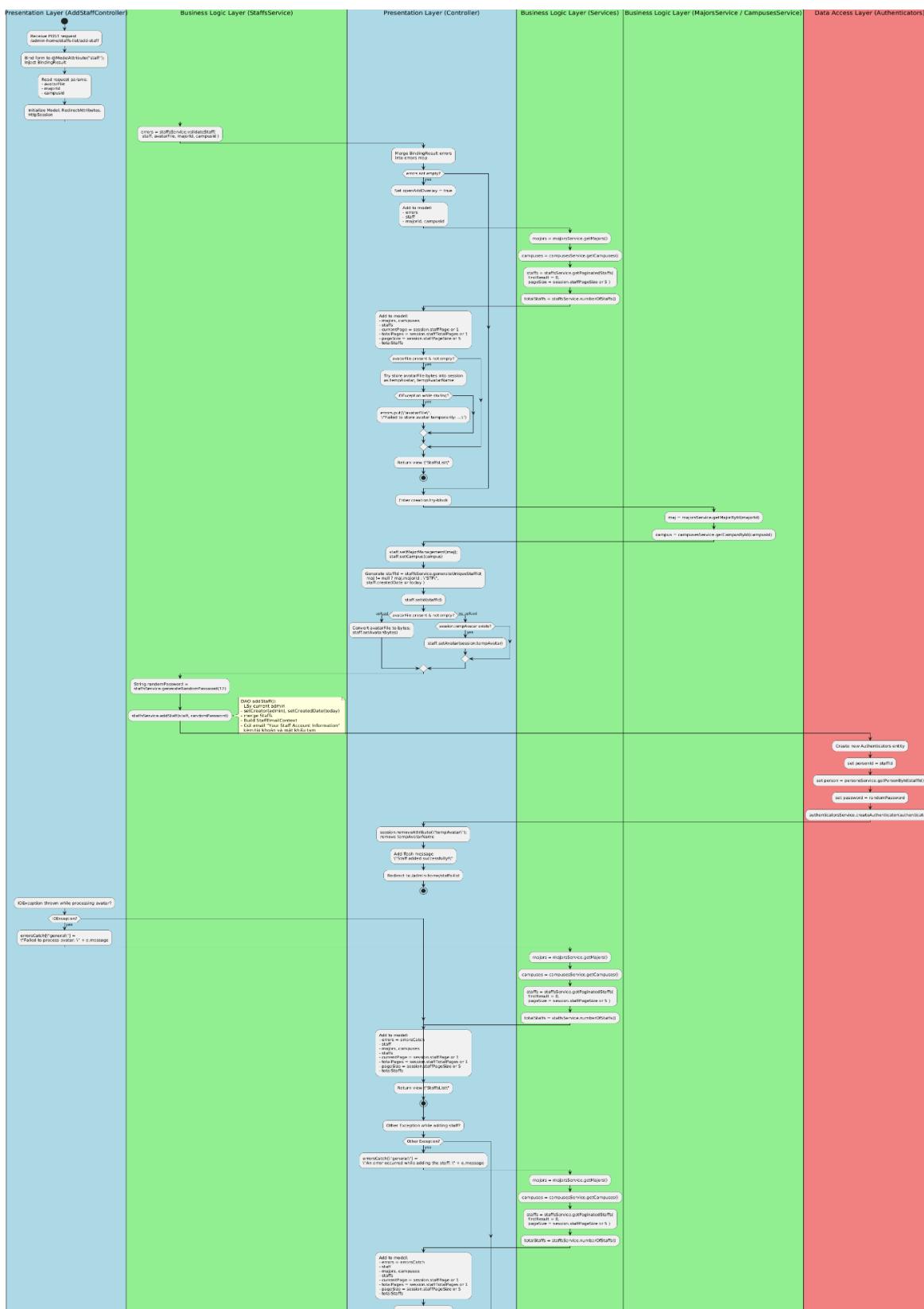


Diagram 30 Staff Creation Workflow – Layered Architecture

MAIN FLOW – RECEIVE REQUEST

- Step 1: Receive POST request to /admin-home/staffs-list/add-staff.
 - Step 2: Bind form data to @ModelAttribute("staff").
 - Step 3: Inject BindingResult for validation errors.
 - Step 4: Read request parameter avatarFile.
 - Step 5: Read request parameter majorId.
 - Step 6: Read request parameter campusId.
 - Step 7: Initialize Model.
 - Step 8: Initialize RedirectAttributes.
 - Step 9: Initialize HttpSession.
-

VALIDATION

- Step 10: Call staffsService.validateStaff(staff, avatarFile, majorId, campusId).
 - Step 11: Store returned result into errors.
 - Step 12: Merge BindingResult validation errors into errors map.
 - Step 13: Check whether errors map is NOT empty.
-

BRANCH A – VALIDATION ERRORS EXIST

- Step 14: Set openAddOverlay = true.
 - Step 15: Add errors to model.
 - Step 16: Add staff object back to model.
 - Step 17: Add majorId and campusId to model.
-

LOAD REQUIRED DROPODOWN DATA

- Step 18: Call majorsService.getMajors() and store into majors.
 - Step 19: Call campusesService.getCampuses() and store into campuses.
 - Step 20: Determine pageSize = session.staffPageSize or default 5.
 - Step 21: Call staffsService.getPaginatedStaffs(firstResult = 0, pageSize).
 - Step 22: Store returned list into staffs.
 - Step 23: Call staffsService.numberOfStaffs() → store into totalStaffs.
-

POPULATE MODEL FOR RETURN

- Step 24: Add majors to model.
 - Step 25: Add campuses to model.
 - Step 26: Add staffs to model.
 - Step 27: Add currentPage = session.staffPage or 1 to model.
 - Step 28: Add totalPages = session.staffTotalPages or 1 to model.
 - Step 29: Add pageSize = session.staffPageSize or 5 to model.
 - Step 30: Add totalStaffs to model.
-

HANDLE TEMPORARY AVATAR STORAGE

- Step 31: Check if avatarFile is present AND not empty.
 - Step 32: If true, attempt to store avatar bytes into session as tempAvatar and tempAvatarName.
 - Step 33: Check if IOException occurred while storing avatar.
 - Step 34: If IOException occurs, add error "Failed to store avatar temporarily" to errors map.
-

RETURN VIEW (VALIDATION FAILED)

Step 35: Return "StaffsList" view.

Step 36: Stop execution.

BRANCH B – VALIDATION PASSED

Step 37: Enter staff creation try-block.

LOAD MAJOR & CAMPUS ENTITIES

Step 38: Call majorsService.getMajorById(majordId) → store in maj.

Step 39: Call campusesService.getCampusById(campusId) → store in campus.

Step 40: Set staff.setMajorManagement(maj).

Step 41: Set staff.setCampus(campus).

GENERATE STAFF ID

Step 42: Call staffsService.generateUniqueStaffId(prefix, createdDate)

Prefix = majordId if major exists, otherwise "STF".

Step 43: Set generated ID into staff.setId(staffId).

PROCESS AVATAR

Step 44: Check if new avatarFile is present AND not empty.

Step 45: If yes, convert avatarFile to bytes.

Step 46: Set staff.setAvatar(bytes).

Step 47: If no avatar uploaded, check if session contains tempAvatar.

Step 48: If session.tempAvatar exists, set staff.setAvatar(session.tempAvatar).

GENERATE PASSWORD + ADD STAFF

Step 49: Call staffsService.generateRandomPassword(12).

Step 50: Store result in randomPassword.

Step 51: Call staffsService.addStaff(staff, randomPassword).

DAO OPERATIONS – ADD STAFF

Step 52: DAO retrieves current admin.

Step 53: DAO sets creator = admin.

Step 54: DAO sets createdDate = today.

Step 55: DAO persists (merge) Staffs entity.

Step 56: DAO builds StaffEmailContext.

Step 57: DAO sends email "Your Staff Account Information".

CREATE AUTHENTICATOR ENTRY

Step 58: Create new Authenticators entity.

Step 59: Set personId = staffId.

Step 60: Call personsService.getPersonById(staffId) and set into authenticator.

Step 61: Set password = randomPassword.

Step 62: Call authenticatorsService.createAuthenticator(authenticators).

FINAL SUCCESS FLOW

- Step 63: Remove tempAvatar from session.
- Step 64: Remove tempAvatarName from session.
- Step 65: Add flash message "Staff added successfully!".
- Step 66: Redirect to /admin-home/staffs-list.
- Step 67: Stop execution.

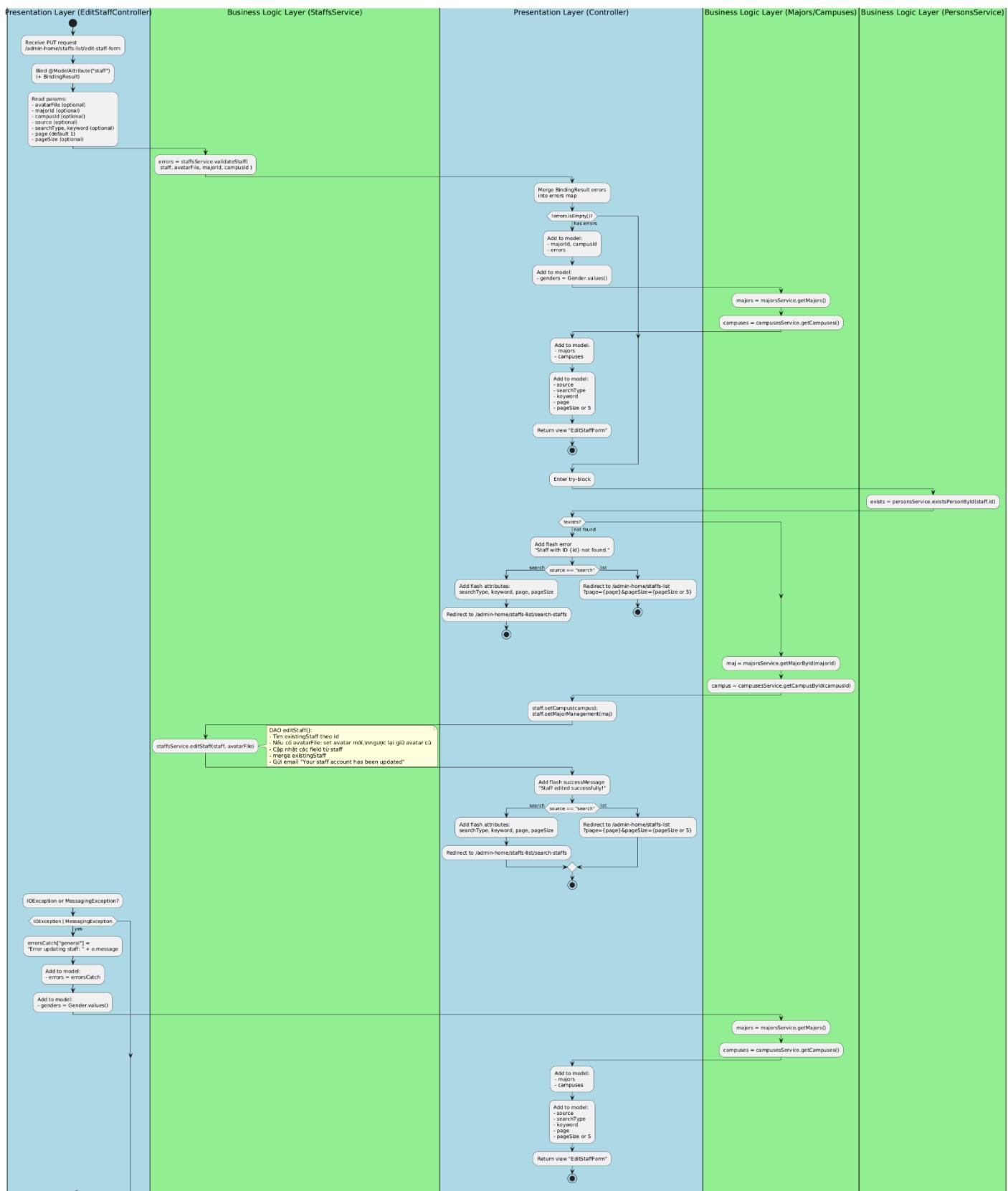


Diagram 31 Staff Update Workflow – Layered Architecture

RECEIVE REQUEST

- Step 1: Receive PUT request to /admin-home/staffs-list/edit-staff-form.
 - Step 2: Bind form data to @ModelAttribute("staff").
 - Step 3: Inject BindingResult for validation results.
 - Step 4: Read request parameter avatarFile (optional).
 - Step 5: Read request parameter majorId (optional).
 - Step 6: Read request parameter campusId (optional).
 - Step 7: Read request parameter source (optional).
 - Step 8: Read request parameter searchType (optional).
 - Step 9: Read request parameter keyword (optional).
 - Step 10: Read request parameter page (default = 1).
 - Step 11: Read request parameter pageSize (optional).
-

VALIDATION PHASE

- Step 12: Call staffsService.validateStaff(staff, avatarFile, majorId, campusId).
 - Step 13: Store returned validation errors in errors.
 - Step 14: Merge BindingResult errors into errors map.
 - Step 15: Check if errors is NOT empty.
-

BRANCH A — VALIDATION ERRORS

- Step 16: Add majorId and campusId to model.
 - Step 17: Add errors to model.
 - Step 18: Add genders = Gender.values() to model.
 - Step 19: Call majorsService.getMajors() and store in majors.
 - Step 20: Call campusesService.getCampuses() and store in campuses.
 - Step 21: Add majors to model.
 - Step 22: Add campuses to model.
 - Step 23: Add source to model.
 - Step 24: Add searchType to model.
 - Step 25: Add keyword to model.
 - Step 26: Add page to model.
 - Step 27: Add pageSize or default 5 to model.
 - Step 28: Return "EditStaffForm".
 - Step 29: Stop execution.
-

BRANCH B — VALIDATION PASSED

- Step 30: Enter try-block for staff update.
-

CHECK IF STAFF EXISTS

- Step 31: Call personsService.existsPersonById(staff.id).
 - Step 32: Store result in exists.
 - Step 33: Check if exists is false.
-

BRANCH C — STAFF NOT FOUND

- Step 34: Add flash error "Staff with ID {id} not found.".
 - Step 35: Check if source == "search".
-

Step 36: If source == "search", add flash attributes: searchType, keyword, page, pageSize.
Step 37: Redirect to /admin-home/staffs-list/search-staffs.
Step 38: Stop execution.
Step 39: If source != "search", redirect to /admin-home/staffs-list?page={page}&pageSize={pageSize or 5}.
Step 40: Stop execution.

BRANCH D — STAFF EXISTS

Step 41: Call majorsService.getMajorById(majorId).
Step 42: Store result in maj.
Step 43: Call campusesService.getCampusById(campusId).
Step 44: Store result in campus.
Step 45: Set staff.setCampus(campus).
Step 46: Set staff.setMajorManagement(maj).

UPDATE STAFF

Step 47: Call staffsService.editStaff(staff, avatarFile).

DAO BEHAVIOR (INSIDE editStaff)

Step 48: DAO finds existing staff by ID.
Step 49: DAO checks avatarFile.
Step 50: If avatarFile exists, DAO updates avatar with new bytes.
Step 51: If no avatarFile, DAO keeps existing avatar.
Step 52: DAO updates all editable fields from staff object.
Step 53: DAO merges updated staff entity.
Step 54: DAO sends email "Your staff account has been updated".

SUCCESS REDIRECT

Step 55: Add flash success message "Staff edited successfully!".
Step 56: Check if source == "search".
Step 57: If source == "search", add flash attributes: searchType, keyword, page, pageSize.
Step 58: Redirect to /admin-home/staffs-list/search-staffs.
Step 59: Stop execution.
Step 60: If source != "search", redirect to /admin-home/staffs-list?page={page}&pageSize={pageSize or 5}.
Step 61: Stop execution.

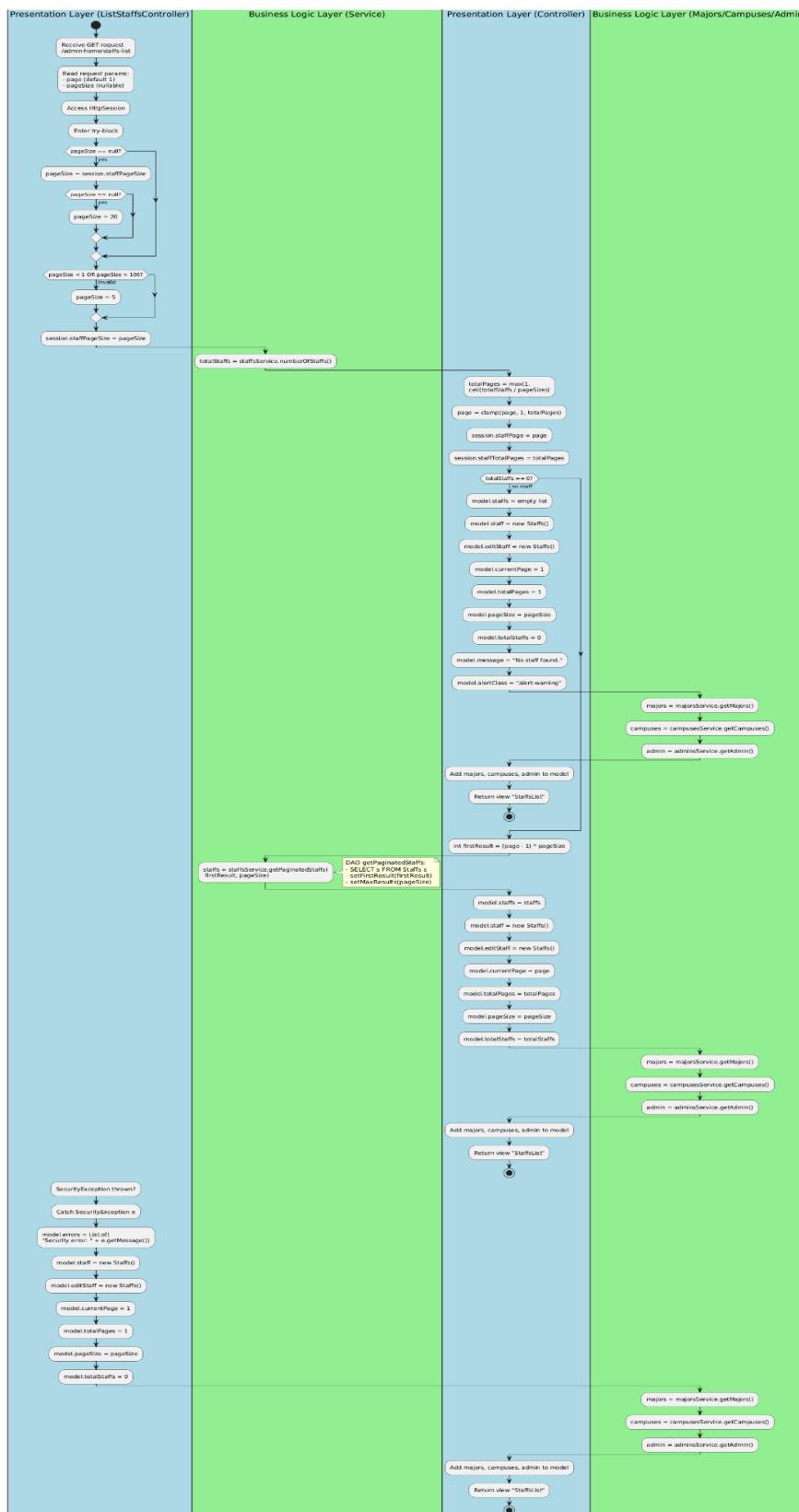


Diagram 32 Staff Listing Workflow – Layered Architecture

MAIN FLOW — LIST STAFFS

- Step 1: Receive GET request to /admin-home/staffs-list.
 - Step 2: Read request parameter page (default = 1).
 - Step 3: Read request parameter pageSize (nullable).
 - Step 4: Access the HttpSession object.
 - Step 5: Enter the try-block.
-

HANDLE PAGE SIZE INPUT

- Step 6: Check if pageSize == null.
 - Step 7: If pageSize is null, set pageSize = session.staffPageSize.
 - Step 8: Check if session.staffPageSize == null.
 - Step 9: If session.pageSize is null, set default pageSize = 20.
 - Step 10: Check if pageSize < 1 OR pageSize > 100.
 - Step 11: If invalid, set pageSize = 5.
 - Step 12: Save session.staffPageSize = pageSize.
-

COUNT TOTAL STAFF

- Step 13: Call staffsService.numberOfStaffs().
 - Step 14: Store the count in totalStaffs.
-

COMPUTE PAGINATION

- Step 15: Compute totalPages = max(1, ceil(totalStaffs / pageSize)).
 - Step 16: Clamp page = clamp(page, 1, totalPages).
 - Step 17: Save session.staffPage = page.
 - Step 18: Save session.staffTotalPages = totalPages.
-

BRANCH A — NO STAFF EXISTS

- Step 19: Check if totalStaffs == 0.
 - Step 20: If totalStaffs == 0, set model.staffs = empty list.
 - Step 21: Set model.staff = new Staffs().
 - Step 22: Set model.editStaff = new Staffs().
 - Step 23: Set model.currentPage = 1.
 - Step 24: Set model.totalPages = 1.
 - Step 25: Set model.pageSize = pageSize.
 - Step 26: Set model.totalStaffs = 0.
 - Step 27: Set model.message = "No staff found."
 - Step 28: Set model.alertClass = "alert-warning".
-

LOAD MAJORS, CAMPUSES, ADMIN FOR EMPTY LIST

- Step 29: Call majorsService.getMajors().
 - Step 30: Call campusesService.getCampuses().
 - Step 31: Call adminsService.getAdmin().
 - Step 32: Add majors to model.
 - Step 33: Add campuses to model.
 - Step 34: Add admin to model.
 - Step 35: Return view "StaffsList".
-

Step 36: Stop execution.

BRANCH B — STAFF EXISTS

Step 37: Compute firstResult = (page - 1) * pageSize.

LOAD PAGINATED STAFF

Step 38: Call

staffsService.getPaginatedStaffs(firstResult, pageSize).

Step 39: DAO executes SELECT s FROM Staffs s with pagination.

Step 40: Store returned list into staffs.

POPULATE MODEL WITH STAFF DATA

Step 41: Set model.staffs = staffs.

Step 42: Set model.staff = new Staffs().

Step 43: Set model.editStaff = new Staffs().

Step 44: Set model.currentPage = page.

Step 45: Set model.totalPages = totalPages.

Step 46: Set model.pageSize = pageSize.

Step 47: Set model.totalStaffs = totalStaffs.

LOAD MAJORS, CAMPUSES, ADMIN FOR LIST VIEW

Step 48: Call majorsService.getMajors().

Step 49: Call campusesService.getCampuses().

Step 50: Call adminsService.getAdmin().

Step 51: Add majors to model.

Step 52: Add campuses to model.

Step 53: Add admin to model.

RETURN MAIN VIEW

Step 54: Return view "StaffsList".

Step 55: Stop execution.

EXCEPTION HANDLING — SECURITYEXCEPTION

Step 56: Detect a SecurityException being thrown.

Step 57: Catch the SecurityException as e.

Step 58: Set model.errors = List.of("Security error: " + e.getMessage()).

Step 59: Set model.staff = new Staffs().

Step 60: Set model.editStaff = new Staffs().

Step 61: Set model.currentPage = 1.

Step 62: Set model.totalPages = 1.

Step 63: Set model.pageSize = pageSize.

Step 64: Set model.totalStaffs = 0.

LOAD MAJORS, CAMPUSES, ADMIN AFTER EXCEPTION

Step 65: Call majorsService.getMajors().

Step 66: Call campusesService.getCampuses().

Step 67: Call adminsService.getAdmin().
Step 68: Add majors to model.
Step 69: Add campuses to model.
Step 70: Add admin to model.
Step 71: Return "StaffsList" view.
Step 72: Stop execution.

SearchStaffsController

001479794

Page 288 | 635 total

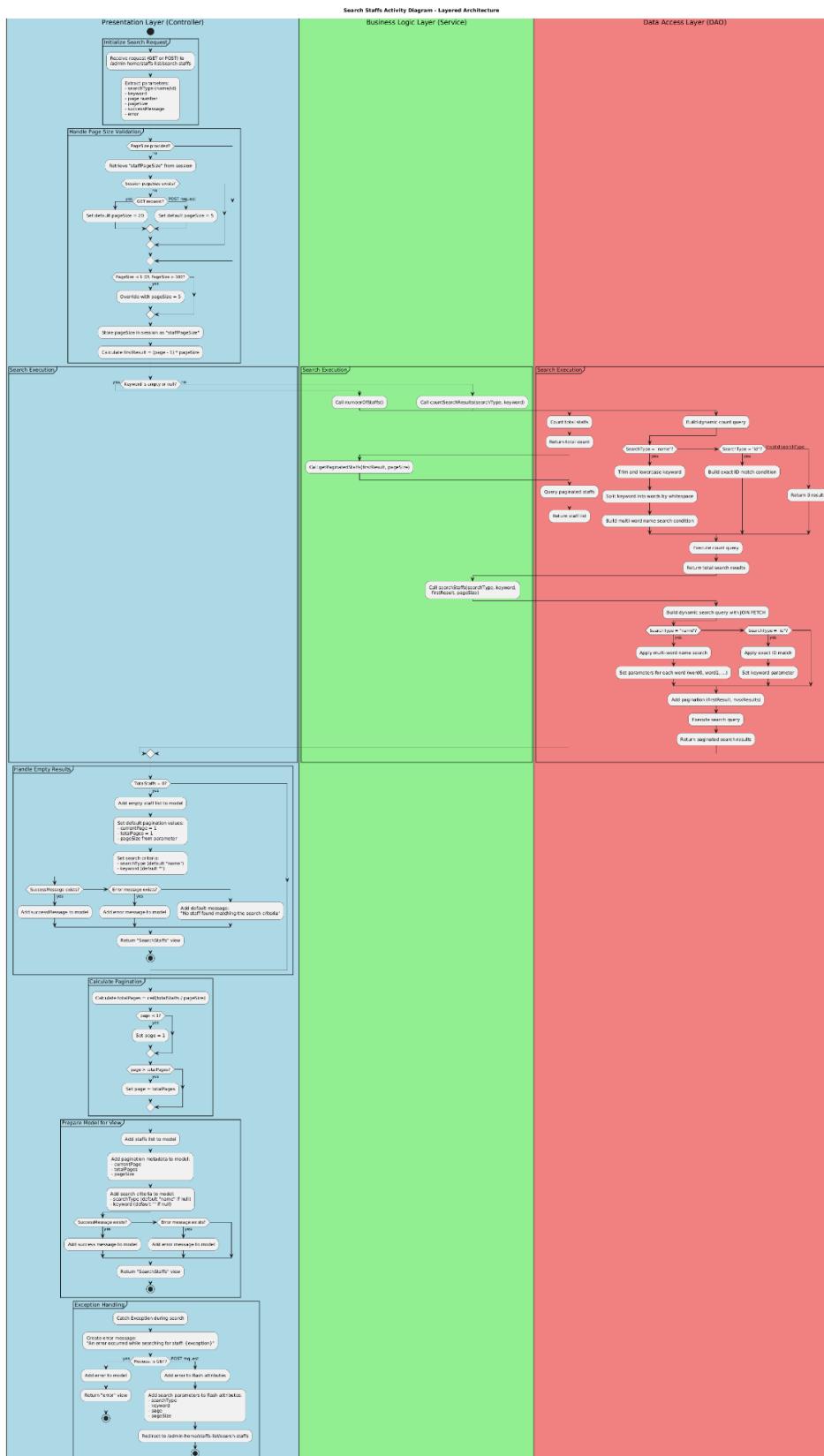


Diagram 33 Staff Search Workflow – Layered Architecture

MAIN FLOW — INITIALIZE SEARCH REQUEST

- Step 1: Receive request (GET or POST) to /admin-home/staffs-list/search-staffs.
 - Step 2: Extract request parameter searchType.
 - Step 3: Extract request parameter keyword.
 - Step 4: Extract request parameter page.
 - Step 5: Extract request parameter pageSize.
 - Step 6: Extract request parameter successMessage.
 - Step 7: Extract request parameter error.
-

HANDLE PAGE SIZE VALIDATION

- Step 8: Check if pageSize was provided in the request.
 - Step 9: If pageSize is null:
 - Step 9.1: Retrieve "staffPageSize" from session.
 - Step 9.2: If session does not contain pageSize:
 - Step 9.2.1: If request method is GET, set pageSize = 20.
 - Step 9.2.2: If request method is POST, set pageSize = 5.
 - Step 10: Validate pageSize:
 - Step 10.1: If pageSize < 1 OR pageSize > 100, set pageSize = 5.
 - Step 11: Store validated pageSize into session as "staffPageSize".
 - Step 12: Compute firstResult = (page - 1) * pageSize.
-

SEARCH EXECUTION — CASE 1: KEYWORD NOT PROVIDED

- Step 13: Check if keyword is null or empty.
 - Step 14: If keyword is null or empty:
 - Step 14.1: Call staffsService.numberOfStaffs().
 - Step 14.1.1: DAO executes count query for all staff.
 - Step 14.1.2: DAO returns total staff count.
 - Step 14.2: Call staffsService.getPaginatedStaffs(firstResult, pageSize).
 - Step 14.2.1: DAO executes paginated query.
 - Step 14.2.2: DAO returns paginated staff list.
-

SEARCH EXECUTION — CASE 2: KEYWORD PROVIDED

- Step 15: If keyword is not empty, call countSearchResults(searchType, keyword).

DAO Logic for Counting

- Step 16: DAO builds dynamic count query.
- Step 16.1: If searchType = "name":
 - Step 16.1.1: Trim keyword.
 - Step 16.1.2: Convert keyword to lowercase.
 - Step 16.1.3: Split keyword into individual words.
 - Step 16.1.4: Construct multi-word name search condition.

Step 16.2: If searchType = "id":

Step 16.2.1: Build exact ID match condition.

Step 16.3: If searchType is invalid:

Step 16.3.1: Set totalResults = 0.

Step 16.3.2: Return totalResults immediately.

Step 17: DAO executes the dynamic count query.

Step 18: DAO returns the total number of matching records.

Executing the Search Query

Step 19: Call service searchStaffs(searchType, keyword, firstResult, pageSize).

Step 20: DAO builds a dynamic SELECT query with JOIN FETCH.

Step 20.1: If searchType = "name":

Step 20.1.1: Apply multi-word search conditions.

Step 20.1.2: Bind each word parameter (word0, word1, ...).

Step 20.2: If searchType = "id":

Step 20.2.1: Apply exact ID filter.

Step 20.2.2: Bind keyword parameter.

Step 21: Apply pagination:

Step 21.1: Set query firstResult.

Step 21.2: Set query maxResults.

Step 22: Execute search query and return paginated result list.

HANDLE EMPTY RESULTS

Step 23: Check if totalStaffs == 0.

Step 24: If totalStaffs == 0:

Build Empty Model Response

Step 24.1: Add empty list to model.staffs.

Default Pagination

Step 24.2: Set default pagination values.

Step 24.2.1: Set currentPage = 1.

Step 24.2.2: Set totalPages = 1.

Step 24.2.3: Set pageSize = pageSize.

Default Search Criteria

Step 24.3: Set default search criteria.

Step 24.3.1: Set searchType = "name".

Step 24.3.2: Set keyword = "".

Message Handling

Step 24.4: If successMessage exists, add it to model.
Step 24.5: Else if error exists, add error to model.
Step 24.6: Else set default message:
"No staff found matching the search criteria."
Step 25: Return view "SearchStaffs".
Step 26: Stop execution.

CALCULATE PAGINATION (WHEN RESULTS EXIST)

Step 27: Compute totalPages = ceil(totalStaffs / pageSize).
Step 28: If page < 1, set page = 1.
Step 29: If page > totalPages, set page = totalPages.

PREPARE MODEL FOR VIEW

Step 30: Add staff list to model.
Step 31: Add pagination metadata:
Step 31.1: Add currentPage.
Step 31.2: Add totalPages.
Step 31.3: Add pageSize.
Step 32: Add search criteria:
Step 32.1: Add searchType (default "name" if null).
Step 32.2: Add keyword (default "" if null).
Step 33: If successMessage exists, add it to model.
Step 34: Else if error exists, add it to model.
Step 35: Return "SearchStaffs" view.
Step 36: Stop execution.

EXCEPTION HANDLING

Step 37: Catch any exception during search flow.
Step 38: Build error message:
"An error occurred while searching for staff: {exception}".

GET request exception flow

Step 39: If request is GET:
Step 39.1: Add error to model.
Step 39.2: Return "error" view.
Step 39.3: Stop execution.

POST request exception flow

Step 40: If request is POST:
Step 40.1: Add error to flash attributes.
Step 40.2: Add search parameters to flash attributes:

Step 40.2.1: Add searchType.
Step 40.2.2: Add keyword.
Step 40.2.3: Add page.
Step 40.2.4: Add pageSize.
Step 40.3: Redirect to /admin-home/staffs-list/search-staffs.
Step 40.4: Stop execution.

Activity diagram related to deputy staff roles

AddDeputyStaffsController

001479794

Page 293 | 635 total

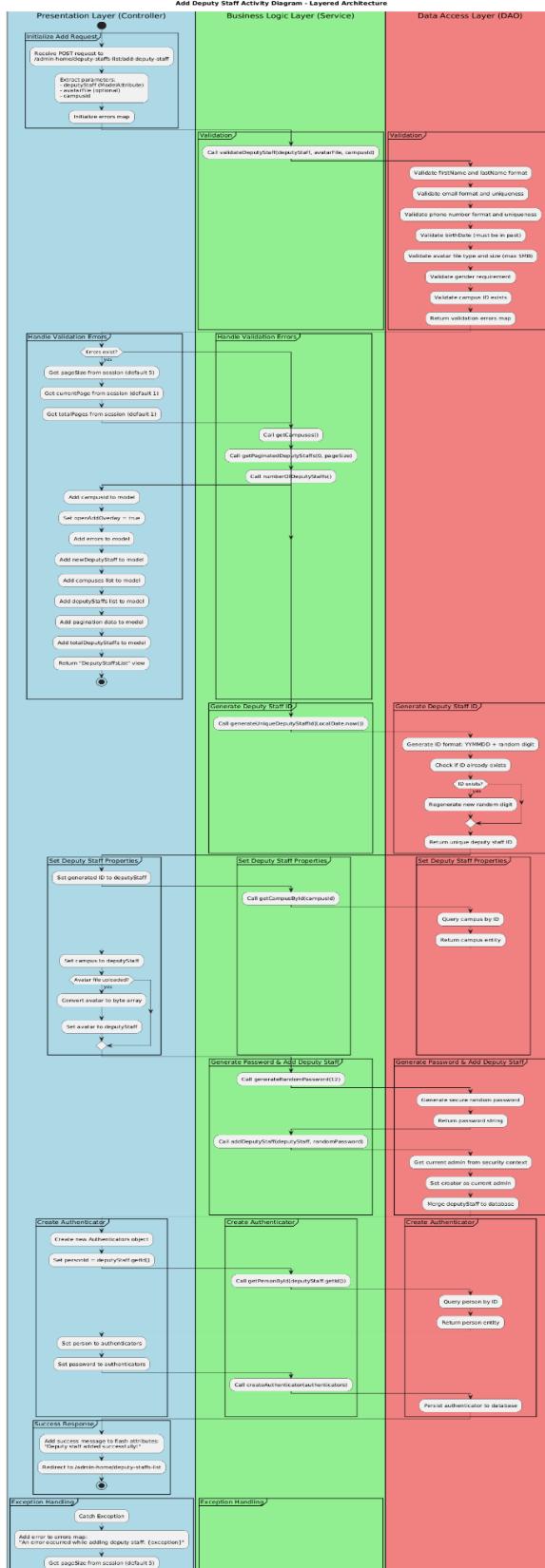


Diagram 34 Deputy Staff Creation Workflow – Layered Architecture

MAIN FLOW — INITIALIZE ADD REQUEST

- Step 1: Receive POST request to /admin-home/deputy-staffs-list/add-deputy-staff.
 - Step 2: Extract deputyStaff (ModelAttribute).
 - Step 3: Extract optional avatarFile.
 - Step 4: Extract campusId.
 - Step 5: Initialize an empty errors map.
-

VALIDATION PHASE

- Step 6: Call service validateDeputyStaff(deputyStaff, avatarFile, campusId).

DAO-Level Validation Steps

- Step 6.1: Validate firstName format.
- Step 6.2: Validate lastName format.
- Step 6.3: Validate email format.
- Step 6.4: Validate email uniqueness in database.
- Step 6.5: Validate phone number format.
- Step 6.6: Validate phone number uniqueness in database.
- Step 6.7: Validate birthDate (must be in the past).
- Step 6.8: Validate avatar file type if provided.
- Step 6.9: Validate avatar file size (max 5MB).
- Step 6.10: Validate gender requirement.
- Step 6.11: Validate that campusId exists in database.

- Step 7: Return validation errors map to controller.
-

HANDLE VALIDATION ERRORS

- Step 8: Check if validation errors exist.

- Step 9: If errors exist:

Retrieve Pagination State from Session

- Step 9.1: Get pageSize from session (default = 5).
- Step 9.2: Get currentPage from session (default = 1).
- Step 9.3: Get totalPages from session (default = 1).

Load Required Data

- Step 9.4: Call service getCampuses().
- Step 9.5: Call service getPaginatedDeputyStaffs(0, pageSize).
- Step 9.6: Call service numberOfDeputyStaffs().

Prepare Model for Error Re-Display

- Step 9.7: Add campusId to model.
- Step 9.8: Set openAddOverlay = true.
- Step 9.9: Add errors map to model.
- Step 9.10: Add newDeputyStaff to model.
- Step 9.11: Add campuses list to model.

Step 9.12: Add deputyStaffs list to model.
Step 9.13: Add pagination data to model.
Step 9.14: Add totalDeputyStaffs to model.

Step 10: Return view "DeputyStaffsList".
Step 11: Stop execution.

GENERATE UNIQUE DEPUTY STAFF ID

Step 12: Call service generateUniqueDeputyStaffId(LocalDate.now()).

DAO Logic

Step 12.1: Generate ID in format YYMMDD + random digit.
Step 12.2: Check if generated ID already exists.
Step 12.3: If ID exists, regenerate until unique.
Step 12.4: Return unique deputy staff ID.

SET DEPUTY STAFF PROPERTIES

Step 13: Assign generated ID to deputyStaff.

Load Campus Entity

Step 14: Call service getCampusById(campusId).

DAO

Step 14.1: Query campus by ID.
Step 14.2: Return campus entity.

Assign Campus

Step 15: Set campus entity into deputyStaff.

Handle Avatar Upload

Step 16: Check if avatarFile is uploaded.
Step 17: If avatar uploaded:
Step 17.1: Convert avatar file to byte array.
Step 17.2: Set avatar bytes into deputyStaff.

GENERATE PASSWORD & ADD DEPUTY STAFF

Step 18: Call service generateRandomPassword(12).

DAO

Step 18.1: Generate secure random password string.
Step 18.2: Return password.

Step 19: Call service addDeputyStaff(deputyStaff, randomPassword).

DAO

-
- Step 19.1: Retrieve current admin from security context.
 - Step 19.2: Set creator of deputyStaff as current admin.
 - Step 19.3: Persist deputyStaff entity (merge).

CREATE AUTHENTICATOR ENTRY

- Step 20: Create new Authenticators object.
 - Step 21: Set personId = deputyStaff.getId().
- Load Person Entity
- Step 22: Call service getPersonById(deputyStaff.getId()).

DAO

- Step 22.1: Query person by ID.
- Step 22.2: Return person entity.

Set Authenticator Details

- Step 23: Assign person entity into authenticators.
- Step 24: Set password into authenticators.

Persist Authenticator

- Step 25: Call service createAuthenticator(authenticators).
- DAO
- Step 25.1: Persist authenticator entry into database.
-

SUCCESS RESPONSE

- Step 26: Add success flash message:
“Deputy staff added successfully!”
- Step 27: Redirect to /admin-home/deputy-staffs-list.
- Step 28: Stop execution.
-

EXCEPTION HANDLING

- Step 29: Catch any exception thrown during process.
- Step 30: Add error message to errors map:
“An error occurred while adding deputy staff: {exception}”

Retrieve Pagination State

- Step 31: Get pageSize from session (default 5).
- Step 32: Get currentPage from session (default 1).
- Step 33: Get totalPages from session (default 1).

Load Required Data

Step 34: Call getCampuses().
Step 35: Call getPaginatedDeputyStaffs(0, pageSize).
Step 36: Call numberOfDeputyStaffs().

Prepare Model for Exception View

Step 37: Set openAddOverlay = true.
Step 38: Add errors map to model.
Step 39: Add newDeputyStaff to model.
Step 40: Add campuses list to model.
Step 41: Add deputyStaffs list to model.
Step 42: Add pagination data to model.
Step 43: Add totalDeputyStaffs to model.

Step 44: Return "DeputyStaffsList" view.
Step 45: Stop execution.

EditDeputyStaffController

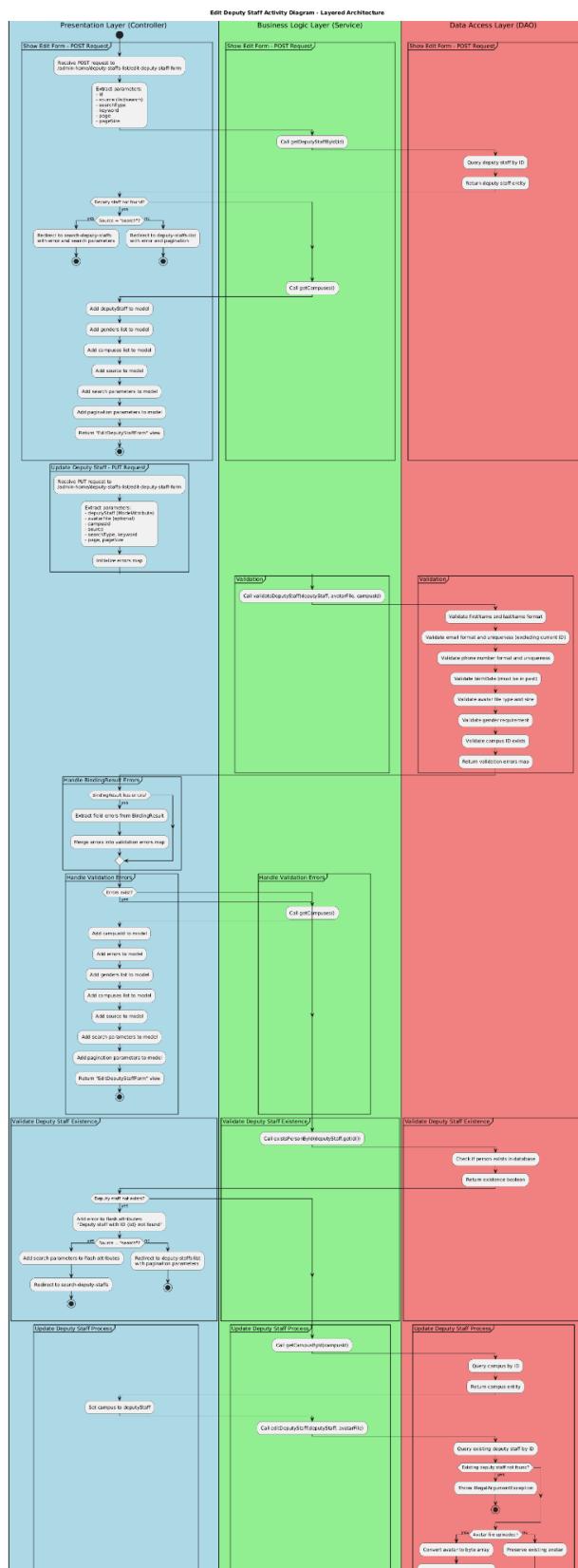


Diagram 35 Deputy Staff Update Workflow – Layered Architecture

SECTION 1 — SHOW EDIT FORM (POST REQUEST)

- Step 1: Receive POST request to /admin-home/deputy-staffs-list/edit-deputy-staff-form.
 - Step 2: Extract parameter id.
 - Step 3: Extract parameter source (list/search).
 - Step 4: Extract parameter searchType.
 - Step 5: Extract parameter keyword.
 - Step 6: Extract parameter page.
 - Step 7: Extract parameter pageSize.
-

Load Deputy Staff by ID

- Step 8: Call service getDeputyStaffById(id).

DAO Process

- Step 8.1: Query deputy staff by ID.
 - Step 8.2: Return deputy staff entity.
-

Handle Missing Deputy Staff

- Step 9: Check if deputy staff entity is null.
 - Step 10: If deputy staff not found:
 - Step 10.1: If source = "search":
 - Step 10.1.1: Redirect to search-deputy-staffs with error message.
 - Step 10.1.2: Include search parameters in redirect.
 - Step 10.1.3: Stop execution.
 - Step 10.2: If source is not "search":
 - Step 10.2.1: Redirect to deputy-staffs-list with error.
 - Step 10.2.2: Include pagination parameters.
 - Step 10.2.3: Stop execution.
-

Load Campuses for Edit Form

- Step 11: Call service getCampuses().
-

Prepare Model for Edit Form

- Step 12: Add deputyStaff entity to model.
- Step 13: Add genders list to model.
- Step 14: Add campuses list to model.
- Step 15: Add source to model.
- Step 16: Add search parameters to model.
- Step 17: Add pagination parameters to model.
- Step 18: Return view "EditDeputyStaffForm".
- Step 19: Stop execution.

SECTION 2 — UPDATE DEPUTY STAFF (PUT REQUEST)

- Step 20: Receive PUT request to /admin-home/deputy-staffs-list/edit-deputy-staff-form.
 - Step 21: Extract ModelAttribute deputyStaff.
 - Step 22: Extract optional avatarFile.
 - Step 23: Extract campusId.
 - Step 24: Extract source.
 - Step 25: Extract searchType and keyword.
 - Step 26: Extract page and pageSize.
 - Step 27: Initialize empty errors map.
-

SECTION 3 — VALIDATION

- Step 28: Call service validateDeputyStaff(deputyStaff, avatarFile, campusId).

DAO-Level Validation

- Step 28.1: Validate firstName format.
- Step 28.2: Validate lastName format.
- Step 28.3: Validate email format.
- Step 28.4: Validate email uniqueness excluding current ID.
- Step 28.5: Validate phone number format.
- Step 28.6: Validate phone number uniqueness.
- Step 28.7: Validate birthDate (must be in past).
- Step 28.8: Validate avatar file type.
- Step 28.9: Validate avatar file size.
- Step 28.10: Validate gender requirement.
- Step 28.11: Validate campus ID existence.

Step 29: Return validation errors map.

SECTION 4 — BINDING RESULT ERRORS

- Step 30: Check if BindingResult has errors.
 - Step 31: If BindingResult has errors:
 - Step 31.1: Extract field errors from BindingResult.
 - Step 31.2: Merge field errors into validation errors map.
-

SECTION 5 — HANDLE VALIDATION ERRORS

- Step 32: Check if validation errors exist.
- Step 33: If errors exist:

Load Supporting Data

- Step 33.1: Call service getCampuses().

Prepare Model for Re-Display

Step 33.2: Add campusId to model.
Step 33.3: Add errors map to model.
Step 33.4: Add genders list to model.
Step 33.5: Add campuses list to model.
Step 33.6: Add source to model.
Step 33.7: Add search parameters to model.
Step 33.8: Add pagination parameters to model.
Step 34: Return view "EditDeputyStaffForm".
Step 35: Stop execution.

SECTION 6 — VALIDATE EXISTENCE BEFORE UPDATE

Step 36: Call service existsPersonById(deputyStaff.getId()).

DAO

Step 36.1: Check database for person existence.
Step 36.2: Return boolean.

Step 37: If deputy staff does not exist:

Step 37.1: Add error to flash attributes:
"Deputy staff with ID {id} not found".

Branch by source

Step 37.2: If source = "search":
Step 37.2.1: Add search parameters to flash attributes.
Step 37.2.2: Redirect to search-deputy-staffs.
Step 37.2.3: Stop execution.

Step 37.3: If source ≠ "search":

Step 37.3.1: Redirect to deputy-staffs-list with pagination.
Step 37.3.2: Stop execution.

SECTION 7 — UPDATE DEPUTY STAFF PROCESS

Load Campus

Step 38: Call service getCampusById(campusId).

DAO

Step 38.1: Query campus by ID.
Step 38.2: Return campus.

Set Campus

Step 39: Assign campus to deputyStaff.

Begin Update Logic

Step 40: Call service editDeputyStaff(deputyStaff, avatarFile).

DAO Operations

Step 40.1: Query existing deputy staff by ID.

Step 40.2: If existing deputy staff not found, throw IllegalArgumentException.

Avatar Handling

Step 40.3: Check if avatar file uploaded.

Step 40.4: If avatar uploaded:

Step 40.4.1: Convert avatar to byte array.

Step 40.4.2: Set new avatar.

Step 40.5: If no avatar uploaded:

Step 40.5.1: Preserve existing avatar.

Update Fields

Step 40.6: Update deputy staff fields selectively.

Step 40.7: Merge updated deputy staff into database.

SECTION 8 — SUCCESS RESPONSE

Step 41: Add success flash message:

"Deputy staff edited successfully!"

Branch by source

Step 42: If source = "search":

Step 42.1: Add searchType, keyword, page, pageSize to flash attributes.

Step 42.2: Redirect to search-deputy-staffs.

Step 42.3: Stop execution.

Step 43: If source ≠ "search":

Step 43.1: Redirect to deputy-staffs-list with pagination.

Step 43.2: Stop execution.

SECTION 9 — EXCEPTION HANDLING

Step 44: Catch IOException.

Step 45: Create new errors map.

Step 46: Add error message:

"Error updating deputy staff: {exception}"

Load Extra Data

Step 47: Call service getCampuses().

Prepare Model

Step 48: Add errors to model.

Step 49: Add genders list to model.

Step 50: Add campuses list to model.

Step 51: Add source to model.

Step 52: Add search parameters to model.

Step 53: Add pagination parameters to model.

Step 54: Return "EditDeputyStaffForm" view.

Step 55: Stop execution.

ListDeputyStaffsController

001479794

Page 304 | 635 total

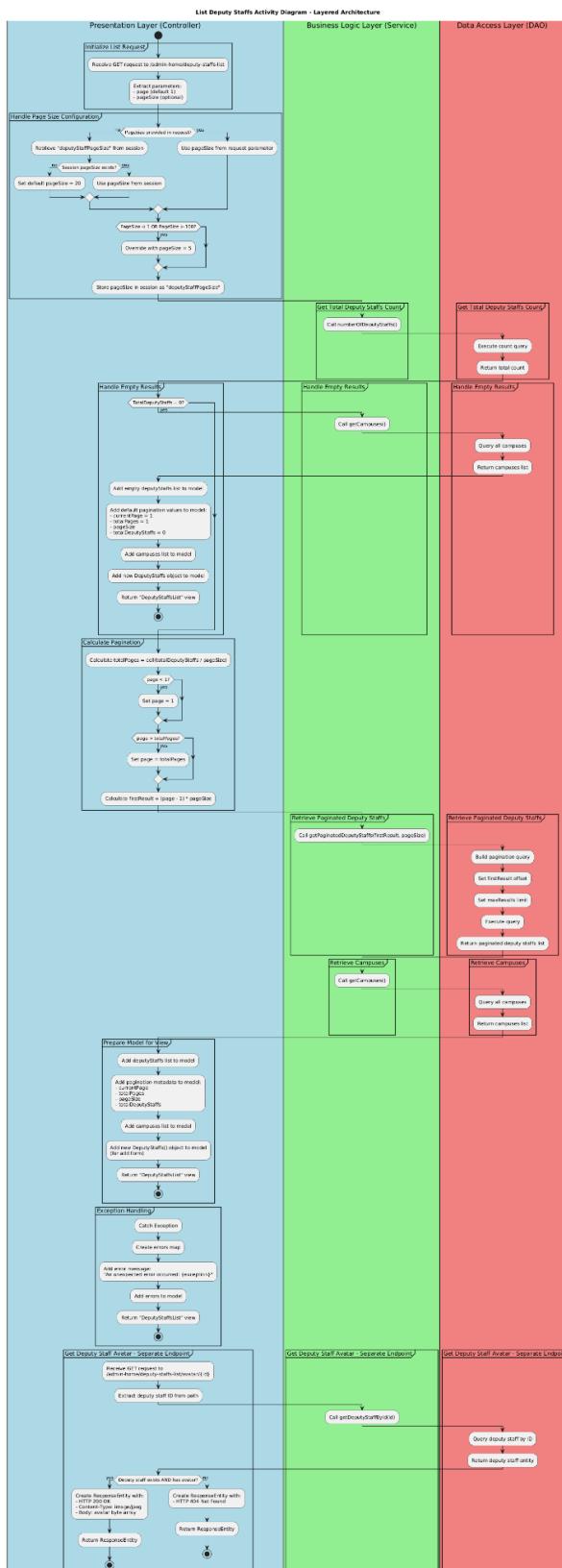


Diagram 36 Deputy Staff Listing Workflow – Layered Architecture

SECTION 1 — INITIALIZE LIST REQUEST

- Step 1: Receive GET request to /admin-home/deputy-staffs-list.
 - Step 2: Extract request parameter page (default = 1 if missing).
 - Step 3: Extract request parameter pageSize (optional).
-

SECTION 2 — HANDLE PAGE SIZE CONFIGURATION

Step 4: Check if pageSize was provided in request.

Case A — pageSize NOT provided

Step 5: Retrieve deputyStaffPageSize from session.
Step 6: Check if session pageSize exists.

Step 6.1: If session pageSize does not exist:
Step 6.1.1: Set default pageSize = 20.

Step 6.2: If session pageSize exists:
Step 6.2.1: Use pageSize from session.

Case B — pageSize provided

Step 7: Use pageSize from request parameter.

Validate Final Page Size

Step 8: Check if pageSize < 1 OR pageSize > 100.
Step 9: If invalid, set pageSize = 5.

Store Page Size

Step 10: Store pageSize into session under "deputyStaffPageSize".

SECTION 3 — GET TOTAL DEPUTY STAFFS COUNT

Step 11: Call service numberOfDeputyStaffs().

DAO Operations

Step 11.1: Execute SQL count query on DeputyStaffs table.
Step 11.2: Return totalDeputyStaffs value.

SECTION 4 — HANDLE EMPTY RESULTS

Step 12: Check if totalDeputyStaffs == 0.

Step 13: If totalDeputyStaffs = 0:

Load Campuses

Step 13.1: Call service getCampuses().

DAO

Step 13.1.1: Query all campuses.

Step 13.1.2: Return campus list.

Prepare Model

Step 13.2: Add empty deputyStaffs list to model.

Step 13.3: Add default pagination values:

Step 13.3.1: Set currentPage = 1.

Step 13.3.2: Set totalPages = 1.

Step 13.3.3: Set pageSize = pageSize.

Step 13.3.4: Set totalDeputyStaffs = 0.

Step 13.4: Add campuses list to model.

Step 13.5: Add new DeputyStaffs() object to model.

Step 13.6: Return "DeputyStaffsList" view.

Step 13.7: Stop execution.

SECTION 5 — CALCULATE PAGINATION (WHEN RESULTS EXIST)

Step 14: Calculate totalPages = ceil(totalDeputyStaffs / pageSize).

Validate current page

Step 15: If page < 1, set page = 1.

Step 16: If page > totalPages, set page = totalPages.

Compute firstResult

Step 17: Compute firstResult = (page - 1) * pageSize.

SECTION 6 — RETRIEVE PAGINATED DEPUTY STAFFS

Step 18: Call service getPaginatedDeputyStaffs(firstResult, pageSize).

DAO

Step 18.1: Build JPA/HQL pagination query.

Step 18.2: Apply firstResult offset.

Step 18.3: Apply maxResults limit.

Step 18.4: Execute query.

Step 18.5: Return paginated deputyStaffs list.

SECTION 7 — RETRIEVE CAMPUSES

Step 19: Call service getCampuses().

DAO

Step 19.1: Query all campuses.

Step 19.2: Return campus list.

SECTION 8 — PREPARE MODEL FOR VIEW

Step 20: Add deputyStaffs list to model.

Add Pagination Metadata

Step 21: Add pagination values to model:

Step 21.1: Add currentPage.

Step 21.2: Add totalPages.

Step 21.3: Add pageSize.

Step 21.4: Add totalDeputyStaffs.

Add Supporting Fields

Step 22: Add campuses list to model.

Step 23: Add new DeputyStaffs() object for add form.

Step 24: Return view "DeputyStaffsList".

Step 25: Stop execution.

SECTION 9 — EXCEPTION HANDLING

Step 26: Catch any exception.

Step 27: Create an errors map.

Step 28: Add error message:

"An unexpected error occurred: {exception}".

Step 29: Add errors to model.

Step 30: Return "DeputyStaffsList" view.

Step 31: Stop execution.

SECTION 10 — GET DEPUTY STAFF AVATAR (SEPARATE ENDPOINT)

Step 32: Receive GET request to /admin-home/deputy-staffs-list/avatar/{id}.

Step 33: Extract deputyStaff ID from path.

Load DeputyStaff by ID

Step 34: Call service getDeputyStaffById(id).

DAO

Step 34.1: Query DeputyStaff by ID.

Step 34.2: Return deputyStaff entity.

Build Avatar Response

Step 35: Check if deputyStaff exists AND has avatar.

If avatar exists:

Step 35.1: Create ResponseEntity with:
Step 35.1.1: HTTP Status 200 OK.
Step 35.1.2: Content-Type = image/jpeg.
Step 35.1.3: Body = avatar byte array.
Step 35.2: Return ResponseEntity.
Step 35.3: Stop execution.

If avatar does not exist:

Step 36: Create ResponseEntity with:
Step 36.1: HTTP Status 404 Not Found.
Step 36.2: No body.
Step 36.3: Return ResponseEntity.
Step 36.4: Stop execution.

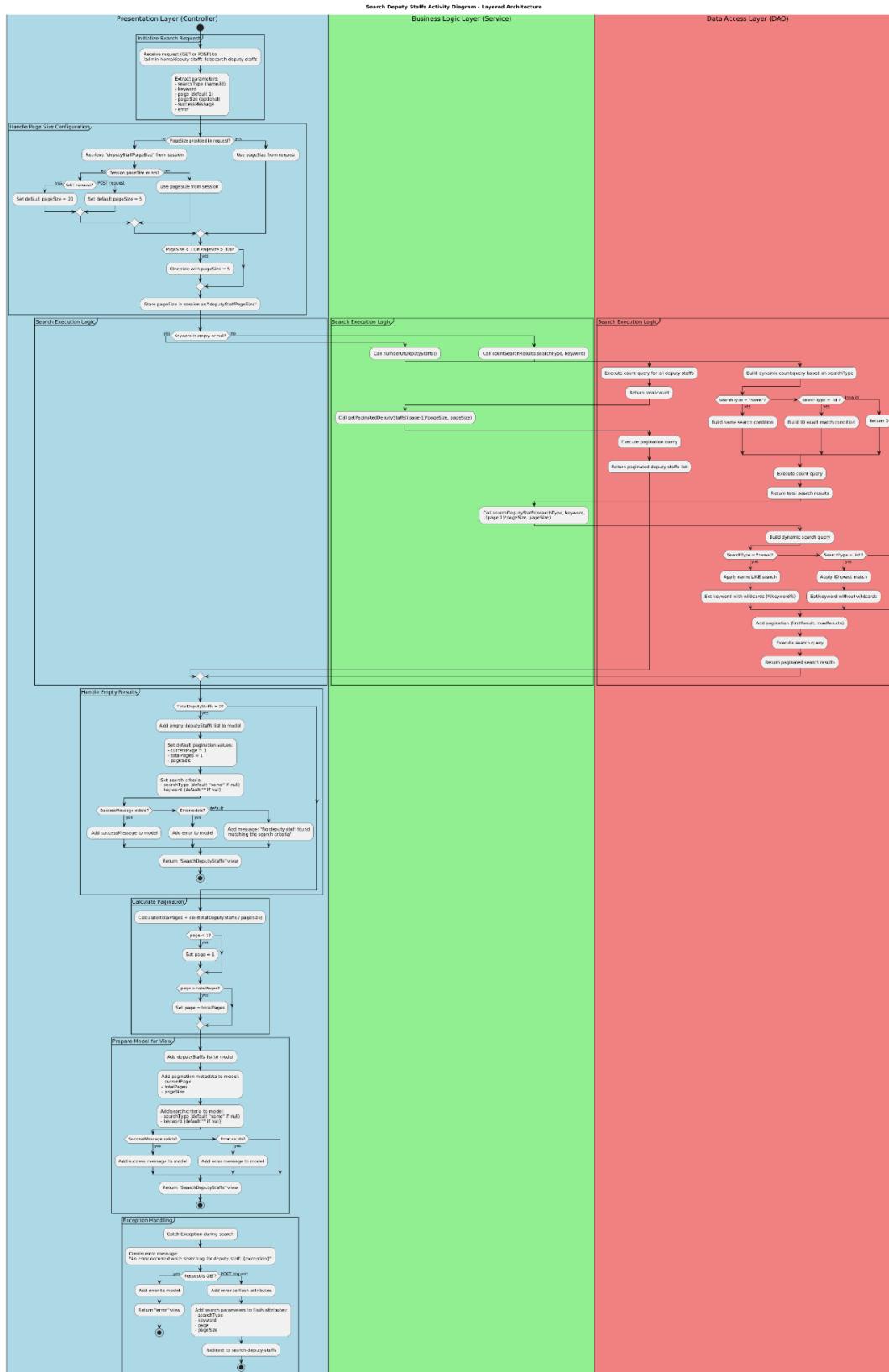


Diagram 37 Deputy Staff Search Workflow – Layered Architecture

SECTION 1 — INITIALIZE SEARCH REQUEST

- Step 1: Receive GET or POST request to /admin-home/deputy-staffs-list/search-deputy-staffs.
 - Step 2: Extract parameter searchType (name/id).
 - Step 3: Extract parameter keyword.
 - Step 4: Extract parameter page (default = 1 if missing).
 - Step 5: Extract parameter pageSize (optional).
 - Step 6: Extract parameter successMessage.
 - Step 7: Extract parameter error.
-

SECTION 2 — HANDLE PAGE SIZE CONFIGURATION

- Step 8: Check if pageSize provided in request.

Case A — pageSize NOT provided

- Step 9: Retrieve "deputyStaffPageSize" from session.
- Step 10: Check if session pageSize exists.

Step 10.1: If session pageSize does NOT exist:

- Step 10.1.1: If request is GET → set pageSize = 20.
- Step 10.1.2: If request is POST → set pageSize = 5.

Step 10.2: If session pageSize exists:

- Step 10.2.1: Use pageSize from session.

Case B — pageSize provided

- Step 11: Use pageSize from request.
-

Validate final pageSize

- Step 12: Check if pageSize < 1 OR pageSize > 100.

- Step 13: If invalid, set pageSize = 5.
-

Store pageSize

- Step 14: Save pageSize into session as "deputyStaffPageSize".
-

SECTION 3 — SEARCH EXECUTION LOGIC

- Step 15: Check if keyword is empty or null.
-

CASE 1 — Keyword is empty (basic list mode)

- Step 16: Call service numberOfDeputyStaffs().

DAO

Step 16.1: Execute count query for all deputy staff.

Step 16.2: Return total count.

Step 17: Call service getPaginatedDeputyStaffs((page - 1) * pageSize, pageSize).

DAO

Step 17.1: Execute pagination query.

Step 17.2: Return paginated deputyStaff list.

CASE 2 — Keyword provided (search mode)

Step 18: Call service countSearchResults(searchType, keyword).

DAO Count Logic

Step 18.1: Build dynamic count query based on searchType.

Step 18.2: If searchType = "name":

Step 18.2.1: Build name search condition (LIKE).

Step 18.3: If searchType = "id":

Step 18.3.1: Build exact ID match condition.

Step 18.4: If searchType invalid:

Step 18.4.1: Return 0 immediately.

Step 18.5: Execute count query.

Step 18.6: Return total search results.

Execute Search Query

Step 19: Call service

searchDeputyStaffs(searchType, keyword, (page-1)*pageSize, pageSize).

DAO

Step 19.1: Build dynamic search query.

Step 19.2: If searchType = "name":

Step 19.2.1: Apply LIKE matching.

Step 19.2.2: Set keyword parameter as %keyword%.

Step 19.3: If searchType = "id":

Step 19.3.1: Apply exact ID comparison.

Step 19.3.2: Set keyword parameter unchanged.

Step 19.4: Add pagination:

Step 19.4.1: Set firstResult.

Step 19.4.2: Set maxResults.

Step 19.5: Execute query.

Step 19.6: Return paginated search results.

SECTION 4 — HANDLE EMPTY RESULTS

Step 20: Check if `totalDeputyStaffs == 0`.

Step 21: If `totalDeputyStaffs = 0`:

Build empty model

Step 21.1: Add empty `deputyStaffs` list to model.

Pagination defaults

Step 21.2: Set `currentPage = 1`.

Step 21.3: Set `totalPages = 1`.

Step 21.4: Set `pageSize = pageSize`.

Search criteria defaults

Step 21.5: Set `searchType = "name"` if null.

Step 21.6: Set `keyword = ""` if null.

Message logic

Step 21.7: If `successMessage` exists → add to model.

Step 21.8: Else if `error` exists → add to model.

Step 21.9: Else add default message:

“No deputy staff found matching the search criteria.”

Step 22: Return “`SearchDeputyStaffs`” view.

Step 23: Stop execution.

SECTION 5 — CALCULATE PAGINATION

Step 24: Compute `totalPages = ceil(totalDeputyStaffs / pageSize)`.

Validate page

Step 25: If `page < 1`, set `page = 1`.

Step 26: If `page > totalPages`, set `page = totalPages`.

SECTION 6 — PREPARE MODEL FOR VIEW

Step 27: Add `deputyStaffs` list to model.

Pagination metadata

Step 28: Add `currentPage` to model.

Step 29: Add `totalPages` to model.

Step 30: Add `pageSize` to model.

Search metadata

Step 31: Add `searchType` (default “`name`”).

Step 32: Add `keyword` (default “`”`”).

Messages

Step 33: If successMessage exists → add to model.

Step 34: Else if error exists → add to model.

Step 35: Return "SearchDeputyStaffs" view.

Step 36: Stop execution.

SECTION 7 — EXCEPTION HANDLING

Step 37: Catch any exception during search.

Step 38: Build error message:

"An error occurred while searching for deputy staff: {exception}"

For GET request

Step 39: If request is GET:

Step 39.1: Add error to model.

Step 39.2: Return "error" view.

Step 39.3: Stop execution.

For POST request

Step 40: If request is POST:

Step 40.1: Add error to flash attributes.

Step 40.2: Add searchType to flash attributes.

Step 40.3: Add keyword to flash attributes.

Step 40.4: Add page to flash attributes.

Step 40.5: Add pageSize to flash attributes.

Step 40.6: Redirect to /search-deputy-staffs.

Step 40.7: Stop execution.

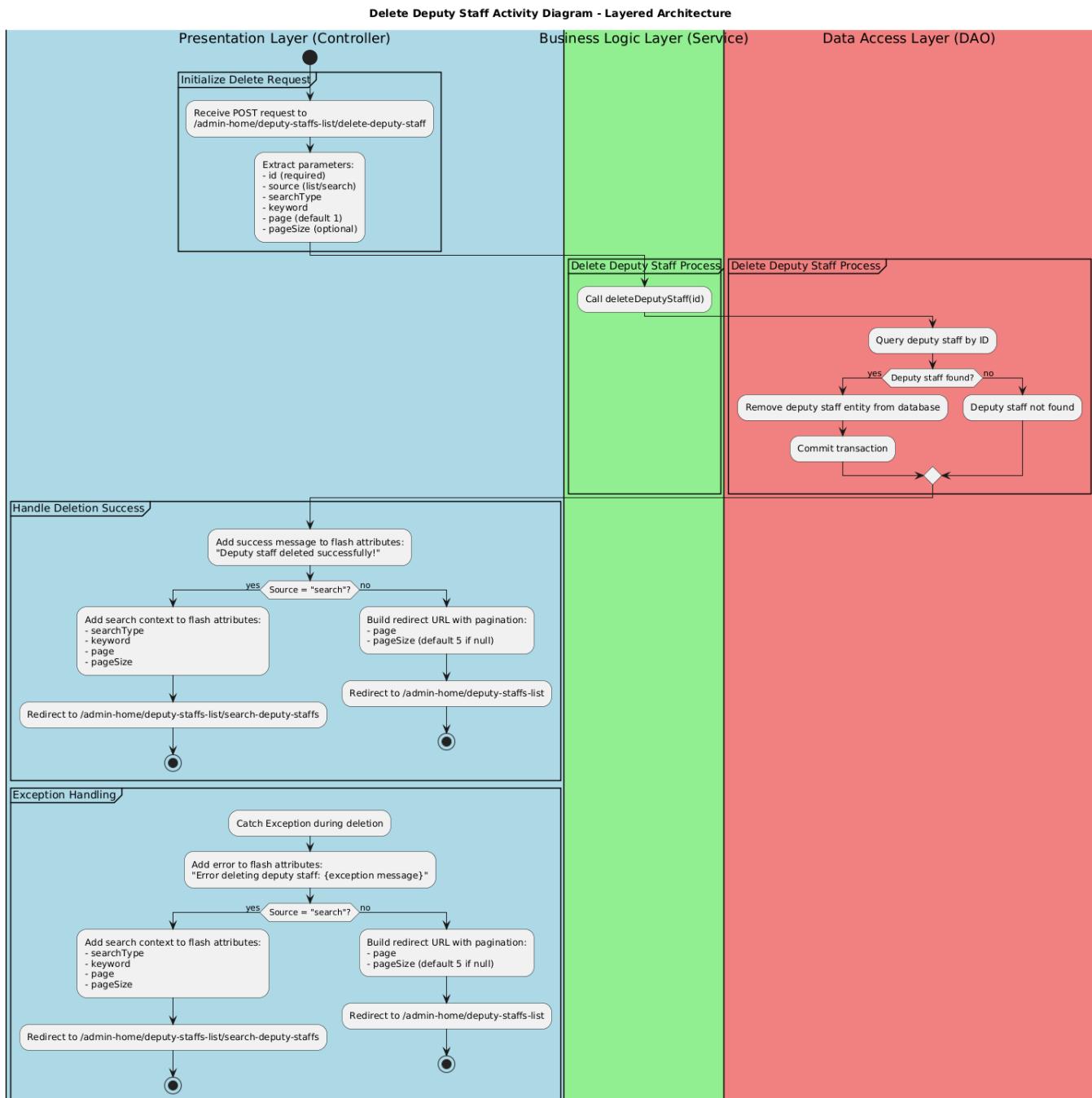


Diagram 38 Deputy Staff Deletion Workflow – Layered Architecture

SECTION 1 — INITIALIZE DELETE REQUEST

Step 1: Receive POST request to
`/admin-home/deputy-staffs-list/delete-deputy-staff`.

Step 2: Extract required parameter id.
Step 3: Extract parameter source (list/search).
Step 4: Extract parameter searchType.
Step 5: Extract parameter keyword.
Step 6: Extract parameter page (default = 1).
Step 7: Extract parameter pageSize (optional).

SECTION 2 — DELETE DEPUTY STAFF PROCESS

Step 8: Call service `deleteDeputyStaff(id)`.

DAO Logic

Step 8.1: Query deputy staff by ID.
Step 8.2: Check if deputy staff entity exists.
Step 8.3: If deputy staff DOES exist:
Step 8.3.1: Remove deputy staff entity from database.
Step 8.3.2: Commit transaction.
Step 8.4: If deputy staff does NOT exist:
Step 8.4.1: Mark as "not found" and return.

SECTION 3 — HANDLE DELETION SUCCESS

Step 9: Add success flash message:

"Deputy staff deleted successfully!"

Branch based on source

Step 10: Check if source = "search".

Case A — Return to Search Page

Step 11: If source = "search":

Add search context

Step 11.1: Add searchType to flash attributes.
Step 11.2: Add keyword to flash attributes.
Step 11.3: Add page to flash attributes.
Step 11.4: Add pageSize to flash attributes.

Redirect

Step 11.5: Redirect to
`/admin-home/deputy-staffs-list/search-deputy-staffs`.

Step 11.6: Stop execution.

Case B — Return to Main List Page

Step 12: If source ≠ "search":

Validate pagination parameters

Step 12.1: Use page from request.

Step 12.2: If pageSize is null, set pageSize = 5 (default).

Step 12.3: Build redirect URL with page & pageSize.

Redirect

Step 12.4: Redirect to

/admin-home/deputy-staffs-list.

Step 12.5: Stop execution.

SECTION 4 — EXCEPTION HANDLING

Step 13: Catch any exception during deletion.

Step 14: Add flash error:

"Error deleting deputy staff: {exception message}"

Branch based on source

Step 15: Check if source = "search".

Case A — Redirect back to Search

Step 16: If source = "search":

Step 16.1: Add searchType to flash attributes.

Step 16.2: Add keyword to flash attributes.

Step 16.3: Add page to flash attributes.

Step 16.4: Add pageSize to flash attributes.

Step 16.5: Redirect to
/admin-home/deputy-staffs-list/search-deputy-staffs.

Step 16.6: Stop execution.

Case B — Redirect back to List

Step 17: If source ≠ "search":

Recover pagination

Step 17.1: Use page from request.

Step 17.2: If pageSize is null, set pageSize = 5.

Step 17.3: Build redirect URL with pagination.

Redirect

Step 17.4: Redirect to
`/admin-home/deputy-staffs-list.`

Step 17.5: Stop execution.

Activity diagram related to major lecturer roles

AddLecturerController

001479794

Page **318** | **635** total

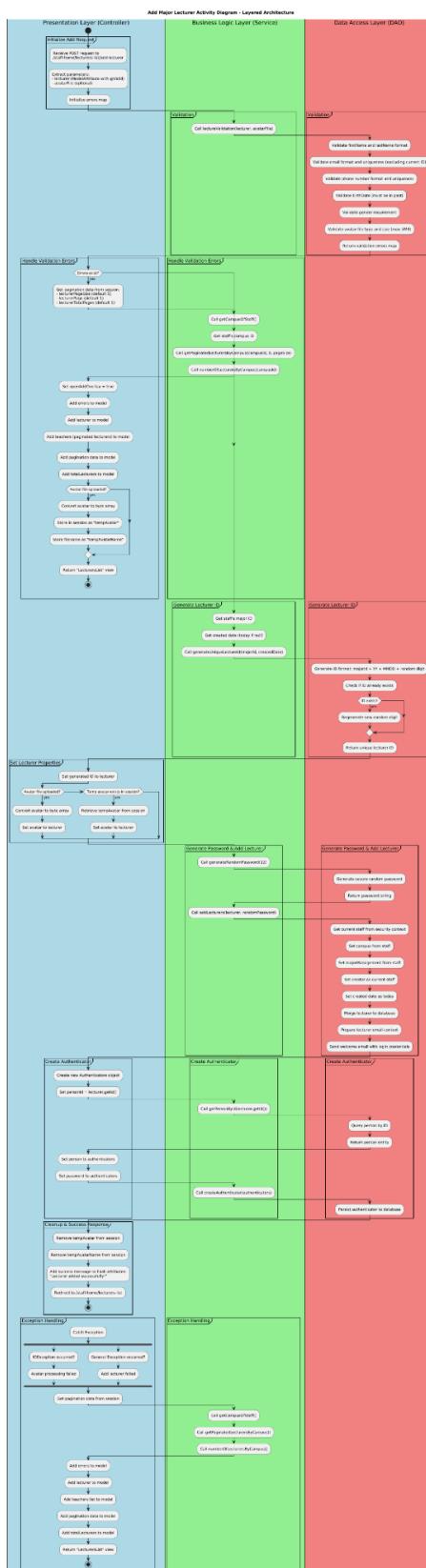


Diagram 39 Mentor Lecturer Creation Workflow – Layered Architecture

SECTION 1 — INITIALIZE ADD REQUEST

- Step 1: Receive POST request to /staff-home/lecturers-list/add-lecturer.
 - Step 2: Extract ModelAttribute lecturer (validated with @Valid).
 - Step 3: Extract optional avatarFile.
 - Step 4: Initialize an empty errors map.
-

SECTION 2 — VALIDATION

- Step 5: Call service lectureValidation(lecturer, avatarFile).

DAO Validation Steps

- Step 5.1: Validate firstName format.
- Step 5.2: Validate lastName format.
- Step 5.3: Validate email format.
- Step 5.4: Validate email uniqueness (excluding current ID).
- Step 5.5: Validate phone number format.
- Step 5.6: Validate phone number uniqueness.
- Step 5.7: Validate birthDate (must be in the past).
- Step 5.8: Validate gender requirement.
- Step 5.9: Validate avatar file type.
- Step 5.10: Validate avatar file size (max 5MB).

- Step 6: Return validation errors map to controller.
-

SECTION 3 — HANDLE VALIDATION ERRORS

- Step 7: Check if errors exist.

- Step 8: If errors exist:

Retrieve Pagination

- Step 8.1: Get lecturerPageSize from session (default = 5).
- Step 8.2: Get lecturerPage from session (default = 1).
- Step 8.3: Get lecturerTotalPages from session (default = 1).

Load Campus & Lecturer List

- Step 8.4: Call service getCampusOfStaff().
- Step 8.5: Retrieve staff's campus ID.
- Step 8.6: Call service getPaginatedLecturersByCampus(campusId, 0, pageSize).
- Step 8.7: Call service numberOfLecturersByCampus(campusId).

Prepare Model

- Step 8.8: Set openAddOverlay = true.
- Step 8.9: Add validation errors to model.
- Step 8.10: Add lecturer object to model.
- Step 8.11: Add paginated lecturers list (teachers) to model.

Step 8.12: Add pagination info to model.

Step 8.13: Add totalLecturers to model.

Handle Avatar Preview

Step 8.14: Check if avatar file was uploaded.

Step 8.15: If avatar uploaded:

Step 8.15.1: Convert avatar to byte array.

Step 8.15.2: Store avatar bytes into session as "tempAvatar".

Step 8.15.3: Store original filename as "tempAvatarName".

Step 9: Return "LecturersList" view.

Step 10: Stop execution.

SECTION 4 — GENERATE LECTURER ID

Step 11: Retrieve staff's major ID.

Step 12: Determine created date (today if null).

Step 13: Call service generateUniqueLectureId(majordId, createdDate).

DAO ID Generation Logic

Step 13.1: Generate ID using format:
majordId + YY + MMDD + randomDigit.

Step 13.2: Check if generated ID exists.

Step 13.3: If ID exists:

Step 13.3.1: Regenerate with new random digit.

Step 13.4: Return unique lecturer ID.

SECTION 5 — SET LECTURER PROPERTIES

Step 14: Assign generated ID to lecturer.

Handle Avatar Assignment

Step 15: Check if avatar file uploaded.

Step 16: If avatar uploaded:

Step 16.1: Convert avatar to byte array.

Step 16.2: Assign byte array into lecturer.avatar.

Step 17: If no avatar uploaded, check temp avatar in session.

Step 18: If temp avatar exists:

Step 18.1: Retrieve "tempAvatar" from session.

Step 18.2: Set lecturer.avatar = retrieved bytes.

SECTION 6 — GENERATE PASSWORD & ADD LECTURER

Step 19: Call service generateRandomPassword(12).

DAO

001479794

Page 321 | 635 total

Step 19.1: Generate secure random password string.

Step 19.2: Return password.

Add Lecturer

Step 20: Call service addLecturers(lecturer, randomPassword).

DAO Insert Logic

Step 20.1: Get current staff from security context.

Step 20.2: Assign staff's campus to lecturer.

Step 20.3: Assign staff's majorManagement to lecturer.

Step 20.4: Set creator = current staff.

Step 20.5: Set createdDate = today.

Step 20.6: Merge lecturer into database.

Email Notification

Step 20.7: Prepare welcome email context.

Step 20.8: Send email containing login credentials.

SECTION 7 — CREATE AUTHENTICATOR ENTRY

Step 21: Create new Authenticators object.

Step 22: Set personId = lecturer.getId().

Load Person Entity

Step 23: Call service getPersonById(lecturer.getId()).

DAO

Step 23.1: Query person by ID.

Step 23.2: Return person entity.

Bind Authenticator Data

Step 24: Assign person to authenticators.

Step 25: Set password to authenticator.

Persist Authenticator

Step 26: Call service createAuthenticator(authenticators).

Step 27: DAO persists authenticator to database.

SECTION 8 — CLEANUP & SUCCESS RESPONSE

Step 28: Remove "tempAvatar" from session.

Step 29: Remove "tempAvatarName" from session.

Step 30: Add flash success message:

"Lecturer added successfully!"

Step 31: Redirect to /staff-home/lecturers-list.

Step 32: Stop execution.

SECTION 9 — EXCEPTION HANDLING

Step 33: Catch any Exception.

Parallel Error Causes

Step 34: Check if IOException occurred.

Step 35: If IOException:

Step 35.1: Record avatar processing failure.

Step 36: Check if general exception occurred.

Step 37: If general error:

Step 37.1: Record "Add lecturer failed".

Recover Pagination

Step 38: Retrieve pagination from session.

Reload Data for Error Page

Step 39: Call getCampusOfStaff().

Step 40: Call getPaginatedLecturersByCampus().

Step 41: Call numberOfLecturersByCampus().

Prepare Model

Step 42: Add errors to model.

Step 43: Add lecturer back to model.

Step 44: Add teachers list to model.

Step 45: Add pagination to model.

Step 46: Add totalLecturers to model.

Step 47: Return "LecturersList" view.

Step 48: Stop execution.

DeleteLecturerController

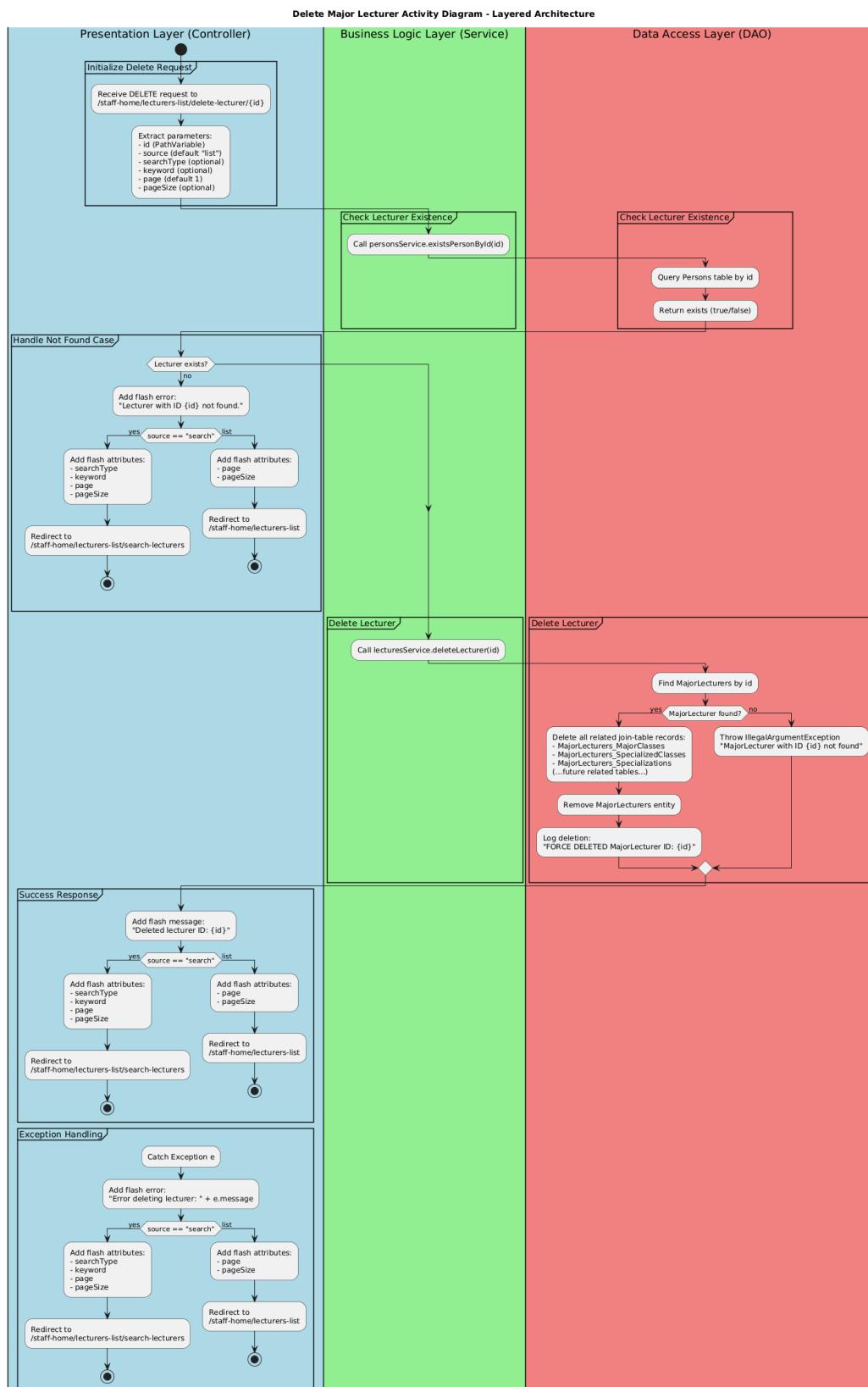


Diagram 40 Major Lecturer Deletion Workflow – Layered Architecture

SECTION 1 — INITIALIZE DELETE REQUEST

Step 1: Receive DELETE request to
`/staff-home/lecturers-list/delete-lecturer/{id}`.

Step 2: Extract id from PathVariable.

Step 3: Extract parameter source (default = "list").

Step 4: Extract optional parameter searchType.

Step 5: Extract optional parameter keyword.

Step 6: Extract parameter page (default = 1).

Step 7: Extract optional parameter pageSize.

SECTION 2 — CHECK LECTURER EXISTENCE

Step 8: Call service `personsService.existsPersonById(id)`.

DAO Logic

Step 8.1: Query Persons table using the provided ID.

Step 8.2: Return boolean result (true/false).

SECTION 3 — HANDLE NOT FOUND CASE

Step 9: Check if lecturer exists.

Step 10: If lecturer does NOT exist:

Add flash error

Step 10.1: Add message:

"Lecturer with ID {id} not found."

Branch by source

Step 10.2: If source == "search":

Step 10.2.1: Add searchType to flash attributes.

Step 10.2.2: Add keyword to flash attributes.

Step 10.2.3: Add page to flash attributes.

Step 10.2.4: Add pageSize to flash attributes.

Step 10.2.5: Redirect to

`/staff-home/lecturers-list/search-lecturers`.

Step 10.2.6: Stop execution.

Step 10.3: If source != "search" (list mode):

Step 10.3.1: Add page to flash attributes.

Step 10.3.2: Add pageSize to flash attributes.

Step 10.3.3: Redirect to

`/staff-home/lecturers-list`.

Step 10.3.4: Stop execution.

SECTION 4 — DELETE LECTURER

Step 11: Call service lecturesService.deleteLecturer(id).

DAO Operations

Step 11.1: Query MajorLecturers table by ID.

Step 11.2: Check if entity exists.

If lecturer exists

Step 11.3: If MajorLecturer found:

Step 11.3.1: Delete related join-table records from:

Step 11.3.1.1: MajorLecturers_MajorClasses.

Step 11.3.1.2: MajorLecturers_SpecializedClasses.

Step 11.3.1.3: MajorLecturers_Specializations.

Step 11.3.1.4: (Any future related join tables).

Step 11.3.2: Remove the MajorLecturers entity from the database.

Step 11.3.3: Log deletion message:

"FORCE DELETED MajorLecturer ID: {id}".

If lecturer does NOT exist

Step 11.4: If MajorLecturer NOT found:

Step 11.4.1: Throw IllegalArgumentException.

Step 11.4.2: Message:

"MajorLecturer with ID {id} not found".

SECTION 5 — SUCCESS RESPONSE

Step 12: Add flash success message:

"Deleted lecturer ID: {id}"

Branch by source

Step 13: Check if source = "search".

Case A — Redirect back to Search

Step 14: If source == "search":

Step 14.1: Add searchType to flash attributes.

Step 14.2: Add keyword to flash attributes.

Step 14.3: Add page to flash attributes.

Step 14.4: Add pageSize to flash attributes.

Step 14.5: Redirect to

/staff-home/lecturers-list/search-lecturers.

Step 14.6: Stop execution.

Case B — Redirect back to List

Step 15: If source != "search":

Step 15.1: Add page to flash attributes.

Step 15.2: Add pageSize to flash attributes.

Step 15.3: Redirect to

/staff-home/lecturers-list.

Step 15.4: Stop execution.

SECTION 6 — EXCEPTION HANDLING

Step 16: Catch any exception e.

Step 17: Add flash error message:

"Error deleting lecturer: " + e.message

Branch by source

Step 18: If source == "search":

Step 18.1: Add searchType to flash attributes.

Step 18.2: Add keyword to flash attributes.

Step 18.3: Add page to flash attributes.

Step 18.4: Add pageSize to flash attributes.

Step 18.5: Redirect to

/staff-home/lecturers-list/search-lecturers.

Step 18.6: Stop execution.

If not search mode

Step 19: If source != "search":

Step 19.1: Add page to flash attributes.

Step 19.2: Add pageSize to flash attributes.

Step 19.3: Redirect to

/staff-home/lecturers-list.

Step 19.4: Stop execution.

EditLecturerController

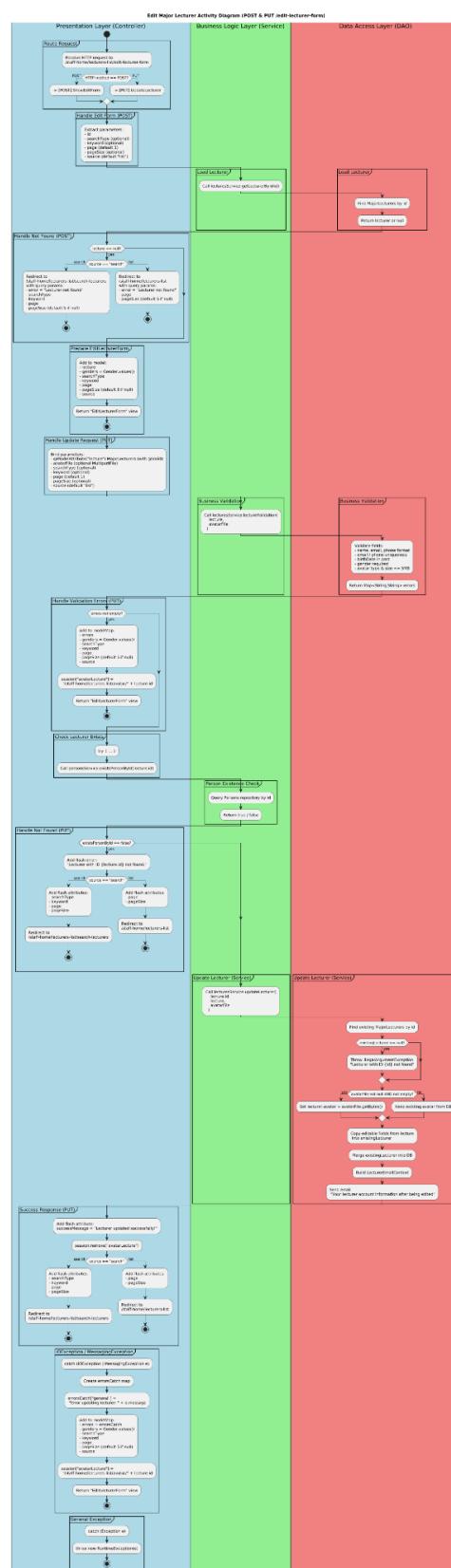


Diagram 41 Major Lecturer Update Workflow – Layered Architecture

SECTION 1 — ROUTE REQUEST

- Step 1: Receive HTTP request to /staff-home/lecturers-list/edit-lecturer-form.
 - Step 2: Determine request method.
 - Step 3: If request method is POST → process Show Edit Form.
 - Step 4: If request method is PUT → process Update Lecturer.
-

SECTION 2 — POST REQUEST: SHOW EDIT FORM

- Step 5: Extract id.
 - Step 6: Extract optional searchType.
 - Step 7: Extract optional keyword.
 - Step 8: Extract page (default = 1).
 - Step 9: Extract optional pageSize.
 - Step 10: Extract source (default = "list").
 - Step 11: Call service lecturesService.getLecturerById(id).
 - Step 12: DAO queries MajorLecturers by ID.
 - Step 13: DAO returns lecturer entity or null.
 - Step 14: If lecturer is null and source = "search":
 - Step 14.1: Redirect to /staff-home/lecturers-list/search-lecturers with:
 - Step 14.1.1: error = "Lecturer not found".
 - Step 14.1.2: searchType.
 - Step 14.1.3: keyword.
 - Step 14.1.4: page.
 - Step 14.1.5: pageSize (default = 5 if null).
 - Step 14.2: Stop execution.
 - Step 15: If lecturer is null and source = "list":
 - Step 15.1: Redirect to /staff-home/lecturers-list with:
 - Step 15.1.1: error = "Lecturer not found".
 - Step 15.1.2: page.
 - Step 15.1.3: pageSize (default = 5 if null).
 - Step 15.2: Stop execution.
 - Step 16: Add the following to model:
 - Step 16.1: lecturer.
 - Step 16.2: genders = Gender.values().
 - Step 16.3: searchType.
 - Step 16.4: keyword.
 - Step 16.5: page.
 - Step 16.6: pageSize (default 5 if null).
 - Step 16.7: source.
 - Step 17: Return "EditLecturerForm" view.
 - Step 18: Stop execution.
-

SECTION 3 — PUT REQUEST: UPDATE LECTURER

- Step 19: Bind ModelAttribute lecture (@Valid).
 - Step 20: Extract optional avatarFile.
 - Step 21: Extract optional searchType.
 - Step 22: Extract optional keyword.
 - Step 23: Extract page (default = 1).
 - Step 24: Extract optional pageSize.
 - Step 25: Extract source (default = "list").
 - Step 26: Call service lecturesService.lectureValidation(lecture, avatarFile).
-

Validation (DAO level)

- Step 26.1: Validate name format.
 - Step 26.2: Validate email format.
 - Step 26.3: Validate phone format.
 - Step 26.4: Validate email uniqueness.
 - Step 26.5: Validate phone uniqueness.
 - Step 26.6: Validate birthDate is in the past.
 - Step 26.7: Validate gender not null.
 - Step 26.8: Validate avatar file type.
 - Step 26.9: Validate avatar file size ≤ 5MB.
 - Step 27: Return validation errors map.
-

SECTION 4 — HANDLE VALIDATION ERRORS (PUT)

- Step 28: If errors map is not empty:
 - Step 28.1: Add errors to model.
 - Step 28.2: Add genders = Gender.values().
 - Step 28.3: Add searchType.
 - Step 28.4: Add keyword.
 - Step 28.5: Add page.
 - Step 28.6: Add pageSize (default 5 if null).
 - Step 28.7: Add source.
 - Step 28.8: Set session attribute "avatarLecture" = /staff-home/lecturers-list/avatar/{lecture.id}.
 - Step 29: Return "EditLecturerForm" view.
 - Step 30: Stop execution.
-

SECTION 5 — CHECK LECTURER EXISTENCE

- Step 31: Call service personsService.existsPersonById(lecture.id).

Step 32: DAO queries Persons by ID.

Step 33: DAO returns true/false.

Step 34: If lecturer does not exist:

001479794

Page 330 | 635 total

Step 34.1: Add flash error:
"Lecturer with ID {lecture.id} not found."

If source = search:

Step 34.2: Add searchType to flash.
Step 34.3: Add keyword.
Step 34.4: Add page.
Step 34.5: Add pageSize.
Step 34.6: Redirect to /staff-home/lecturers-list/search-lecturers.
Step 34.7: Stop execution.

If source = list:

Step 34.8: Add page to flash.
Step 34.9: Add pageSize.
Step 34.10: Redirect to /staff-home/lecturers-list.
Step 34.11: Stop execution.

SECTION 6 — UPDATE LECTURER (SERVICE)

Step 35: Call service lecturesService.updateLecturer(lecture.id, lecture, avatarFile).

Step 36: DAO retrieves existing lecturer by ID.
Step 37: If not found → throw IllegalArgumentException.

Avatar Processing

Step 38: If avatarFile is not null and not empty:
Step 38.1: Convert avatarFile to bytes.
Step 38.2: Assign bytes to lecturer.avatar.

Step 39: If avatarFile not provided:
Step 39.1: Keep existing avatar.

Apply Lecturer Changes

Step 40: Copy editable fields from input lecture into existing record.
Step 41: Merge updated lecturer into database.

Email Notification

Step 42: Build LecturerEmailContext.
Step 43: Send email titled:
"Your lecturer account information after being edited".

SECTION 7 — SUCCESS RESPONSE

Step 44: Add flash success message:
"Lecturer updated successfully!"
Step 45: Remove session attribute "avatarLecture".

If source = search

Step 46: Add searchType to flash.
Step 47: Add keyword.
Step 48: Add page.
Step 49: Add pageSize.
Step 50: Redirect to /staff-home/lecturers-list/search-lecturers.
Step 51: Stop execution.

If source = list

Step 52: Add page.
Step 53: Add pageSize.
Step 54: Redirect to /staff-home/lecturers-list.
Step 55: Stop execution.

SECTION 8 — IO / MESSAGING EXCEPTION HANDLING

Step 56: Catch IOException or MessagingException.
Step 57: Create errorsCatch map.
Step 58: Add error:
"Error updating lecturer: {exception message}."
Step 59: Add errorsCatch to model.
Step 60: Add genders = Gender.values().
Step 61: Add searchType.
Step 62: Add keyword.
Step 63: Add page.
Step 64: Add pageSize (default 5 if null).
Step 65: Add source.
Step 66: Set session "avatarLecture" = /staff-home/lecturers-list/avatar/{lecture.id}.
Step 67: Return "EditLecturerForm" view.
Step 68: Stop execution.

SECTION 9 — GENERAL EXCEPTION HANDLING

Step 69: Catch any other Exception.
Step 70: Throw new RuntimeException(e).
Step 71: Stop execution.

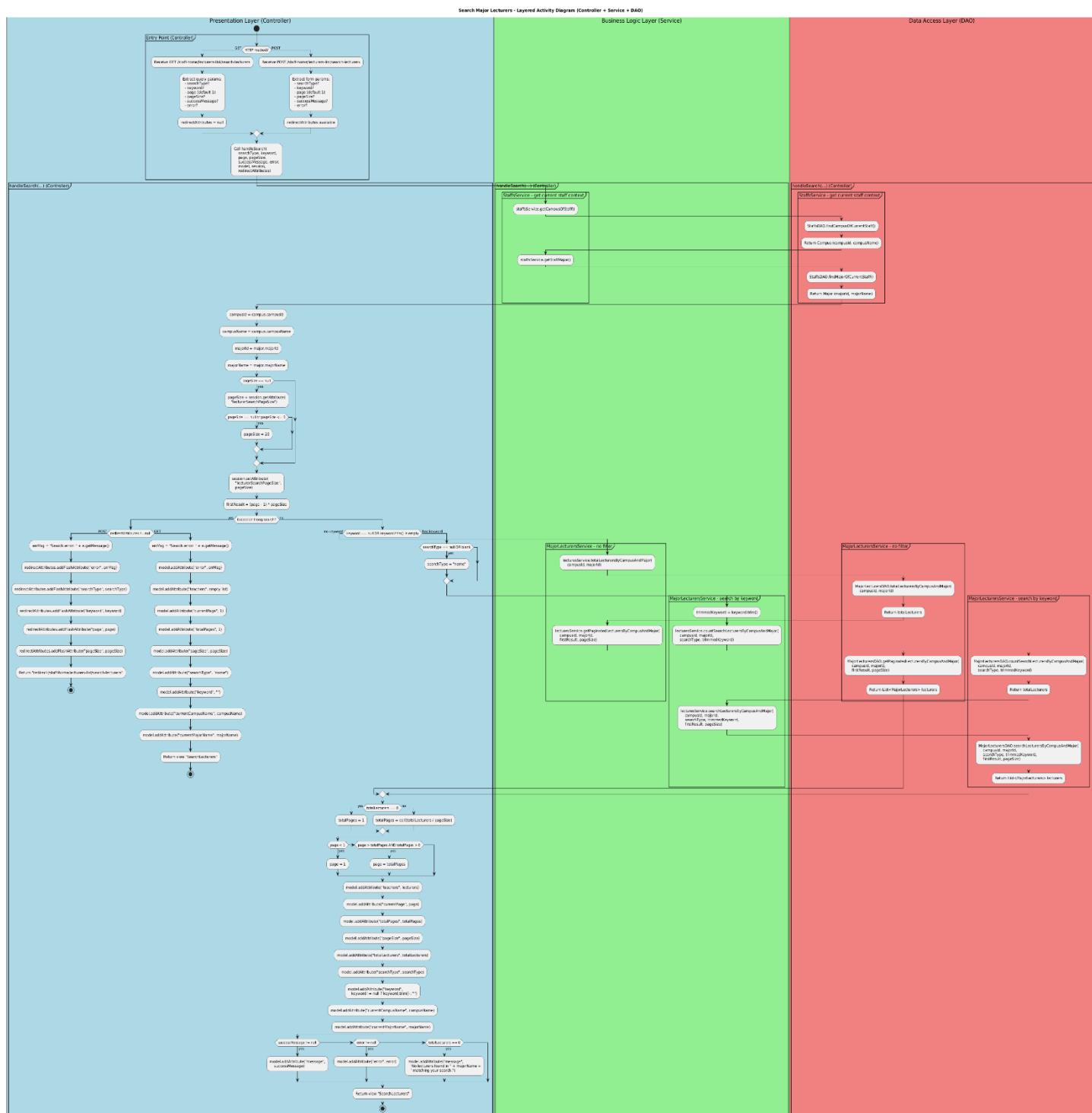


Diagram 42 Major Lecturer Search Workflow – Layered Architecture

SECTION 1 — ENTRY POINT (CONTROLLER)

Step 1: Receive HTTP request to
/staff-home/lecturers-list/search-lecturers.

Step 2: Determine HTTP method.

GET request:

Step 3: Extract query parameters:

Step 3.1: searchType (optional).

Step 3.2: keyword (optional).

Step 3.3: page (default = 1).

Step 3.4: pageSize (optional).

Step 3.5: successMessage (optional).

Step 3.6: error (optional).

Step 3.7: Set redirectAttributes = null.

POST request:

Step 4: Extract form parameters (same fields as GET).

Step 5: redirectAttributes available for storing flash attributes.

Step 6: Call controller method:

handleSearch(searchType, keyword, page, pageSize, successMessage, error, model, session, redirectAttributes).

SECTION 2 — LOAD STAFF CONTEXT (SERVICE + DAO)

Step 7: Call staffsService.getCampusOfStaff().

DAO steps:

Step 7.1: Query StaffsDAO → campus of current staff.

Step 7.2: Return Campus (campusId, campusName).

Step 8: Call staffsService.getStaffMajor().

DAO steps:

Step 8.1: Query StaffsDAO → major of current staff.

Step 8.2: Return Major (majorId, majorName).

Step 9: Assign values:

Step 9.1: campusId = campus.campusId.

Step 9.2: campusName = campus.campusName.

Step 9.3: majorId = major.majorId.

Step 9.4: majorName = major.majorName.

SECTION 3 — HANDLE PAGE SIZE (CONTROLLER + SESSION)

Step 10: If pageSize is null:

Step 10.1: Read "lecturerSearchPageSize" from session.

Step 10.2: If session pageSize is null OR <= 0 → set pageSize = 20.

Step 11: Save pageSize back into session as "lecturerSearchPageSize".

Step 12: Compute firstResult = (page - 1) * pageSize.

SECTION 4 — HANDLE TRY/CATCH BLOCK

Step 13: Check if an exception occurs inside search.

Exception Path

Step 14: If exception thrown:

Case A — POST request (redirectAttributes not null)

Step 14.1: Build error message: "Search error: " + e.getMessage().

Step 14.2: Add flash attribute "error".

Step 14.3: Add flash attribute "searchType".

Step 14.4: Add flash attribute "keyword".

Step 14.5: Add flash attribute "page".

Step 14.6: Add flash attribute "pageSize".

Step 14.7: Redirect to /staff-home/lecturers-list/search-lecturers.

Step 14.8: Stop execution.

Case B — GET request (redirectAttributes == null)

Step 15.1: Add error message to model.

Step 15.2: Add empty teachers list.

Step 15.3: Set currentPage = 1.

Step 15.4: Set totalPages = 1.

Step 15.5: Set pageSize.

Step 15.6: Set searchType = "name".

Step 15.7: Set keyword = "".

Step 15.8: Add currentCampusName = campusName.

Step 15.9: Add currentMajorName = majorName.

Step 15.10: Return "SearchLecturers" view.

Step 15.11: Stop execution.

SECTION 5 — MAIN TRY-BLOCK (NO EXCEPTION)

SECTION 5.1 — CASE 1: NO KEYWORD SUPPLIED

Step 16: If keyword is null OR keyword.trim() is empty:

Count total lecturers (no filter)

Step 17: Call service totalLecturersByCampusAndMajor(campusId, majorId).

DAO:

Step 17.1: Query MajorLecturersDAO.

Step 17.2: Return totalLecturers.

Fetch paginated lecturers (no search)

Step 18: Call getPaginatedLecturersByCampusAndMajor(campusId, majorId, firstResult, pageSize).

DAO:

Step 18.1: Query database with offset + limit.

Step 18.2: Return lecturers list.

SECTION 5.2 — CASE 2: KEYWORD SUPPLIED

Step 19: If searchType is null/blank → set searchType = "name".

Step 20: trimmedKeyword = keyword.trim().

Count lecturers by search filter

Step 21: Call service:

countSearchLecturersByCampusAndMajor(campusId, majorId, searchType, trimmedKeyword).

DAO:

Step 21.1: Build dynamic query by searchType.

Step 21.2: Execute count.

Step 21.3: Return totalLecturers.

Fetch paginated lecturers by search filter

Step 22: Call service:

searchLecturersByCampusAndMajor(campusId, majorId, searchType, trimmedKeyword, firstResult, pageSize).

DAO:

Step 22.1: Build dynamic search query.

Step 22.2: Apply searchType condition.

Step 22.3: Apply pagination.

Step 22.4: Execute query.

Step 22.5: Return lecturers list.

SECTION 6 — PAGINATION CALCULATION

Step 23: If totalLecturers == 0 → totalPages = 1.

Step 24: Else totalPages = ceil(totalLecturers / pageSize).

Step 25: If page < 1 → set page = 1.

Step 26: Else if page > totalPages AND totalPages > 0 → set page = totalPages.

SECTION 7 — POPULATE MODEL

Step 27: Add lecturers list to model under "teachers".
Step 28: Add currentPage = page.
Step 29: Add totalPages.
Step 30: Add pageSize.
Step 31: Add totalLecturers.
Step 32: Add searchType.
Step 33: Add keyword trimmed or "" if null.
Step 34: Add currentCampusName.
Step 35: Add currentMajorName.

SECTION 8 — HANDLE MESSAGES / ERRORS

Step 36: If successMessage exists → add "message".
Step 37: Else if error exists → add "error".
Step 38: Else if totalLecturers == 0 → add default message:
Step 38.1: "No lecturers found in {majorName} matching your search."

SECTION 9 — RETURN VIEW

Step 39: Return "SearchLecturers" view.
Step 40: Stop execution.

Activity diagram related to minor lecturer roles

AddMinorLecturerController

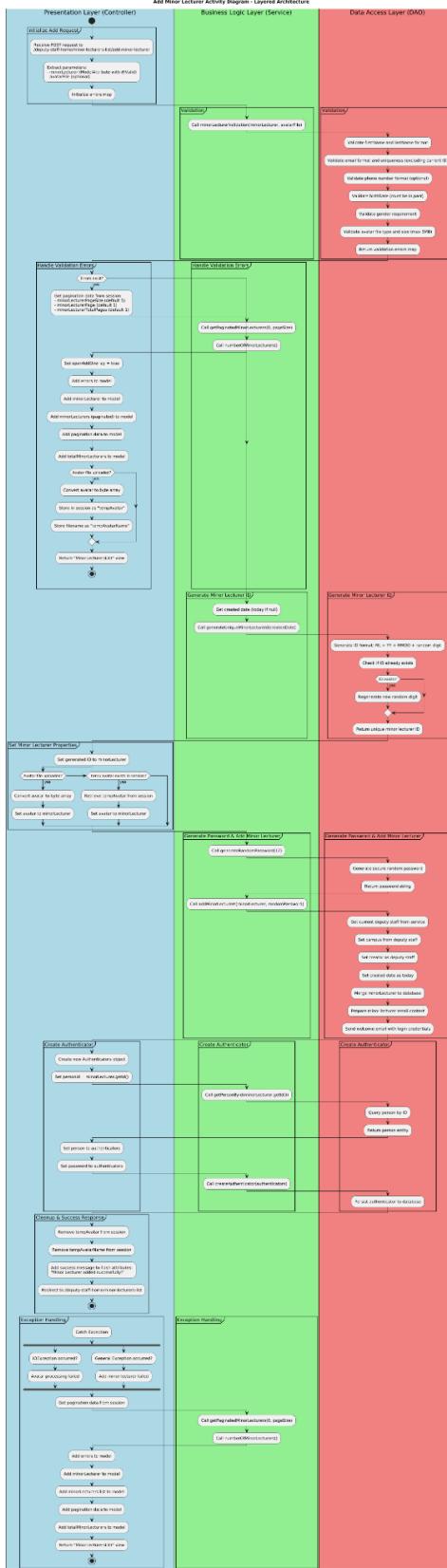


Diagram 43 Minor Lecturer Creation Workflow – Layered Architecture

MAIN FLOW — INITIAL REQUEST

- Step 1: Receive POST request to /deputy-staff-home/minor-lecturers-list/add-minor-lecturer.
 - Step 2: Extract minorLecturer (@Valid @ModelAttribute).
 - Step 3: Extract avatarFile (optional).
 - Step 4: Initialize an empty errors map.
-

VALIDATION PHASE

- Step 5: Call minorLecturerValidation(minorLecturer, avatarFile) in service.
 - Step 6: DAO validates firstName format.
 - Step 7: DAO validates lastName format.
 - Step 8: DAO validates email format.
 - Step 9: DAO validates email uniqueness (exclude current ID if updating).
 - Step 10: DAO validates phone number format (optional).
 - Step 11: DAO validates phone number uniqueness (exclude current ID).
 - Step 12: DAO validates birthDate must be in the past.
 - Step 13: DAO validates gender is not null.
 - Step 14: DAO validates avatar file type (must be image).
 - Step 15: DAO validates avatar file size ≤ 5MB.
 - Step 16: Return validation error map.
-

HANDLE VALIDATION ERRORS

- Step 17: Check if errors map is non-empty.
If errors exist → Step 18 to Step 35
- Step 18: Retrieve minorLecturerPageSize from session (default 5).
- Step 19: Retrieve minorLecturerPage from session (default 1).
- Step 20: Retrieve minorLecturerTotalPages from session (default 1).
- Step 21: Call getPaginatedMinorLecturers(0, pageSize) from service.
- Step 22: DAO executes query to load paginated minor lecturers.
- Step 23: Call numberOfMinorLecturers() from service.
- Step 24: DAO executes count query for minor lecturers.
- Step 25: Set openAddOverlay = true in model.
- Step 26: Add errors to model.
- Step 27: Add submitted minorLecturer object to model.

Step 28: Add paginated minor lecturers to model.
Step 29: Add pagination metadata to model.
Step 30: Add totalMinorLecturers to model.
Step 31: Check if avatarFile was uploaded.
Step 32: If yes, convert avatar to byte array.
Step 33: Store tempAvatar in session.
Step 34: Store tempAvatarName in session.
Step 35: Return "MinorLecturersList" view and stop execution.

GENERATE MINOR LECTURER ID

Step 36: Determine createdDate (use today if null).
Step 37: Call generateUniqueMinorLectureId(createdDate) in service.

DAO Steps:

Step 38: Extract year (YY) from createdDate.
Step 39: Extract month/day (MMDD) from createdDate.
Step 40: Generate a random digit.
Step 41: Build ID = ML + YY + MMDD + randomDigit.
Step 42: Check ID existence via personsService.existsPersonById.
Step 43: If exists → generate a new random digit.
Step 44: Return unique minor lecturer ID.

SET MINOR LECTURER PROPERTIES

Step 45: Set ID into minorLecturer.
Step 46: Check if avatarFile was uploaded.
If yes → Step 47
If no → Step 49
Step 47: Convert avatar to byte array.
Step 48: Set avatar into minorLecturer.
Step 49: Check if session contains tempAvatar.
Step 50: If yes → set avatar from session.

GENERATE PASSWORD & ADD MINOR LECTURER

Step 51: Call generateRandomPassword(12) in service.

001479794

Page 340 | 635 total

DAO Steps:

Step 52: Generate secure random sequence of letters, digits, symbols.

Step 53: Shuffle characters.

Step 54: Return password.

Service Flow:

Step 55: Call addMinorLecturers(minorLecturer, randomPassword).

DAO Steps:

Step 56: Set createdDate = today.

Step 57: Get current deputy staff from security context.

Step 58: Set minorLecturer.campus = deputyStaff.campus.

Step 59: Set minorLecturer.creator = deputyStaff.

Step 60: Persist minorLecturer into database via entityManager.merge().

Step 61: Prepare MinorLecturerEmailContext object.

Step 62: Send welcome email to lecturer with login credentials.

CREATE AUTHENTICATOR

Step 63: Create new Authenticators object.

Step 64: Set personId = minorLecturer.id.

Step 65: Call getPersonById(id) from PersonsService.

DAO Steps:

Step 66: Load person entity by ID.

Step 67: Return person.

Controller Steps:

Step 68: Set person to authenticator.

Step 69: Set password to authenticator.

Step 70: Call createAuthenticator(authenticators) from service.

DAO Steps:

Step 71: Persist authenticator to database.

CLEANUP & SUCCESS RESPONSE

Step 72: Remove tempAvatar from session.

Step 73: Remove tempAvatarName from session.

Step 74: Add success flash message: "*Minor Lecturer added successfully!*".

Step 75: Redirect to /deputy-staff-home/minor-lecturers-list.

Step 76: Stop execution.

EXCEPTION HANDLING

Step 77: Catch any Exception.

Parallel branch:

Step 78: If IOException → avatar processing failed.

Step 79: If other Exception → adding minor lecturer failed.

Step 80: Retrieve pagination data from session.

Step 81: Call getPaginatedMinorLecturers(0, pageSize).

Step 82: Call numberOfMinorLecturers().

Step 83: Add errors map to model.

Step 84: Add minorLecturer back to model.

Step 85: Add minorLecturers list to model.

Step 86: Add pagination data to model.

Step 87: Add totalMinorLecturers to model.

Step 88: Return "MinorLecturersList" view.

Step 89: Stop execution.

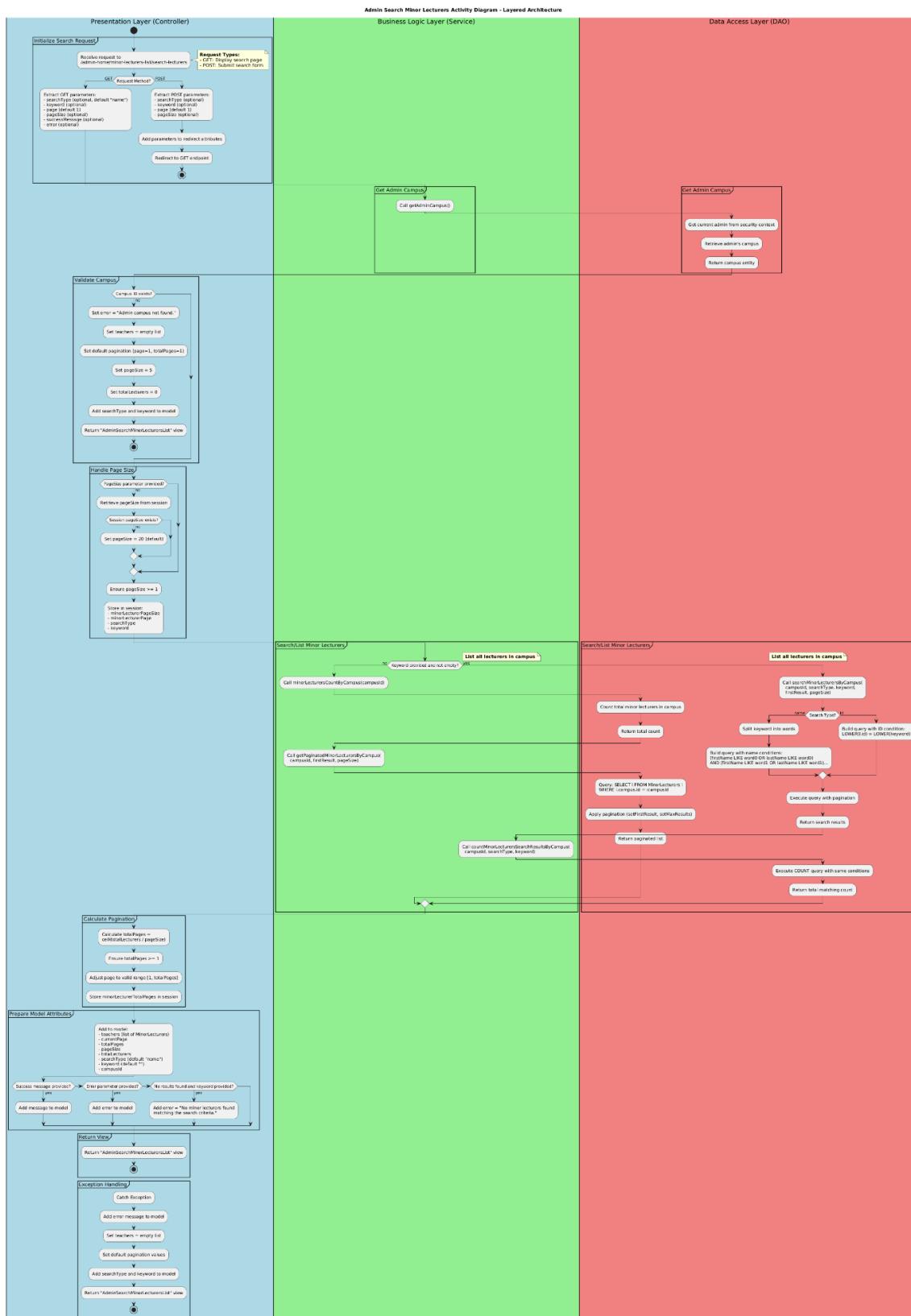


Diagram 44 Minor Lecturer Search Workflow

Step 1: Receive request to /admin-home/minor-lecturers-list/search-lecturers.

Step 2: Determine request method.

PROCESS POST SUBMISSION (REFURBISHMENT)

Step 3: If method == POST → extract searchType, keyword, page, pageSize.

Step 4: Add all parameters + flash attribute (successMessage/error if any) to redirect attribute.

Step 5: Redirect to the same GET endpoint with parameters.

Step 6: Stop execution (POST branch ends).

GET REQUEST HANDLING

Step 7: If method == GET → extract parameters:

- searchType (default is "name")
- keyword (can be null)
- page (default is 1)
- pageSize (can be null)
- successMessage (can be null)
- error (can be null)

GET ADMIN CAMPUS

Step 8: Call adminService.getAdminCampus() (get current admin's campus from SecurityContext).

Step 9: If campus == null or campus.id == null → go to BRANCH A (Campus not found).

HANDLE PAGE SIZE INPUT

Step 10: Check if pageSize is == null.

Step 11: If null → get from session.minorLecturerPageSize.

Step 12: If session.minorLecturerPageSize == null → set default pageSize = 20.

Step 13: If pageSize < 1 OR pageSize > 100 → set pageSize = 5 (reserve).

Step 14: Save session.minorLecturerPageSize = pageSize.

Step 15: Save session.minorLecturerSearchType = searchType.

Step 16: Save session.minorLecturerKeyword = keyword (can be null).

DETERMINING SEARCH OR LIST MODE

Step 17: Check if keyword != null && keyword.trim() != "".

Step 18: If there is NO keyword → go to LABEL B (List all in field).

Step 19: If there is a keyword → go to LABEL C (Search by criteria).

COUNT TOTAL RECORDS (PREPARE TO SHARE)

Step 20: Calculate firstResult = (page - 1) * pageSize.

BRANCH B — LIST ALL SPORTS LECTURERS IN CAMPUS

Step 21: Call minorLecturersService.minorLecturersCountByCampus(campusId).

Step 22: Store the result → totalLecturers.

Step 23: Call minorLecturersService.getPaginatedMinorLecturersByCampus(campusId, firstResult, pageSize).

Step 24: DAO execute: SELECT l FROM MinorLecturers l WHERE l.campus.id = :campusId + pagination.

Step 25: Store the returned list → teachers.

Step 26: Go to Step 35 (Pagination).

BRANCH C — SEARCH FOR ELEMENTARY SCHOOL TEACHERS BY CRITERIA

Step 27: Call minorLecturersService.searchMinorLecturersByCampus(campusId, searchType, keyword, firstResult, pageSize).

Step 28: Inside DAO → if searchType == "name":

- Split keyword into words

- Build dynamic WHERE with OR between firstName/lastName for each word

Step 29: If searchType == "id" → WHERE LOWER(l.id) = LOWER(:keyword).

Step 30: Execute query + pagination → return list → teachers.

Step 31: Call minorLecturersService.countMinorLecturersSearchResultsByCampus(campusId, searchType, keyword).

Step 32: DAO executes COUNT query with same condition.

Step 33: Store count → totalLecturers.

Step 34: Continue.

COMPLETE PAGE CALCULATION (GENERAL)

Step 35: Calculate totalPages = max(1, ceil(totalLecturers / pageSize)).

Step 36: Clip page = max(1, min(page, totalPages)).

Step 37: Save session.minorLecturerPage = page.

Step 38: Save session.minorLecturerTotalPages = totalPages.

BRANCH A — CAMPUS NOT FOUND

Step 39: If campus not found (from Step 9) →

Step 40: Set model.error = "Admin campus not found.".

Step 41: Set model.teachers = empty list.

Step 42: Set model.currentPage = 1, model.totalPages = 1, model.pageSize = 5, model.totalLecturers = 0.

Step 43: Add searchType & keyword to model.

Step 44: Go to Step 60 (Return to view).

MODEL FOUND (NORMAL)

Step 45: Set model.teachers = teachers.

Step 46: Set model.currentPage = page.

Step 47: Set model.totalPages = totalPages.

Step 48: Set model.pageSize = pageSize.

Step 49: Set model.totalLecturers = totalLecturers.

Step 50: Set model.searchType = searchType (or session value).

Step 51: Set model.keyword = keyword (or "").

Step 52: Set model.campusId = campusId.

MESSAGE HANDLING

Step 53: If successMessage exists → add model.successMessage.

Step 54: Otherwise, if error exists → add model.error.

Step 55: Otherwise, if provided keyword && totalLecturers == 0 → set model.error = "No sub-lecturers found that match the search criteria.".

RETURN MAIN VIEW

Step 60: Return "AdminSearchMinorLecturersList" view type.

Step 61: Stop execution.

EXCEPTION HANDLING

Step 62: Catch any Exception (including SecurityException, DataAccessException, etc.).

Step 63: Set model.error = "An error occurred: " + getMessage().

Step 64: Set model.teachers = empty list.

Step 65: Set default pagination: currentPage=1, totalPages=1, pageSize=pageSize (or 20), totalLecturers=0.

Step 66: Keep searchType and keywords in model.

Step 67: Return "AdminSearchMinorLecturersList" view.

Step 68: Stop execution.

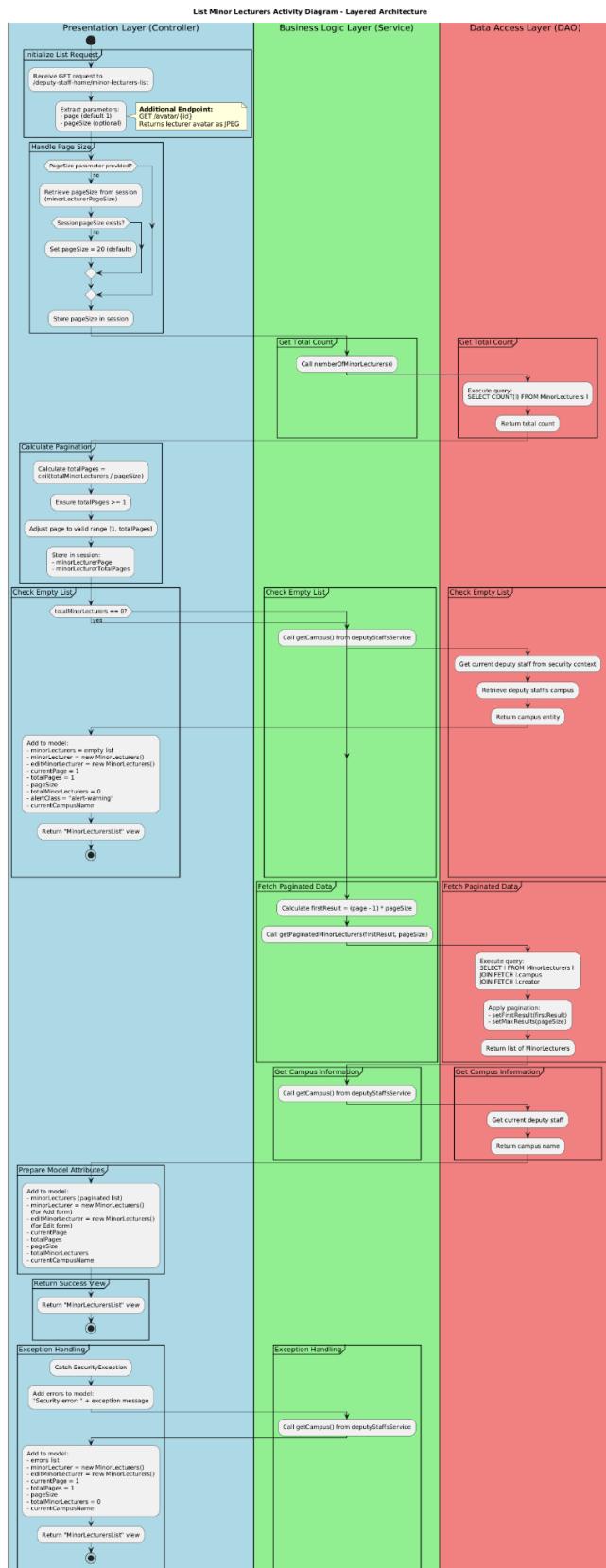


Diagram 45 Minor Lecturer Listing Workflow

MAIN FLOW — DEPUTY STAFF LIST MINOR LECTURERS

Step 1: Receive GET request to /deputy-staff-home/minor-lecturers-list.

Step 2: Read request parameter page (default = 1).

Step 3: Read request parameter pageSize (nullable).

Step 4: Access the HttpSession object.

Step 5: Enter the try-block.

HANDLE PAGE SIZE INPUT

Step 6: Check if pageSize == null.

Step 7: If pageSize is null, set pageSize = session.minorLecturerPageSize.

Step 8: Check if session.minorLecturerPageSize == null.

Step 9: If session minorLecturerPageSize is null, set default pageSize = 20.

Step 10: Check if pageSize < 1 OR pageSize > 100.

Step 11: If invalid, set pageSize = 5.

Step 12: Save session.minorLecturerPageSize = pageSize.

COUNT TOTAL MINOR LECTURERS

Step 13: Call minorLecturersService.numberOfMinorLecturers().

Step 14: DAO executes SELECT COUNT(I) FROM MinorLecturers I.

Step 15: Store the count in totalMinorLecturers.

COMPUTE PAGINATION

Step 16: Compute totalPages = max(1, ceil(totalMinorLecturers / pageSize)).

Step 17: Clamp page = clamp(page, 1, totalPages).

Step 18: Save session.minorLecturerPage = page.

Step 19: Save session.minorLecturerTotalPages = totalPages.

BRANCH A — NO MINOR LECTURER EXISTS

Step 20: Check if totalMinorLecturers == 0.

Step 21: If totalMinorLecturers == 0, go to Step 30.

LOAD PAGINATED MINOR LECTURERS (NORMAL CASE)

Step 22: Compute firstResult = (page - 1) * pageSize.

Step 23: Call minorLecturersService.getPaginatedMinorLecturers(firstResult, pageSize).

Step 24: DAO executes

SELECT I FROM MinorLecturers I

JOIN FETCH I.campus

JOIN FETCH I.creator

with setFirstResult(firstResult) and setMaxResults(pageSize).

Step 25: Store returned list into minorLecturers.

GET CURRENT CAMPUS NAME (SHARED FOR BOTH BRANCHES)

Step 26: Call deputyStaffsService.getCampus().

Step 27: DAO retrieves current deputy staff from security context → return campus entity.

Step 28: Extract campus name → currentCampusName.

Step 29: Continue to Step 40 (Populate model normal case).

BRANCH A — EMPTY LIST HANDLING

Step 30: Call deputyStaffsService.getCampus() (same as Step 26-27).

Step 31: Set model.minorLecturers = empty list.

Step 32: Set model.minorLecturer = new MinorLecturers().

Step 33: Set model.editMinorLecturer = new MinorLecturers().

Step 34: Set model.currentPage = 1.

Step 35: Set model.totalPages = 1.

Step 36: Set model.pageSize = pageSize.

Step 37: Set model.totalMinorLecturers = 0.

Step 38: Set model.alertClass = "alert-warning".

Step 39: Set model.currentCampusName = currentCampusName.

Step 40: Go to Step 55 (Return view).

POPULATE MODEL WITH DATA (NORMAL CASE)

Step 41: Set model.minorLecturers = minorLecturers (from Step 25).

Step 42: Set model.minorLecturer = new MinorLecturers() (for Add form).

Step 43: Set model.editMinorLecturer = new MinorLecturers() (for Edit form).

Step 44: Set model.currentPage = page.

Step 45: Set model.totalPages = totalPages.

Step 46: Set model.pageSize = pageSize.

Step 47: Set model.totalMinorLecturers = totalMinorLecturers.

Step 48: Set model.currentCampusName = currentCampusName.

RETURN MAIN VIEW

Step 55: Return view "MinorLecturersList".

Step 56: Stop execution.

EXCEPTION HANDLING — SECURITYEXCEPTION

Step 57: Detect a SecurityException being thrown.

Step 58: Catch the SecurityException as e.

Step 59: Set model.errors = List.of("Security error: " + e.getMessage()).

Step 60: Call deputyStaffsService.getCampus() để vẫn lấy được campus name.

Step 61: Set model.minorLecturer = new MinorLecturers().

Step 62: Set model.editMinorLecturer = new MinorLecturers().

Step 63: Set model.currentPage = 1.

Step 64: Set model.totalPages = 1.

Step 65: Set model.pageSize = pageSize.

Step 66: Set model.totalMinorLecturers = 0.

Step 67: Set model.currentCampusName = returned campus name (hoặc "" nếu null).

Step 68: Return "MinorLecturersList" view.

Step 69: Stop execution.

DeleteMinorLecturerController

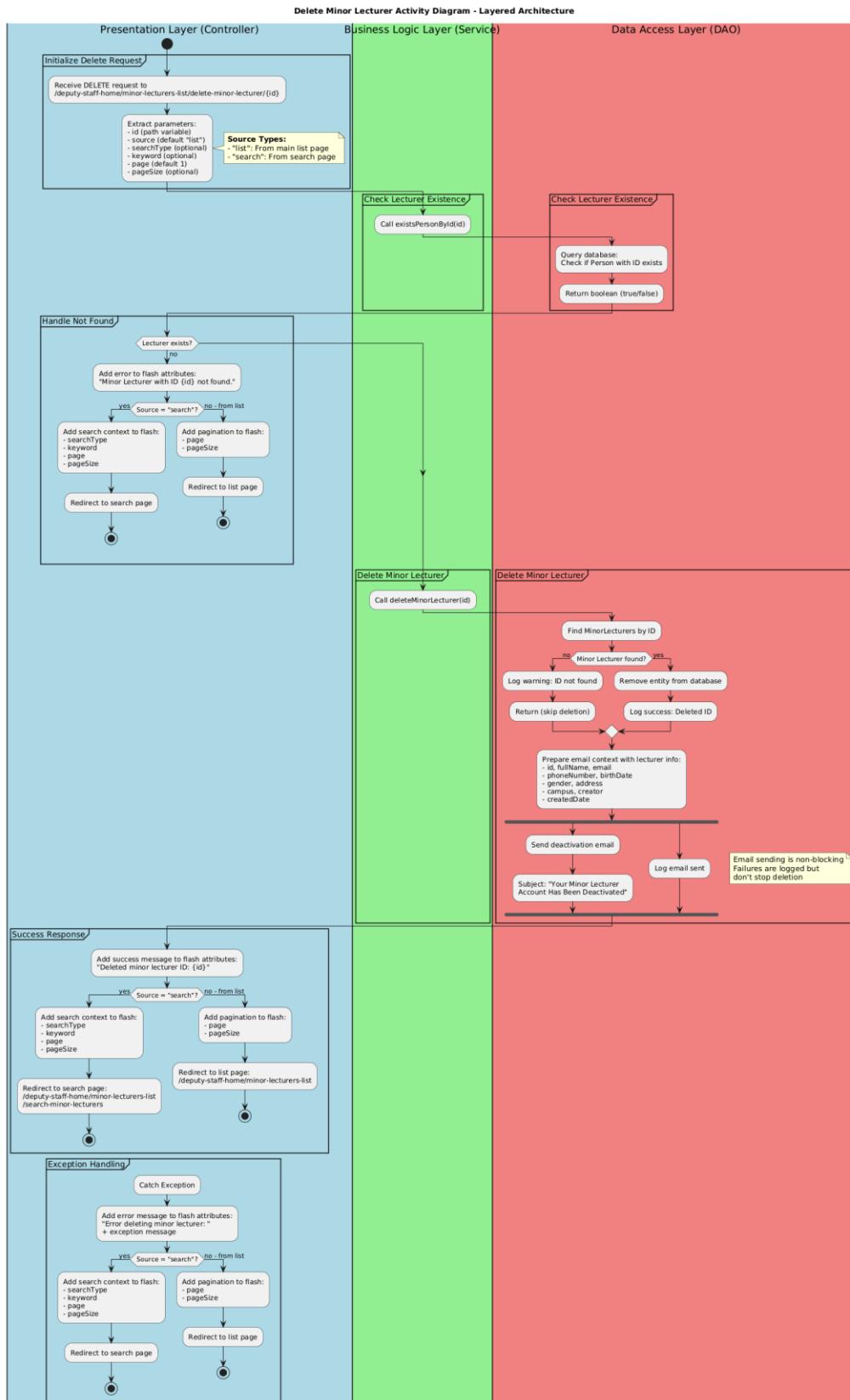


Diagram 46 Minor Lecturer Deletion Workflow

MAIN FLOW — DEPUTY STAFF DELETE MINOR LECTURER

Step 1: Receive DELETE request to /deputy-staff-home/minor-lecturers-list/delete-minor-lecturer/{id}.

Step 2: Extract path variable id.

Step 3: Extract request parameters:

- source (default "list")
- searchType (nullable)
- keyword (nullable)
- page (default 1)
- pageSize (nullable)

Step 4: Access the HttpSession object.

Step 5: Enter the try-block.

CHECK EXISTENCE

Step 6: Call personService.existsPersonById(id).

Step 7: DAO executes: SELECT COUNT(p) > 0 FROM Person p WHERE p.id = :id.

Step 8: If result == false → go to BRANCH A (Not Found).

BRANCH A — MINOR LECTURER NOT FOUND

Step 9: Add flash error = "Minor Lecturer with ID {id} not found.".

Step 10: If source == "search" →

- Step 11: Add flash searchType, keyword, page, pageSize.
- Step 12: Redirect to /deputy-staff-home/minor-lecturers-list/search-minor-lecturers.
- Step 13: Stop execution.

Step 14: Else (source == "list") →

- Step 15: Add flash page, pageSize.
- Step 16: Redirect to /deputy-staff-home/minor-lecturers-list.
- Step 17: Stop execution.

DELETE MINOR LECTURER

Step 18: Call minorLecturersService.deleteMinorLecturer(id).

Step 19: Inside service → find MinorLecturers entity by id.

Step 20: If entity not found → log warning "Attempt to delete non-existent MinorLecturer ID: {id}" → continue (no exception).

Step 21: If found → remove entity from database (entityManager.remove() hoặc repository.delete()).

Step 22: Log info: "Successfully deleted MinorLecturer ID: {id}".

PREPARE AND SEND DEACTIVATION EMAIL (NON-BLOCKING)

Step 23: Extract all necessary fields from entity: id, fullName, email, phoneNumber, birthDate, gender, address, campus.name, creator.name, createdDate.

Step 24: Build email context map/object.

Step 25: Fork async task → emailService.sendMinorLecturerDeactivationEmail(email, context).

Step 26: Subject cố định: "Your Minor Lecturer Account Has Been Deactivated".

Step 27: Log "Deactivation email queued for ID: {id}".

Step 28: Any exception trong email sending → catch & log only, không ném ra ngoài (deletion vẫn thành công).

SUCCESS REDIRECT

Step 29: Add flash success = "Deleted minor lecturer ID: {id}".

Step 30: If source == "search" →

 Step 31: Add flash searchType, keyword, page, pageSize.

 Step 32: Redirect to /deputy-staff-home/minor-lecturers-list/search-minor-lecturers.

 Step 33: Stop execution.

Step 34: Else (source == "list") →

 Step 35: Add flash page, pageSize.

 Step 36: Redirect to /deputy-staff-home/minor-lecturers-list.

 Step 37: Stop execution.

EXCEPTION HANDLING

Step 38: Catch any Exception e (DataAccessException, ConstraintViolationException, SecurityException, v.v.).

Step 39: Add flash error = "Error deleting minor lecturer: " + e.getMessage().

Step 40: Log full stack trace với level ERROR.

Step 41: If source == "search" →

 Step 42: Add flash searchType, keyword, page, pageSize.

 Step 43: Redirect to search page.

 Step 44: Stop execution.

Step 45: Else →

 Step 46: Add flash page, pageSize.

 Step 47: Redirect to list page.

 Step 48: Stop execution.

Activity diagram related to student roles

AddStudentController

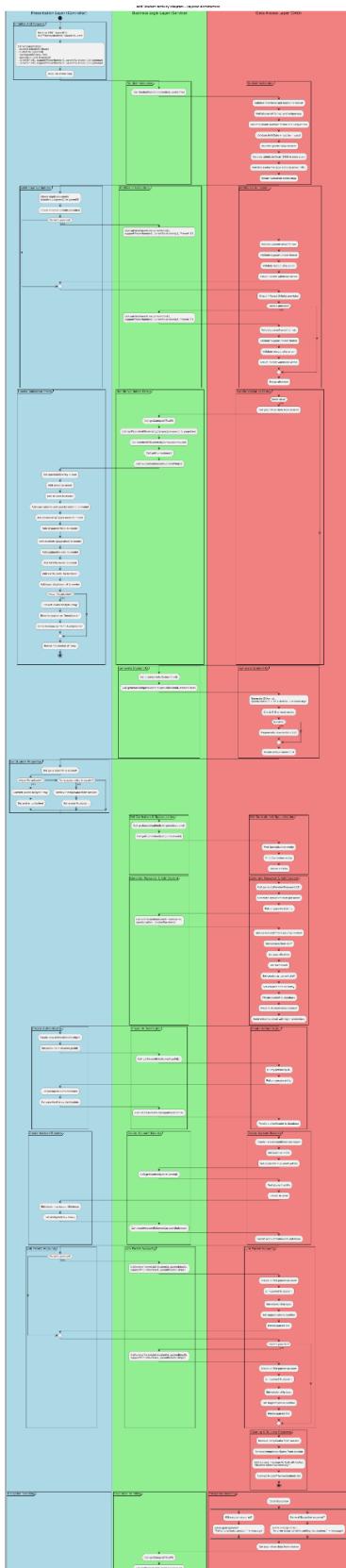


Diagram 47 Minor Lecturer Update Workflow

MAIN FLOW — STAFF ADD STUDENT

Step 1: Receive POST request to /staff-home/students-list/add-student.

Step 2: Bind @ModelAttribute Student student.

Step 3: Bind @RequestParam(required = false) MultipartFile avatarFile.

Step 4: Extract required curriculumId, specializationId.

Step 5: Extract optional parent fields (email1/2, phone1/2, relationship1/2).

Step 6: Initialize LinkedHashMap<String, String> errors = new LinkedHashMap<>().

Step 7: Enter try-block.

VALIDATION PHASE

Step 8: Call studentValidationService.validate(student, avatarFile) → returns Map<String, String>.

Step 9: Perform additional controller-level validation: duplicate email between student vs parent1 vs parent2.

Step 10: If parentEmail1 provided → call parentLinkValidationService.validateParentLink(email1, phone1, relationship1, "Parent 1").

Step 11: If parentEmail2 provided → call parentLinkValidationService.validateParentLink(email2, phone2, relationship2, "Parent 2").

Step 12: Merge all error maps into final errors.

BRANCH A — VALIDATION FAILED

Step 13: If errors.isEmpty() → go to Step 50 (Return same page with errors).

GENERATE UNIQUE STUDENT ID

Step 14: Determine createdDate = today nếu student.createdDate == null.

Step 15: Call studentService.generateUniqueId(specializationId, createdDate).

Step 16: Inside DAO → generate format: specializationCode + YY + MMDD + randomDigit.

Step 17: Loop until find unused ID (max 10 attempts, else throw exception).

Step 18: Set student.id = generatedId.

HANDLE AVATAR

Step 19: If avatarFile != null && !avatarFile.isEmpty() →

 Step 20: Convert to byte[] và set student.avatar = bytes.

Step 21: Else if session contains "tempAvatar" →

 Step 22: Set student.avatar = session.tempAvatar.

FETCH CURRICULUM & SPECIALIZATION ENTITIES

Step 23: Call specializationService.getById(specializationId) → throw if not found.

Step 24: Call curriculumService.getById(curriculumId) → throw if not found.

GENERATE PASSWORD & PERSIST STUDENT

Step 25: Call passwordService.generateRandomPassword(12) → strong password.

Step 26: Call studentService.addStudent(student, curriculumEntity, specializationEntity, rawPassword).

Step 27: Inside service:

- Get current staff from SecurityContext
- Set student.campus = staff.campus
- Set student.specialization = specializationEntity
- Set student.curriculum = curriculumEntity
- Set student.creator = currentStaff
- Set student.createdDate = today
- Merge/persist student

Step 28: Fire-and-forget welcome email với username = student.id, password = rawPassword.

CREATE AUTHENTICATOR ACCOUNT

Step 29: Create new Authenticators entity.

Step 30: Set authenticators.personId = student.id.

Step 31: Call personService.get.GetById(student.id) → attach to authenticators.person.

Step 32: Set authenticators.password = BCrypt.encode(rawPassword).

Step 33: Call authenticatorsService.create(authenticators).

CREATE ACCOUNT BALANCE

Step 34: Create new AccountBalances entity.

Step 35: Set balance = 0.0, studentId = student.id, lastUpdated = now().

Step 36: Attach student entity.

Step 37: Call accountBalancesService.create(accountBalances).

CREATE PARENT LINKS

Step 38: If parentEmail1 provided →

Step 39: Call parentLinkService.createParentLink(student.id, email1, phone1, relationship1).

Step 40: If parentEmail2 provided →

Step 41: Call parentLinkService.createParentLink(student.id, email2, phone2, relationship2).

SUCCESS CLEANUP & REDIRECT

Step 42: Remove "tempAvatar" and "tempAvatarName" from session.

Step 43: Add flash success = "Student added successfully!".

Step 44: Redirect to /staff-home/students-list.

Step 45: Stop execution.

BRANCH A — VALIDATION ERRORS HANDLING

Step 50: Retrieve page & pageSize from session (fallback 1 và 20).

Step 51: Call staffService.getCampusOfCurrentStaff().

Step 52: Call studentService.getPaginatedStudentsByCampus(campusId, 0, pageSize).

Step 53: Call studentService.numberStudentsByCampus(campusId).

Step 54: Call curriculumService.getCurriculums().

Step 55: Call specializationService.getByMajor(staffMajor).

Step 56: Set model.openAddOverlay = true.

Step 57: Add errors map → model.

Step 58: Add student object (with entered data) → model.

Step 59: Add curriculumId, specializationId → model.

Step 60: Add all parent fields → model.

Step 61: Add RelationshipType enum values → model.

Step 62: If avatarFile uploaded → convert to byte[] → session "tempAvatar" + "tempAvatarName".

Step 63: Add paginated students list, pagination data, totalStudents, curriculums, specializations → model.

Step 64: Return view "StudentsList".

Step 65: Stop execution.

EXCEPTION HANDLING

Step 66: Catch IOException → errors.put("general", "Failed to process avatar: " + message).

Step 67: Catch any other Exception → errors.put("general", "An error occurred while adding the student: " + message).

Step 68: Log full stack trace.

Step 69: Execute same steps 50–64 (reload page with full context + errors).

Step 70: Return "StudentsList" view.

Step 71: Stop execution.

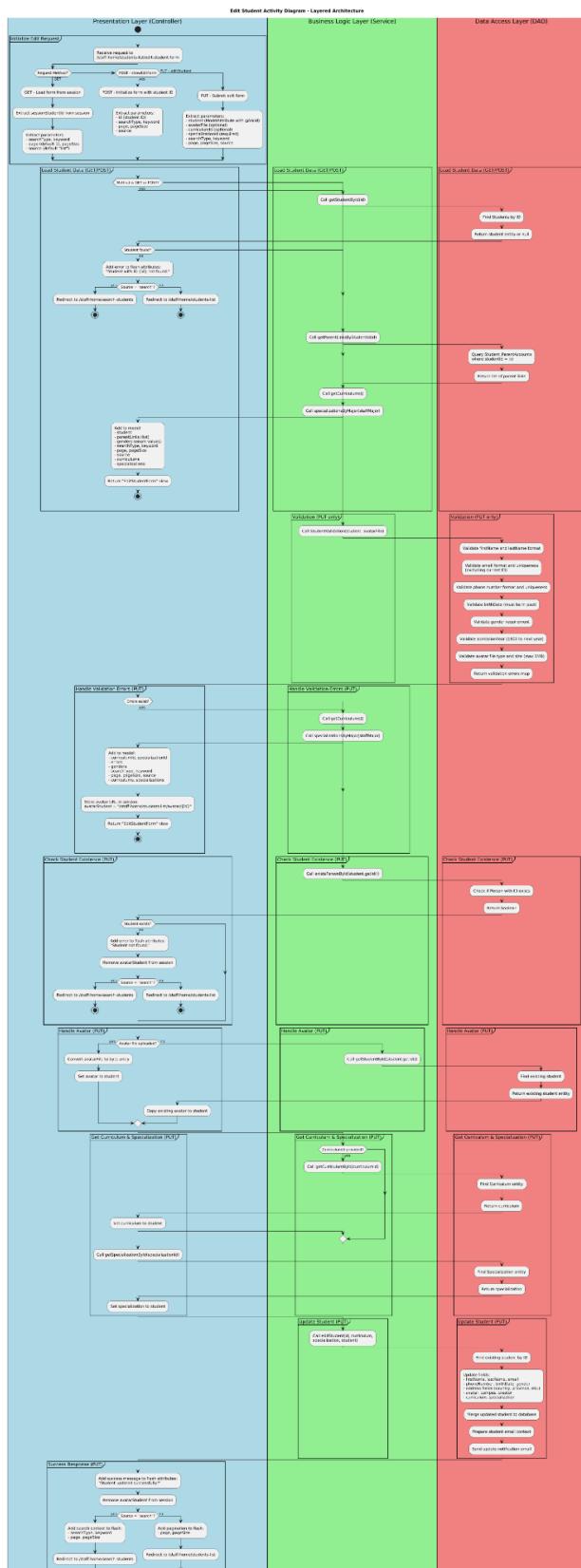


Diagram 48 Student Update Workflow

MAIN FLOW — STAFF EDIT STUDENT (3 METHODS IN ONE ENDPOINT)

Step 1: Receive request to /staff-home/students-list/edit-student-form.

Step 2: Determine HTTP method.

METHOD GET — LOAD EDIT FORM FROM SESSION

Step 3: If GET →

Step 4: Extract sessionStudentId from HttpSession.

Step 5: If sessionStudentId == null → redirect to list (no context).

Step 6: Set id = sessionStudentId.

Step 7: Extract searchType, keyword, page, pageSize, source (default "list").

Step 8: Go to Step 30 (Common load student data).

METHOD POST — INITIALIZE EDIT FORM (SHOW FORM)

Step 9: If POST →

Step 10: Extract required parameter id (student ID).

Step 11: Extract searchType, keyword, page, pageSize, source.

Step 12: Save id vào session.sessionStudentId = id.

Step 13: Go to Step 30 (Common load student data).

METHOD PUT — SUBMIT EDIT FORM

Step 14: If PUT →

Step 15: Bind @ModelAttribute @Valid Student student.

Step 16: Bind @RequestParam(required = false) MultipartFile avatarFile.

Step 17: Extract curriculumId (nullable), specializationId (required).

Step 18: Extract searchType, keyword, page, pageSize, source.

Step 19: Enter try-block for PUT.

VALIDATION (PUT ONLY)

Step 20: Call studentValidationService.validate(student, avatarFile) → returns error map.

Step 21: If BindingResult has errors OR validation map not empty → go to BRANCH A (Validation failed).

CHECK STUDENT EXISTENCE (PUT)

Step 22: Call personService.existsPersonById(student.getId()).

Step 23: If false → go to BRANCH B (Student not found).

HANDLE AVATAR (PUT)

Step 24: If avatarFile != null && !avatarFile.isEmpty() →

Step 25: Convert to byte[] → student.avatar = bytes.

Step 26: Else →

Step 27: Call studentService.getStudentById(student.getId()).

Step 28: Copy existing avatar bytes → student.avatar.

FETCH CURRICULUM & SPECIALIZATION (PUT)

Step 29: If curriculumId provided → call curriculumService.getById() → set student.curriculum.

Step 30: Call specializationService.getById(specializationId) → set student.specialization.

UPDATE STUDENT (PUT)

Step 31: Call studentService.editStudent(student).

Step 32: Inside service → find existing entity → copy all updatable fields (name, email, phone, birthDate, gender, address, avatar, curriculum, specialization).

Step 33: Merge entity → persist changes.

Step 34: Fire-and-forget update notification email to student.

SUCCESS RESPONSE (PUT)

Step 35: Add flash success = "Student updated successfully!".

Step 36: Remove session.sessionStudentId và session.avatarStudent.

Step 37: If source == "search" →

Step 38: Add flash searchType, keyword, page, pageSize.

Step 39: Redirect to /staff-home/search-students.

Step 40: Stop execution.

Step 41: Else →

Step 42: Add flash page, pageSize.

Step 43: Redirect to /staff-home/students-list.

Step 44: Stop execution.

COMMON LOAD STUDENT DATA (GET + POST showForm)

Step 45: Call studentService.getStudentById(id).

Step 46: If null → go to BRANCH B (Student not found).

Step 47: Call parentLinkService.getParentLinksByStudentId(id) → return list.

Step 48: Call curriculumService.getCurriculums().

Step 49: Call specializationService.getByMajor(staffMajor).

Step 50: Add to model:

- student

- parentLinks

- Gender enum values

- searchType, keyword, page, pageSize, source
- curriculums list
- specializations list

Step 51: Set session.avatarStudent = "/staff-home/students-list/avatar/" + id.

Step 52: Return view "EditStudentForm".

Step 53: Stop execution.

BRANCH A — VALIDATION FAILED (PUT)

Step 54: Call curriculumService.getCurriculums().

Step 55: Call specializationService.getByMajor(staffMajor).

Step 56: Add to model: errors map, curriculumId, specializationId, Gender enum, curriculums, specializations, search context.

Step 57: Keep session.avatarStudent = current avatar URL.

Step 58: Return "EditStudentForm" view.

Step 59: Stop execution.

BRANCH B — STUDENT NOT FOUND (ALL METHODS)

Step 60: Add flash error = "Student with ID {id} not found.".

Step 61: Remove session.sessionStudentId và session.avatarStudent.

Step 62: If source == "search" → redirect to search page với context.

Step 63: Else → redirect to /staff-home/students-list với page/pageSize.

Step 64: Stop execution.

EXCEPTION HANDLING (PUT ONLY)

Step 65: Catch IOException → errors.put("general", "Failed to process avatar: " + msg).

Step 66: Catch DataAccessException → errors.put("general", "Database error: " + msg).

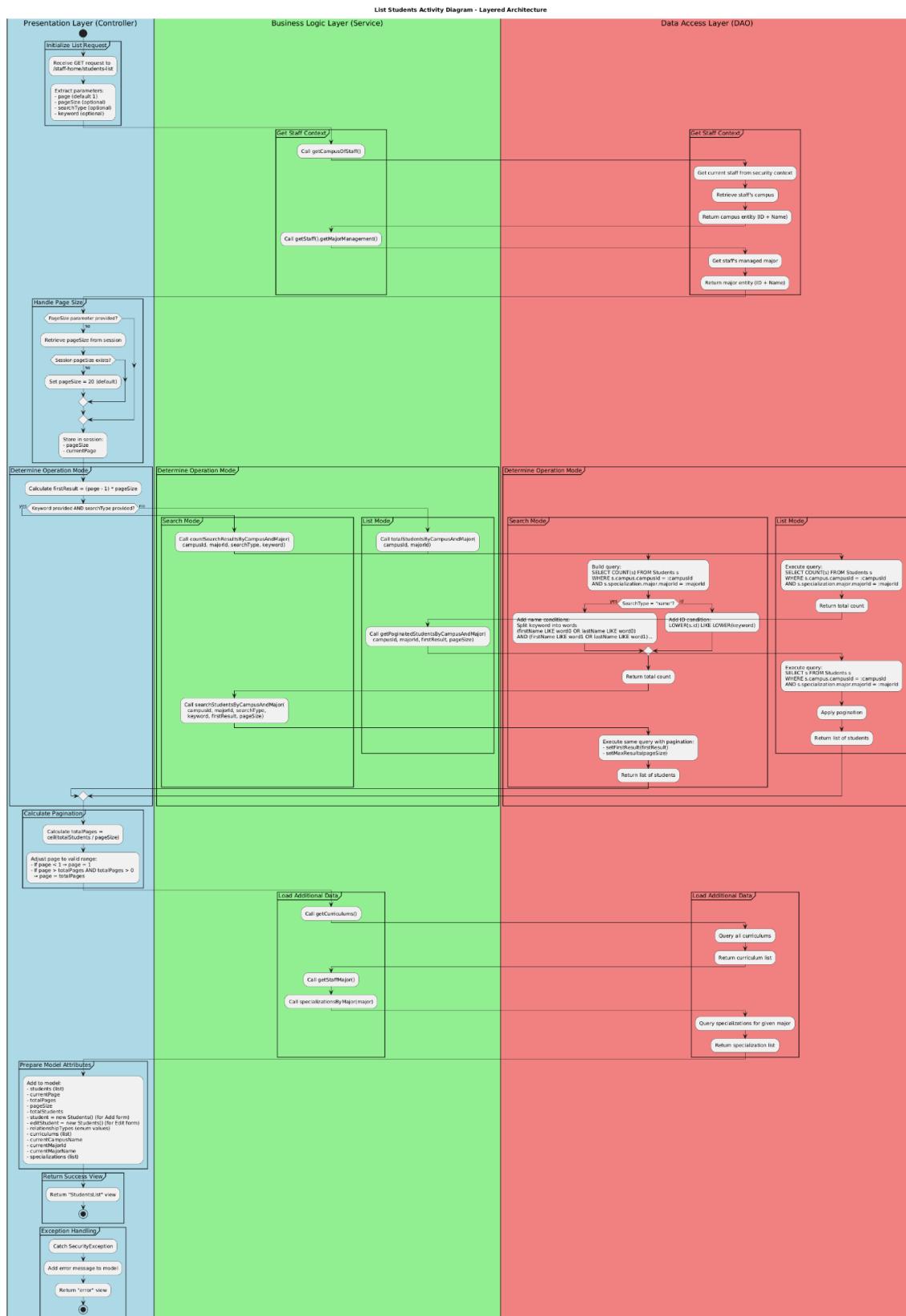
Step 67: Catch any Exception → errors.put("general", "Unexpected error: " + msg).

Step 68: Log full stack trace.

Step 69: Execute Step 54–58 (reload form with errors).

Step 70: Return "EditStudentForm" view.

Step 71: Stop execution.


Diagram 49 Student Listing Workflow

MAIN FLOW — STAFF LIST STUDENTS (với cả chế độ List thường + Search)

Step 1: Receive GET request to /staff-home/students-list.

Step 2: Read request parameter page (default = 1).

Step 3: Read request parameter pageSize (nullable).

Step 4: Read request parameter searchType (nullable).

Step 5: Read request parameter keyword (nullable).

Step 6: Access the HttpSession object.

Step 7: Enter the try-block.

GET CURRENT STAFF CONTEXT

Step 8: Call staffsService.getCampusOfStaff().

Step 9: Extract campusId và campusName từ entity trả về.

Step 10: Call staffsService.getStaff().getMajorManagement().getMajorId() → majorId.

Step 11: Call staffsService.getStaffMajor() → lấy Major entity → majorName.

HANDLE PAGE SIZE INPUT

Step 12: Check if pageSize == null.

Step 13: If null → lấy từ session.pageSize.

Step 14: If session.pageSize == null → set default pageSize = 20.

Step 15: Validate pageSize: nếu < 1 hoặc > 100 → set fallback = 5.

Step 16: Save session.pageSize = pageSize.

Step 17: Save session.currentPage = page (sẽ được override sau khi clamp).

DETERMINE MODE: SEARCH OR NORMAL LIST

Step 18: Compute firstResult = (page - 1) * pageSize.

Step 19: Check if (keyword != null && !keyword.trim().isBlank() && searchType != null).

Step 20: If YES → go to BRANCH B (Search mode).

Step 21: If NO → go to BRANCH A (Normal list mode).

BRANCH A — NORMAL LIST MODE

Step 22: Call studentsService.totalStudentsByCampusAndMajor(campusId, majorId).

Step 23: DAO executes COUNT query với WHERE campus + major.

Step 24: Store result → totalStudents.

Step 25: Call studentsService.getPaginatedStudentsByCampusAndMajor(campusId, majorId, firstResult, pageSize).

Step 26: DAO executes SELECT s FROM Students s WHERE campus + major + pagination.

Step 27: Store returned list → students.

Step 28: Go to Step 40.

BRANCH B — SEARCH MODE

Step 29: Trim keyword = keyword.trim().

Step 30: Call studentsService.countSearchResultsByCampusAndMajor(campusId, majorId, searchType, keyword).

Step 31: Inside DAO →

- Nếu searchType == "name" → split keyword thành words → build dynamic AND (OR firstName/lastName LIKE word)

- Nếu searchType == "id" → LOWER(s.id) LIKE LOWER(keyword)

- Thêm điều kiện campusId + majorId

Step 32: Execute COUNT query → return totalStudents.

Step 33: Call studentsService.searchStudentsByCampusAndMajor(campusId, majorId, searchType, keyword, firstResult, pageSize).

Step 34: Execute cùng query như trên nhưng SELECT s + pagination → return list → students.

Step 35: Continue.

COMPUTE PAGINATION (COMMON)

Step 40: Compute totalPages = (int) Math.ceil((double) totalStudents / pageSize).

Step 41: If totalPages == 0 → totalPages = 1.

Step 42: Clamp page:

- if (page < 1) → page = 1

- if (page > totalPages && totalPages > 0) → page = totalPages

Step 43: Override session.currentPage = page (sau khi clamp).

LOAD ADDITIONAL DATA (SHARED)

Step 44: Call curriculumService.getCurriculums() → list toàn bộ curriculum.

Step 45: Call specializationService.specializationsByMajor(majorId) → list specialization của major hiện tại.

POPULATE MODEL

Step 46: Set model.students = students.

Step 47: Set model.currentPage = page.

Step 48: Set model.totalPages = totalPages.

Step 49: Set model.pageSize = pageSize.

Step 50: Set model.totalStudents = totalStudents.

Step 51: Set model.student = new Students() (cho Add overlay).

Step 52: Set model.editStudent = new Students() (cho Edit overlay).

Step 53: Set model.relationshipTypes = RelationshipToStudent.values().

Step 54: Set model.curriculums = curriculum list.

Step 55: Set model.currentCampusName = campusName.

Step 56: Set model.currentMajorId = majorId.

Step 57: Set model.currentMajorName = majorName.

Step 58: Set model.specializations = specialization list.

RETURN MAIN VIEW

Step 59: Return view "StudentsList".

Step 60: Stop execution.

EXCEPTION HANDLING — SECURITYEXCEPTION

Step 61: Detect a SecurityException being thrown (ví dụ: staff không có campus/major).

Step 62: Catch the SecurityException as e.

Step 63: Set model.error = e.getMessage().

Step 64: Return view "error".

Step 65: Stop execution.

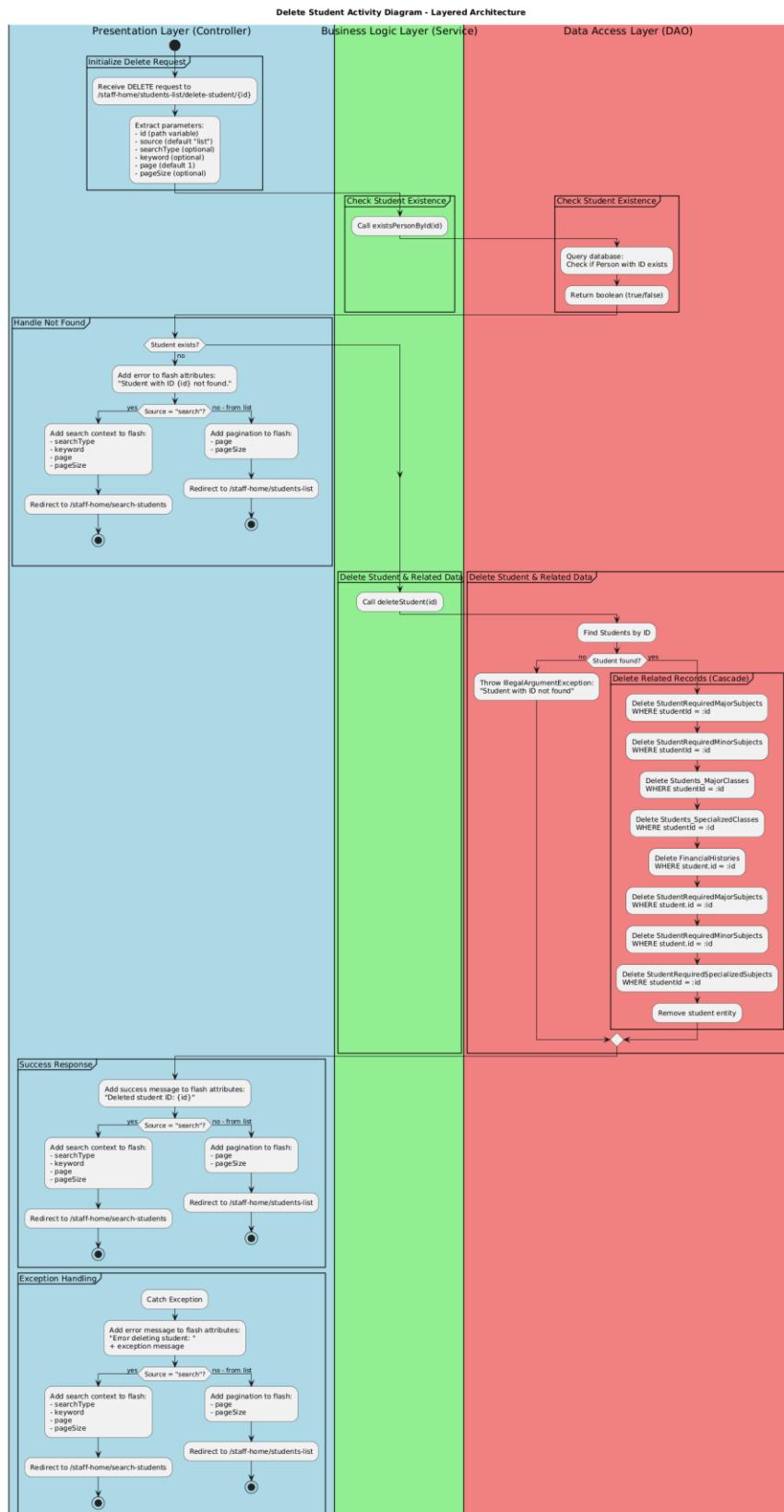


Diagram 50 Student Deletion Workflow

MAIN FLOW — STAFF DELETE STUDENT

Step 1: Receive DELETE request to /staff-home/students-list/delete-student/{id}.

Step 2: Extract path variable id.

Step 3: Extract request parameters:

- source (default "list")
- searchType (nullable)
- keyword (nullable)
- page (default 1)
- pageSize (nullable)

Step 4: Access the HttpSession object.

Step 5: Enter the try-block.

CHECK EXISTENCE

Step 6: Call personService.existsPersonById(id).

Step 7: DAO executes: SELECT COUNT(p) > 0 FROM Person p WHERE p.id = :id.

Step 8: If result == false → go to BRANCH A (Not Found).

BRANCH A — STUDENT NOT FOUND

Step 9: Add flash error = "Student with ID {id} not found.".

Step 10: If source == "search" →

Step 11: Add flash searchType, keyword, page, pageSize.

Step 12: Redirect to /staff-home/search-students.

Step 13: Stop execution.

Step 14: Else (source == "list") →

Step 15: Add flash page, pageSize.

Step 16: Redirect to /staff-home/students-list.

Step 17: Stop execution.

DELETE STUDENT & CASCADE ALL RELATED DATA

Step 18: Call studentsService.deleteStudent(id).

Step 19: Inside service → find Students entity by id.

Step 20: If entity not found → throw IllegalArgumentException("Student with ID not found") → sẽ bị catch ở Step 38.

Step 21: Begin transaction (REQUIRED).

CASCADE DELETE RELATED RECORDS

Step 22: Delete from StudentRequiredMajorSubjects WHERE studentId = :id.

Step 23: Delete from StudentRequiredMinorSubjects WHERE studentId = :id.
Step 24: Delete from Students_MajorClasses WHERE studentId = :id.
Step 25: Delete from Students_SpecializedClasses WHERE studentId = :id.
Step 26: Delete from FinancialHistories WHERE student.id = :id.
Step 27: Delete from StudentRequiredSpecializedSubjects WHERE studentId = :id.
Step 28: Delete from Student_ParentAccounts WHERE studentId = :id (nếu có).
Step 29: Delete from Authenticators WHERE personId = :id (nếu có).
Step 30: Delete from AccountBalances WHERE studentId = :id (nếu có).
Step 31: Finally remove Students entity itself (entityManager.remove() hoặc repository.delete()).
Step 32: Commit transaction.
Step 33: Log info: "Successfully deleted Student ID: {id} and all related data".

SEND DEACTIVATION EMAIL (NON-BLOCKING)

Step 34: Extract student info (fullName, email) trước khi entity bị xóa hoàn toàn (có thể cache tạm).
Step 35: Fork async task → emailService.sendStudentDeactivationEmail(email, fullName).
Step 36: Subject: "Your Student Account Has Been Deactivated".
Step 37: Any exception trong email → log only, không ảnh hưởng deletion.

SUCCESS REDIRECT

Step 38: Add flash success = "Deleted student ID: {id}".
Step 39: If source == "search" →
 Step 40: Add flash searchType, keyword, page, pageSize.
 Step 41: Redirect to /staff-home/search-students.
 Step 42: Stop execution.
Step 43: Else (source == "list") →
 Step 44: Add flash page, pageSize.
 Step 45: Redirect to /staff-home/students-list.
 Step 46: Stop execution.

EXCEPTION HANDLING

Step 47: Catch IllegalArgumentException, DataAccessException, ConstraintViolationException, SecurityException, or any Exception e.
Step 48: Rollback transaction nếu đang mở.
Step 49: Add flash error = "Error deleting student: " + e.getMessage().
Step 50: Log full stack trace với level ERROR.

Step 51: If source == "search" →

Step 52: Add flash searchType, keyword, page, pageSize.

Step 53: Redirect to /staff-home/search-students.

Step 54: Stop execution.

Step 55: Else →

Step 56: Add flash page, pageSize.

Step 57: Redirect to /staff-home/students-list.

Step 58: Stop execution.

Activity diagram related to minor classes module

AddClassController

001479794

Page **370** | **635** total

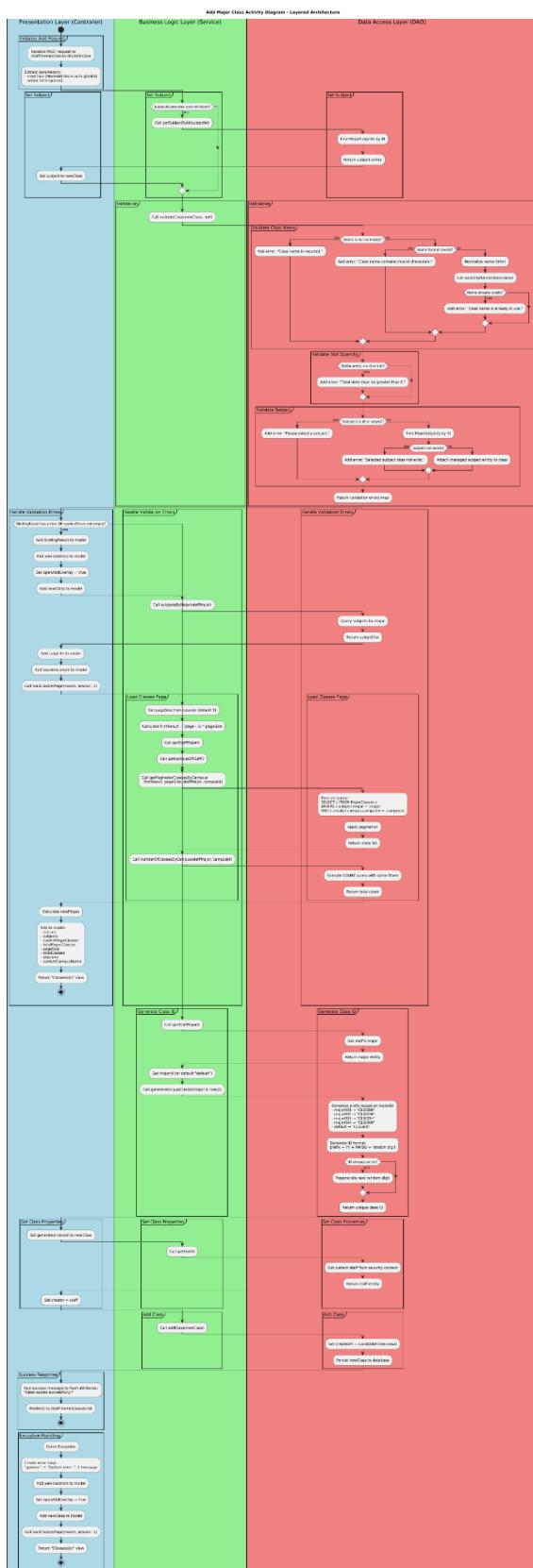


Diagram 51 Major Class Creation Workflow

001479794

Page 371 | 635 total

Step 1: Receive POST request to /staff-home/classes-list/add-class.

Step 2: Bind @ModelAttribute("newClass") @Valid MajorClasses newClass.

Step 3: Extract required parameter subjectId.

Step 4: Enter try-block.

ATTACH SUBJECT ENTITY

Step 5: If subjectId is provided and not blank →

Step 6: Call majorSubjectsService.getSubjectById(subjectId).

Step 7: If subject found → set newClass.subject = managed entity.

VALIDATION PHASE

Step 8: Call classValidationService.validateClass(newClass, null) → returns Map<String, String>.

Step 9: Inside validation service →

- If className is null or blank → error "Class name is required."
- If className contains invalid characters → error "Class name contains invalid characters."
- Trim and normalize name
- Call repository.existsByNameClass(normalizedName) → if true → error "Class name is already in use."

Step 10: If slotQuantity ≤ 0 or null → error "Total slots must be greater than 0."

Step 11: If newClass.subject == null → error "Please select a subject."

Step 12: Merge all validation errors into serviceErrors map.

BRANCH A — VALIDATION FAILED

Step 13: If BindingResult.hasErrors() OR serviceErrors is not empty → go to Step 50.

GENERATE UNIQUE CLASS ID

Step 14: Call staffsService.getStaffMajor() → retrieve current staff's Major entity.

Step 15: Extract majorId.

Step 16: Call classService.generateUniqueId(majorId, LocalDate.now()).

Step 17: Inside generator →

- Map majorId to prefix:

major001 → "CLSGBH"

major002 → "CLSGCH"

major003 → "CLSGDH"

major004 → "CLSGKH"

others → "CLSGEN"

- Format: prefix + YY + MMDD + randomDigit (0–9)

- Loop up to 10 times until a unique ID is found

Step 18: Set newClass.classId = generated unique ID.

SET CREATOR

Step 19: Call staffsService.getStaff() → get current staff from SecurityContext.

Step 20: Set newClass.creator = currentStaff.

PERSIST NEW CLASS

Step 21: Call majorClassesService.addClass(newClass).

Step 22: Inside service →

- Set newClass.createdAt = LocalDateTime.now()
- Persist entity (repository.save() or entityManager.persist())

Step 23: Commit transaction.

SUCCESS RESPONSE

Step 24: Add flash attribute success = "Class added successfully!".

Step 25: Redirect to /staff-home/classes-list.

Step 26: Stop execution.

BRANCH A — VALIDATION OR EXCEPTION → RETURN SAME PAGE WITH ERRORS

Step 50: Add BindingResult and serviceErrors to model.

Step 51: Set model.openAddOverlay = true.

Step 52: Add newClass (with user-entered data) to model.

Step 53: Call majorSubjectsService.subjectsByMajor(staffMajor) → load all subjects of current major.

Step 54: Add subjects list to model.

Step 55: Add SessionType enum values to model.

RELOAD CURRENT CLASSES PAGE (FORCE PAGE 1)

Step 56: Retrieve pageSize from session (default = 5).

Step 57: Compute firstResult = 0.

Step 58: Call staffsService.getStaffMajor() and getCampusOfStaff().

Step 59: Call majorClassesService.getPaginatedClassesByCampus(firstResult, pageSize, staffMajor, campusId).

Step 60: DAO executes:

```
SELECT c FROM MajorClasses c
WHERE c.subject.major = :major
AND c.creator.campus.campusId = :campusId
```

+ pagination

Step 61: Call majorClassesService.numberOfClassesByCampus(staffMajor, campusId) → totalClasses.

Step 62: Compute totalPages = max(1, ceil(totalClasses / pageSize)).

Step 63: Add to model:

- classes (list)
- subjects
- currentPageClasses = 1
- totalPagesClasses
- pageSize
- totalClasses
- sessions enum
- currentCampusName

Step 64: Return view "ClassesList".

Step 65: Stop execution.

EXCEPTION HANDLING

Step 66: Catch any Exception (DataIntegrityViolationException, ConstraintViolationException, RuntimeException, etc.).

Step 67: Put serviceErrors.put("general", "System error: " + e.getMessage()).

Step 68: Log full stack trace with ERROR level.

Step 69: Execute exactly Step 50–64 (reload page with error message).

Step 70: Return "ClassesList" view.

Step 71: Stop execution.

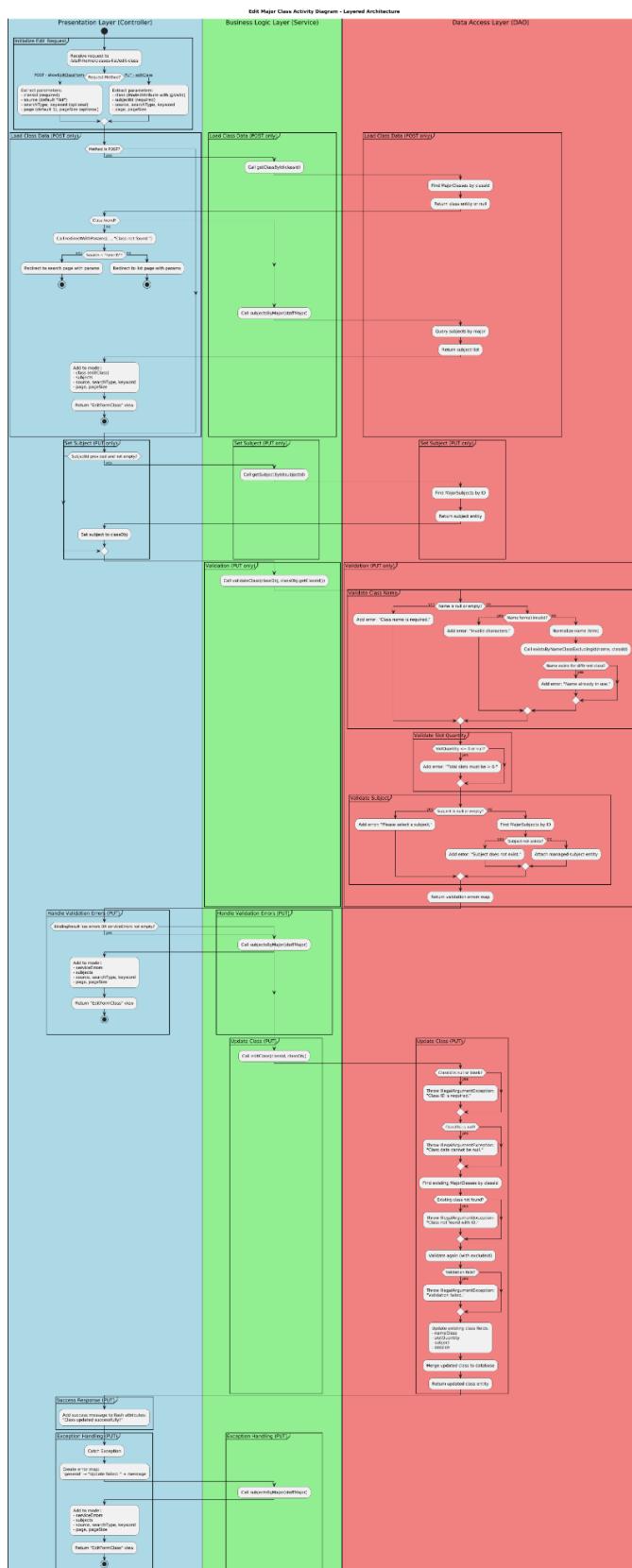


Diagram 52 Major Class Update Workflow

Step 1: Receive request to /staff-home/classes-list/edit-class.

Step 2: Determine HTTP method.

METHOD POST — SHOW EDIT FORM

Step 3: If POST → extract required parameter classId.

Step 4: Extract source (default "list"), searchType, keyword, page (default 1), pageSize.

Step 5: Call majorClassesService.getClassById(classId).

Step 6: If class not found → go to BRANCH B (Class Not Found).

Step 7: Call majorSubjectsService.subjectsByMajor(staffMajor) → get subject list.

Step 8: Add to model:

- class (as editClass)
- subjects list
- source, searchType, keyword, page, pageSize

Step 9: Return view "EditFormClass".

Step 10: Stop execution.

METHOD PUT — SUBMIT EDIT FORM

Step 11: If PUT → bind @ModelAttribute("class") @Valid MajorClasses classObj.

Step 12: Extract required subjectId.

Step 13: Extract source, searchType, keyword, page, pageSize.

Step 14: Enter try-block.

ATTACH SUBJECT ENTITY (PUT)

Step 15: If subjectId is provided and not blank →

Step 16: Call majorSubjectsService.getSubjectById(subjectId).

Step 17: Set classObj.subject = managed subject entity.

VALIDATION PHASE (PUT ONLY)

Step 18: Call classValidationService.validateClass(classObj, classObj.getClassId()) → returns error map.

Step 19: Inside validation →

- Name null/empty → error "Class name is required."
- Name invalid characters → error "Invalid characters."
- Normalize name
- Call repository.existsByNameClassExcludingId(name, classId) → if true → error "Name already in use."

Step 20: If slotQuantity ≤ 0 or null → error "Total slots must be > 0."

Step 21: If classObj.subject == null → error "Please select a subject."

Step 22: Merge all errors into serviceErrors map.

BRANCH A — VALIDATION FAILED (PUT)

Step 23: If BindingResult.hasErrors() OR serviceErrors not empty → go to Step 60.

UPDATE CLASS IN DATABASE

Step 24: Call majorClassesService.editClass(classObj.getId(), classObj).

Step 25: Inside service →

- Validate classId not null/blank → else throw IllegalArgumentException
- Validate classObj not null → else throw
- Find existing entity by classId → if not found → throw "Class not found"
- Re-validate name uniqueness excluding current ID
- Copy updatable fields: nameClass, slotQuantity, subject, session
- Merge entity to database

Step 26: Commit transaction.

SUCCESS RESPONSE

Step 27: Add flash success = "Class updated successfully!".

Step 28: If source == "search" →

Step 29: Add flash searchType, keyword, page, pageSize.

Step 30: Redirect to /staff-home/classes-list/search-classes (or appropriate search endpoint).

Step 31: Stop execution.

Step 32: Else (source == "list") →

Step 33: Add flash page, pageSize.

Step 34: Redirect to /staff-home/classes-list.

Step 35: Stop execution.

BRANCH B — CLASS NOT FOUND (POST METHOD)

Step 40: Add flash error = "Class not found.".

Step 41: If source == "search" →

Step 42: Add flash searchType, keyword, page, pageSize.

Step 43: Redirect to search page.

Step 44: Stop execution.

Step 45: Else →

Step 46: Add flash page, pageSize.

Step 47: Redirect to /staff-home/classes-list.

Step 48: Stop execution.

BRANCH A — VALIDATION FAILED OR EXCEPTION (PUT)

Step 60: Call majorSubjectsService.subjectsByMajor(staffMajor) → reload subjects.

Step 61: Add to model:

- serviceErrors (or general error)
- subjects list
- classObj (with user input preserved)
- source, searchType, keyword, page, pageSize

Step 62: Return view "EditFormClass".

Step 63: Stop execution.

EXCEPTION HANDLING (PUT ONLY)

Step 64: Catch any Exception (IllegalArgumentException, DataAccessException, etc.).

Step 65: Create serviceErrors.put("general", "Update failed: " + e.getMessage()).

Step 66: Log full stack trace with ERROR level.

Step 67: Execute exactly Step 60–62 (reload form with error).

Step 68: Return "EditFormClass" view.

Step 69: Stop execution.

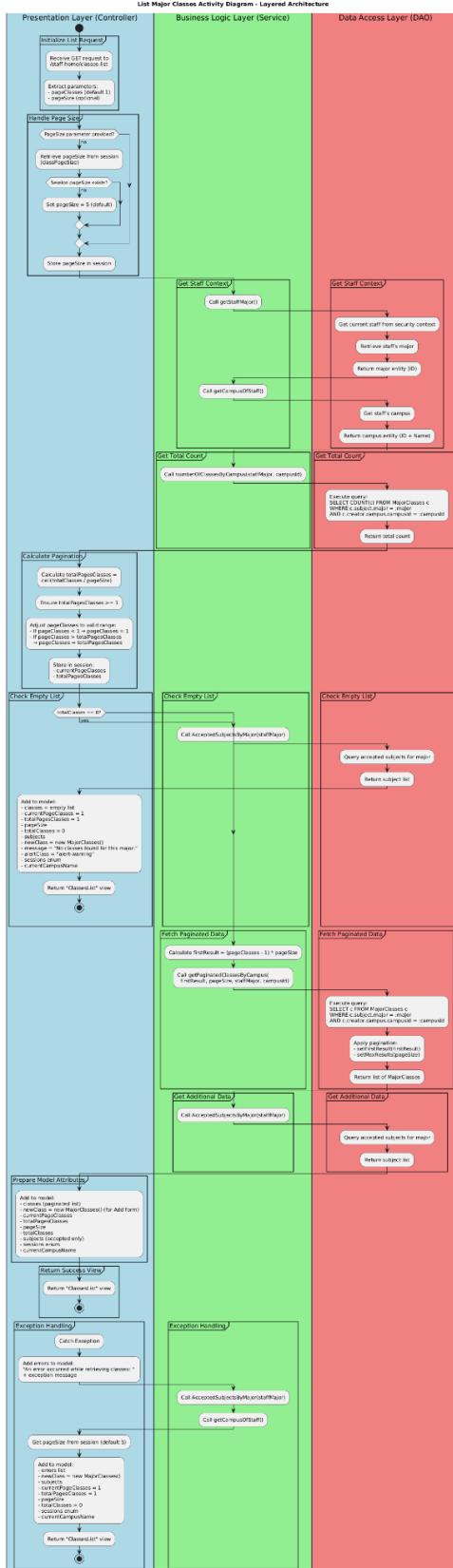


Diagram 53 Major Class Listing Workflow

001479794

Page 379 | 635 total

- Step 1: Receive GET request to /staff-home/classes-list.
- Step 2: Read request parameter pageClasses (default = 1).
- Step 3: Read request parameter pageSize (nullable).
- Step 4: Access HttpSession object.
- Step 5: Enter try-block.

HANDLE PAGE SIZE INPUT

- Step 6: Check if pageSize == null.
- Step 7: If null → retrieve from session.classPageSize.
- Step 8: If session.classPageSize == null → set default pageSize = 5.
- Step 9: Validate: if pageSize < 1 or > 100 → fallback to 5.
- Step 10: Save session.classPageSize = pageSize.

GET CURRENT STAFF CONTEXT

- Step 11: Call staffsService.getStaffMajor() → return Major entity of current staff.
- Step 12: Extract staffMajor (Major entity).
- Step 13: Call staffsService.getCampusOfStaff() → return Campus entity.
- Step 14: Extract campusId and currentCampusName.

COUNT TOTAL CLASSES

- Step 15: Call majorClassesService.numberOfClassesByCampus(staffMajor, campusId).
- Step 16: DAO executes:

```
SELECT COUNT(c) FROM MajorClasses c
WHERE c.subject.major = :major
AND c.creator.campus.campusId = :campusId
```

- Step 17: Store result → totalClasses.

COMPUTE PAGINATION

- Step 18: Compute totalPagesClasses = max(1, ceil(totalClasses / pageSize)).
- Step 19: Clamp pageClasses:
 - if pageClasses < 1 → pageClasses = 1
 - if pageClasses > totalPagesClasses && totalPagesClasses > 0 → pageClasses = totalPagesClasses

- Step 20: Save session.currentPageClasses = pageClasses.

- Step 21: Save session.totalPagesClasses = totalPagesClasses.

BRANCH A — NO CLASSES EXIST

- Step 22: Check if totalClasses == 0.

Step 23: If yes → go to Step 50.

LOAD PAGINATED CLASSES (NORMAL CASE)

Step 24: Compute firstResult = (pageClasses - 1) * pageSize.

Step 25: Call majorClassesService.getPaginatedClassesByCampus(firstResult, pageSize, staffMajor, campusId).

Step 26: DAO executes:

```
SELECT c FROM MajorClasses c
WHERE c.subject.major = :major
AND c.creator.campus.campusId = :campusId
+ setFirstResult + setMaxResults
```

Step 27: Store returned list → classes.

LOAD ACCEPTED SUBJECTS (SHARED FOR BOTH BRANCHES)

Step 28: Call majorSubjectsService.AcceptedSubjectsByMajor(staffMajor).

Step 29: DAO queries only subjects with status = ACCEPTED for this major.

Step 30: Store result → subjects list.

POPULATE MODEL — NORMAL CASE

Step 31: Set model.classes = classes (paginated list).

Step 32: Set model.newClass = new MajorClasses() (for Add overlay).

Step 33: Set model.currentPageClasses = pageClasses.

Step 34: Set model.totalPagesClasses = totalPagesClasses.

Step 35: Set model.pageSize = pageSize.

Step 36: Set model.totalClasses = totalClasses.

Step 37: Set model.subjects = subjects (accepted only).

Step 38: Set model.sessions = SessionType.values().

Step 39: Set model.currentCampusName = currentCampusName.

Step 40: Return view "ClassesList".

Step 41: Stop execution.

BRANCH A — EMPTY LIST HANDLING

Step 50: Call majorSubjectsService.AcceptedSubjectsByMajor(staffMajor) → get subjects list.

Step 51: Set model.classes = empty list.

Step 52: Set model.newClass = new MajorClasses().

Step 53: Set model.currentPageClasses = 1.

Step 54: Set model.totalPagesClasses = 1.

Step 55: Set model.pageSize = pageSize.
Step 56: Set model.totalClasses = 0.
Step 57: Set model.subjects = subjects.
Step 58: Set model.message = "No classes found for this major.".br/>Step 59: Set model.alertClass = "alert-warning".
Step 60: Set model.sessions = SessionType.values().
Step 61: Set model.currentCampusName = currentCampusName.
Step 62: Return view "ClassesList".
Step 63: Stop execution.

EXCEPTION HANDLING

Step 64: Catch any Exception (SecurityException, DataAccessException, etc.).
Step 65: Set model.errors = List.of("An error occurred while retrieving classes: " + e.getMessage()).
Step 66: Log full stack trace with ERROR level.
Step 67: Retrieve pageSize from session (fallback 5).
Step 68: Call majorSubjectsService.AcceptedSubjectsByMajor(staffMajor) → reload subjects.
Step 69: Call staffsService.getCampusOfStaff() → get campus name.
Step 70: Set model.classes = empty list.
Step 71: Set model.newClass = new MajorClasses().
Step 72: Set model.currentPageClasses = 1.
Step 73: Set model.totalPagesClasses = 1.
Step 74: Set model.pageSize = pageSize.
Step 75: Set model.totalClasses = 0.
Step 76: Set model.subjects = subjects list.
Step 77: Set model.sessions = SessionType.values().
Step 78: Set model.currentCampusName = currentCampusName (or empty if null).
Step 79: Return view "ClassesList".
Step 80: Stop execution.

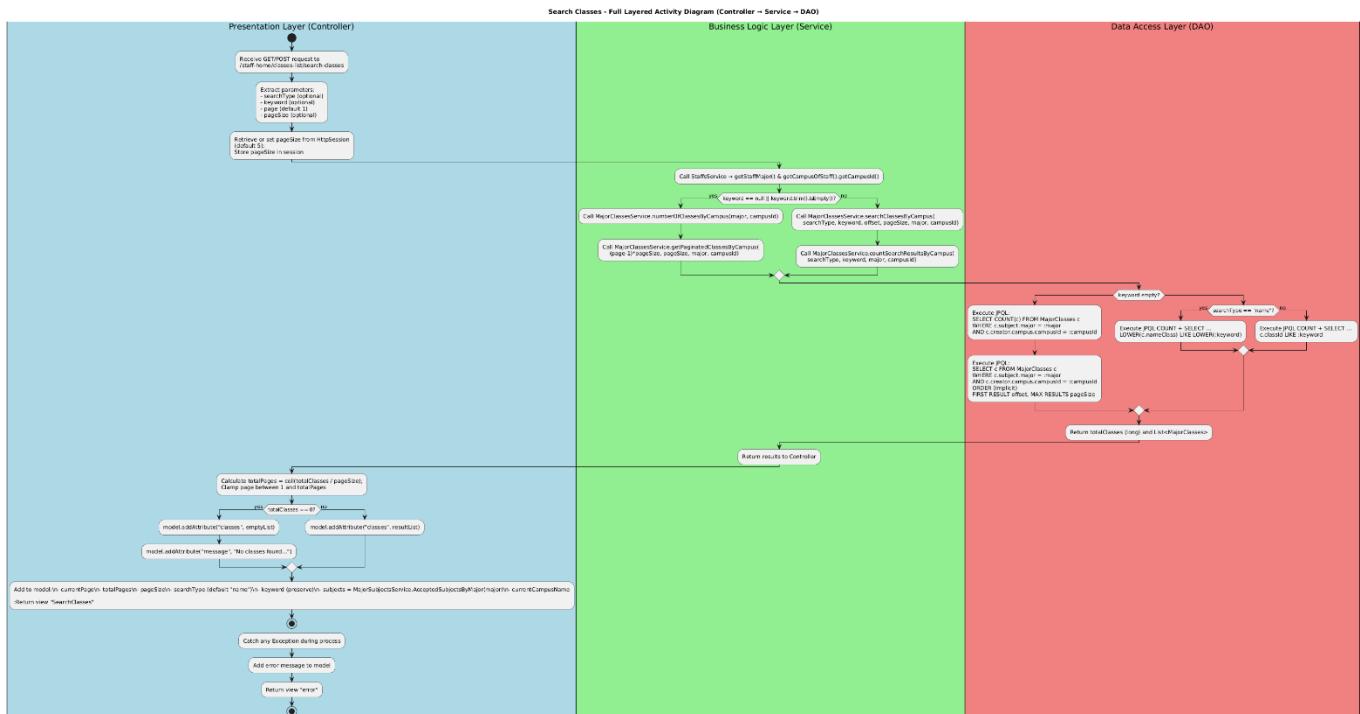


Diagram 54 Major Class Search Workflow

Step 1: Receive GET or POST request to /staff-home/classes-list/search-classes.

Step 2: Extract parameter searchType (nullable).

Step 3: Extract parameter keyword (nullable).

Step 4: Extract parameter page (default = 1).

Step 5: Extract parameter pageSize (nullable).

Step 6: Access HttpSession.

Step 7: Enter try-block.

HANDLE PAGE SIZE

Step 8: Check if pageSize == null.

Step 9: If null → retrieve from session.classSearchPageSize.

Step 10: If session.classSearchPageSize == null → set default pageSize = 5.

Step 11: Validate: if pageSize < 1 or > 100 → fallback to 5.

Step 12: Save session.classSearchPageSize = pageSize.

GET CURRENT STAFF CONTEXT

Step 13: Call `staffsService.getStaffMajor()` → return current staff's Major entity.

Step 14: Call staffsService.getCampusOfStaff() → return Campus entity

Step 15: Extract campusId and currentCampusName

DETERMINE SEARCH OR LIST MODE

Step 16: Check if (keyword == null || keyword.trim().isEmpty()).

Step 17: If YES → go to BRANCH A (List all classes).

Step 18: If NO → go to BRANCH B (Search mode).

BRANCH A — LIST ALL CLASSES (NO KEYWORD)

Step 19: Compute offset = (page - 1) * pageSize.

Step 20: Call majorClassesService.numberOfClassesByCampus(staffMajor, campusId).

Step 21: DAO executes:

```
SELECT COUNT(c) FROM MajorClasses c
WHERE c.subject.major = :major
AND c.creator.campus.campusId = :campusId
```

Step 22: Store result → totalClasses.

Step 23: Call majorClassesService.getPaginatedClassesByCampus(offset, pageSize, staffMajor, campusId).

Step 24: DAO executes:

```
SELECT c FROM MajorClasses c
WHERE c.subject.major = :major
AND c.creator.campus.campusId = :campusId
ORDER BY c.classId DESC
+ setFirstResult(offset) + setMaxResults(pageSize)
```

Step 25: Store returned list → classes.

Step 26: Go to Step 40.

BRANCH B — SEARCH MODE

Step 27: Normalize keyword = keyword.trim().toLowerCase().

Step 28: Compute offset = (page - 1) * pageSize.

Step 29: Call majorClassesService.countSearchResultsByCampus(searchType, keyword, staffMajor, campusId).

Step 30: Inside DAO →

- If searchType == "name" → WHERE LOWER(c.nameClass) LIKE LOWER('%' + keyword + '%')
- If searchType == "id" → WHERE c.classId LIKE '%' + keyword + '%'
- Always AND c.subject.major = :major AND c.creator.campus.campusId = :campusId

Step 31: Execute COUNT query → return totalClasses.

Step 32: Call majorClassesService.searchClassesByCampus(searchType, keyword, offset, pageSize, staffMajor, campusId).

Step 33: Execute same WHERE condition + ORDER BY + pagination → return list → classes.

Step 34: Continue.

COMPUTE PAGINATION (COMMON)

Step 40: Compute totalPages = totalClasses == 0 ? 1 : (int) Math.ceil((double) totalClasses / pageSize).

Step 41: Clamp page:

- if (page < 1) → page = 1
- if (page > totalPages && totalPages > 0) → page = totalPages

Step 42: Save session.currentSearchPageClasses = page.

Step 43: Save session.totalSearchPagesClasses = totalPages.

LOAD ACCEPTED SUBJECTS

Step 44: Call majorSubjectsService.AcceptedSubjectsByMajor(staffMajor).

Step 45: DAO returns only subjects with status = ACCEPTED for this major.

POPULATE MODEL

Step 46: If totalClasses == 0 →

- Step 47: Set model.classes = Collections.emptyList().
- Step 48: Set model.message = "No classes found matching your search criteria."
- Step 49: Set model.alertClass = "alert-info".

Step 50: Else →

- Step 51: Set model.classes = classes (search result).

Step 52: Set model.currentPage = page.

Step 53: Set model.totalPages = totalPages.

Step 54: Set model.pageSize = pageSize.

Step 55: Set model.totalClasses = totalClasses.

Step 56: Set model.searchType = searchType != null ? searchType : "name".

Step 57: Set model.keyword = keyword != null ? keyword.trim() : "".

Step 58: Set model.subjects = accepted subjects list.

Step 59: Set model.currentCampusName = currentCampusName.

Step 60: Return view "SearchClasses".

Step 61: Stop execution.

EXCEPTION HANDLING

- Step 62: Catch any Exception (SecurityException, DataAccessException, etc.).
- Step 63: Log full stack trace with ERROR level.
- Step 64: Set model.error = "An error occurred while searching classes: " + e.getMessage().
- Step 65: Set model.classes = Collections.emptyList().
- Step 66: Set model.currentPage = 1.
- Step 67: Set model.totalPages = 1.
- Step 68: Set model.pageSize = pageSize (from session or 5).
- Step 69: Set model.totalClasses = 0.
- Step 70: Set model.searchType = searchType != null ? searchType : "name".
- Step 71: Set model.keyword = keyword != null ? keyword.trim() : "".
- Step 72: Try to reload subjects via majorSubjectsService.AcceptedSubjectsByMajor() (may fail silently).
- Step 73: Set model.subjects = loaded list or emptyList().
- Step 74: Set model.currentCampusName = currentCampusName or "Unknown Campus".
- Step 75: Return view "error" (or fallback to "SearchClasses" with error block).
- Step 76: Stop execution.

Activity diagram related to minor class roles

AddMinorClassController

001479794

Page 386 | 635 total

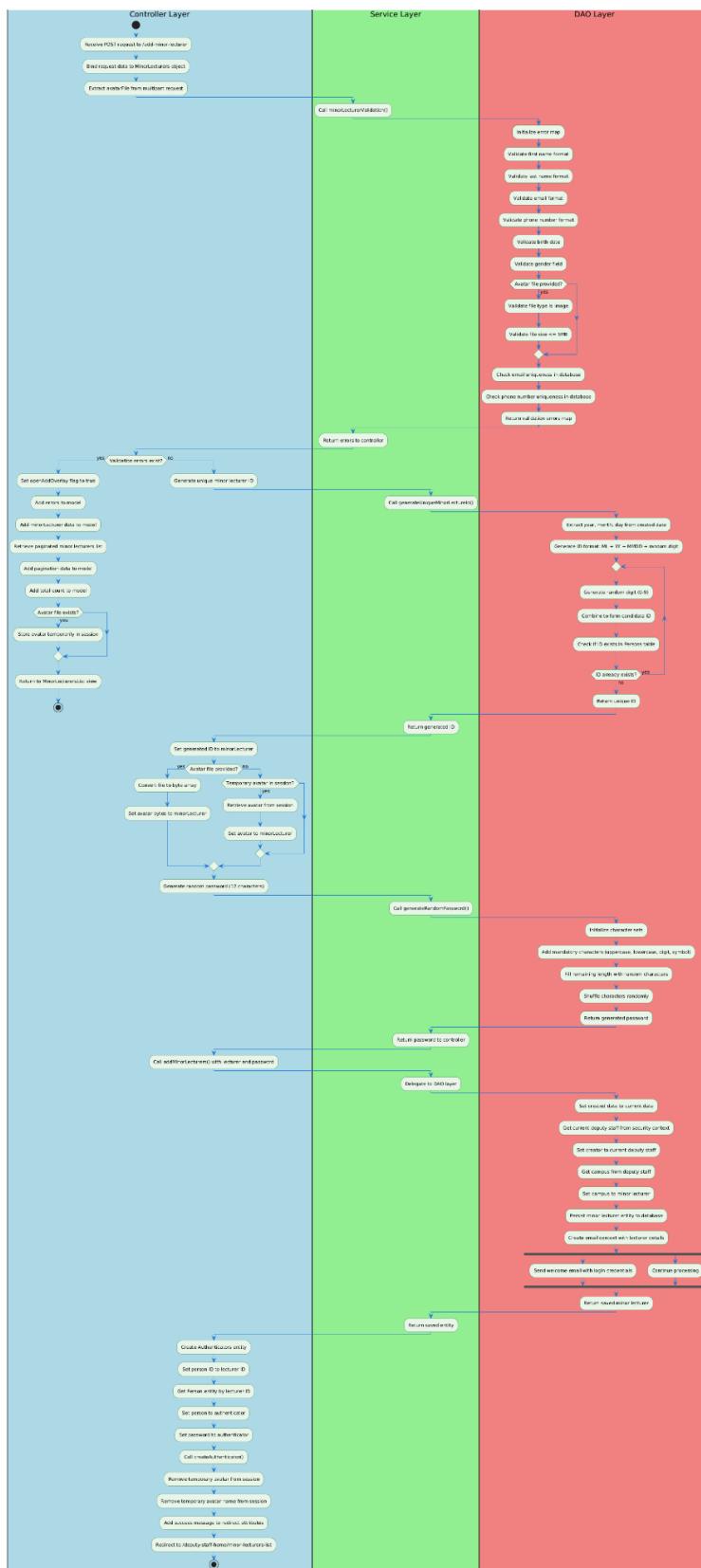


Diagram 55 Add Minor Lecturer Workflow

CONTROLLER LAYER — RECEIVE REQUEST

- Step 1: Receive POST request to /add-minor-lecturer.
 - Step 2: Bind request data into MinorLecturers object.
 - Step 3: Extract avatarFile from multipart request.
-

SERVICE LAYER — VALIDATION CALL

- Step 4: Call minorLecturerValidation(minorLecturer, avatarFile).
-

DAO LAYER — VALIDATION PROCESS

- Step 5: Initialize an empty error map.
 - Step 6: Validate first name format (non-empty, alphabetic).
 - Step 7: Validate last name format.
 - Step 8: Validate email format (regex).
 - Step 9: Validate phone number format.
 - Step 10: Validate birth date (valid date & legal age).
 - Step 11: Validate gender field.
 - Step 12: Check if avatar file is provided.
 - Step 13: If avatar exists → validate file type is an image (JPEG/PNG).
 - Step 14: Validate avatar file size ≤ 5MB.
 - Step 15: Check email uniqueness in DB.
 - Step 16: Check phone uniqueness in DB.
 - Step 17: Return validation error map to service.
-

SERVICE LAYER — RETURN VALIDATION RESULTS

- Step 18: Service returns error map to controller.
-

CONTROLLER LAYER — VALIDATION BRANCH DECISION

- Step 19: Check if validation errors exist.
-

BRANCH A — VALIDATION FAILED

- Step 20: Set openAddOverlay = true.
- Step 21: Add validation errors to model.
- Step 22: Add returned minorLecturer object to model.
- Step 23: Retrieve paginated minor lecturers list.
- Step 24: Add pagination metadata to model.
- Step 25: Add total minor lecturers count to model.

Step 26: Check if avatar file exists.

Step 27: If yes → temporarily store avatar in session.

Step 28: Return view "MinorLecturersList" to show overlay with errors.

Step 29: Stop execution.

BRANCH B — VALIDATION PASSED

Step 30: Generate unique minor lecturer ID.

Step 31: Call generateUniqueMinorLectureId() service function.

DAO LAYER — GENERATE UNIQUE ID

Step 32: Extract year, month, day from created date.

Step 33: Generate base format: ML + YY + MMDD + randomDigit.

Step 34: Enter loop:

- Generate random digit (0–9).
- Combine to form candidate ID.
- Check if ID exists in Persons table.

Step 35: Repeat until an unused ID is found.

Step 36: Return unique ID to service.

SERVICE LAYER — RETURN GENERATED ID

Step 37: Return unique ID to controller.

CONTROLLER LAYER — ASSIGN ID / AVATAR HANDLING

Step 38: Set generated ID to minorLecturer.

Step 39: Check if avatar file is provided.

Step 40: If yes → convert file to byte[] and assign to minorLecturer.

Step 41: Else → check if temporary avatar exists in session.

Step 42: If exists → retrieve from session and assign.

CONTROLLER LAYER — GENERATE PASSWORD

Step 43: Generate random 12-character password.

Step 44: Call generateRandomPassword().

DAO LAYER — PASSWORD GENERATION

Step 45: Initialize character sets (upper, lower, digits, symbols).

Step 46: Add at least one mandatory character of each type.

- Step 47: Fill remaining characters with random selection.
 - Step 48: Shuffle to randomize order.
 - Step 49: Return generated password to service.
-

SERVICE LAYER — PASSWORD RETURN

- Step 50: Return generated password to controller.
-

CONTROLLER LAYER — SAVE MINOR LECTURER

- Step 51: Call addMinorLecturers(minorLecturer, password) service method.
-

SERVICE → DAO — PERSIST ENTITY

- Step 52: Set created date to current date.
- Step 53: Retrieve current deputy staff from security context.
- Step 54: Set creator field to deputy staff.
- Step 55: Retrieve campus from deputy staff.
- Step 56: Assign campus to minorLecturer.
- Step 57: Persist MinorLecturers entity to database.

Step 58: Create email context with lecturer information.

FORK — PARALLEL OPERATIONS

- Step 59: Background task → Send welcome email (credentials included).
 - Step 60: Continue processing without waiting.
-

DAO → SERVICE — RETURN ENTITY

- Step 61: Return saved MinorLecturers entity.
 - Step 62: Service returns saved entity to controller.
-

CONTROLLER LAYER — AUTHENTICATOR CREATION

- Step 63: Create new Authenticators object.
 - Step 64: Set personID = lecturerID.
 - Step 65: Retrieve Person entity by lecturerID.
 - Step 66: Set person reference to authenticator.
 - Step 67: Set generated password to authenticator.
 - Step 68: Call createAuthenticator(authenticator).
-

CLEANUP + REDIRECT

Step 69: Remove temporary avatar from session.

Step 70: Remove temporary avatar name from session.

Step 71: Add success message to redirect attributes.

Step 72: Redirect to /deputy-staff-home/minor-lecturers-list.

Step 73: Stop execution.

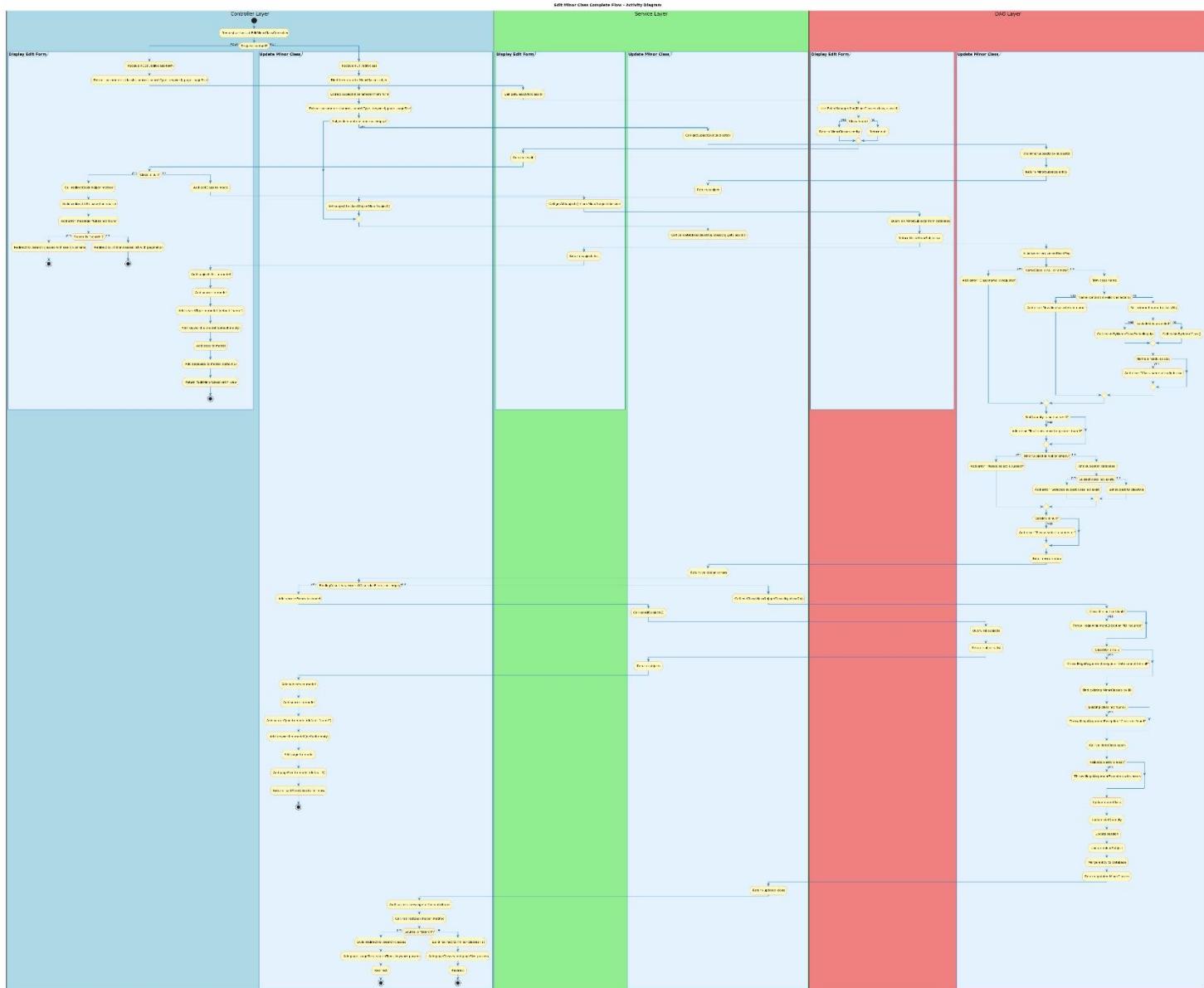


Diagram 56 Add Minor Class Complete Workflow

Step 1: Request arrives at EditMinorClassController.

Step 2: Determine whether the request method is POST or PUT.

Step 3: If the method is POST, receive POST /edit-class-form.

Step 4: Extract parameters classId, source, searchType, keyword, page, pageSize.

Step 5: Call service method getClassById(classId).

Step 6: DAO loads the class using EntityManager.find(MinorClasses.class, classId).

Step 7: If the entity exists, return the MinorClasses object; otherwise return null.

Step 8: Service returns the lookup result to the controller.

Step 9: Controller checks if the class is null.

Step 10: If class is null, call redirectBack helper method.

Step 11: Build redirect URL depending on source.

Step 12: Add error message "Class not found".

Step 13: If source == "search", redirect to /search-classes with search parameters and stop execution.

Step 14: Otherwise, redirect to /minor-classes-list with pagination parameters and stop execution.

Step 15: If class exists, add editClass object to the model.

Step 16: Call service getAllSubjects().

Step 17: DAO queries all MinorSubjects from the database.

Step 18: DAO returns the list of subjects.

Step 19: Service returns subject list.

Step 20: Controller adds the subjects list to the model.

Step 21: Add source to the model.

Step 22: Add searchType to the model (default "name" if null).

Step 23: Add keyword to the model (default empty string if null).

Step 24: Add page to the model.

Step 25: Add pageSize to the model (default 5).

Step 26: Return "EditMinorClassForm" view and stop execution.

Step 27: If request is PUT, receive PUT /edit-class.

Step 28: Bind form fields to a MinorClasses object (classObj).

Step 29: Extract subjectId from the form.

Step 30: Extract source, searchType, keyword, page, pageSize.

Step 31: If subjectId is not null and not blank, call getSubjectById(subjectId).

Step 32: DAO loads subject entity using EntityManager.find(MinorSubjects.class, subjectId).

Step 33: Service returns the subject.

Step 34: Controller sets this subject into the class object.

Step 35: Call service method validateClass(classObj, classObj.getClassId()).

Step 36: DAO initializes a LinkedHashMap for errors.

Step 37: Check if NameClass is null or empty; if yes, add "Class name is required" to errors.

Step 38: If not empty, trim the name.

Step 39: Validate name for invalid characters; if invalid, add "Invalid characters in name".

Step 40: If name is valid, assign trimmed name to classObj.

Step 41: If excluding ID is required, call existsByNameClassExcludingId().

Step 42: Otherwise call existsByNameClass().

Step 43: If name already exists, add "Class name already in use" to errors.

Step 44: If slotQuantity is null or ≤ 0 , add "Total slots must be greater than 0".

Step 45: Check if minorSubject is null; if yes, add "Please select a subject".

Step 46: If subject is provided, DAO checks existence.

Step 47: If subject does not exist, add "Selected subject does not exist".

Step 48: If exists, assign it into classObj.

Step 49: If session field is null, add "Please select a semester".

Step 50: DAO returns the map of validation errors.

Step 51: Service returns validation errors to the controller.

Step 52: Controller checks if BindingResult has errors OR validation errors are not empty.

Step 53: If errors exist, add all service errors to the model.

Step 54: Call service getAllSubjects() again.

Step 55: DAO queries all subjects and returns list.

Step 56: Service returns subject list.

Step 57: Controller adds subjects list to the model.

Step 58: Add source to the model.

Step 59: Add searchType (default "name").

Step 60: Add keyword (default empty string).

Step 61: Add page and pageSize (default 5).

Step 62: Return "EditMinorClassForm" view and stop execution.

Step 63: If no errors exist, call service method editClass(classObj.getClassId(), classObj).

Step 64: DAO checks if classId is null or blank; if yes, throw IllegalArgumentException("ID required").

Step 65: DAO checks if classObj is null; if yes, throw "Data cannot be null".

Step 66: DAO attempts to find existing class by ID.

Step 67: If not found, throw "Class not found".

Step 68: DAO calls validateClass again internally.

Step 69: If validation errors remain, throw IllegalArgumentException with those errors.

Step 70: DAO updates the following fields:

- nameClass
- slotQuantity
- session
- minorSubject

Step 71: DAO merges updated entity into the database.

Step 72: DAO returns the updated MinorClasses entity.

Step 73: Service returns updated entity to controller.

Step 74: Controller adds success message to flash attributes.

Step 75: Call redirectBack helper method.

Step 76: Check if source == "search".

Step 77: If source is search, build redirect URL to /search-classes including:

- page
- pageSize
- searchType
- keyword

Step 78: Redirect to /search-classes and stop execution.

Step 79: If not search source, build redirect URL to /minor-classes-list including:

- pageClasses
- pageSize

Step 80: Redirect to /minor-classes-list.

Step 81: Stop execution.

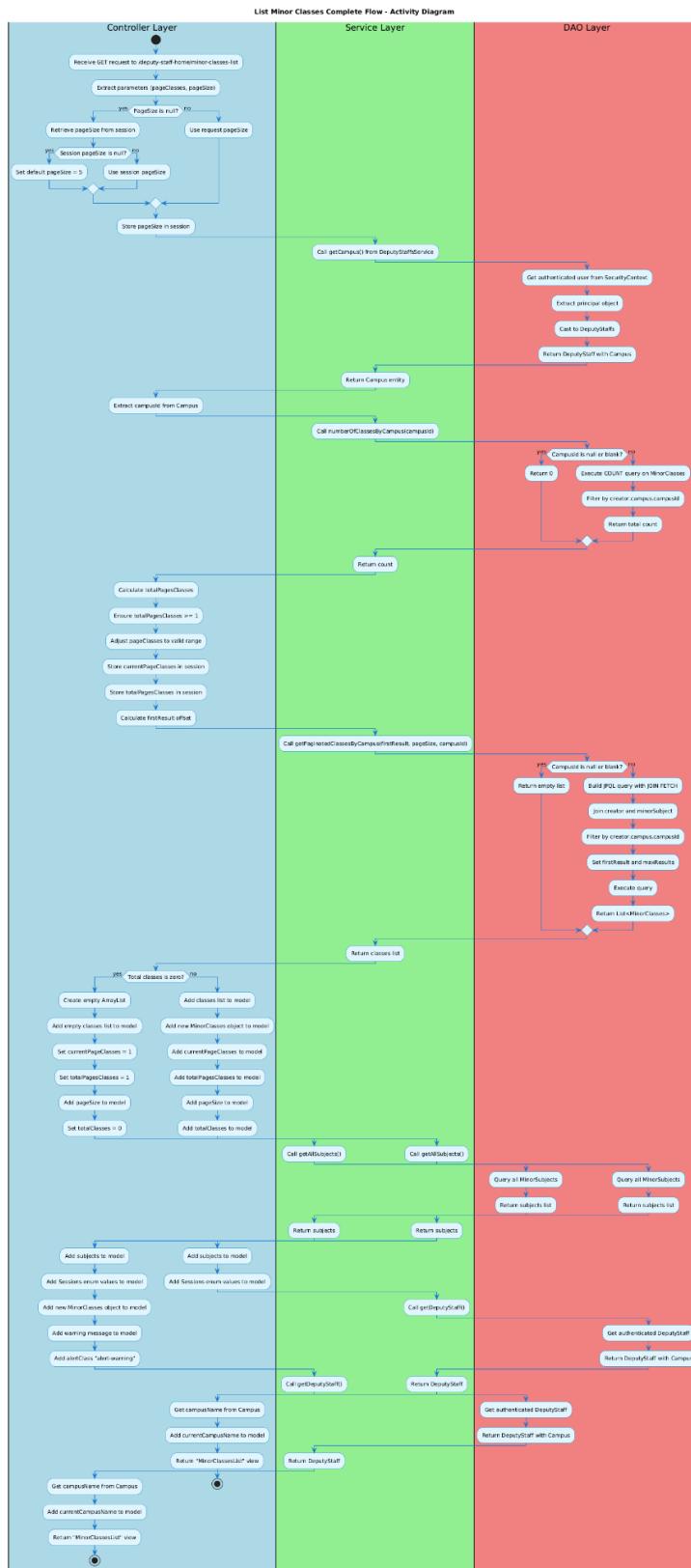


Diagram 57 List Minor Classes Workflow

Step 1: Receive GET request to /deputy-staff-home/minor-classes-list.

Step 2: Extract parameters pageClasses and pageSize.

Step 3: Check if pageSize is null.

Step 4: If pageSize is null, retrieve pageSize from session.

Step 5: Check if the session pageSize is null.

Step 6: If session pageSize is null, set default pageSize = 5.

Step 7: Otherwise use the session pageSize.

Step 8: If request provides valid pageSize, use the request pageSize instead.

Step 9: Store pageSize in session.

Step 10: Call DeputyStaffsService.getCampus().

Step 11: DAO retrieves the authenticated user from the SecurityContext.

Step 12: Extract principal object.

Step 13: Cast principal to DeputyStaffs.

Step 14: DAO returns DeputyStaff together with its Campus entity.

Step 15: Service returns the Campus.

Step 16: Controller extracts campusId from Campus.

Step 17: Call numberOfClassesByCampus(campusId).

Step 18: DAO checks if campusId is null or blank.

Step 19: If campusId is invalid, return count = 0.

Step 20: Otherwise, execute COUNT query on MinorClasses table.

Step 21: Filter by creator.campus.campusId.

Step 22: DAO returns the total count.

Step 23: Service returns the class count to the controller.

Step 24: Calculate totalPagesClasses = ceil(totalClasses / pageSize).

Step 25: Ensure totalPagesClasses >= 1.

Step 26: Adjust pageClasses so it stays within valid bounds.

Step 27: Store currentPageClasses in session.

Step 28: Store totalPagesClasses in session.

Step 29: Calculate firstResult = (pageClasses - 1) * pageSize.

Step 30: Call getPaginatedClassesByCampus(firstResult, pageSize, campusId).

Step 31: DAO checks if campusId is null or blank.

Step 32: If invalid, return an empty list.

Step 33: Otherwise, build JPQL query with JOIN FETCH.

Step 34: Join with creator and minorSubject.

Step 35: Filter by creator.campus.campusId.

Step 36: Apply firstResult offset.

Step 37: Apply maxResults limit.

Step 38: Execute query.

Step 39: DAO returns List<MinorClasses>.

Step 40: Service returns the list to the controller.

Step 41: Check if totalClasses is zero.

BRANCH A — NO CLASSES AVAILABLE

- Step 42:** Create an empty ArrayList.
- Step 43:** Add empty list to model.
- Step 44:** Set currentPageClasses = 1.
- Step 45:** Set totalPagesClasses = 1.
- Step 46:** Add pageSize to model.
- Step 47:** Set totalClasses = 0.
- Step 48:** Call getAllSubjects().
- Step 49:** DAO queries all MinorSubjects.
- Step 50:** DAO returns subjects list.
- Step 51:** Service returns the subjects list.
- Step 52:** Add subjects list to model.
- Step 53:** Add Sessions enum values to model.
- Step 54:** Add a new MinorClasses object to model.
- Step 55:** Add a warning message to model.
- Step 56:** Add alertClass = "alert-warning" to model.
- Step 57:** Call getDeputyStaff().
- Step 58:** DAO fetches authenticated DeputyStaff.
- Step 59:** DAO returns DeputyStaff with Campus.
- Step 60:** Service returns DeputyStaff.
- Step 61:** Extract campusName from Campus.
- Step 62:** Add currentCampusName to model.
- Step 63:** Return "MinorClassesList" view.
- Step 64:** Stop execution.

BRANCH B — CLASSES AVAILABLE

- Step 65:** Add the classes list to model.
- Step 66:** Add a new MinorClasses object to model.
- Step 67:** Add currentPageClasses to model.
- Step 68:** Add totalPagesClasses to model.
- Step 69:** Add pageSize to model.
- Step 70:** Add totalClasses to model.
- Step 71:** Call getAllSubjects().
- Step 72:** DAO queries all MinorSubjects.
- Step 73:** DAO returns subjects list.
- Step 74:** Service returns subjects.
- Step 75:** Add subjects list to model.
- Step 76:** Add Sessions enum values to model.
- Step 77:** Call getDeputyStaff().

Step 78: DAO retrieves authenticated DeputyStaff.

Step 79: DAO returns DeputyStaff with Campus.

Step 80: Service returns DeputyStaff.

Step 81: Extract campusName from Campus.

Step 82: Add currentCampusName to model.

Step 83: Return "MinorClassesList" view.

Step 84: Stop execution.

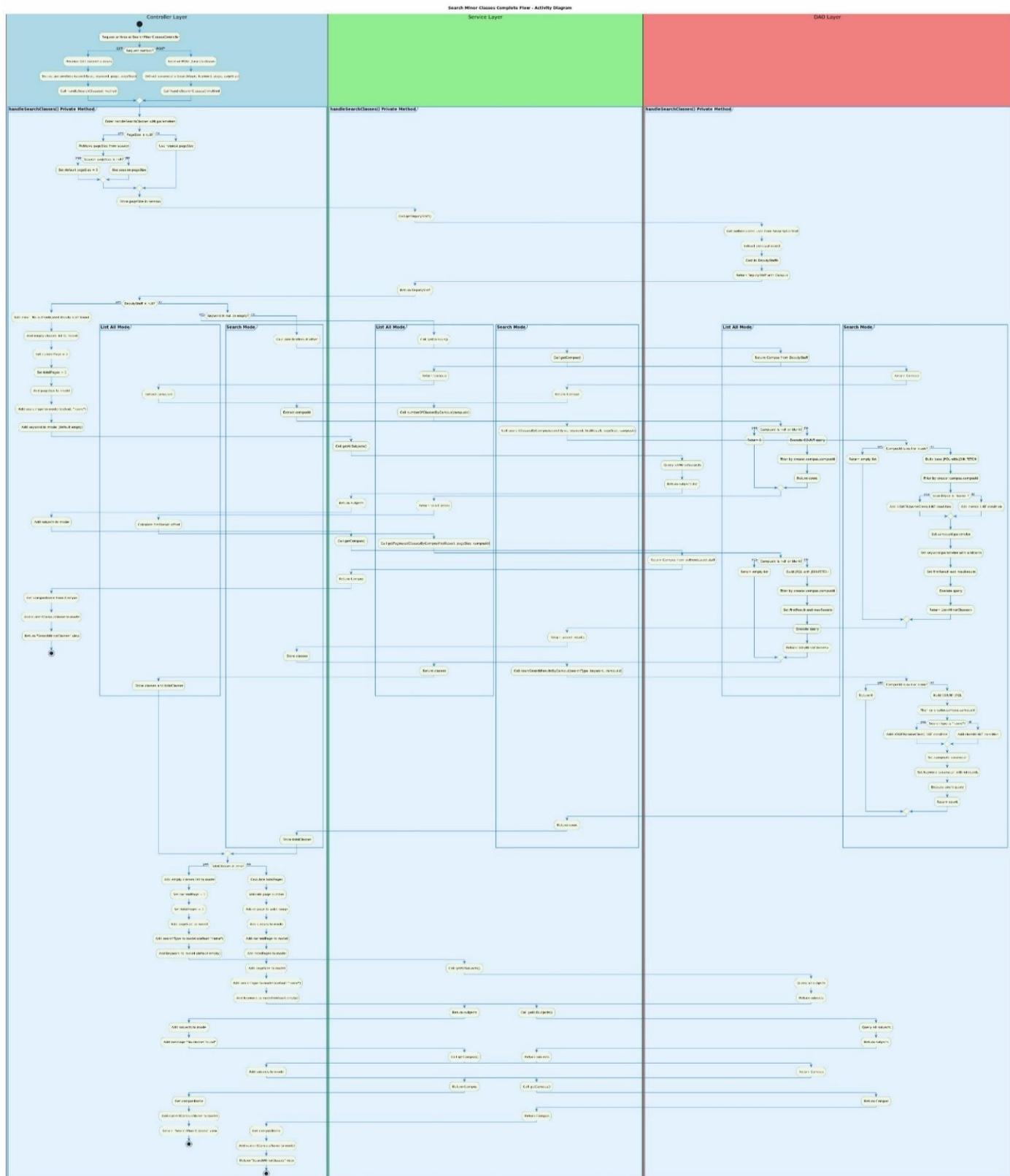


Diagram 58 Load Minor Classes Complex View – Full Layered Activity Diagram

Step 1: Request arrives at SearchMinorClassesController.

Step 2: Determine whether request method is GET or POST.

Step 3: If GET request is received at /search-classes, extract parameters searchType, keyword, page, pageSize.

Step 4: Call private method handleSearchClasses() with these parameters.

Step 5: If POST request is received at /search-classes, extract parameters searchType, keyword, page, pageSize.

Step 6: Call private method handleSearchClasses() with these parameters.

Step 7: Enter handleSearchClasses(searchType, keyword, page, pageSize).

Step 8: Check whether pageSize is null.

Step 9: If pageSize is null, retrieve pageSize from session.

Step 10: Check whether session pageSize is null.

Step 11: If session pageSize is null, set default pageSize = 5.

Step 12: Otherwise, use the session pageSize.

Step 13: If request provided a pageSize, use the request value.

Step 14: Store final pageSize in session.

Step 15: Call service getDeputyStaff().

Step 16: DAO retrieves authenticated user from SecurityContext.

Step 17: Extract principal object.

Step 18: Cast principal to DeputyStaffs.

Step 19: Return DeputyStaff with its Campus.

Step 20: Service returns DeputyStaff to the controller.

Step 21: Controller checks if DeputyStaff is null.

Step 22: If DeputyStaff is null, add error "No authenticated deputy staff found" to model.

Step 23: Add an empty classes list to model.

Step 24: Set currentPage = 1.

Step 25: Set totalPages = 1.

Step 26: Add pageSize to model.

Step 27: Add searchType to model (default "name").

Step 28: Add keyword to model (default empty string).

Step 29: Call getAllSubjects().

Step 30: DAO queries all MinorSubjects.

Step 31: DAO returns subject list.

Step 32: Service returns subjects list.

Step 33: Controller adds subjects to model.

Step 34: Call service getCampus().

Step 35: DAO retrieves Campus from authenticated DeputyStaff.

Step 36: Service returns Campus.

- Step 37:** Controller extracts campusName.
Step 38: Add currentCampusName to the model.
Step 39: Return "SearchMinorClasses" view.
Step 40: Stop execution.
-

Step 41: If DeputyStaff exists, check whether keyword is null or empty.

BRANCH A — LIST ALL MODE (keyword empty)

- Step 42:** Call service getCampus() again.
Step 43: DAO returns Campus from DeputyStaff.
Step 44: Service returns Campus.
Step 45: Controller extracts campusId.
Step 46: Call service numberOfClassesByCampus(campusId).
Step 47: DAO checks if campusId is null or blank.
Step 48: If invalid, return count = 0.
Step 49: Otherwise, execute COUNT query filtered by creator.campus.campusId.
Step 50: DAO returns count.
Step 51: Service returns totalClasses.
Step 52: Controller calculates firstResult = (page - 1) * pageSize.
Step 53: Call service getPaginatedClassesByCampus(firstResult, pageSize, campusId).
Step 54: DAO checks if campusId is null or blank.
Step 55: If invalid, return empty list.
Step 56: Otherwise, build JPQL with JOIN FETCH; filter by campus; set offset and limit; execute query.
Step 57: DAO returns list of MinorClasses.
Step 58: Service returns list.
Step 59: Controller stores classes list and totalClasses.
-

BRANCH B — SEARCH MODE (keyword provided)

- Step 60:** Controller calculates firstResult = (page - 1) * pageSize.
Step 61: Call service getCampus().
Step 62: DAO returns Campus.
Step 63: Service returns Campus.
Step 64: Controller extracts campusId.
Step 65: Call service searchClassesByCampus(searchType, keyword, firstResult, pageSize, campusId).
Step 66: DAO checks if campusId is null or blank.
Step 67: If invalid, return empty list.
Step 68: Otherwise, build base JPQL with JOIN FETCH; filter by campus.

- Step 69:** If searchType is "name", add LOWER(nameClass) LIKE condition.
Step 70: If searchType is "id", add classId LIKE condition.
Step 71: Set campusId parameter.
Step 72: Set keyword parameter with wildcards.
Step 73: Set offset and limit.
Step 74: Execute query.
Step 75: DAO returns matching classes.

Step 76: Service returns search results.
Step 77: Controller stores classes list.

Step 78: Call service countSearchResultsByCampus(searchType, keyword, campusId).

Step 79: DAO checks campusId; if invalid, return count = 0.
Step 80: Otherwise, build COUNT JPQL; filter by campus.
Step 81: If searchType "name", add LIKE condition.
Step 82: If searchType "id", add classId condition.
Step 83: Set campusId.
Step 84: Set keyword with wildcards.
Step 85: Execute COUNT query.
Step 86: DAO returns count.

Step 87: Service returns totalClasses.
Step 88: Controller stores totalClasses.
-

Step 89: Check whether totalClasses is zero.

BRANCH C — NO RESULTS FOUND

- Step 90:** Add empty classes list to model.
Step 91: Set currentPage = 1.
Step 92: Set totalPages = 1.
Step 93: Add pageSize to model.
Step 94: Add searchType to model (default "name").
Step 95: Add keyword to model (default empty).

Step 96: Call service getAllSubjects().
Step 97: DAO queries subjects and returns list.
Step 98: Service returns subjects.
Step 99: Add subjects to model.
Step 100: Add message "No classes found".

Step 101: Call service getCampus().
Step 102: DAO returns Campus.
Step 103: Service returns Campus.

Step 104: Extract campusName.
Step 105: Add currentCampusName to model.

Step 106: Return "SearchMinorClasses" view.

Step 107: Stop execution.

BRANCH D — RESULTS FOUND

Step 108: Calculate totalPages = ceil(totalClasses / pageSize).

Step 109: Validate page number.

Step 110: If page < 1, set page = 1.

Step 111: If page > totalPages, set page = totalPages.

Step 112: Add classes list to model.

Step 113: Add currentPage to model.

Step 114: Add totalPages to model.

Step 115: Add pageSize to model.

Step 116: Add searchType to model (default "name").

Step 117: Add keyword to model (default empty).

Step 118: Call service getAllSubjects().

Step 119: DAO retrieves subjects list.

Step 120: Service returns subjects.

Step 121: Controller adds subjects to model.

Step 122: Call service getCampus().

Step 123: DAO returns Campus.

Step 124: Service returns Campus.

Step 125: Extract campusName.

Step 126: Add currentCampusName to model.

Step 127: Return "SearchMinorClasses" view.

Step 128: Stop execution.

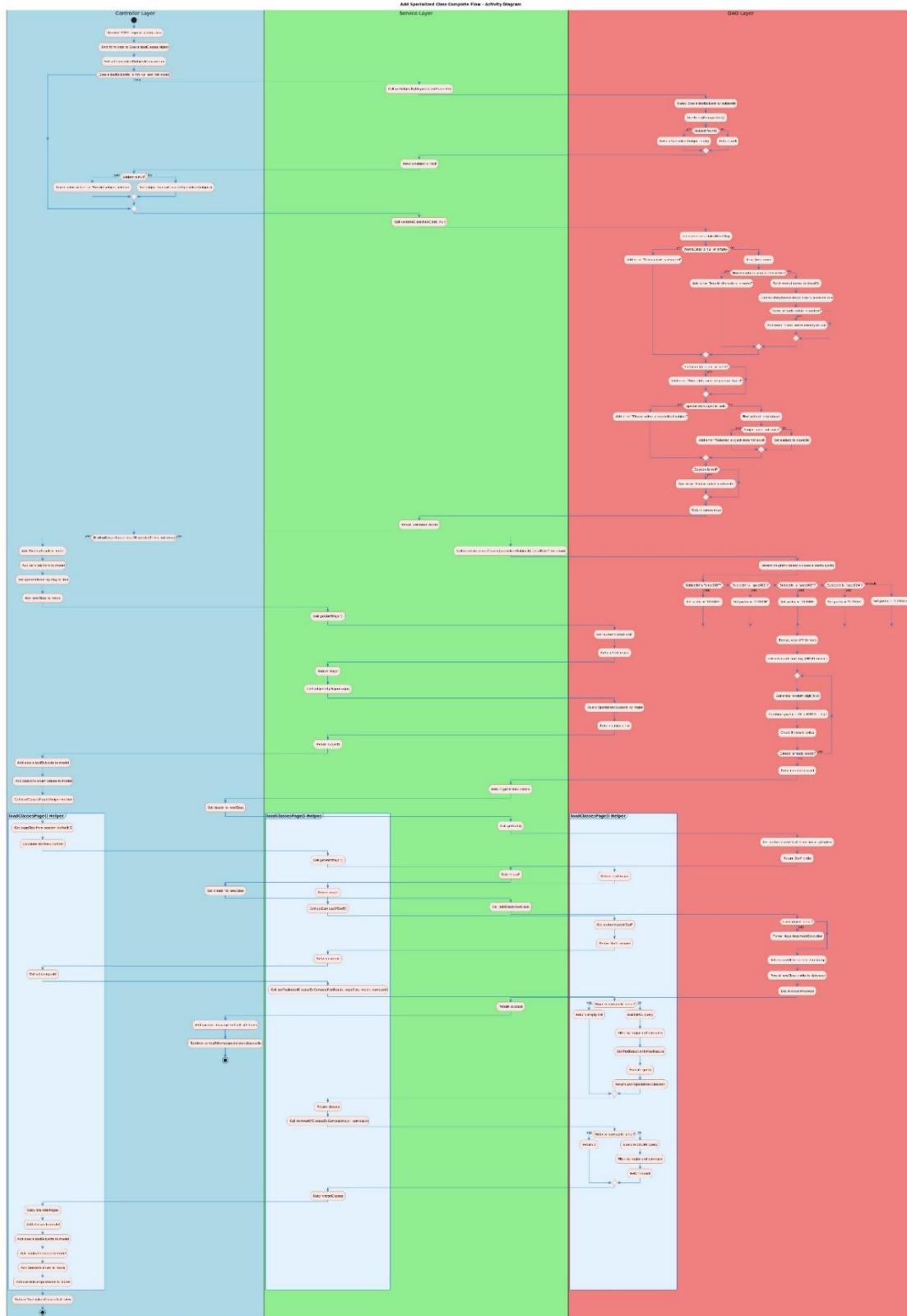


Diagram 59 Add Minor Class – Full Layered Activity Diagram

- Step 1:** Receive POST request to /add-class.
- Step 2:** Bind form data into a SpecializedClasses object (newClass).
- Step 3:** Extract parameter specializedSubjectId.
- Step 4:** Check if specializedSubjectId is not null and not blank.
- Step 5:** If true, call service getSubjectById(specializedSubjectId).
- Step 6:** DAO queries database using EntityManager.find(SpecializedSubject.class, subjectId).
- Step 7:** If subject exists, return the SpecializedSubject entity.
- Step 8:** If subject does not exist, return null.
- Step 9:** Service returns subject or null to the controller.
- Step 10:** Controller checks whether subject is null.
- Step 11:** If subject is null, reject the value with error message "Invalid subject selected".
- Step 12:** If subject is not null, set the subject into newClass.setSpecializedSubject().
- Step 13:** Call service validateClass(newClass, null).
- Step 14:** DAO initializes a LinkedHashMap for errors.
- Step 15:** Check if class name (nameClass) is null or empty.
- Step 16:** If yes, add error "Class name is required".
- Step 17:** If not empty, trim the class name.
- Step 18:** Check if the trimmed name contains invalid characters.
- Step 19:** If invalid characters are found, add error "Invalid characters in name".
- Step 20:** If name is valid, assign trimmed name to classObj.
- Step 21:** Call existsByNameClass() from ClassesService.
- Step 22:** If name already exists, add error "Class name already in use".
- Step 23:** Check if slotQuantity is null or ≤ 0 .
- Step 24:** If yes, add error "Total slots must be greater than 0".
- Step 25:** Check if specializedSubject is null.
- Step 26:** If yes, add error "Please select a specialized subject".
- Step 27:** If not null, attempt to find the subject in the database.
- Step 28:** If subject does not exist, add error "Selected subject does not exist".
- Step 29:** If subject exists, set the subject into classObj.
- Step 30:** Check if session field is null.
- Step 31:** If yes, add error "Please select a semester".
- Step 32:** DAO returns the validation errors map.
- Step 33:** Service returns validation errors to the controller.
- Step 34:** Controller checks if BindingResult has errors or serviceErrors map is not empty.
- Step 35:** If errors exist, add BindingResult to model.
- Step 36:** Add serviceErrors to model.
- Step 37:** Set flag openAddOverlay = true.
- Step 38:** Add newClass to model.
- Step 39:** Call service getStaffMajor().
- Step 40:** DAO retrieves authenticated staff.

- Step 41:** DAO returns staff major.
Step 42: Service returns major.
Step 43: Call service subjectsByMajor(major).
Step 44: DAO queries SpecializedSubjects by major.
Step 45: DAO returns list of subjects.
Step 46: Service returns subjects list.
Step 47: Controller adds specializedSubjects list to model.
Step 48: Add Sessions enum values to the model.
Step 49: Call helper method loadClassesPage().
-

Steps inside loadClassesPage()

- Step 50:** Retrieve pageSize from session (default 5).
Step 51: Calculate firstResult = (page - 1) * pageSize.
Step 52: Call service getStaffMajor().
Step 53: DAO returns staff major.
Step 54: Service returns major.
Step 55: Call service getCampusOfStaff().
Step 56: DAO retrieves authenticated staff and returns campus.
Step 57: Service returns campus.
Step 58: Extract campusId.
Step 59: Call getPaginatedClassesByCampus(firstResult, pageSize, major, campusId).
Step 60: DAO checks if major or campusId is null.
Step 61: If null, return empty list.
Step 62: If valid, build JPQL query.
Step 63: Filter by major and campusId.
Step 64: Set offset and maxResults.
Step 65: Execute query.
Step 66: DAO returns list of SpecializedClasses.
Step 67: Service returns classes list.
Step 68: Call numberOfClassesByCampus(major, campusId).
Step 69: DAO checks major or campusId; if invalid, return 0.
Step 70: Otherwise execute COUNT query with filters.
Step 71: DAO returns class count.
Step 72: Service returns totalClasses.
Step 73: Controller calculates totalPages.
Step 74: Add classes to model.
Step 75: Add specializedSubjects to model.
Step 76: Add pagination data to model.
Step 77: Add Sessions enum to model.
Step 78: Add currentCampusName to model.

Return to main flow after loadClassesPage()

Step 79: Return "SpecializedClassesList" view.

Step 80: Stop execution.

No validation errors branch

Step 81: Call service generateUniqueId(specializedSubjectId, LocalDateTime.now()).

Step 82: DAO determines prefix based on subjectId.

Step 83: If subjectId == "spec001", set prefix "CLSSBH".

Step 84: If subjectId == "spec002", set prefix "CLSSCH".

Step 85: If subjectId == "spec003", set prefix "CLSSDH".

Step 86: If subjectId == "spec004", set prefix "CLSSKH".

Step 87: Otherwise, set prefix "CLSSGEN".

Step 88: Extract year (YY).

Step 89: Extract month and day (MMDD).

Step 90: Start loop to generate unique ID:

- generate random digit 0–9
- combine prefix + YY + MMDD + digit
- check if ID exists
- repeat if ID already exists

Step 91: DAO returns unique classId.

Step 92: Service returns classId to controller.

Step 93: Controller sets the classId into newClass.

Step 94: Call service getStaff().

Step 95: DAO retrieves authenticated staff.

Step 96: DAO returns staff entity.

Step 97: Service returns staff.

Step 98: Controller sets the creator onto newClass.

Step 99: Call service addClass(newClass).

Step 100: DAO checks if newClass is null; if yes, throw IllegalArgumentException.

Step 101: Set createdAt timestamp.

Step 102: Persist newClass into the database.

Step 103: Log success message.

Step 104: Service returns success.

Step 105: Controller adds success message to flash attributes.

Step 106: Controller redirects to /staff-home/specialized-classes-list.

Step 107: Stop execution.

EditSpecializedClassController

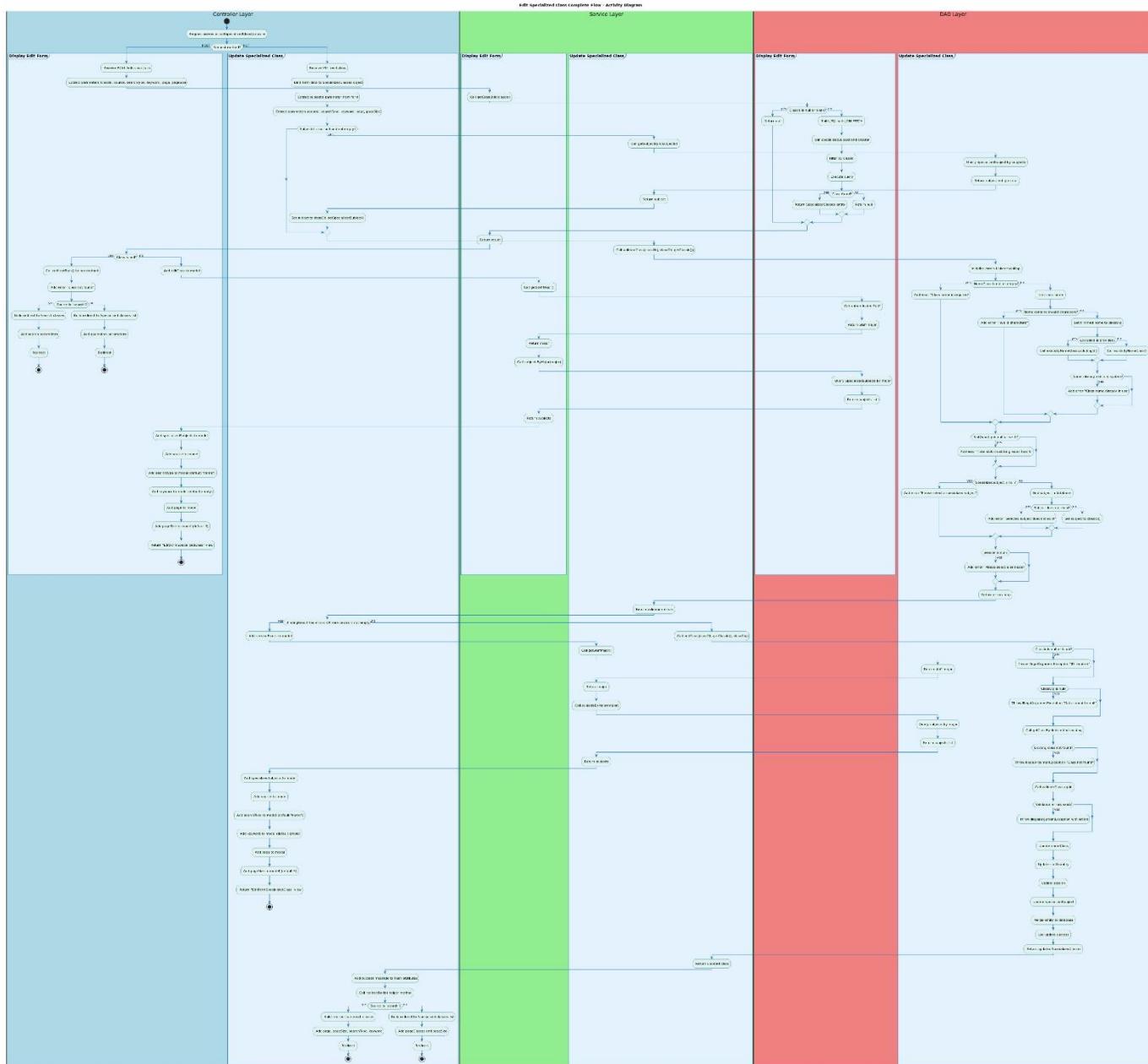


Diagram 60 Add Major Class Composite Flow – Full Layered Activity Diagram

Step 1: Request arrives at EditSpecializedClassController.

Step 2: Determine whether the request method is POST or PUT.

Step 3: If POST request is received at /edit-class-form, extract parameters classId, source, searchType, keyword, page, pageSize.

Step 4: Call service getClassById(classId).

Step 5: DAO checks if classId is null or blank.

Step 6: If null or blank, return null.

Step 7: If valid, build JPQL with JOIN FETCH for specializedSubject and creator.

Step 8: Filter by classId.

Step 9: Execute the query.

Step 10: If class is found, return the SpecializedClasses entity.

Step 11: If class is not found, return null.

Step 12: Service returns the result to controller.

Step 13: Controller checks if class is null.

Step 14: If class is null, call redirectBack() helper.

Step 15: Add error "Class not found".

Step 16: If source == "search", build redirect to /search-classes with search parameters and redirect, then stop execution.

Step 17: Otherwise, build redirect to /specialized-classes-list with pagination parameters and redirect, then stop execution.

Step 18: If class exists, add editClass to model.

Step 19: Call service getStaffMajor().

Step 20: DAO gets authenticated staff.

Step 21: DAO returns staff major.

Step 22: Service returns the major.

Step 23: Call service subjectsByMajor(major).

Step 24: DAO queries SpecializedSubjects by major.

Step 25: DAO returns a list of subjects.

Step 26: Service returns subjects.

Step 27: Controller adds specializedSubjects list to model.

Step 28: Add source to model.

Step 29: Add searchType (default "name").

Step 30: Add keyword (default empty).

Step 31: Add page to model.

Step 32: Add pageSize to model (default 5).

Step 33: Return "EditFormSpecializedClass" view and stop execution.

Step 34: If PUT request is received at /edit-class, bind form data into a SpecializedClasses object (classObj).

Step 35: Extract subjectId from form.

Step 36: Extract source, searchType, keyword, page, pageSize.

Step 37: If subjectId is not null and not empty, call service getSubjectById(subjectId).

Step 38: DAO queries SpecializedSubject by subjectId.

Step 39: DAO returns entity or null.

Step 40: Service returns subject.

Step 41: Controller sets subject into classObj.setSpecializedSubject().

Step 42: Call service validateClass(classObj, classObj.getClassId()).

Step 43: DAO initializes a LinkedHashMap for errors.

Step 44: Check if nameClass is null or empty; if yes, add "Class name is required".

Step 45: If name is provided, trim the name.

Step 46: Check for invalid characters; if invalid, add "Invalid characters".

Step 47: If name is valid, set trimmed name into classObj.

Step 48: If excluding ID is required, call existsByNameClassExcludingId().

Step 49: Otherwise call existsByNameClass().

Step 50: If name already exists, add "Class name already in use".

Step 51: Check if slotQuantity is null or ≤ 0 ; if yes, add "Total slots must be greater than 0".

Step 52: Check if specializedSubject is null; if yes, add "Please select a specialized subject".

Step 53: If specializedSubject not null, attempt to find subject in DB.

Step 54: If subject does not exist, add "Selected subject does not exist".

Step 55: If subject exists, set it into classObj.

Step 56: Check if session field is null; if yes, add "Please select a semester".

Step 57: DAO returns errors map.

Step 58: Service returns validation errors.

Step 59: Controller checks if BindingResult has errors or serviceErrors not empty.

Step 60: If errors exist, add serviceErrors to model.

Step 61: Call service getStaffMajor().

Step 62: DAO returns staff major.

Step 63: Service returns major.

Step 64: Call service subjectsByMajor(major).

Step 65: DAO queries subjects by major.

Step 66: DAO returns list of subjects.

Step 67: Service returns subjects.

Step 68: Controller adds specializedSubjects to model.

Step 69: Add source to model.

Step 70: Add searchType (default "name").

Step 71: Add keyword (default empty).

Step 72: Add page to model.

Step 73: Add pageSize to model (default 5).

Step 74: Return "EditFormSpecializedClass" view and stop execution.

Step 75: If no validation errors, call service editClass(classObj.getClassId(), classObj).

Step 76: DAO checks if classId is null or blank; if yes, throw "ID required" and abort.

Step 77: DAO checks if classObj is null; if yes, throw "Data cannot be null" and abort.

Step 78: DAO calls getClassById(id) to find existing class.

Step 79: If existing class not found, throw "Class not found".

Step 80: DAO calls validateClass again.

Step 81: If validation errors still exist, throw IllegalArgumentException with errors.

Step 82: DAO updates nameClass.

Step 83: DAO updates slotQuantity.

Step 84: DAO updates session.

Step 85: DAO updates specializedSubject.

Step 86: DAO merges updated entity into the database.

Step 87: DAO logs update success.

Step 88: DAO returns updated SpecializedClasses entity.

Step 89: Service returns updated class.

Step 90: Controller adds success message to flash attributes.

Step 91: Controller calls redirectBack() helper method.

Step 92: If source == "search", build redirect to /search-classes including page, pageSize, searchType, keyword, then redirect and stop execution.

Step 93: Otherwise, build redirect to /specialized-classes-list including pageClasses and pageSize, then redirect and stop execution.

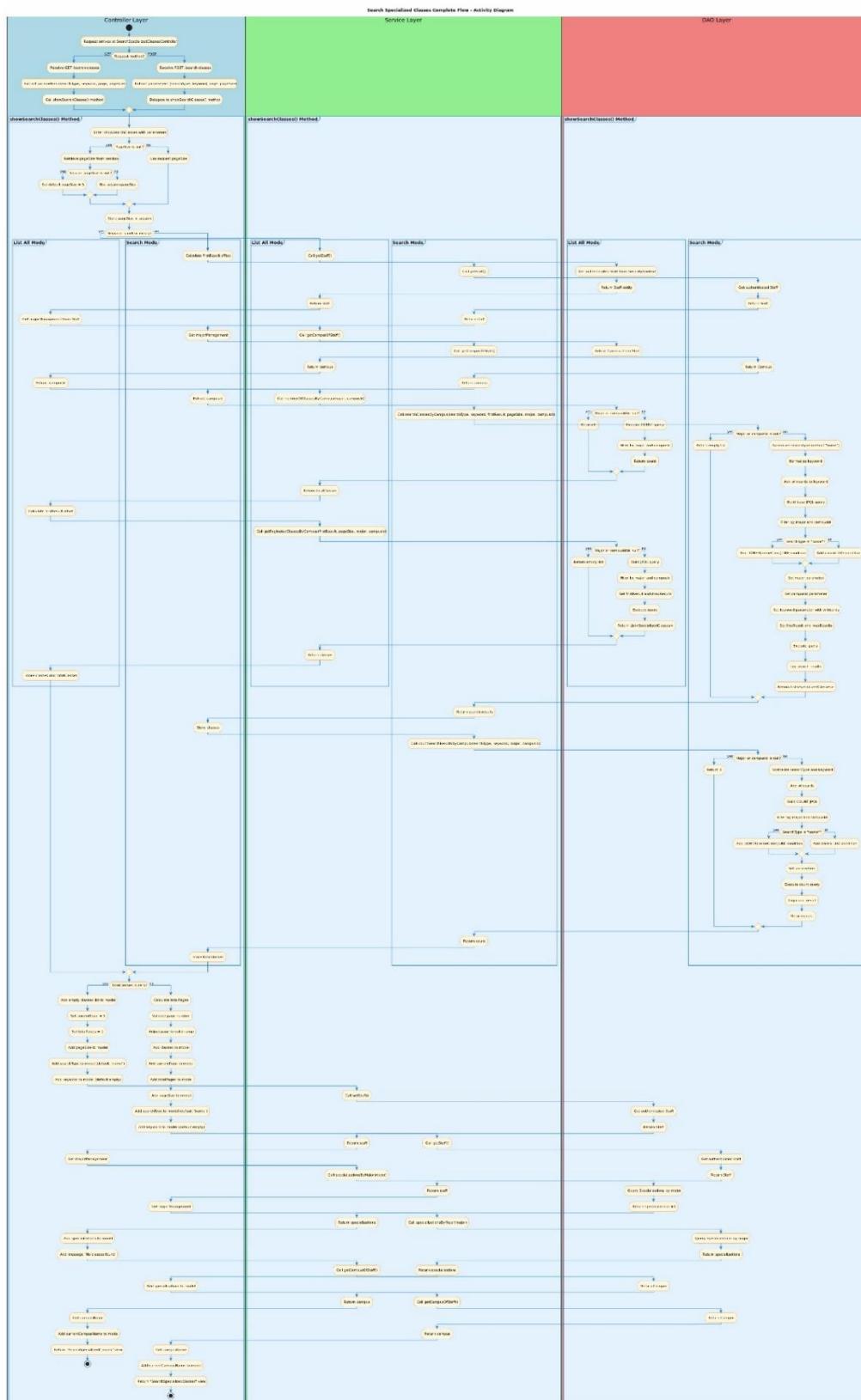


Diagram 61 Search Specialized Classes Complete Flow – Activity Diagram

Step 1: Request arrives at SearchSpecializedClassesController.

Step 2: Determine whether the request method is GET or POST.

Step 3: If GET request is received at /search-classes, extract parameters searchType, keyword, page, pageSize.

Step 4: Call showSearchClasses(searchType, keyword, page, pageSize).

Step 5: If POST request is received at /search-classes, extract parameters searchType, keyword, page, pageSize.

Step 6: Delegate to showSearchClasses(searchType, keyword, page, pageSize).

Step 7: Enter showSearchClasses(searchType, keyword, page, pageSize).

Step 8: Check whether pageSize is null.

Step 9: If pageSize is null, retrieve pageSize from session.

Step 10: Check whether session pageSize is null.

Step 11: If session pageSize is null, set default pageSize = 5.

Step 12: If session pageSize is not null, use session pageSize.

Step 13: If request already provided a non-null pageSize, use request pageSize.

Step 14: Store pageSize in session.

Step 15: Check if keyword is null or empty.

Step 16: If keyword is null or empty (List All Mode):

Step 17: Call service getStaff().

Step 18: DAO gets authenticated Staff from SecurityContext.

Step 19: DAO returns Staff entity.

Step 20: Service returns Staff to controller.

Step 21: Controller gets majorManagement from Staff.

Step 22: Call service getCampusOfStaff().

Step 23: DAO returns Campus from Staff.

Step 24: Service returns Campus.

Step 25: Controller extracts campusId.

Step 26: Call service numberOfClassesByCampus(major, campusId).

Step 27: DAO checks if major or campusId is null.

Step 28: If major or campusId is null, return 0.

Step 29: Otherwise, execute COUNT query filtered by major and campusId.

Step 30: DAO returns count.

Step 31: Service returns totalClasses.

Step 32: Controller calculates firstResult = (page - 1) * pageSize.

Step 33: Call service getPaginatedClassesByCampus(firstResult, pageSize, major, campusId).

Step 34: DAO checks if major or campusId is null.

Step 35: If null, return an empty list.

Step 36: Otherwise, build JPQL query filtered by major and campusId.

- Step 37:** Set firstResult and maxResults.
Step 38: Execute query.
Step 39: DAO returns List<SpecializedClasses>.
Step 40: Service returns classes.
Step 41: Controller stores classes list and totalClasses.
-

- Step 42:** If keyword is not null and not empty (Search Mode):
Step 43: Controller calculates firstResult = (page - 1) * pageSize.
Step 44: Call service getStaff().
Step 45: DAO gets authenticated Staff.
Step 46: DAO returns Staff.
Step 47: Service returns Staff.
Step 48: Controller gets majorManagement from Staff.
Step 49: Call service getCampusOfStaff().
Step 50: DAO returns Campus.
Step 51: Service returns Campus.
Step 52: Controller extracts campusId.
Step 53: Call service searchClassesByCampus(searchType, keyword, firstResult, pageSize, major, campusId).
Step 54: DAO checks if major or campusId is null.
Step 55: If null, return empty list.
Step 56: Otherwise, normalize searchType (default "name").
Step 57: Normalize keyword, then add wildcards to keyword.
Step 58: Build base JPQL query filtered by major and campusId.
Step 59: If searchType is "name", add LOWER(nameClass) LIKE condition.
Step 60: If searchType is "id", add classId LIKE condition.
Step 61: Set major parameter.
Step 62: Set campusId parameter.
Step 63: Set keyword parameter with wildcards.
Step 64: Set firstResult and maxResults.
Step 65: Execute query.
Step 66: Log search results.
Step 67: DAO returns List<SpecializedClasses>.
Step 68: Service returns search results.
Step 69: Controller stores classes list.
Step 70: Call service countSearchResultsByCampus(searchType, keyword, major, campusId).
Step 71: DAO checks if major or campusId is null.
Step 72: If null, return 0.
Step 73: Otherwise, normalize searchType and keyword, then add wildcards.
Step 74: Build COUNT JPQL filtered by major and campusId.

Step 75: If searchType is "name", add LOWER(nameClass) LIKE condition.

Step 76: If searchType is "id", add classId LIKE condition.

Step 77: Set query parameters (major, campusId, keyword).

Step 78: Execute count query.

Step 79: Log count result.

Step 80: DAO returns count.

Step 81: Service returns count.

Step 82: Controller stores totalClasses.

Step 83: Check if totalClasses is zero.

Step 84: If totalClasses is zero, add empty classes list to model.

Step 85: Set currentPage = 1.

Step 86: Set totalPages = 1.

Step 87: Add pageSize to model.

Step 88: Add searchType to model (default "name").

Step 89: Add keyword to model (default empty).

Step 90: Call service getStaff().

Step 91: DAO gets authenticated Staff.

Step 92: DAO returns Staff.

Step 93: Service returns Staff.

Step 94: Controller gets majorManagement from Staff.

Step 95: Call service specializationsByMajor(major).

Step 96: DAO queries Specializations by major.

Step 97: DAO returns list of specializations.

Step 98: Service returns specializations.

Step 99: Controller adds specializations to model.

Step 100: Add message "No classes found" to model.

Step 101: Call service getCampusOfStaff().

Step 102: DAO returns Campus.

Step 103: Service returns Campus.

Step 104: Controller gets campusName.

Step 105: Add currentCampusName to model.

Step 106: Return "SearchSpecializedClasses" view.

Step 107: Stop execution.

Step 108: If totalClasses is not zero, calculate totalPages = ceil(totalClasses / pageSize).

Step 109: Validate page number.

Step 110: If page < 1, set page = 1.

Step 111: If page > totalPages, set page = totalPages.

Step 112: Add classes list to model.

Step 113: Add currentPage = page to model.

Step 114: Add totalPages to model.

Step 115: Add pageSize to model.
Step 116: Add searchType to model (default "name").
Step 117: Add keyword to model (default empty).
Step 118: Call service getStaff().
Step 119: DAO gets authenticated Staff.
Step 120: DAO returns Staff.
Step 121: Service returns Staff.
Step 122: Controller gets majorManagement from Staff.
Step 123: Call service specializationsByMajor(major).
Step 124: DAO queries Specializations by major.
Step 125: DAO returns specializations list.
Step 126: Service returns specializations.
Step 127: Controller adds specializations to model.
Step 128: Call service getCampusOfStaff().
Step 129: DAO returns Campus.
Step 130: Service returns Campus.
Step 131: Controller gets campusName.
Step 132: Add currentCampusName to model.
Step 133: Return "SearchSpecializedClasses" view.
Step 134: Stop execution.

Activity diagram related to campus

AddCampusesController

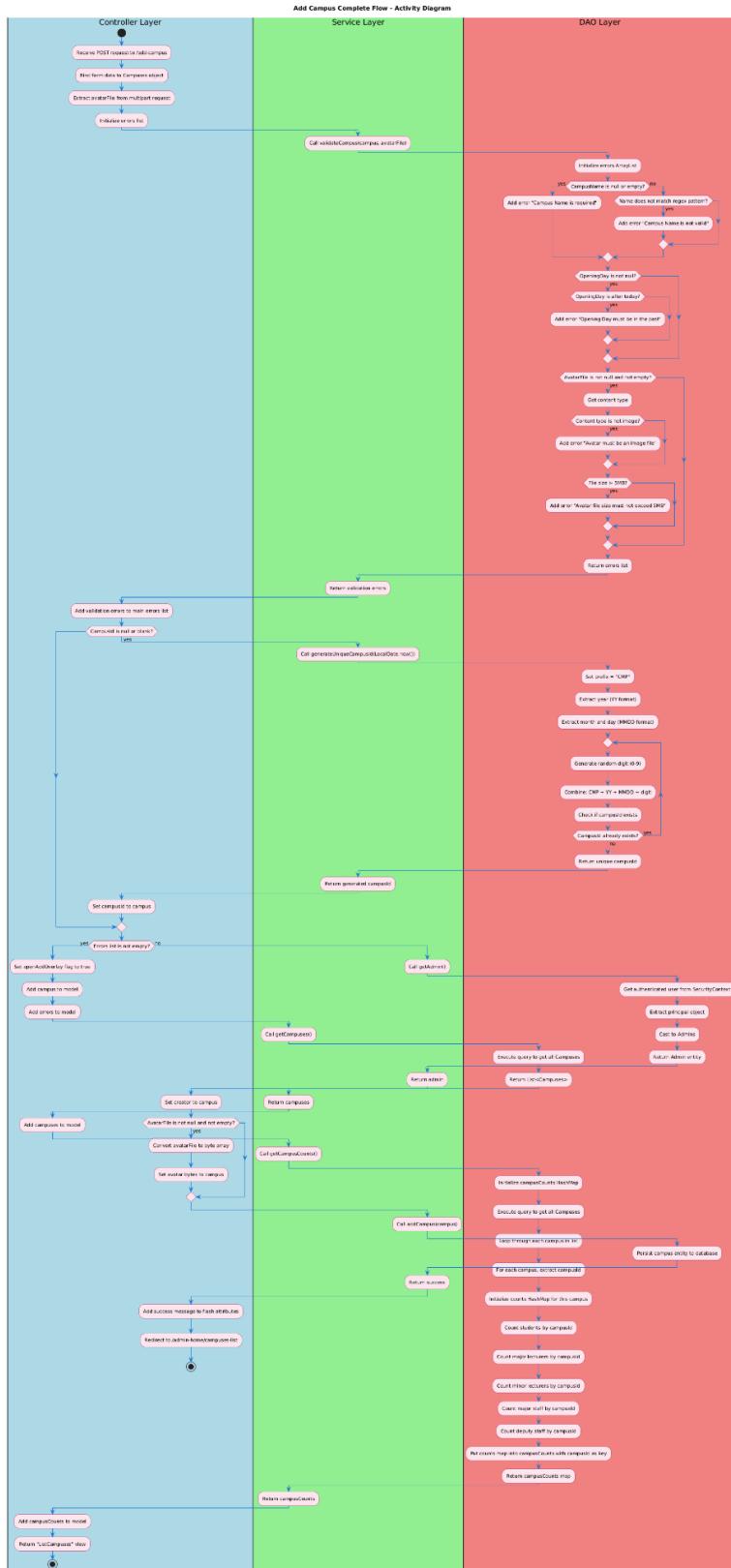


Diagram 62 Add Campus Complete Flow – Activity Diagram

- Step 1:** Receive POST request to /add-campus.
Step 2: Bind form data into a Campuses object.
Step 3: Extract avatarFile from the multipart request.
Step 4: Initialize an errors list.
Step 5: Call service method validateCampus(campus, avatarFile).
Step 6: DAO initializes an ArrayList to store validation errors.
Step 7: Check if campusName is null or empty.
Step 8: If true, add error "Campus Name is required".
Step 9: Otherwise, check whether campusName matches required regex.
Step 10: If regex does not match, add error "Campus Name is not valid".
Step 11: Check if openingDay is not null.
Step 12: If openingDay is after today, add error "Opening Day must be in the past".
Step 13: Check if avatarFile is not null and not empty.
Step 14: If true, get the content type.
Step 15: If content type is not an image type, add "Avatar must be an image file".
Step 16: If file size is greater than 5MB, add "Avatar file size must not exceed 5MB".
Step 17: DAO returns the validation errors list.
Step 18: Service returns the validation errors.
Step 19: Controller adds validation errors to the main errors list.
Step 20: Check if campusId is null or blank.
Step 21: If true, call service generateUniqueCampusId(LocalDate.now()).
Step 22: DAO sets prefix "CMP".
Step 23: DAO extracts year in YY format.
Step 24: DAO extracts month and day in MMDD format.
Step 25: Start loop to generate ID:
 - generate random digit 0–9
 - combine "CMP" + YY + MMDD + digit
 - check if campusId exists
 - repeat while ID exists**Step 26:** DAO returns unique campusId.
Step 27: Service returns generated campusId.
Step 28: Controller sets campusId into the campus object.
Step 29: Check if errors list is not empty.

BRANCH A — ERRORS PRESENT

Step 30: Set flag openAddOverlay = true.
Step 31: Add campus object back to model.
Step 32: Add errors list to model.
Step 33: Call service getCampuses().
Step 34: DAO executes query to retrieve all Campuses.
Step 35: DAO returns List<Campuses>.
Step 36: Service returns the campuses.
Step 37: Controller adds campuses to the model.
Step 38: Call service getCampusCounts().
Step 39: DAO initializes a HashMap named campusCounts.
Step 40: DAO executes query to retrieve all Campuses.
Step 41: DAO loops through every campus.
Step 42: For each campus, extract campusId.
Step 43: Initialize a counts HashMap for that campus.
Step 44: Count students by campusId.
Step 45: Count major lecturers by campusId.
Step 46: Count minor lecturers by campusId.
Step 47: Count major staff by campusId.
Step 48: Count deputy staff by campusId.
Step 49: Put the counts map into campusCounts using campusId as key.
Step 50: DAO returns campusCounts.
Step 51: Service returns campusCounts.
Step 52: Controller adds campusCounts to the model.
Step 53: Controller returns "ListCampuses" view.
Step 54: Stop execution.

BRANCH B — NO ERRORS

Step 55: Call service getAdmin().
Step 56: DAO retrieves authenticated user from SecurityContext.
Step 57: DAO extracts principal.
Step 58: DAO casts principal to Admins.
Step 59: DAO returns Admin entity.
Step 60: Service returns the admin.
Step 61: Controller sets admin as the creator of the campus.
Step 62: Check if avatarFile is not null and not empty.
Step 63: If true, convert avatarFile to a byte array.
Step 64: Set avatar bytes into the campus object.
Step 65: Call service method addCampus(campus).
Step 66: DAO persists campus entity to the database.
Step 67: Service returns success.

Step 68: Controller adds success message to flash attributes.

Step 69: Controller redirects to /admin-home/campuses-list.

Step 70: Stop execution.

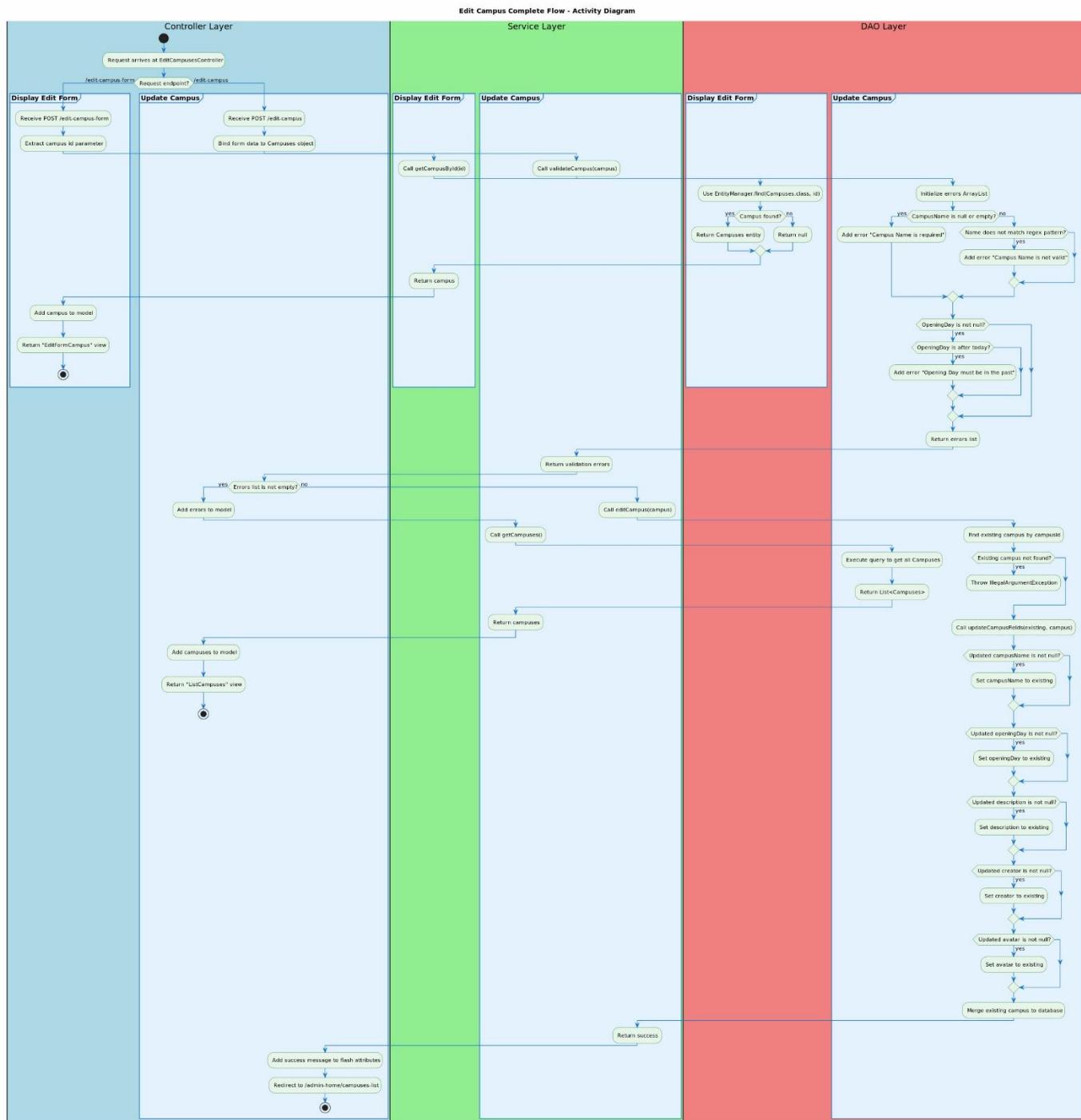


Diagram 63 Edit Campus Complete Flow – Activity Diagram

Step 1: Request arrives at EditCampusesController.

Step 2: Determine whether the endpoint is /edit-campus-form or /edit-campus.

BRANCH A — DISPLAY EDIT FORM (/edit-campus-form)

Step 3: Receive POST request to /edit-campus-form.

Step 4: Extract the campus ID parameter from the request.

Step 5: Call service method getCampusById(id).

Step 6: DAO calls EntityManager.find(Campuses.class, id).

Step 7: If a campus with the given ID exists, DAO returns the Campuses entity.

Step 8: If no campus is found, DAO returns null.

Step 9: Service returns the result (campus or null).

Step 10: Controller adds the campus object to the model.

Step 11: Controller returns "EditFormCampus" view.

Step 12: Stop execution.

BRANCH B — UPDATE CAMPUS (/edit-campus)

Step 13: Receive POST request to /edit-campus.

Step 14: Bind form fields to a Campuses object.

Step 15: Call service method validateCampus(campus).

Step 16: DAO initializes an ArrayList to store validation errors.

Step 17: Check if campusName is null or empty.

Step 18: If yes, add "Campus Name is required" to errors.

Step 19: If not empty, check whether the campus name matches the required regex pattern.

Step 20: If regex does not match, add "Campus Name is not valid".

Step 21: Check if openingDay is not null.

Step 22: If openingDay is after today, add "Opening Day must be in the past".

Step 23: DAO returns the validation errors list.

Step 24: Service returns the validation errors to the controller.

Step 25: Controller checks whether the errors list is not empty.

SUB-BRANCH B1 — ERRORS FOUND

Step 26: Add errors to the model.

Step 27: Call service getCampuses().

Step 28: DAO executes a query to retrieve all campuses.

Step 29: DAO returns List<Campuses>.

Step 30: Service returns the campus list.

Step 31: Controller adds the campus list to the model.

Step 32: Controller returns "ListCampuses" view.

Step 33: Stop execution.

SUB-BRANCH B2 — NO ERRORS (VALIDATION PASSED)

Step 34: Call service editCampus(campus).

Step 35: DAO searches for existing campus using campusId.

Step 36: If existing campus is not found, DAO throws IllegalArgumentException, terminating the process.

Step 37: DAO calls updateCampusFields(existing, campus).

Step 38: If the updated campusName is not null, assign it to the existing entity.

Step 39: If the updated openingDay is not null, assign it to the existing entity.

Step 40: If the updated description is not null, assign it to the existing entity.

Step 41: If the updated creator is not null, assign it to the existing entity.

Step 42: If the updated avatar is not null, assign it to the existing entity.

Step 43: DAO merges the updated campus entity into the database.

Step 44: Service returns success.

Step 45: Controller adds a success message to flash attributes.

Step 46: Controller redirects to /admin-home/campuses-list.

Step 47: Stop execution.

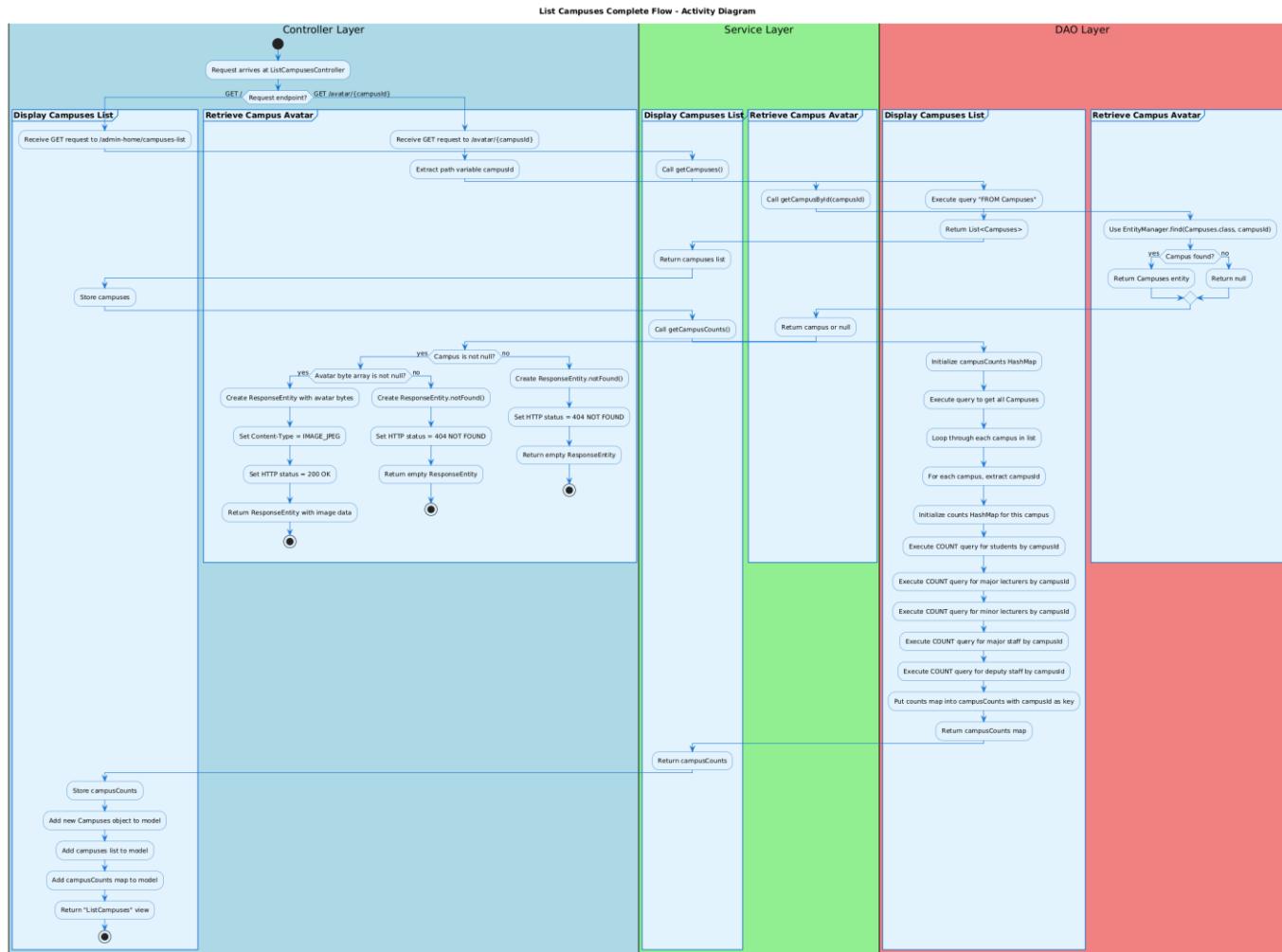


Diagram 64 List Campuses Complete Flow – Activity Diagram

Step 1: Request arrives at ListCampusesController.

Step 2: Determine whether the request is GET /admin-home/campuses-list or GET /avatar/{campusId}.

BRANCH A — DISPLAY CAMPUSES LIST (GET /admin-home/campuses-list)

Step 3: Receive GET request to /admin-home/campuses-list.

Step 4: Call service method getCampuses().

Step 5: DAO executes query "FROM Campuses".

Step 6: DAO returns List<Campuses>.

Step 7: Service returns the campuses list.

Step 8: Controller stores the campuses list.

Step 9: Call service method getCampusCounts().

Step 10: DAO initializes a HashMap named campusCounts.

Step 11: DAO executes query to retrieve all campuses.

Step 12: DAO loops through each campus.

Step 13: For each campus, extract campusId.

Step 14: Initialize a new counts map for that campus.

Step 15: Execute COUNT query for students by campusId.

Step 16: Execute COUNT query for major lecturers by campusId.

Step 17: Execute COUNT query for minor lecturers by campusId.

Step 18: Execute COUNT query for major staff by campusId.

Step 19: Execute COUNT query for deputy staff by campusId.

Step 20: Insert the counts map into campusCounts using campusId as key.

Step 21: DAO returns the completed campusCounts map.

Step 22: Service returns campusCounts to the controller.

Step 23: Controller stores campusCounts.

Step 24: Controller adds a new empty Campuses object to the model.

Step 25: Controller adds the campuses list to the model.

Step 26: Controller adds the campusCounts map to the model.

Step 27: Controller returns the "ListCampuses" view.

Step 28: Stop execution.

BRANCH B — RETRIEVE CAMPUS AVATAR (GET /avatar/{campusId})

Step 29: Receive GET request to /avatar/{campusId}.

Step 30: Extract the path variable campusId.

Step 31: Call service method getCampusById(campusId).

Step 32: DAO calls EntityManager.find(Campuses.class, campusId).

Step 33: If a campus entity is found, DAO returns the Campuses entity.

Step 34: If not found, DAO returns null.

Step 35: Service returns the campus entity or null.

Step 36: Controller checks if the campus is not null.

Step 37: If campus exists, check whether its avatar byte array is not null.

Step 38: If avatar bytes exist, create a ResponseEntity with the bytes, set Content-Type = IMAGE_JPEG, set HTTP status 200, and return the ResponseEntity.

Step 39: If avatar bytes do not exist, return ResponseEntity.notFound() with status 404.

Step 40: If campus is null, return ResponseEntity.notFound() with status 404.

Step 41: Stop execution.

Activity diagram related to major subject

AddMajorSubjectController

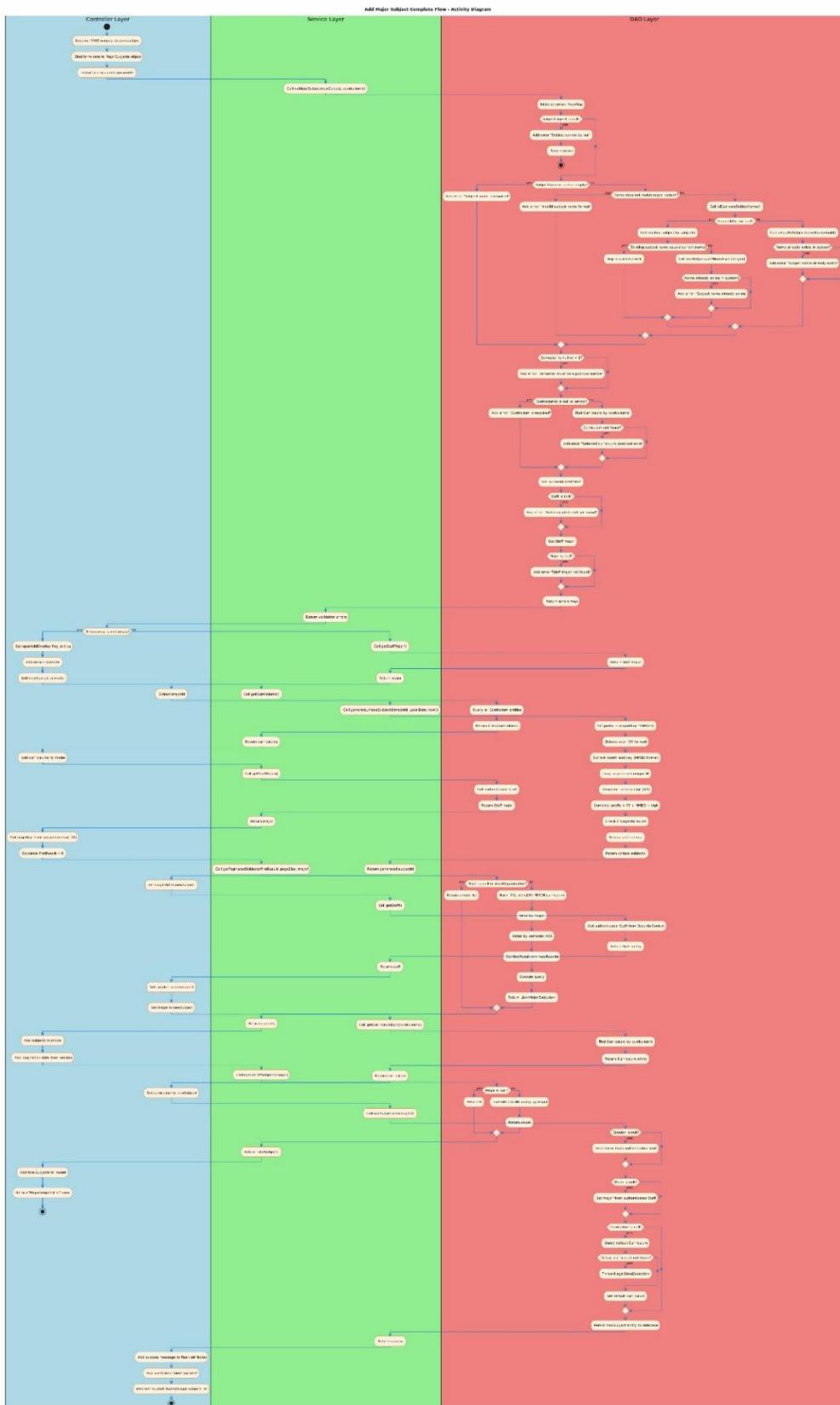


Diagram 65 Add Major Subject Complete Flow – Activity Diagram

- Step 1:** Receive POST request to /add-subject.
Step 2: Bind form data to MajorSubjects object.
Step 3: Extract curriculumId parameter.
Step 4: Call service method validateSubject(newSubject, curriculumId).
Step 5: DAO initializes an empty HashMap for errors.
Step 6: Check whether the subject object is null.
Step 7: If null, add error "**Subject cannot be null**", return the errors map, and stop validation.
Step 8: Validate the subject name.
Step 9: If subjectName is null or empty, add error "**Subject name is required**".
Step 10: Otherwise, check the name against the regex pattern.
Step 11: If name does not match the valid pattern, add error "**Invalid subject name format**".
Step 12: If the name has valid format, check for duplicate names.
Step 13: If a subjectId is provided, retrieve the existing subject.
Step 14: If the existing subject has the same name, skip duplicate checking.
Step 15: Otherwise, call existsBySubjectNameExcludingId().
Step 16: If name already exists, add error "**Subject name already exists**".
Step 17: If no subjectId is provided, call existsBySubjectNameExcludingId() directly.
Step 18: If the name exists, add error "**Subject name already exists**".
Step 19: Validate semester.
Step 20: If semester is null or lower than 1, add error "**Semester must be a positive number**".
Step 21: Validate curriculumId.
Step 22: If curriculumId is null or empty, add error "**Curriculum is required**".
Step 23: Otherwise, find Curriculum by curriculumId.
Step 24: If not found, add error "**Selected curriculum does not exist**".
Step 25: Retrieve the authenticated staff.
Step 26: If staff is null, add error "**Authenticated staff not found**".
Step 27: Retrieve staff major.
Step 28: If major is null, add error "**Staff major not found**".
Step 29: DAO returns the complete map of validation errors.
Step 30: Service returns the validation errors to the controller.

CONTROLLER DECISION

- Step 31:** Controller checks whether the error map is empty.

BRANCH A — VALIDATION FAILED

- Step 32:** Set openAddOverlay = true.
Step 33: Add errors to the model.
Step 34: Add the newSubject back to the model.
Step 35: Call service method getCurriculums().

Step 36: DAO queries all Curriculum entities and returns a list.

Step 37: Service returns the curriculum list.

Step 38: Controller adds the curriculum list to the model.

Step 39: Call service method getStaffMajor().

Step 40: DAO retrieves the authenticated staff and returns staff major.

Step 41: Service returns the major.

Step 42: Retrieve pageSize from session (default 20).

Step 43: Set firstResult = 0.

Step 44: Call service method getPaginatedSubjects(firstResult, pageSize, major).

Step 45: DAO checks major and pagination validity.

Step 46: If invalid, return empty list.

Step 47: Otherwise, build JPQL with JOIN FETCH curriculum, filter by major, order by semester ascending, set pagination, execute query, and return the subject list.

Step 48: Service returns the paginated subjects.

Step 49: Controller adds subjects to the model.

Step 50: Controller adds pagination data from session.

Step 51: Call service method numberOfSubjects(major).

Step 52: DAO checks major; if null, return 0.

Step 53: Otherwise, execute COUNT query by major and return total subject count.

Step 54: Service returns totalSubjects.

Step 55: Controller adds totalSubjects to the model.

Step 56: Controller returns the "MajorSubjectsList" view.

Step 57: Stop execution.

BRANCH B — VALIDATION PASSED

Step 58: Call service method getStaffMajor().

Step 59: DAO returns staff major.

Step 60: Service returns major.

Step 61: Controller extracts majorId.

Step 62: Call service method generateUniqueId(majorId, LocalDate.now()).

Step 63: DAO sets prefix = majorId (or "SUBGEN" if missing).

Step 64: Extract year in YY format.

Step 65: Extract month and day in MMDD format.

Step 66: Start loop for ID generation.

Step 67: Generate a random digit 0–9.

Step 68: Combine into prefix + YY + MMDD + digit.

Step 69: Check whether subjectId already exists.

Step 70: Repeat until ID is unique.

Step 71: DAO returns the unique subjectId.

Step 72: Service returns the generated subjectId.

Step 73: Controller sets the subjectId to newSubject.

Step 74: Call service method getStaff().

Step 75: DAO retrieves authenticated staff.

Step 76: Service returns staff.

Step 77: Controller sets creator to newSubject.

Step 78: Controller sets major to newSubject.

Step 79: Call service method getCurriculumById(curriculumId).

Step 80: DAO retrieves Curriculum entity by ID.

Step 81: Service returns the curriculum.

Step 82: Controller sets curriculum to newSubject.

Step 83: Call service method addSubject(newSubject).

Step 84: DAO checks creator; if null, populate from authenticated staff.

Step 85: DAO checks major; if null, populate from staff major.

Step 86: DAO checks curriculum; if null, query for default curriculum.

Step 87: If no default curriculum found, throw IllegalStateException.

Step 88: DAO persists the new subject entity.

Step 89: Service returns success.

Step 90: Controller adds a success message to flash attributes.

Step 91: Controller sets alertClass = "alert-success".

Step 92: Controller redirects to /staff-home/major-subjects-list.

Step 93: Stop execution.

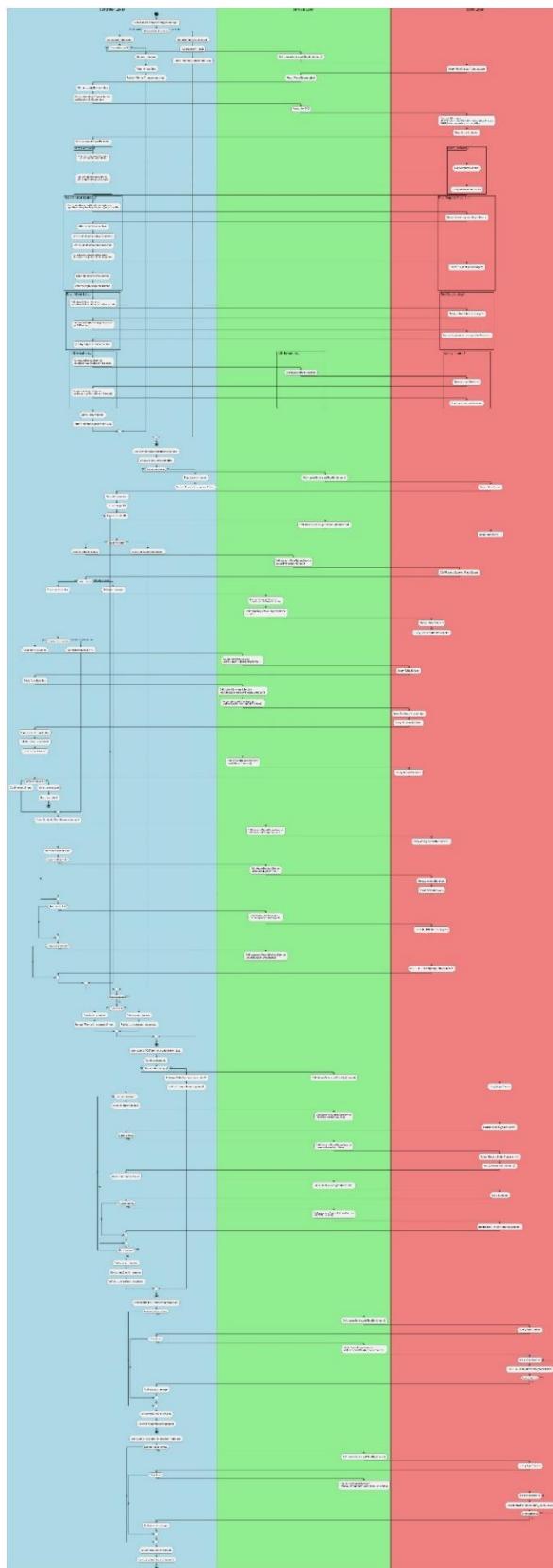


Diagram 66 Edit Major Subject Complete Flow – Activity Diagram

001479794

Page 431 | 635 total

BRANCH A — classId EXISTS IN SESSION

Step 3: Controller retrieves classId from the session.

Step 4: Controller checks whether a class with this classId exists in the database.

SUB-BRANCH A1 — CLASS FOUND

Step 5: Controller calls the service method to get the class by ID.

Step 6: Service calls the DAO to retrieve the class entity by ID.

Step 7: DAO queries the MajorClasses table for the given ID.

Step 8: DAO returns the MajorClasses entity to the service.

Step 9: Service returns the class object to the controller.

Step 10: Controller extracts the associated subject and subjectId from the class.

Step 11: Controller calls the students-major-classes service to get students in the class.

Step 12: Service forwards the request to the DAO.

Step 13: DAO executes a query to retrieve all students belonging to the given major class.

Step 14: DAO returns the list of students in the class to the service.

Step 15: Service returns the list of students to the controller.

Step 16: Controller stores the IDs of these students in a Set (students currently in the class).

PARTITION — FETCH LECTURERS

Step 17: Controller calls the lecturers-classes service to list lecturers in the class.

Step 18: Service forwards the call to the DAO.

Step 19: DAO queries the lecturers currently assigned to this class.

Step 20: DAO returns the list of lecturers in the class to the service.

Step 21: Service returns the lecturers-in-class list to the controller.

Step 22: Controller calls the lecturers-classes service to list lecturers not in the class.

Step 23: Service forwards the call to the DAO.

Step 24: DAO queries the lecturers not assigned to this class.

Step 25: DAO returns the list of lecturers not in the class to the service.

Step 26: Service returns the lecturers-not-in-class list to the controller.

PARTITION — FETCH REQUIRED STUDENTS

Step 27: Controller calls the student-required-major-subjects service to get students who are required to take this subject.

Step 28: Service forwards the request to the DAO.

Step 29: DAO queries the StudentRequiredMajorSubjects records for the given subject.

Step 30: DAO returns the required-subject records to the service.

Step 31: Service returns the required-student list to the controller.

Step 32: Controller filters out required students who are already in the class (using the Set of student IDs).

Step 33: Controller filters out required students whose IDs are in the retake-student list.

Step 34: Controller filters out required students whose IDs are in the temporary-retake-student list.

Step 35: For each remaining required student, controller calls the academic-transcripts service to check whether the student has passed the subject.

Step 36: Service forwards the check to the DAO.

Step 37: DAO checks whether the student has a passing record for this subject.

Step 38: DAO returns the pass status to the service.

Step 39: Service returns the result to the controller.

Step 40: Controller filters out students who have already passed the subject.

Step 41: Controller stores the remaining students as eligibleRequiredStudents.

PARTITION — FETCH RETAKE LISTS

Step 42: Controller calls the retake-subjects service to get retake records for the given subjectId.

Step 43: Service forwards the request to the DAO.

Step 44: DAO queries RetakeSubjects by subject.

Step 45: DAO returns the retake list to the service.

Step 46: Service returns the retake list to the controller.

Step 47: Controller calls the temporary-retake-subjects service to get all pending temporary retake records.

Step 48: Service forwards the request to the DAO.

Step 49: DAO queries all pending TemporaryRetakeSubjects.

Step 50: DAO returns this list to the service.

Step 51: Service returns the pending temporary-retake list to the controller.

Step 52: Controller filters retake and temporary-retake records by the current subjectId and removes any students already in the class.

PARTITION — SPLIT BY BALANCE

Step 53: Controller calls the retake-subjects service to get students with sufficient balance.

Step 54: Service checks the balance of each relevant student and calls the DAO for account data.

Step 55: DAO queries account balances and returns them to the service.

Step 56: Service determines which students have sufficient balance and returns this group to the controller.

Step 57: Controller calls the retake-subjects service to get students with insufficient balance.

Step 58: Service retrieves and compares balances via the DAO as needed.

Step 59: DAO returns the account data to the service.

Step 60: Service determines which students have insufficient balance and returns this group to the controller.

FINALIZE MODEL (CLASS FOUND CASE)

Step 61: Controller adds all collected data (class, subject, students in class, eligible required students, retake lists, temporary retake lists, balance groups, lecturers in class, lecturers not in class, etc.) to the model.

Step 62: Controller returns the "MemberArrangement" view with the populated model.

Step 63: Stop execution.

SUB-BRANCH A2 — CLASS NOT FOUND

Step 64: Controller sets an error message indicating the selected class does not exist.

Step 65: Controller prepares empty lists for students, lecturers, retake students, and other collections.

Step 66: Controller adds the error message and empty data to the model.

Step 67: Controller returns the "MemberArrangement" view.
Step 68: Stop execution.

BRANCH B — NO classId IN SESSION

Step 69: Controller sets an error message indicating that no class has been selected.
Step 70: Controller prepares an empty model or empty lists for all data collections.
Step 71: Controller returns the "MemberArrangement" view with the error message.
Step 72: Stop execution.

FLOW 2 — POST /add-students-to-class

Step 73: User submits POST request to /add-students-to-class.
Step 74: Controller validates classId and studentIds from the request.
Step 75: Controller checks whether validation passes.

BRANCH C — VALIDATION PASSED

Step 76: Controller calls the classes service to get the class by classId.
Step 77: Service calls the DAO to retrieve the MajorClasses entity.
Step 78: DAO queries the class record and returns it to the service.
Step 79: Service returns the class object to the controller.
Step 80: Controller retrieves the authenticated staff information.
Step 81: Controller extracts the subjectId (and subject) from the class.
Step 82: Controller initializes counters and collections to track successfully added students and errors.

LOOP OVER EACH STUDENT

Step 83: Controller begins a loop over each submitted studentId.
Step 84: For the current studentId, controller calls the students service to get the student by ID.
Step 85: Service calls the DAO to query the student record.
Step 86: DAO returns the student (or null) to the service.
Step 87: Service returns the student (or null) to the controller.
Step 88: Controller checks whether the student exists.

SUB-BRANCH C1 — STUDENT EXISTS

Step 89: Controller checks whether the student is already enrolled in the class.
Step 90: Controller calls the students-major-classes service to check existence by student and class.
Step 91: Service calls the DAO to perform an existence or count query in the enrollments table.
Step 92: DAO returns the existence result to the service.
Step 93: Service returns the existence result to the controller.
Step 94: Controller checks whether the student is not yet in the class.

SUB-BRANCH C1.1 — STUDENT NOT IN CLASS

- Step 95: Controller determines the retake status for this student and subject.
- Step 96: Controller calls the retake-subjects service to check existence by student and subject.
- Step 97: Controller calls the temporary-retake-subjects service to check existence for this student and subject.
- Step 98: Services call DAOs to query retake and temporary-retake tables.
- Step 99: DAOs return existence information to the services.
- Step 100: Services return retake and temporary-retake existence flags to the controller.
- Step 101: Controller decides whether this is a new retake student (requiring fee) or comes from an existing retake/temporary list.

SUB-BRANCH C1.1.a — NEW RETAKE STUDENT (FEE APPLIES)

- Step 102: Controller calculates the base re-study fee needed for this subject and student.
- Step 103: Controller calls the tuition-by-year service to get tuition information including re-study fees for the relevant year.
- Step 104: Service calls the DAO to load tuition configuration for that year.
- Step 105: DAO returns tuition configuration to the service.
- Step 106: Service returns the tuition configuration to the controller.
- Step 107: Controller uses this configuration to compute the re-study fee.
- Step 108: Controller checks whether the student has an active scholarship for the current year.
- Step 109: Controller calls the student-scholarship service to get active scholarships for the student and year.
- Step 110: Controller calls the scholarship-by-year service to get finalized scholarship information for the relevant scholarship and year.
- Step 111: Services call their DAOs to query scholarship mappings and scholarship-by-year data.
- Step 112: DAOs return scholarship data to the services.
- Step 113: Services return scholarship information to the controller.
- Step 114: Controller applies any applicable scholarship discount.
- Step 115: Controller calculates the final fee amount to be deducted (finalFeeToDeduct).
- Step 116: Controller checks whether the student's account balance is sufficient.
- Step 117: Controller calls the account-balances service to verify that the balance covers finalFeeToDeduct.
- Step 118: Service calls the DAO to query the student's account balance.
- Step 119: DAO returns the account balance to the service.
- Step 120: Service compares the balance to the required fee and returns the sufficiency result to the controller.
- Step 121: Controller checks whether the balance is sufficient.

SUB-BRANCH C1.1.a.i — SUFFICIENT BALANCE

- Step 122: Controller proceeds with the enrollment for this student.

SUB-BRANCH C1.1.a.ii — INSUFFICIENT BALANCE

- Step 123: Controller adds an error message indicating that the student's balance is insufficient.
- Step 124: Controller skips enrolling this student into the class.
- Step 125: Controller continues with the next studentId in the loop.

SUB-BRANCH C1.1.b — FROM EXISTING RETAKE/TEMPORARY LIST (NO NEW FEE)

Step 126: Controller identifies that the student comes from an existing retake or temporary list where fee handling is already covered or not required.

Step 127: Controller sets finalFeeToDeduct to zero for this enrollment.

ENROLLMENT FOR ACCEPTED STUDENT

Step 128: Controller creates a new enrollment entity linking the student to the class.

Step 129: Controller calls the students-major-classes service to add the student to the class.

Step 130: Service calls the DAO to persist the new enrollment.

Step 131: DAO saves the enrollment entity to the database.

Step 132: Service returns success to the controller.

Step 133: Controller increments the addedCount of successfully enrolled students.

EXTRA ACTIONS FOR NEW RETAKE WITH FEE

Step 134: Controller checks whether this enrollment corresponds to a new retake student with a fee deduction.

Step 135: If yes, controller calls the retake-subjects service to deduct the fee and log the payment.

Step 136: Service calls the DAO to update the student's account balance.

Step 137: DAO updates the account balance record.

Step 138: DAO inserts a payment history entry for the transaction.

Step 139: Service returns success to the controller.

CLEANUP FOR RETAKE / TEMPORARY LISTS

Step 140: Controller checks whether the student came from the permanent retake list.

Step 141: If yes, controller calls the retake-subjects service to delete the corresponding retake entry for this student and subject.

Step 142: Service calls the DAO to remove the retake record.

Step 143: DAO deletes the retake entry from the database.

Step 144: Service returns to the controller.

Step 145: Controller checks whether the student came from the temporary retake list.

Step 146: If yes, controller calls the temporary-retake-subjects service to delete the temporary entry for this student and subject.

Step 147: Service calls the DAO to remove the temporary retake record.

Step 148: DAO deletes the temporary retake entry.

Step 149: Service returns to the controller.

Step 150: Controller completes processing for this student and moves on to the next studentId.

SUB-BRANCH C1.2 — STUDENT ALREADY IN CLASS

Step 151: If the existence check shows the student is already in the class, controller adds an error message indicating duplicate enrollment.

Step 152: Controller skips creating an enrollment for this student and continues with the next studentId.

SUB-BRANCH C2 — STUDENT DOES NOT EXIST

Step 153: If the student record is not found, controller adds an error message that the student was not found.

Step 154: Controller skips further processing for this studentId and moves to the next one.

LOOP TERMINATION AND RESULT

Step 155: Controller continues the loop while there are more studentId values.

Step 156: When all studentId values have been processed, the loop ends.

Step 157: Controller checks whether any errors were accumulated during processing.

BRANCH C3 — ERRORS PRESENT

Step 158: Controller populates the model with error messages and any partial data needed.

Step 159: Controller returns the "MemberArrangement" view with the error information.

Step 160: Stop execution.

BRANCH C4 — NO ERRORS

Step 161: Controller adds a success message indicating how many students were added to the class.

Step 162: Controller redirects to the member arrangement page.

Step 163: Stop execution.

BRANCH D — VALIDATION FAILED (INITIAL CHECK)

Step 164: Controller populates the model with validation errors for classId and/or studentIds.

Step 165: Controller returns the "MemberArrangement" view with validation errors.

Step 166: Stop execution.

FLOW 3 — POST /remove-student-from-class

Step 167: User submits POST request to /remove-student-from-class.

Step 168: Controller validates the submitted studentIds.

Step 169: Controller checks whether the list of studentIds is not empty.

BRANCH E — studentIds PROVIDED

Step 170: Controller calls the classes service to get the class by classId.

Step 171: Service calls the DAO to retrieve the class entity.

Step 172: DAO queries MajorClasses and returns the class record.

Step 173: Service returns the class object to the controller.

Step 174: Controller begins a loop over each submitted studentId.

Step 175: For the current studentId, controller checks whether the student is currently enrolled in the class.

Step 176: Controller calls the students-major-classes service to check existence by student and class.

Step 177: Service calls the DAO to perform the existence/count check.

Step 178: DAO returns whether the enrollment exists.

Step 179: Service returns the existence result to the controller.
Step 180: Controller checks whether the student is in the class.

SUB-BRANCH E1 — STUDENT IS IN CLASS

Step 181: Controller calls the students-major-classes service to remove the student from the class.

Step 182: Service calls the DAO to retrieve the enrollment entity for this student and class.

Step 183: DAO loads the enrollment entity.

Step 184: DAO removes the enrollment entity from the database.

Step 185: Service returns success to the controller.

Step 186: Controller increments the removedCount of students removed from the class.

Step 187: Controller calls the students service to get the full student entity by ID.

Step 188: Service calls the DAO to query the student.

Step 189: DAO returns the student entity.

Step 190: Service returns the student to the controller.

Step 191: Controller checks whether the student entity is found.

Step 192: If the student is found, controller calls the temporary-retake-subjects service to add this student to the temporary list for this subject.

Step 193: Service calls the DAO to insert a TemporaryRetakeSubjects record.

Step 194: DAO inserts the temporary retake entry.

Step 195: Service returns to the controller.

Step 196: Controller continues with the next studentId.

SUB-BRANCH E2 — STUDENT NOT IN CLASS

Step 197: If the student is not enrolled in the class, controller does not perform a removal for this studentId.

Step 198: Controller moves on to the next studentId.

LOOP TERMINATION AND RESULT

Step 199: Controller continues the loop while there are more studentId values.

Step 200: When all studentIds have been processed, the loop ends.

Step 201: Controller adds a success message indicating how many students were removed.

Step 202: Controller sets the current class ID in the session for redirect purposes.

Step 203: Controller redirects to the member arrangement page.

Step 204: Stop execution.

BRANCH F — NO studentIds PROVIDED

Step 205: Controller adds an error message indicating that at least one student must be selected.

Step 206: Controller redirects to the member arrangement page.

Step 207: Stop execution.

FLOW 4 — POST /add-lecturers-to-class

Step 208: User submits POST request to /add-lecturers-to-class.

Step 209: Controller checks whether the submitted lecturerIds list is not empty.

BRANCH G — lecturerIds PROVIDED

Step 210: Controller calls the classes service to get the class by classId.

Step 211: Service calls the DAO to retrieve the class entity.

Step 212: DAO queries MajorClasses and returns the class record.

Step 213: Service returns the class object to the controller.

Step 214: Controller checks whether the class is found.

SUB-BRANCH G1 — CLASS FOUND

Step 215: Controller calls the lecturers—classes service to add the given lecturers to the class.

Step 216: Service iterates over each lecturerId and forwards insertion requests to the DAO.

Step 217: DAO loops through each lecturerId and inserts the association between lecturer and class into the join table.

Step 218: DAO completes all insert operations and returns to the service.

Step 219: Service returns success to the controller.

Step 220: Controller adds a success message indicating that lecturers have been added.

SUB-BRANCH G2 — CLASS NOT FOUND

Step 221: If the class is not found, controller may treat this as an error internally, but the flow continues to redirect (as in the diagram).

FINALIZE

Step 222: Controller sets the current class ID in the session for redirect purposes.

Step 223: Controller redirects to the member arrangement page.

Step 224: Stop execution.

BRANCH H — NO lecturerIds PROVIDED

Step 225: If no lecturerIds are submitted, controller does not perform any add operation.

Step 226: Controller sets the current class ID in the session if available.

Step 227: Controller redirects to the member arrangement page.

Step 228: Stop execution.

FLOW 5 — POST /remove-lecturer-from-class

Step 229: User submits POST request to /remove-lecturer-from-class.

Step 230: Controller checks whether the submitted lecturerIds list is not empty.

BRANCH I — lecturerIds PROVIDED

Step 231: Controller calls the classes service to get the class by classId.

Step 232: Service calls the DAO to retrieve the class entity.

Step 233: DAO queries MajorClasses and returns the class record.

Step 234: Service returns the class object to the controller.

Step 235: Controller checks whether the class is found.

SUB-BRANCH I1 — CLASS FOUND

Step 236: Controller calls the lecturers—classes service to remove the given lecturers from the class.

Step 237: Service iterates over each lecturerId and forwards delete requests to the DAO.

Step 238: DAO loops through each lecturerId and deletes the association between lecturer and class from the join table.

Step 239: DAO completes all delete operations and returns to the service.

Step 240: Service returns success to the controller.

Step 241: Controller adds a success message indicating that lecturers have been removed.

SUB-BRANCH I2 — CLASS NOT FOUND

Step 242: If the class is not found, controller may treat this as an error internally, but the flow moves on to redirect.

FINALIZE

Step 243: Controller sets the current class ID in the session for redirect purposes.

Step 244: Controller redirects to the member arrangement page.

Step 245: Stop execution.

BRANCH J — NO lecturerIds PROVIDED

Step 246: If no lecturerIds are submitted, controller does not perform any remove operation.

Step 247: Controller sets the current class ID in the session if available.

Step 248: Controller redirects to the member arrangement page.

Step 249: Stop execution.

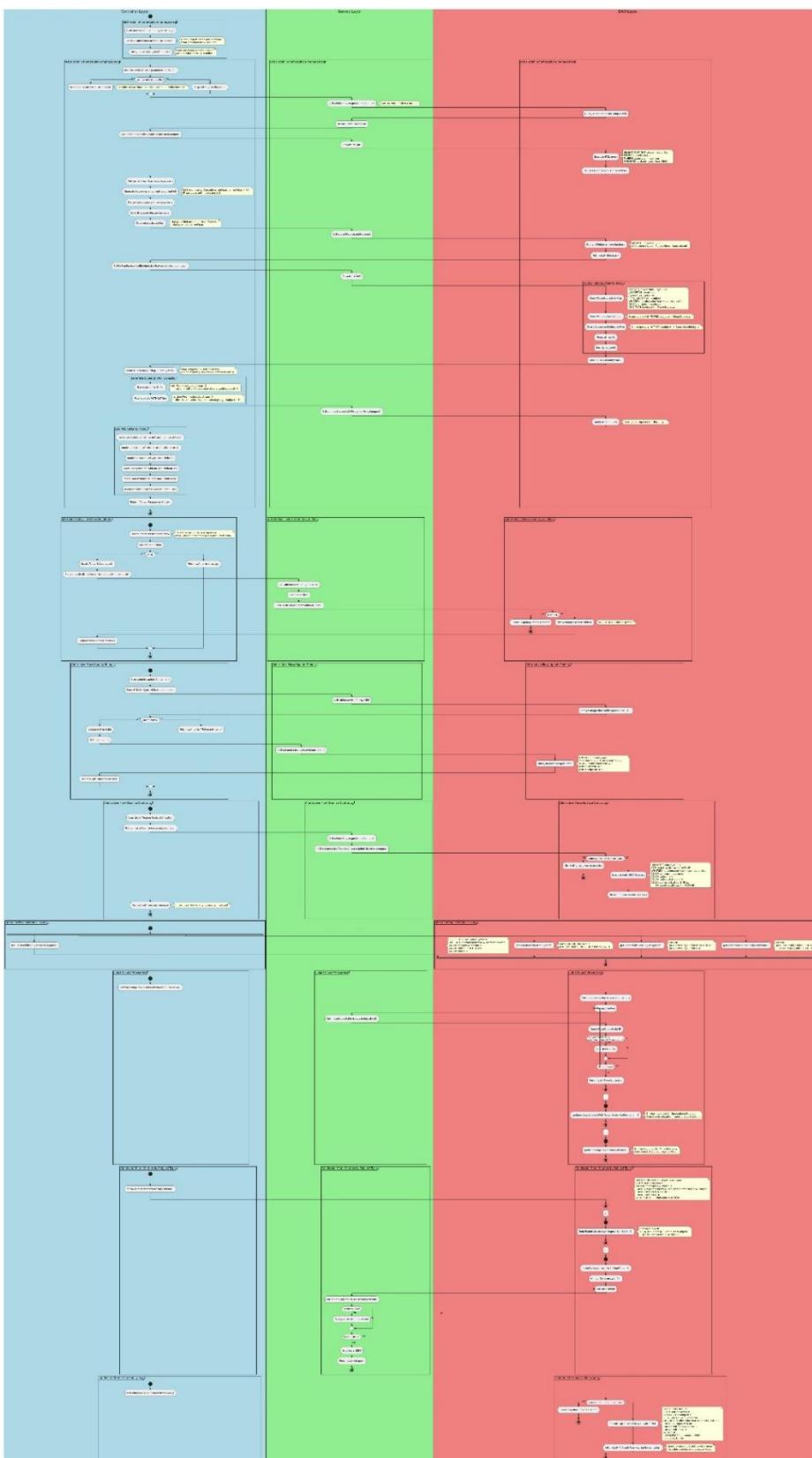


Diagram 67 Add Minor Subject Complete Flow – Activity Diagram

Step 1: User accesses the tuition management page via GET /admin-home/tuition-management.
Step 2: Controller retrieves admissionYear from the HTTP session (if previously stored).
Step 3: Controller delegates processing to the POST handler method listSubjects() (tuition-management POST), passing the retrieved admissionYear and the session.
Step 4: Stop execution of the GET handler.

MAIN FLOW — POST /admin-home/tuition-management (LIST VIEW)

Step 5: Controller receives a POST request to /admin-home/tuition-management with an optional admissionYear parameter.
Step 6: Controller checks whether admissionYear was provided in the request.

BRANCH A — admissionYear PROVIDED

Step 7: Controller stores the provided admissionYear into the HTTP session.

BRANCH B — admissionYear NOT PROVIDED

Step 8: Controller keeps the existing admissionYear value from the session (if any) unchanged.

DETERMINE CAMPUS AND AVAILABLE YEARS

Step 9: Controller calls the admins service to retrieve the current admin's campus.
Step 10: Admins service calls the DAO to query admin records and obtain campus information.
Step 11: DAO returns the corresponding campus entity to the service.
Step 12: Service returns the campus object to the controller.

Step 13: Controller calls the tuition service to find all admission years that have tuition data for this campus.

Step 14: Tuition service forwards the request to the DAO.
Step 15: DAO executes a query to retrieve all distinct admission years for tuition entries belonging to the given campus, ordered from newest to oldest.
Step 16: DAO returns the list of admission years to the service.
Step 17: Service returns the list of admission years to the controller.

Step 18: Controller gets the current calendar year from the system date.
Step 19: Controller generates a list of "future" years from the current year up to current year + 5.
Step 20: Controller filters out any future years that are already present in the admission-year list.
Step 21: Controller merges the remaining future years into the existing admission-year list.
Step 22: Controller sorts the combined list of years in descending order.

Step 23: Controller determines the selectedYear:

- If a valid admissionYear was provided or stored in session, use that;
 - Otherwise, use the current year as selectedYear.
-

LOAD SUBJECTS AND TUITION FOR SELECTED YEAR

Step 24: Controller calls the subject service to retrieve all subjects.
Step 25: Subject service calls the DAO to query all subject entities in the system (all types):

major, minor, specialized, etc.).

Step 26: DAO returns the complete list of subjects to the service.

Step 27: Service returns the list of subjects to the controller.

Step 28: Controller calls the tuition service to find all tuition entries for the selectedYear and campus.

Step 29: Tuition service forwards this request to the DAO.

Step 30: DAO queries tuition records for major subjects for the given admission year and campus, including related entities (such as campus and creator) and the associated subject.

Step 31: DAO queries tuition records for minor subjects for the same admission year and campus.

Step 32: DAO queries tuition records for specialized subjects for the same admission year and campus.

Step 33: DAO merges the results from the major, minor, and specialized queries into a single list of tuition entries.

Step 34: DAO sorts the merged tuition list by subject identifier.

Step 35: DAO returns the sorted tuition list to the service.

Step 36: Service returns the list of tuition entries to the controller.

BUILD TUITION MAP AND SPLIT SUBJECTS

Step 37: Controller creates a map structure that associates each subjectId with its corresponding tuition entry (TuitionByYear).

Step 38: Controller fills the tuition map by iterating over the tuition list and mapping each subject to its tuition record.

Step 39: Controller creates a list of subjects **with** tuition fee defined for the selected year and campus:

- Include subjects whose IDs are present as keys in the tuition map.

Step 40: Controller creates a list of subjects **without** tuition fee defined:

- Include subjects whose IDs are not present in the tuition map.
-

LOAD CAMPUSES FOR DROPODOWN AND PREPARE MODEL

Step 41: Controller calls the campus service to retrieve campuses used in the tuition context (for dropdown or exception lists).

Step 42: Campus service calls the DAO to query campus records.

Step 43: DAO returns the set of campuses to the service.

Step 44: Service returns the list of campuses to the controller.

Step 45: Controller adds the admission-year list to the model as admissionYears.

Step 46: Controller adds the selectedYear to the model.

Step 47: Controller adds the list of subjects with tuition fee to the model (e.g., withFee).

Step 48: Controller adds the list of subjects without tuition fee to the model (e.g., withoutFee).

Step 49: Controller adds the tuition map (subject → tuition entry) to the model.

Step 50: Controller adds the campus list to the model (e.g., Campuses).

- Step 51: Controller returns the "TuitionManagement" view with the prepared model.
Step 52: Stop execution of the POST listing flow.
-

ALTERNATIVE FLOW — CREATE NEW TUITION ENTRY

- Step 53: User submits a "create tuition" form to the appropriate POST endpoint for creating tuition (separate from the listing endpoint).
Step 54: Controller validates the submitted input data (admission year, subject, campus, tuition amounts, etc.).
Step 55: Controller checks whether the input data is valid.

BRANCH C — INPUT VALID

- Step 56: Controller creates a new TuitionByYear object in memory.
Step 57: Controller sets the composite identifier of the tuition record (including admission year, subject ID, and campus ID).
Step 58: Controller calls the admins service to obtain the current admin as the creator.
Step 59: Service returns the admin entity to the controller.
Step 60: Controller sets the creator field of the TuitionByYear object.
Step 61: Controller calls the tuition service to create and persist the new tuition entry.
Step 62: Tuition service validates that the composite identifier is present.
Step 63: If the composite identifier is null, the tuition service throws an exception indicating invalid data and stops processing.
Step 64: If the identifier is valid, tuition service calls the DAO to persist the new tuition record.
Step 65: DAO inserts the tuition record into the tuition table.
Step 66: DAO returns control to the service.
Step 67: Service returns success to the controller.
Step 68: Controller redirects back (for example to the tuition management page) with a success message indicating that tuition has been created.
Step 69: Stop execution of the create-tuition flow.

BRANCH D — INPUT INVALID

- Step 70: Controller prepares error messages describing validation problems.
Step 71: Controller returns to the tuition form or tuition management page with the error information.
Step 72: Stop execution of the create-tuition flow.
-

ALTERNATIVE FLOW — UPDATE EXISTING TUITION ENTRY

- Step 73: User submits an "update tuition" form to the corresponding POST endpoint.
Step 74: Controller extracts the composite TuitionByYear identifier from the request parameters (admission year, subject ID, campus ID).
Step 75: Controller calls the tuition service to find the tuition entry by its ID.
Step 76: Tuition service calls the DAO to load the tuition record by its identifier.

Step 77: DAO retrieves the TuitionByYear record (if it exists) and returns it to the service.
Step 78: Service returns the tuition entity to the controller.

Step 79: Controller checks whether the tuition entry exists.

BRANCH E — TUITION ENTRY EXISTS

Step 80: Controller updates the tuition fields (standard tuition, re-study tuition, and any other editable fields) with values from the form.

Step 81: Controller sets modifier information (e.g., who updated and when).

Step 82: Controller calls the tuition service to perform the update.

Step 83: Tuition service calls the DAO to merge the modified tuition entity.

Step 84: DAO updates the corresponding record in the tuition table in the database.

Step 85: DAO returns to the service.

Step 86: Service returns success to the controller.

Step 87: Controller redirects with a success message indicating that the tuition entry has been updated.

Step 88: Stop execution of the update-tuition flow.

BRANCH F — TUITION ENTRY NOT FOUND

Step 89: Controller prepares an error message indicating that the requested tuition record was not found.

Step 90: Controller returns to the previous page or form with the error message.

Step 91: Stop execution of the update-tuition flow.

ALTERNATIVE FLOW — FINALIZE CONTRACTS

Step 92: User clicks a “Finalize Contracts” button on the tuition management interface.

Step 93: Controller obtains the admissionYear value from the session or request parameter.

Step 94: Controller calls the admins service to get the current admin’s campus.

Step 95: Service returns the campus entity to the controller.

Step 96: Controller calls the tuition service to finalize contracts for the given admissionYear and campus.

Step 97: Tuition service checks whether admissionYear or campus is null.

BRANCH G — INVALID PARAMETERS

Step 98: If admissionYear or campus is null, tuition service throws an exception indicating invalid input.

Step 99: Flow for finalization stops with an error.

BRANCH H — VALID PARAMETERS

Step 100: Tuition service calls the DAO to perform a bulk update on the tuition records for the specified admissionYear and campus.

Step 101: DAO executes an update that sets the contract status (e.g., to ACTIVE) for records meeting the required conditions (such as having positive tuition and re-study tuition and not already active).

Step 102: DAO returns the number of affected rows (number of contracts finalized) to the

service.

Step 103: Service returns the count of finalized contracts to the controller.

Step 104: Controller redirects with a success message, informing the user how many contracts were finalized.

Step 105: Stop execution of the finalize-contracts flow.

3.10 Algorithm and Data Structure Design

Algorithm Design and Data Structures - Timetable Function

The timetable management system is designed based on a hierarchical architecture with optimized data structures and algorithms to ensure accuracy, efficiency, and scalability. The core data structure of the system uses the JOINED inheritance model in JPA, with the abstract class `Timetable` as the base class for three specific types of timetables: `MajorTimetable`, `MinorTimetable`, and `SpecializedTimetable`. Each timetable record is uniquely identified by `TimetableID` and contains important attributes including `RoomID` (reference to the classroom), `SlotID` (reference to the class shift), `DayOfTheWeek` (day of the week), `WeekOfYear` (week of the year), and `Year` (school year). This structure allows the system to manage the school calendar in multiple dimensions: by class, by room, by lecturer, by student, and by time, creating a complex but organized network of relationships. The conflict detection algorithm is the heart of the system, ensuring that there are no overlaps in time and space. When a class is requested to be scheduled in a particular slot, the system performs a series of sequential and comprehensive checks. First, the system checks whether the class is already scheduled in the specified slot and day using the `getTimetableByClassSlotDayWeek` method, using a JPQL query with conditions combining `classId`, `slotId`, `dayOfWeek`, `weekOfYear`, and `year`. If the class is already scheduled, the process stops and returns a list of empty rooms. Next, the system retrieves a list of all lecturers assigned to that class through the Many-to-Many relationship between `MajorLecturers` and `MajorClasses`. For each lecturer, the system executes the `lecturerHasConflict` algorithm, a complex boolean function using UNION ALL to check for conflicts across all three types of timetables (Major, Minor, Specialized). This algorithm queries three timetable tables simultaneously to find out if a lecturer is teaching another class in the same slot, same day, same week, and same year. If any lecturer has a conflict, the entire process is rejected to ensure consistency. Similarly, the system applies the `studentHasConflict` algorithm to check for conflicts for all students in the class. This algorithm also uses UNION ALL to scan through the three types of timetables, checking if any students have registered for another class at the same time. This ensures that no students have duplicate schedules, an important requirement for ensuring the quality of education. After passing all the conflict checks for lecturers and students, the system then searches for available classrooms through the `getAvailableRoomsInCampus` algorithm. This algorithm queries all rooms on the same campus, then eliminates rooms that have been booked for the corresponding slot, day, week, and year. The result is a list of available rooms, sorted by priority for the user to choose from.

A special feature of the system is the `SlotOfTheDayThatCanBeSuccessfullyArranged` algorithm, which creates a two-dimensional matrix (7 days × number of slots) to suggest classrooms for each time slot. This algorithm uses a nested loop to iterate through all days of the week (from Monday to Sunday) and all available slots in the system. For each (day, slot) pair, the system calls `getAvailableRoomsForSlot` to check availability. If there is an available room, the matrix stores the `RoomID` of the first room in the list; otherwise, a null value is assigned. This matrix is passed to the user interface for visual display, making it easy for the scheduler to identify available slots and slots blocked due to conflicts. The batch scheduling algorithm is implemented in the controller

layer, allowing users to schedule multiple slots at once for a class. This algorithm uses a Set<String> data structure called usedRoomSlotWeek to keep track of (roomId, slotId, week, year) pairs that have been used in the same scheduling session, preventing duplicate bookings in the same week. The outer loop iterates over the days of the week, while the inner loop iterates over the selected slots. For each (day, slot) pair, the system checks the validity of the input data, verifies that the slot is not in the past, and checks whether the room has been used via conflictKey. If all conditions are satisfied, a new Timetable object is created with a random UUID as the identifier, and saved to the database via the SaveMajorTimetable method. This algorithm ensures the atomicity of the transaction, meaning that if one slot cannot be scheduled, the other slots can still be processed independently. The legacy data structure uses JPA's JOINED strategy, where the Timetable table contains common attributes, while the MajorTimetable, MinorTimetable, and SpecializedTimetable tables contain unique attributes and are linked to the parent table through PrimaryKeyJoinColumn. This model allows the system to take advantage of Java polymorphism, with abstract methods such as getClassId(), getClassName(), getClassType(), and getDetailUrl() implemented differently in each subclass. This makes the code flexible and easy to maintain, as the common processing logic can be written once in the base class, while the specific logic is implemented in the subclasses. The query and filtering algorithms make extensive use of JPQL (Java Persistence Query Language) with optimization techniques such as JOIN FETCH to avoid the N+1 query problem, a common problem in ORMs. When retrieving the timetable list, the system uses JOIN FETCH to preload relations such as room, slot, classEntity, and creator in a single query, instead of having to execute multiple individual queries. This significantly reduces the number of round-trips to the database and improves overall performance. Queries are also optimized with indexes on frequently used columns in the WHERE condition, such as weekOfYear, year, dayOfWeek, and slotId. Statistics and reporting algorithms are implemented through methods such as getDashboardSummary, getTop5BusyLecturers, getSlotsPerDayOfWeek, and getTop5UsedRooms. These algorithms use SQL aggregation functions such as COUNT, GROUP BY, and ORDER BY to calculate important metrics. For example, getSlotsPerDayOfWeek uses a long[7] array to store the number of slots assigned for each day of the week, with the array index corresponding to the order of the DaysOfWeek enum. This algorithm groups the timetable records by dayOfWeek, counts the number for each group, and then maps the results into the results array using day.ordinal() to determine the exact position. Similarly, getTop5BusyLecturers uses GROUP BY and ORDER BY COUNT(t) DESC to determine the five busiest instructors, with the results limited by setMaxResults(5).

The Slots data structure is designed to be simple but effective, with SlotID as the primary key, SlotName for description, and StartTime and EndTime to define the duration of each class. This structure allows the system to flexibly manage different class shifts, from morning to evening shifts, with possible different durations. The time conflict checking algorithm uses direct comparisons between LocalTime objects, ensuring accuracy and efficiency. Finally, the system uses an automatic notification mechanism via the sendScheduleNotification method, which is called after the class schedule is created or updated. This algorithm retrieves the list of all students in the class through the Students_MajorClasses relationship, and then sends email notifications to each student using the built-in email service. This process is executed asynchronously so as not to slow down the system's response, and includes error handling to ensure that an error in sending an email to one student does not affect the sending of emails to other students.

Overall, the algorithm design and data structure of the timetable system represent a delicate balance between complexity and efficiency, between flexibility and consistency. The algorithms are optimized to handle large amounts of data in real time, while still ensuring data accuracy and integrity. The legacy structure allows the system to be easily expanded to support new types of classes in the future, while conflict checking algorithms ensure that the system always maintains a reasonable and conflict-free schedule.

Algorithm Design and Data Structures - User Assignment to Classes Function

The user assignment to classes functionality represents a sophisticated multi-entity relationship management system that employs composite key structures, inheritance hierarchies, and complex validation algorithms to ensure data integrity and business rule compliance. At the core of this system lies the abstract entity `Students_Classes`, which serves as the base class for three concrete implementations: `Students_MajorClasses`, `Students_MinorClasses`, and `Students_SpecializedClasses`. This inheritance pattern, implemented using JPA's JOINED inheritance strategy, allows the system to maintain a unified interface while supporting distinct behaviors for different class types. The composite primary key structure, encapsulated in the `StudentsClassesId` embeddable class, consists of two components: `studentId` and `classId`, forming a unique identifier that prevents duplicate enrollments and ensures referential integrity through the `@EmbeddedId` annotation. This design pattern eliminates the need for surrogate keys and provides natural key semantics that directly reflect the business domain.

The assignment algorithm for students follows a multi-stage validation and processing pipeline that begins with prerequisite checking and ends with financial transaction processing. When a staff member initiates the assignment of students to a class, the system first retrieves the target class entity and validates its existence. The algorithm then iterates through each student ID in the input list, performing a series of validation checks. The first validation step involves checking whether the student already exists in the class through the `existsByStudentAndClass` method, which executes a COUNT query on the `Students_MajorClasses` table using the composite key. If a match is found, the algorithm skips that student and logs an error message, preventing duplicate enrollments. The second validation step determines whether the student is already registered for the subject through either the retake subjects system (`RetakeSubjects`) or the temporary retake subjects system (`TemporaryRetakeSubjects`). This check is crucial for financial processing, as students who are already in the retake system should not be charged additional fees.

The financial processing algorithm represents one of the most complex aspects of the assignment system, incorporating scholarship discount calculations and account balance verification. For new enrollments (students not in retake or temporary systems), the algorithm first retrieves the re-study fee for the subject using the `getReStudyFee` method. This fee is then subject to scholarship discount calculations if the student has an active scholarship. The algorithm queries the `Students_Scholarships` table to find active scholarships for the student based on their admission

year, then retrieves the corresponding `ScholarshipByYear` record to obtain the discount percentage. The final fee calculation applies the discount formula: $\text{finalFee} = \text{originalFee} * (100 - \text{discountPercentage}) / 100$, with rounding to two decimal places to ensure currency precision. Before proceeding with enrollment, the algorithm verifies that the student's account balance is sufficient to cover the calculated fee through the `hasSufficientBalance` method, which queries the `AccountBalances` table. If the balance is insufficient, the enrollment is rejected and an error message is generated.

The enrollment creation process utilizes the entity manager's persist operation to create a new `Students_MajorClasses` instance. The algorithm constructs the composite key using the `StudentsClassesId` constructor, sets the student and class references, assigns the staff member who performed the action through the `addedBy` field, and timestamps the creation using `LocalDateTime.now()`. This timestamp serves multiple purposes: audit trail maintenance, notification triggering, and historical record keeping. After successful enrollment, the algorithm performs financial deduction for new retake students by calling `deductAndLogPayment`, which atomically decrements the student's account balance and creates a corresponding entry in the `FinancialHistories` table. This two-phase operation ensures transactional consistency, as both the balance update and history logging must succeed together or fail together.

The cleanup algorithm handles the removal of records from retake and temporary retake systems after successful enrollment. If a student was found in the `RetakeSubjects` table, the algorithm calls `deleteByStudentAndSubject` to remove the retake registration record. Similarly, if the student was in the `TemporaryRetakeSubjects` table, the corresponding temporary record is deleted. This cleanup process ensures that the retake systems remain synchronized with actual class enrollments, preventing orphaned records and maintaining data consistency across multiple tables. The algorithm uses separate service methods for each cleanup operation, allowing for independent error handling and logging.

The removal algorithm for students follows a similar pattern but includes additional logic for handling temporary retake subject creation. When a student is removed from a class, the algorithm first verifies the existence of the enrollment relationship using `existsByStudentAndClass`. If the relationship exists, it is removed through the `removeStudentFromClass` method, which executes a JPQL query to find the specific `Students_MajorClasses` record and then calls `entityManager.remove()`. After successful removal, the algorithm automatically creates a temporary retake subject record for the student, which allows them to be re-enrolled in the same subject without additional payment. This temporary record includes a descriptive reason field that indicates the student was removed from a specific class, providing audit trail information for administrative review.

The lecturer assignment algorithm follows a simpler pattern but maintains the same composite key structure through the `LecturersClassesId` embeddable class. The `addLecturersToClass` method accepts a list of lecturer IDs and iterates through each ID, creating a new `MajorLecturers_MajorClasses` instance for each lecturer. The algorithm validates the existence

of each lecturer through the `getLecturerById` method before creating the assignment relationship. Each assignment record includes the composite key, references to both the lecturer and class entities, a creation timestamp, and the staff member who performed the assignment. The batch processing nature of this algorithm allows multiple lecturers to be assigned to a class in a single operation, improving efficiency for administrative tasks.

The lecturer removal algorithm uses a similar iteration pattern but performs deletion operations instead of creation. For each lecturer ID in the removal list, the algorithm constructs the composite key and uses `entityManager.find()` to locate the corresponding `MajorLecturers_MajorClasses` record. If the record exists, it is removed using `entityManager.remove()`. The algorithm includes null checks and empty list validation to prevent unnecessary database operations and ensure robust error handling.

The query algorithms for retrieving assignment information employ sophisticated JPQL queries with subqueries and NOT IN clauses to filter students and lecturers based on complex criteria. The `getStudentsNotInClassAndSubject` method demonstrates this complexity by combining multiple conditions: it filters students by major and campus, ensures they have the subject in their required subjects list through a subquery, and excludes students who are already enrolled in the class or any other class for the same subject. This multi-condition filtering ensures that only eligible students are presented for assignment, reducing administrative errors and maintaining academic integrity.

The data structure design utilizes lazy loading strategies through `FetchType.LAZY` annotations on relationship mappings, which improves performance by deferring the loading of related entities until they are explicitly accessed. This is particularly important for the `Students_Classes` entity, which maintains relationships with both `Students` and `Classes` entities. The lazy loading strategy prevents the N+1 query problem that would occur if all related entities were eagerly loaded, especially when processing large lists of assignments. The `@OnDelete(action = OnDeleteAction.CASCADE)` annotations ensure referential integrity by automatically removing assignment records when either the student or class is deleted, preventing orphaned records in the database.

The notification system integrated into the assignment process uses a separate query mechanism to retrieve class assignment notifications for both students and lecturers. The `getClassNotificationsForStudent` and `getClassNotificationsForLecturer` methods execute JPQL queries that concatenate notification messages using SQL's CONCAT function, combining class names, subject names, and creation timestamps into human-readable notification strings. These queries filter records by a `notificationType` field set to 'NOTIFICATION_002', allowing the system to distinguish between different types of notifications and provide targeted messaging to users.

The statistical and reporting algorithms provide administrative insights through aggregation queries that count and rank assignments. The `countLecturersTeachingAtLeastOneClass`

method uses COUNT with DISTINCT to determine the number of unique lecturers who have at least one class assignment, filtering by campus and major management scope. The `getTop5LecturersByClassCount` method employs GROUP BY and ORDER BY clauses to rank lecturers by their number of class assignments, providing administrators with visibility into workload distribution. Similarly, the `countMajorClassesWithoutAnyLecturer` method uses a NOT IN subquery to identify classes that lack lecturer assignments, enabling proactive resource allocation.

The data structure's support for polymorphism through abstract methods allows the system to handle different class types uniformly while maintaining type-specific behaviors. The abstract `Students_Classes` class defines methods such as `getSubjectName()` and `getSubjectType()`, which are implemented differently in each subclass. This design enables code that works with the base class to automatically adapt to different class types without requiring instanceof checks or type casting, following the Open/Closed Principle of object-oriented design.

The transactional management of the assignment operations ensures atomicity through the `@Transactional` annotation at the service and DAO layers. This annotation ensures that all database operations within a single assignment transaction either complete successfully or are rolled back entirely, maintaining data consistency even in the event of errors. The transaction boundaries are carefully managed to include both the enrollment creation and financial deduction operations, ensuring that students are not enrolled without proper payment processing, and vice versa.

The batch processing capabilities of the assignment algorithms allow multiple students or lecturers to be processed in a single operation, improving efficiency and reducing database round-trips. The algorithms accumulate error messages during processing and present them collectively to the user, rather than failing on the first error. This approach provides better user experience by allowing administrators to see all validation issues at once and make corrections accordingly. The success and error message handling uses Spring's `RedirectAttributes` for flash attributes, which persist messages across redirects and are automatically cleared after being displayed to the user.

In summary, the user assignment to classes functionality represents a comprehensive system that combines sophisticated data structures, complex validation algorithms, financial processing logic, and robust error handling to provide a reliable and efficient mechanism for managing academic enrollments. The use of composite keys, inheritance hierarchies, lazy loading, and transactional management ensures both performance and data integrity, while the polymorphic design allows for extensibility and maintainability of the codebase.

Algorithm Design and Data Structures - Tuition Fees by Year Function

The tuition fees by year management system represents a sophisticated financial data management solution that employs a composite key structure, contract status tracking, and multi-dimensional query algorithms to handle annual tuition fee configurations across different campuses, subjects, and admission years. At the core of this system lies the `TuitionByYear` entity, which utilizes a composite primary key encapsulated in the `TuitionByYearId` embeddable class. This composite key consists of three components: `subjectId`, `admissionYear`, and `campusId`, forming a unique identifier that ensures one tuition fee record per subject per year per campus. This design pattern eliminates the need for surrogate keys and provides natural key semantics that directly reflect the business domain, where tuition fees are inherently tied to specific subjects, admission cohorts, and campus locations.

The data structure design incorporates two distinct fee types within a single entity: `tuition` for regular course fees and `reStudyTuition` for retake or re-study fees. Both fields are nullable `Double` types, allowing the system to represent subjects that may not have fees configured yet or subjects that have different fee structures. The `ContractStatus` enumeration field tracks the lifecycle state of tuition fee contracts, with values such as `DRAFT` for preliminary configurations and `ACTIVE` for finalized contracts that cannot be modified. This status-based workflow ensures data integrity by preventing modifications to finalized contracts while allowing administrators to prepare and review fee structures before activation.

The composite key implementation in `TuitionByYearId` follows JPA best practices with proper `equals()` and `hashCode()` methods that consider all three key components. The `@EmbeddedId` annotation in the `TuitionByYear` entity allows the composite key to be used seamlessly with JPA's entity manager operations, while the `@MapsId` annotations on the `subject` and `campus` relationship mappings ensure that the foreign key relationships are properly synchronized with the composite key components. The `admissionYear` field is marked as `insertable = false, updatable = false` because it is part of the composite key and should only be set during entity creation, preventing accidental modifications that could break referential integrity.

The tuition fee retrieval algorithm employs sophisticated JPQL queries with multiple filtering conditions to support various administrative use cases. The `tuitionFeesByCampus` method executes a straightforward query that filters records by campus ID and admission year, returning all tuition fee records for a specific campus and year combination. This query serves as the foundation for campus-specific fee management interfaces, allowing administrators to view and modify fees for their respective campuses. The query uses lazy loading for related entities through `FetchType.LAZY` annotations, optimizing performance by deferring the loading of `Subjects` and `Campuses` entities until they are explicitly accessed.

The filtering algorithms for identifying subjects with and without fees demonstrate the system's ability to handle incomplete data configurations. The `getTutionsWithFeeByYearAndCampus` method uses a JPQL query with `IS NOT NULL` and `> 0` conditions to identify subjects that have valid tuition fees configured. Conversely, the `getTutionsWithoutFeeByYear` method uses `IS

NULL OR <= 0` conditions to identify subjects that lack fee configurations. These complementary queries enable administrators to quickly identify which subjects require fee configuration and which subjects are ready for contract finalization. The algorithms are designed to handle edge cases such as zero fees, which are treated as missing configurations to prevent accidental free course assignments.

The subject type filtering algorithms utilize JPA's `TREAT` function and `TYPE` discriminator to handle the polymorphic `Subjects` entity hierarchy. The `findAllAdmissionYearsWithMajorTuition` method demonstrates this by using `TREAT(t.subject AS MajorSubjects).major = :major` to filter tuition records for major subjects belonging to a specific major. This type-safe casting approach allows the system to query across the inheritance hierarchy while maintaining type safety and avoiding runtime errors. Similarly, the `findAllAdmissionYearsWithSpecializedTuition` method uses `TREAT` to filter specialized subjects by their associated major through the specialization relationship.

The curriculum-based subject retrieval algorithms combine tuition fee data with curriculum and major filtering to provide targeted subject lists for academic planning. The `getMajorSubjectsWithTuitionByYearAndCurriculum` method first retrieves all tuition records with fees for the specified year and campus using `getTuititionsWithFeeByYearAndCampus`. It then iterates through these records, loading each subject entity and filtering by major and curriculum matches. This two-phase approach allows the algorithm to leverage the optimized database query for tuition retrieval while performing in-memory filtering for curriculum and major matching, which may involve complex relationship traversals that are more efficiently handled in application code.

The contract finalization algorithm represents a critical business process that transitions tuition fee configurations from draft to active status. The `finalizeContracts` method executes a bulk update query using JPQL's `UPDATE` statement to change the `contractStatus` of all qualifying records to `ACTIVE`. The query includes multiple validation conditions: it only updates records where both `tuition > 0` and `reStudyTuition > 0`, ensuring that incomplete fee configurations cannot be finalized. Additionally, it includes a condition `(t.contractStatus IS NULL OR t.contractStatus != :status)` to prevent redundant updates and ensure idempotency. This bulk update approach is more efficient than individual entity updates when processing large numbers of records, reducing database round-trips and transaction overhead.

The tuition fee update algorithm in the controller layer implements a comprehensive validation and processing pipeline. The `updateTuition` method first validates that the admission year is not in the past, preventing modifications to historical fee structures that may have already been used for financial calculations. It then iterates through all subjects in the system, extracting tuition fee values from the request parameters using a naming convention `tuitionFee_<subjectId>`. For each subject with a provided fee value, the algorithm parses the string to a `Double`, validates that it is non-negative, and constructs the composite key. It then checks if an existing `TuitionByYear` record exists for this key combination. If the record exists and has an `ACTIVE` contract status, the update is rejected to maintain data integrity. If the record exists with a `DRAFT`

status, it is updated using `entityManager.merge()`. If no record exists, a new entity is created with `DRAFT` status and persisted using `entityManager.persist()`.

The admission year management algorithm provides functionality for discovering and managing available admission years across the system. The `findAllAdmissionYears` method executes a `DISTINCT` query to retrieve all unique admission years for a given campus, ordered in descending order to present the most recent years first. This query serves as the foundation for year selection interfaces, allowing administrators to choose which year's fees they wish to view or modify. The controller layer extends this functionality by generating future years (current year through current year + 5) that may not yet have tuition records, ensuring that administrators can proactively configure fees for upcoming admission cohorts.

The multi-type subject retrieval algorithm in `findByAdmissionYear` demonstrates a sophisticated approach to handling polymorphic entity hierarchies. The method executes three separate queries, one for each subject type (Major, Minor, Specialized), using JPA's `TYPE` discriminator to filter by entity type. Each query uses `JOIN FETCH` to eagerly load related entities ('campus', 'creator', 'subject') to avoid N+1 query problems. The results from all three queries are combined into a single list, which is then sorted by subject ID using Java's `Comparator` and `Collectors.toList()`. This approach ensures consistent ordering across different subject types while maintaining type safety and query optimization.

The re-study fee management algorithms provide parallel functionality to regular tuition fee management, with separate methods for identifying subjects with and without re-study fees configured. The `getTuitionsWithReStudyFeeByYear` and `getTuitionsWithoutReStudyFeeByYear` methods mirror the structure of their tuition fee counterparts but filter on the `reStudyTuition` field instead. This separation allows administrators to manage regular and re-study fees independently, recognizing that these fee types may have different configuration timelines and approval processes. The algorithms support the business requirement that both fee types must be configured before contract finalization can occur.

The reference query algorithm for student-facing tuition information uses advanced JPQL features to provide optimized data retrieval. The `tuitionReferenceForStudentsByCampus` method employs `DISTINCT` to eliminate duplicate records, `JOIN FETCH` to eagerly load subject and acceptor entities, and `COALESCE` in the `ORDER BY` clause to handle null semester values by treating them as 999 (placing them at the end of the sorted list). This query is designed for read-heavy operations where students need to view tuition information, prioritizing query performance and data completeness over write optimization.

The validation algorithms in the contract finalization process ensure data completeness before allowing contracts to be activated. The controller's `finalizeContracts` method first validates that the admission year is not in the past and that tuition records exist for the selected year. It then iterates through all tuition records, checking that both `tuition` and `reStudyTuition` fields are not

null and greater than zero. Any missing or invalid fees are collected into an error list and presented to the administrator, preventing partial contract finalization that could lead to inconsistent fee structures. This validation step is critical for maintaining data integrity and ensuring that all subjects have complete fee configurations before contracts become active.

The data structure's support for audit trails through the `creator` field allows the system to track which administrator created or last modified each tuition fee record. This field maintains a many-to-one relationship with the `Admins` entity, enabling accountability and historical tracking of fee configuration changes. The `@OnDelete(action = OnDeleteAction.CASCADE)` annotation ensures that if an admin is deleted, the associated tuition records are handled appropriately, though in practice, admin deletion is typically prevented by business rules.

The lazy loading strategy employed throughout the entity relationships optimizes memory usage and query performance. By using `FetchType.LAZY` on the `subject`, `campus`, and `creator` relationships, the system defers loading these entities until they are explicitly accessed. This is particularly important for list views where hundreds of tuition records may be displayed, as eagerly loading all related entities would result in excessive database queries and memory consumption. The system uses `JOIN FETCH` selectively in queries where related entities are known to be needed, providing explicit control over when eager loading occurs.

The error handling and validation algorithms throughout the system use defensive programming techniques to ensure robust operation. Null checks are performed on critical parameters before executing queries, with `IllegalArgumentException` thrown for invalid inputs. The parsing of tuition fee values from request parameters includes try-catch blocks to handle `NumberFormatException`, allowing the system to continue processing other subjects even if one has invalid data. Error messages are collected and presented collectively to administrators, providing comprehensive feedback about all validation issues in a single operation rather than failing on the first error.

The session management algorithm in the controller layer maintains user context across multiple requests by storing the selected admission year in the HTTP session. This allows administrators to navigate between different views (tuition management, contract finalization, re-study fee management) while maintaining their year selection, improving user experience by reducing the need to repeatedly select the same year. The session attribute is updated whenever a new year is selected and retrieved when displaying pages that require year context.

In summary, the tuition fees by year management system represents a comprehensive solution that combines sophisticated data structures, complex query algorithms, contract lifecycle management, and robust validation to provide a reliable and efficient mechanism for managing annual tuition fee configurations. The use of composite keys, polymorphic entity handling, bulk update operations, and defensive programming techniques ensures both performance and data integrity, while the separation of regular and re-study fee management allows for flexible

administrative workflows that accommodate different business requirements and approval processes.

Algorithm Design and Data Structures - Parent Assignment to Students Function

The parent assignment to students functionality represents a sophisticated relationship management system that employs composite key structures, email-based parent account discovery, automatic account creation, password generation, and comprehensive validation algorithms to establish and maintain parent-student relationships. At the core of this system lies the `Student_ParentAccounts` entity, which utilizes a composite primary key encapsulated in the `StudentParentAccountsId` embeddable class. This composite key consists of two components: `studentId` and `parentId`, forming a unique identifier that ensures one relationship record per student-parent pair. This design pattern prevents duplicate relationships while allowing a single parent to be linked to multiple students and a single student to have multiple parents, reflecting the real-world scenario where families may have multiple children in the system and students may have multiple guardians.

The data structure design incorporates relationship metadata through the `RelationshipToStudent` enumeration field, which categorizes the nature of the parent-student relationship (e.g., MOTHER, FATHER, GUARDIAN). This enumeration provides semantic meaning to the relationship and enables the system to support different types of family structures and guardianship arrangements. The `supportPhoneNumber` field allows for a separate contact number specifically for parent-student communication, distinct from the parent's primary phone number stored in the `ParentAccounts` entity. This separation enables parents to maintain different contact numbers for different purposes, such as a dedicated number for school-related communications.

The `ParentAccounts` entity extends the abstract `Persons` entity through JPA's JOINED inheritance strategy, inheriting all person-related attributes while adding parent-specific fields such as `createdDate` and `creator`. This inheritance pattern allows the system to leverage the common person data structure while maintaining type-specific behaviors. The `@PrimaryKeyJoinColumn` annotation ensures that the parent account's primary key is the same as the person's ID, maintaining referential integrity and enabling polymorphic queries across the person hierarchy.

The parent account creation algorithm implements an intelligent email-based discovery and creation mechanism that minimizes duplicate accounts and simplifies the assignment process. When a staff member assigns a parent to a student, the `createParentLink` method first checks if a parent account already exists with the provided email address using the `findByEmail` method, which executes a case-insensitive query on the email field. If an existing parent account is found, the algorithm reuses that account, preventing duplicate parent records for the same email address. This approach recognizes that parents may have multiple children in the system and should not need separate accounts for each child.

If no existing parent account is found, the algorithm creates a new `ParentAccounts` entity with minimal required information. The email address is set and normalized to lowercase to ensure consistency, and a unique parent ID is generated using the `generateUniqueParentId` method. This ID generation algorithm follows a structured format: `PAR` prefix, followed by a two-digit year code, a four-digit date code (month and day), and a three-digit random number. The algorithm uses a do-while loop with `SecureRandom` to ensure uniqueness, checking against the `Persons` table to avoid collisions. This format provides both human readability (indicating creation date) and uniqueness guarantees.

The automatic password generation algorithm creates secure, randomly generated passwords for new parent accounts. The `generateRandomPassword` method enforces a minimum length of 8 characters and ensures password complexity by guaranteeing the presence of at least one uppercase letter, one lowercase letter, one digit, and one special character. The algorithm first adds one character from each required category to ensure complexity requirements are met, then fills the remaining length with random characters from the complete character set. Finally, it shuffles all characters using `Collections.shuffle()` with a `SecureRandom` instance to ensure randomness and prevent predictable patterns. This approach balances security requirements with usability, as the generated password is automatically sent to the parent via email.

The authentication record creation process automatically creates an `Authenticators` entity for each new parent account, establishing the credentials necessary for system access. The algorithm sets the `personId` to match the parent's ID, creates a bidirectional relationship with the parent entity, and stores the generated password. This automatic credential provisioning eliminates the need for manual account setup and ensures that parents receive login information immediately upon being assigned to a student.

The email notification system integrated into the parent assignment process provides automated communication at multiple stages of the relationship lifecycle. When a new parent account is created, the `sendParentCreationEmail` method constructs a `ParentEmailContext` object containing all relevant parent information and sends an email with the login credentials. This email includes the parent's account ID, generated password, and instructions for accessing the system. The email sending process is wrapped in a try-catch block to ensure that email failures do not prevent account creation, as email delivery is considered a non-critical operation that can be retried or handled separately.

The relationship link creation algorithm constructs a `Student_ParentAccounts` entity using the composite key, establishing the many-to-many relationship between students and parents. The algorithm sets the student and parent references using `@MapId` annotations to ensure synchronization with the composite key components, assigns the staff member who performed the action through the `addedBy` field, timestamps the creation using `LocalDateTime.now()`, and sets the relationship type and support phone number if provided. The `linkStudentToParent` method persists this relationship using `entityManager.persist()`, creating the database record that establishes the parent-student link.

The validation algorithms implement comprehensive data integrity checks before allowing parent assignments. The `validateParentLink` method performs multi-level validation: it first checks if any parent-related fields are provided using `isAnyFieldProvided`, ensuring that the operation has sufficient data to proceed. If email is provided, it validates the email format using a regex pattern `^([A-Za-z0-9+_.-]+@(.+)\$)` and checks for email uniqueness across all person types using `personsService.existsByEmail()`. This cross-type email validation prevents conflicts where a student, staff member, or admin might already be using the email address. The validation also checks phone number format using the pattern `^(\+?[\d]{10,15}\$)` and validates the relationship enumeration by attempting to parse it using `RelationshipToStudent.valueOf()`.

The email availability checking algorithm in the controller layer provides an additional validation step before attempting parent account creation. The `isParentEmailAvailable` method checks if an email address can be used for a parent account by verifying that it is not already associated with any other account type in the system. This pre-validation prevents unnecessary processing and provides immediate feedback to administrators, improving user experience and reducing system load.

The edit-or-create algorithm (`editOrCreateParentLink`) implements intelligent relationship management that handles both new assignments and updates to existing relationships. The algorithm first checks if an existing relationship link exists for the student. If an existing link is found and the email matches the new email, it updates the relationship metadata (relationship type and support phone number) using `editParentLink`. If the email differs, it removes the old link, checks if the old parent account should be deleted (if no other students are linked), and creates a new link with the new parent account. This approach ensures that relationship changes are handled gracefully while maintaining data integrity and preventing orphaned parent accounts.

The parent account cleanup algorithm (`deleteIfUnlinked`) implements automatic garbage collection for parent accounts that are no longer associated with any students. When a parent-student relationship is removed, the algorithm calls `countLinkedStudents` to check if the parent account has any remaining student links, excluding the student that was just unlinked. If the count is zero, indicating that the parent account is orphaned, the algorithm deletes the associated authenticator record and then deletes the parent account itself. This cleanup process prevents the accumulation of unused accounts in the database while ensuring that parent accounts with active relationships are preserved.

The student retrieval algorithm (`getStudentsByParentId`) enables parents to view all students linked to their account. The method executes a JPQL query that joins the `Student_ParentAccounts` table with the `Students` table, filtering by parent ID and ordering results by creation date in descending order. This query provides parents with a chronological view of their student relationships, with the most recently added students appearing first. The query uses lazy loading for the student entity, deferring the loading of related student data until it is explicitly accessed.

The link finding algorithm (`findLinkByStudentAndParent`) provides efficient lookup of specific relationship records using the composite key components. The method executes a JPQL query that filters by both student ID and parent ID, returning a single `Student_ParentAccounts` record if the relationship exists, or null if it does not. This method is used throughout the system to check for existing relationships before creating new ones, preventing duplicate links and enabling update operations.

The relationship enumeration validation algorithm ensures that only valid relationship types can be assigned. The controller's validation process attempts to parse the relationship string using `RelationshipToStudent.valueOf(relationship.toUpperCase())`, which throws an `IllegalArgumentException` if the relationship type is invalid. This validation is performed early in the request processing pipeline, before any database operations, to provide immediate feedback and prevent invalid data from entering the system.

The session management algorithm in the controller layer maintains student context across multiple requests by storing the student ID in the HTTP session. This allows staff members to navigate between different views (student edit form, parent assignment, etc.) while maintaining their student selection, improving user experience by reducing the need to repeatedly select the same student. The session attribute is set after each operation and retrieved when displaying pages that require student context.

The error handling and validation algorithms throughout the system use defensive programming techniques to ensure robust operation. Null checks are performed on critical parameters before executing queries, with appropriate error messages returned to the user interface. The parsing of relationship enums includes try-catch blocks to handle `IllegalArgumentException`, allowing the system to provide user-friendly error messages rather than exposing technical exceptions. Error messages are collected and presented through Spring's `RedirectAttributes` flash attributes, which persist messages across redirects and are automatically cleared after being displayed.

The data structure's support for audit trails through the `addedBy` and `createdAt` fields allows the system to track which staff member created each parent-student relationship and when it was established. This audit information is valuable for administrative oversight, troubleshooting, and historical record keeping. The `@OnDelete(action = OnDeleteAction.CASCADE)` annotations ensure referential integrity by automatically removing relationship records when either the student or parent is deleted, preventing orphaned links in the database.

The lazy loading strategy employed throughout the entity relationships optimizes memory usage and query performance. By using `FetchType.LAZY` on the `student`, `parent`, and `addedBy` relationships, the system defers loading these entities until they are explicitly accessed. This is particularly important for list views where many parent-student relationships may be displayed, as

eagerly loading all related entities would result in excessive database queries and memory consumption.

The polymorphic query support through the ‘Persons’ inheritance hierarchy allows the system to query parent accounts using person-based queries when needed. The ‘ParentAccounts’ entity can be treated as a ‘Persons’ entity in queries, enabling unified person management while maintaining type-specific behaviors. This polymorphism is leveraged in email and phone number validation, where the system checks for uniqueness across all person types to prevent conflicts.

In summary, the parent assignment to students functionality represents a comprehensive system that combines sophisticated data structures, intelligent account discovery and creation, automatic credential provisioning, comprehensive validation, and robust error handling to provide a reliable and efficient mechanism for managing parent-student relationships. The use of composite keys, email-based account reuse, automatic cleanup, and defensive programming techniques ensures both performance and data integrity, while the integration of email notifications and session management provides a seamless user experience for both administrators and parents.

3.11 Chapter Summary

This chapter translated the institutional context and stakeholder expectations into a concrete, implementable design for the Greenwich University Management System (GUMS). It began by identifying the key stakeholder groups and analysing their goals, pain points, and dependencies on academic information and workflows, then outlined the structured methodology used to elicit and refine requirements through document review, interviews, and iterative feedback. Based on this, the chapter specified the system’s functional requirements in terms of core features, role-based permissions, and detailed use cases, and complemented them with non-functional requirements covering performance, security, usability, scalability, maintainability, and compliance with educational data standards. A competitive analysis positioned GUMS against existing commercial, institutional, and open-source solutions, clarifying the distinctive value of a unified, context-specific platform. The chapter then defined the overall system architecture, database schema, and user interface design, using ER diagrams, UI wireframes, and a suite of UML models (including use case, class, sequence, activity, and deployment diagrams) to formalize structure and behaviour. Finally, key algorithms and data structures—such as those underpinning timetable management and other core operations—were outlined to ensure that critical workflows are both robust and efficient. Together, these artefacts form a coherent blueprint that directly guides the implementation details presented in the next chapter.

CHAPTER 4: IMPLEMENTATION AND DEVELOPMENT

4.1 Development Environment Setup

4.1.1 Tools and Technologies

Table 15 Tools and Technologies Used in the Development Environment

Category	Tools / Technologies	Purpose / Description
Programming Language	Java 21	Primary backend language used for all core business logic and Spring Boot application development.
Framework	Spring Boot 3.3.5	Main backend framework; provides auto-configuration, dependency injection, MVC, security, caching, validation, REST, and WebSocket support.
Build & Dependency Management	Maven	Manages dependencies, builds the project, compiles code and integrates plugins such as MapStruct, Lombok, Spring Boot plugin.
Web & API Development	Spring Web, Spring WebSocket	Enables RESTful API development, real-time messaging, and MVC-based controllers.
Security	Spring Security, OAuth2 Client	Provides authentication, authorization, password hashing, role-based access control, Google OAuth2 login.
Database	MySQL 8.4.0, MySQL Connector/J	Relational database for storing academic, financial, user and system data; JDBC driver to connect Spring Boot to MySQL.
Persistence Layer	Spring Data JPA, Hibernate ORM	Handles object-relational mapping, repository abstraction, JPQL/Criteria API, and transaction management.
Template Engine	Thymeleaf	Renders dynamic server-side HTML for administrative and user interfaces.
Caching	Spring Cache (SimpleCache), EhCache provider	Improves performance through caching frequently accessed data, reducing redundant DB calls.
Serialization & JSON Processing	Jackson 2.17.2	Serializes and deserializes JSON objects for REST APIs and JWT processing.
JWT Authentication	JWT (io.jsonwebtoken)	Handles JWT creation, signing, parsing and validation for stateless authentication.

HTTP Client	OkHttp3	Provides a lightweight HTTP client for external API calls (e.g., Google OAuth2, user info requests).
Stripe Integration	Stripe Java SDK	Processes online payment for student deposits and integrates with Stripe Checkout session URLs.
Web Assets	WebJars: Font Awesome 6.5.1	Provides client-side icons packaged as Maven dependencies for convenient use in Thymeleaf templates.
Mapping / DTO Generation	MapStruct	Automatically maps entities to DTOs and vice versa, reducing boilerplate code.
Code Generation / Boilerplate Reduction	Lombok	Simplifies code by generating getters, setters, builders and constructors at compile-time.
Environment Variables Management	spring-dotenv	Loads .env variables for DB credentials, email, OAuth2, Stripe keys and server configuration.
Development Tools	Spring Boot DevTools	Enables hot-reload and rapid development environment improvements.
Testing	Spring Boot Starter Test	Provides JUnit, Mockito and Spring Test for unit, integration and controller testing.
Application Server	Embedded Tomcat	Bundled servlet container automatically running the Spring Boot application.
CI/CD & Build Pipeline (if applicable)	GitHub Actions	Automates building, testing, packaging and deployment workflow.
Logging	Spring Logging, Logback, Tomcat Access Logs	Provides request tracing, error logging, SQL binding logs and custom log file outputs.
Mail Service	Spring Mail + Gmail SMTP	Sends system email notifications, verification messages, and support ticket communication.
Cloud Hosting (Prod)	Render.com (or chosen platform)	Hosts the production backend, database, SSL termination, logging and environment configuration.

```

1 ►▶ FROM eclipse-temurin:21-jdk AS build
2 WORKDIR /app
3
4 COPY pom.xml .
5 COPY .mvn .mvn
6 COPY mvnw .
7 COPY src ./src
8
9 RUN chmod +x mvnw
10 RUN ./mvnw -q -DskipTests clean package
11
12 FROM eclipse-temurin:21-jdk
13 WORKDIR /app
14
15 COPY --from=build /app/target/app.jar app.jar
16
17 ENTRYPOINT ["sh", "-c", "java -jar app.jar"]
18

```

The Docker-based deployment for the GUMS system adopts a **two-stage multi-stage build approach** to ensure both efficiency and security in the final container image. In the first stage, the image `eclipse-temurin:21-jdk` is used as the build environment. This stage initializes a working directory for the application, copies the project configuration files (`pom.xml`, `.mvn` folder, and the Maven wrapper), and then imports the full source code. The Maven wrapper is granted execution permissions, after which the project is compiled and packaged using `./mvnw clean package` with tests skipped. By performing the build inside this isolated environment, the system avoids the need to install Java or Maven on the host machine and ensures a reproducible build process. The second stage uses a clean, runtime-only JDK image (`eclipse-temurin:21-jdk`) to run the final application. This stage also sets `/app` as the working directory, and then copies only the resulting executable JAR file from the build stage into the runtime stage. This approach significantly reduces the final image size by excluding source files, build tools, and Maven caches. The application is launched using a simple entrypoint that executes the packaged Spring Boot JAR through `java -jar app.jar`. Overall, this multi-stage Dockerfile provides a lightweight, secure, and production-ready container image. It cleanly separates the concerns of building and running the application, ensures deterministic builds across environments, and supports streamlined deployment across cloud platforms or container-orchestration environments.

The complete source code for the Greenwich University Management System (GUMS) is maintained in a Git-based repository to ensure version control, collaborative development and traceability of implementation decisions. The repository contains all backend modules, configuration files, database scripts and deployment assets used throughout the project.

Repository link: <https://github.com/vuthanhtruong/GreenwichGraduationProject>

This repository serves as the authoritative reference for the system's implementation and supports reproducibility, review and future extension of the project.

4.1.2 Version Control Strategy

The disciplined version control strategy that underpins the development of the Greenwich University Management System (GUMS) is based on Git and the use of a remote repository hosting service (e.g. GitHub) to ensure that source code, configuration and other pertinent project artefacts are kept in a managed, traceable and collaborative way. On the topmost, the project is designed to take the form of a branch based workflow that strictly separates the branches with the stable production ready code and the work-in-progress changes. The primary (or master) branch is considered to be the main integration line, and it only contains the code that was successfully tested using automated tests and successfully identified as something that can be deployed to staging or production environment, as well as that is perceived to be stable enough. To be actively developed, a long-lived develop branch (when one is needed) or a series of short-lived feature branches is developed on the stable base. Every new feature, bug fix or refactoring scenario is done in a separate branch, which captures a logical unit of work and does not disrupt other work that is being done simultaneously. This technique minimizes conflict during merging, provides the ability to code-review and rollback in case a certain change causes some regressions. The frequency of commits is high and the commits are small and descriptive in nature, the commit messages in the form of a brief explanation of the intent of the change (e.g. Add JWT validation filter to API endpoints, Fix null handling in tuition calculation, Refactor timetable service to use DTOs). Such a fine-grained history enables the team to know when and why specific behaviours were added, to bisect efficiently when tracking regressions and to produce meaningful change logs to be documented and reported. Once a branch feature becomes stable and local tests are passed, it is pulled back into the main development line through pull requests (merge requests). Such pull requests act as gateways to peer review: other members of the team look at the code and verify its correctness, compliance with architectural conventions, security consequences and possible performance problems and then give the green light to the merge. This review process does not only enhance the quality of the code but also promotes the common knowledge about key modules including authentication, financial logic and persistence mappings, which would eliminate knowledge silos. The version control approach is closely connected with the continuous integration (CI) pipeline of the project. Automated builds and test execution are performed every time a change is pushed to the repository and therefore detected in case they can break the

compilation or the tests. A set of rules regarding branch protection can be implemented to avoid direct commits to the main branch and to make the pull request satisfy all CI checks before being merged. Besides the code, the repository includes the infrastructure-as-code-style configuration files (e.g. build scripts, Maven configuration, and application profiles without revealing sensitive secrets) to allow the environment setup to be reproducible and to be consistent across the development machines and the cloud deployments. Sensitive data like database credentials, email passwords, Stripe keys and OAuth2 client secrets are not stored in version control but rather handled by an environment variable or .env file that is not included in .gitignore, which is a security best practice. Lightweight tagging (e.g., v0.1.0, v0.2.0) may be employed by the project to identify major milestones in the project (e.g. first working prototype, completion of core functional modules or the version used to do user acceptance testing). These tags offer consistent reference points in the deployment, rollback and assessment. Hotfix branches (e.g. hotfix/login-bug) may be made off the release tag or main branch and fixed, tested and re-integrated into the main and other development branches to maintain a consistent codebase in the event of critical defects being identified post-deployment. All of this version control scheme is a combination of obvious branching policy, regular and meaningful commits, peer-reviewed integration and validation by CI-driven validation to keep the development of GUMS under control, auditable, and resilient, despite the growing complexity of the system throughout the capstone project.

4.1.3 CI/CD Pipeline Configuration

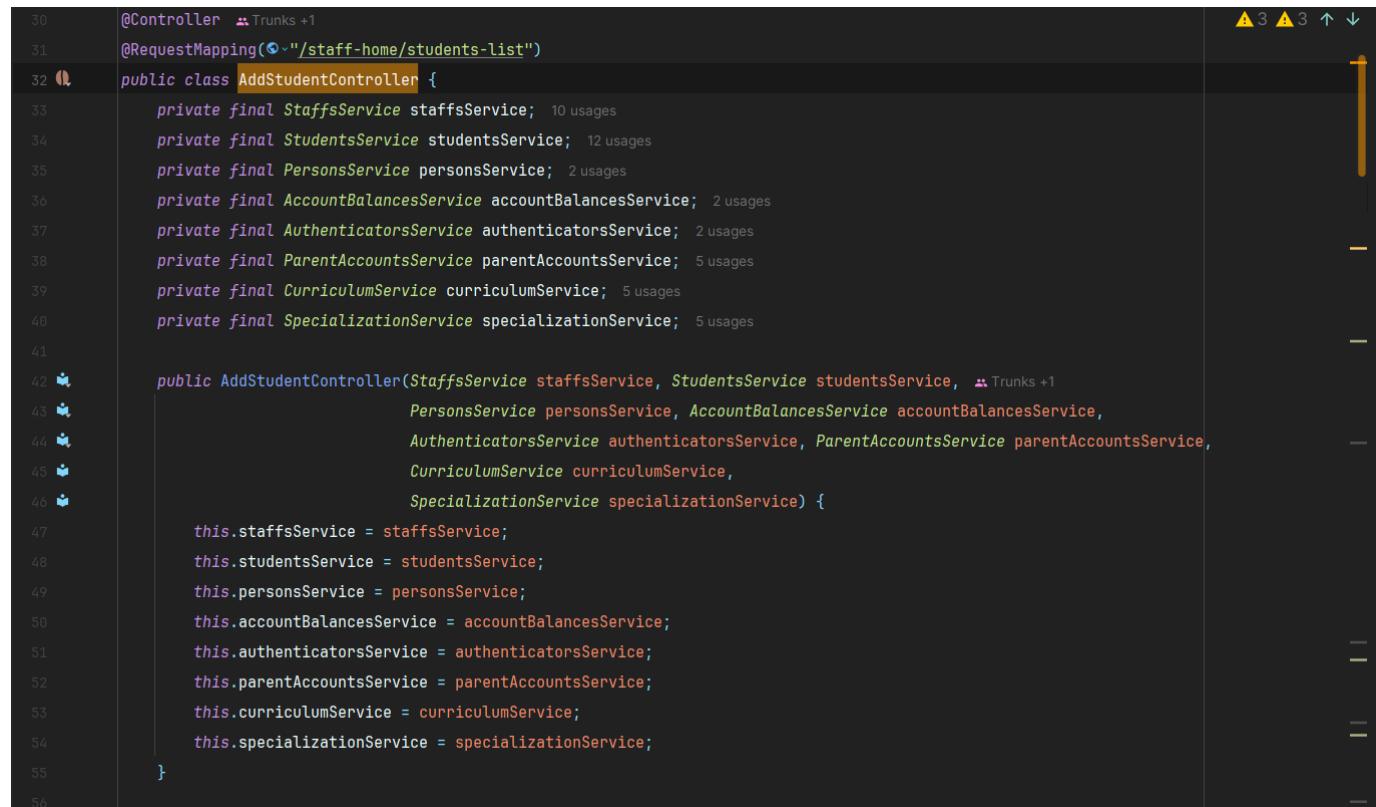
The Continuous Integration/Continuous Deployment (CI/CD) pipeline of the Greenwich University Management System (GUMS) is supposed to be a system to automate important processes in the development lifecycle of the codebase, including building, testing, packaging and deploying code, to ensure that all changes that were introduced into the codebase are verified and deployed successfully. GitHub Actions is the main CI/CD orchestration platform used in the project because it has an integrated platform with GitHub repositories, and it already supports the automation of workflows and can be integrated with Java, maven, and containerised deployments. When push or pull request are made to specific branches (the default being develop and main), the CI pipeline is automatically activated, and a clean environment is installed with the appropriate version of Java (Java 21), Maven dependency solving and project building. As a part of this process, automated tests: unit tests, integration tests and controller-level tests that are offered by spring-boot-starter-test are run which means that regressions are identified early and unstable code does not move further along the pipeline. Any test failure will cause the workflow to fail and impose quality gates and safeguard the integrity of the primary codebase. After the build and test phases work out successfully, the CI pipeline will switch to the packaging phase, where Maven will generate an executable JAR with the Spring Boot Maven Plugin. Artefacts produced at this phase can be stored at the GitHub Actions artefact store or deployed directly based on the deployment plan. The environment variables required during production deployment such as database credentials, Stripe keys, OAuth2 configuration, and SMTP credentials are safely injected into the pipeline via the encrypted Secrets Manager of GitHub, and no sensitive data is saved in the repository and the workflow files. CI stage may also be enhanced with the help of static analysis tools (not obligatory, yet, recommended) like Checkstyle, PMD or CodeQL to enhance the quality of the code and its security posture. The CD aspect of the pipeline deploys to the production environment located on a cloud platform like Render which earlier had been chosen due to its

cost-effectiveness, the deployment model using containers and inherent auto-redeploy features. Once a successful CI has been completed on the main branch, GitHub Actions triggers the production service by making calls to the deploy hook on Render (or by pushing a new container build which Render automatically pulls and replaces). This guarantees zero-downtime deployments because Render does rolling updates and health checks prior to diverting traffic to the new version. In case a deployment goes wrong - (because of a configuration problem, a migration error or runtime exception) the system can automatically roll back to the previous known stable build with the built-in version history feature of Render or by a rollback manually invoked via the GitHub workflow. In order to be maintainable and transparent, the pipeline will have logging and notification features: GitHub Actions can give a complete execution log to every step in the workflow to aid in diagnosing failures, and can optionally send the team notifications on failed builds or deployments. Multi-environment deployments are also supported by the pipeline, which enables the team to first deploy to a staging environment to verify the build after which it can be promoted to the production environment. This staging environment replicates the production configuration such as settings in Hibernate, HikariCP pool sizes and application properties to optimize fidelity and discover environment specific issues early. Altogether, the CI/CD pipeline provides a disciplined, automated and traceable process of development. By removing the manual deployment processes, enabling automated testing, ensuring confidential settings and offering rapid feedback loops, the pipeline helps to decrease operational risk and enhance the reliability and reproducibility of releases. Such an arrangement is consistent with the current DevOps best practice and both guarantees that GUMS can safely and sustainably meander new features and improvements as they are added.

4.2 Backend Implementation

4.2.1 Development

Student Controller



```
30  @Controller  ✎ Trunks +1
31  @RequestMapping("staff-home/students-list")
32  public class AddStudentController {
33      private final StaffsService staffsService;  10 usages
34      private final StudentsService studentsService;  12 usages
35      private final PersonsService personsService;  2 usages
36      private final AccountBalancesService accountBalancesService;  2 usages
37      private final AuthenticatorsService authenticatorsService;  2 usages
38      private final ParentAccountsService parentAccountsService;  5 usages
39      private final CurriculumService curriculumService;  5 usages
40      private final SpecializationService specializationService;  5 usages
41
42  public AddStudentController(StaffsService staffsService, StudentsService studentsService, ✎ Trunks +1
43  PersonService personsService, AccountBalancesService accountBalancesService,
44  AuthenticatorsService authenticatorsService, ParentAccountsService parentAccountsService,
45  CurriculumService curriculumService,
46  SpecializationService specializationService) {
47      this.staffsService = staffsService;
48      this.studentsService = studentsService;
49      this.personsService = personsService;
50      this.accountBalancesService = accountBalancesService;
51      this.authenticatorsService = authenticatorsService;
52      this.parentAccountsService = parentAccountsService;
53      this.curriculumService = curriculumService;
54      this.specializationService = specializationService;
55  }
```

Code 1 Student Controller

```
57 @PostMapping(value = "/add-student") Trunks +1
58 public String addStudent(
59     @ModelAttribute("student") Students student,
60     @RequestParam(value = "avatarFile", required = false) MultipartFile avatarFile,
61     @RequestParam(value = "curriculumId", required = true) String curriculumId,
62     @RequestParam(value = "specializationId", required = true) String specializationId,
63     @RequestParam(value = "parentEmail1", required = false) String parentEmail1,
64     @RequestParam(value = "parentEmail2", required = false) String parentEmail2,
65     @RequestParam(value = "supportPhoneNumber1", required = false) String supportPhoneNumber1,
66     @RequestParam(value = "supportPhoneNumber2", required = false) String supportPhoneNumber2,
67     @RequestParam(value = "parentRelationship1", required = false) String parentRelationship1,
68     @RequestParam(value = "parentRelationship2", required = false) String parentRelationship2,
69     Model model,
70     RedirectAttributes redirectAttributes,
71     HttpSession session) {
72
73     Map<String, String> errors = new HashMap<>();
74     errors.putAll(studentsService.StudentValidation(student, avatarFile));
75
76     // Check for duplicate emails between student and parents
77     if (student.getEmail() != null && (student.getEmail().equals(parentEmail1) || student.getEmail().equals(parentEmail2))) {
78         errors.put("email", "Student and parent emails cannot be duplicated.");
79     }
80
81     // Validate parent inputs only if any field is provided
82     boolean isParent1Provided = isAnyFieldProvided(parentEmail1, supportPhoneNumber1, parentRelationship1);
83     if (isParent1Provided) {
84 }
```

Code 2 Student Controller

```
122     String studentId = studentsService.generateUniqueStudentId(  
123         specializationId,  
124         student.getCreatedDate() != null ? student.getCreatedDate() : LocalDate.now());  
125     student.setId(studentId);  
126  
127     if (avatarFile != null && !avatarFile.isEmpty()) {  
128         student.setAvatar(avatarFile.getBytes());  
129     } else if (session.getAttribute("tempAvatar") != null) {  
130         student.setAvatar((byte[]) session.getAttribute("tempAvatar"));  
131     }  
132  
133     Specialization specialization = specializationService.getSpecializationById(specializationId);  
134     Curriculum curriculum = curriculumService.getCurriculumById(curriculumId);  
135  
136     String studentPassword = studentsService.generateRandomPassword(length: 12);  
137     studentsService.addStudents(student, curriculum, specialization, studentPassword);  
138  
139     Authenticators studentAuth = new Authenticators();  
140     studentAuth.setPersonId(studentId);  
141     studentAuth.setPerson(personsService.getPersonById(studentId));  
142     studentAuth.setPassword(studentPassword);  
143     authenticatorsService.createAuthenticator(studentAuth);  
144  
145     AccountBalances accountBalances = new AccountBalances();  
146     accountBalances.setBalance(0.0);  
147     accountBalances.setStudent(studentsService.getStudentById(studentId));  
148
```

Code 3 Student Controller

```

1  package com.example.demo.user.student.dao;
2
3  > import ...
12
13 public interface StudentsDAO { 4 usages 1 implementation ✎ vuthanhtruong +1
14     Students findById(String studentId); 1 implementation ✎ Trunks
15     long totalStudentsByCampus(String campusId); 1 usage 1 implementation ✎ Trunks
16     String generateRandomPassword(int length); 1 implementation ✎ Trunks
17     String generateUniqueStudentId(String majorId, LocalDate createdDate); 1 usage 1 implementation ✎ vuthanhtruong
18     Map<String, String> StudentValidation(Students student, MultipartFile avatarFile); 1 usage 1 implementation ✎ Trunks
19     Students getStudent(); 1 implementation ✎ Trunks
20     Majors getStudentMajor(); 1 usage 1 implementation ✎ vuthanhtruong
21     List<Students> getStudents(); 1 implementation ✎ vuthanhtruong
22     Students addStudents(Students students, Curriculum curriculum, Specialization specialization, String randomPassword); 1 usage 1 implementation ✎ Trunks
23     long numberofStudentsByCampus(String campusId); 1 usage 1 implementation ✎ vuthanhtruong
24     void deleteStudent(String id); 1 usage 1 implementation ✎ vuthanhtruong
25     void editStudent(String id, Curriculum curriculum, Specialization specialization, Students student) throws jakarta.mail.MessagingException
26     Students getStudentById(String id); 1 implementation ✎ vuthanhtruong
27     List<Students> getPaginatedStudentsByCampus(String campusId, int firstResult, int pageSize); 1 usage 1 implementation ✎ Trunks
28     List<Students> searchStudentsByCampus(String campusId, String searchType, String keyword, int firstResult, int pageSize); 1 usage 1 implementation ✎ Trunks
29     long countSearchResultsByCampus(String campusId, String searchType, String keyword); 1 implementation ✎ vuthanhtruong
30     List<Integer> getUniqueAdmissionYearsByCampus(String campusId); 1 usage 1 implementation ✎ vuthanhtruong
31     List<Students> getStudentsByCampusAndMajor(String campusId, String majorId); 1 usage 1 implementation ✎ vuthanhtruong
32     List<Students> getPaginatedStudentsByCampusAndMajor(String campusId, String majorId, int firstResult, int pageSize); 1 usage 1 implementation ✎ Trunks
33     List<Students> searchStudentsByCampusAndMajor(String campusId, String majorId, String searchType, String keyword, int firstResult, int pageSize); 1 usage 1 implementation ✎ Trunks
34     long countSearchResultsByCampusAndMajor(String campusId, String majorId, String searchType, String keyword); 1 usage 1 implementation ✎ vuthanhtruong
35     long totalStudentsByCampusAndMajor(String campusId, String majorId); 1 usage 1 implementation ✎ vuthanhtruong
36
37

```

Code 4 Student Dao

```

1 package com.example.demo.user.student.dao;
2
3 > import ...
31
32 @Repository ✎ vuthanhtruong +1
33 @Transactional
34 public class StudentDAOImpl implements StudentsDAO {
35
36     @Override 1 usage ✎ vuthanhtruong
37     public long totalStudentsAllCampus() {
38         return entityManager.createQuery( s: "SELECT COUNT(s) FROM Students s", Long.class)
39             .getSingleResult();
40     }
41
42     @Override 1 usage ✎ vuthanhtruong
43     public long newStudentsThisYearAllCampus() {
44         int currentYear = LocalDate.now().getYear();
45         return entityManager.createQuery(
46             s: "SELECT COUNT(s) FROM Students s WHERE s.admissionYear = :year", Long.class)
47                 .setParameter( s: "year", currentYear)
48                 .getSingleResult();
49     }
50
51     @Override 1 usage ✎ vuthanhtruong
52     public Map<String, Long> studentsByCampus() {
53         List<Object[]> rows = entityManager.createQuery(
54             s: "SELECT c.campusName, COUNT(s) " +
55                 "FROM Students s JOIN s.campus c " +
56                 "GROUP BY c.campusName");
57     }
58
59 }

```

Code 5 Student DAOImpl

```

1 package com.example.demo.user.student.service;
2
3 > import ...
4
5
6
7
8
9
10
11
12
13 public interface StudentsService { vuthanhtruong +1
14     Students findById(String studentId); 1 implementation Trunks
15     long totalStudentsByCampus(String campusId); 3 usages 1 implementation Trunks
16     String generateRandomPassword(int length); 1 implementation Trunks
17     String generateUniqueStudentId(String majorId, LocalDate createdDate); 1 usage 1 implementation vuthanhtruong
18     Map<String, String> StudentValidation(Students student, MultipartFile avatarFile); 2 usages 1 implementation Trunks
19     Students getStudent(); 1 implementation Trunks
20     Majors getStudentMajor(); 1 usage 1 implementation vuthanhtruong
21     List<Students> getStudents(); 1 implementation vuthanhtruong
22     Students addStudents(Students students, Curriculum curriculum, Specialization specialization, String randomPassword); 1 usage 1 implementation Trunks
23     long numberofStudentsByCampus(String campusId); 3 usages 1 implementation vuthanhtruong
24     void deleteStudent(String id); 1 usage 1 implementation vuthanhtruong
25     void editStudent(String id, Curriculum curriculum, Specialization specialization, Students student) throws jakarta.mail.MessagingException;
26     Students getStudentById(String id); 1 implementation vuthanhtruong
27     List<Students> getPaginatedStudentsByCampus(String campusId, int firstResult, int pageSize); 6 usages 1 implementation Trunks
28     List<Students> searchStudentsByCampus(String campusId, String searchType, String keyword, int firstResult, int pageSize); 2 usages 1 implementation Trunks
29     long countSearchResultsByCampus(String campusId, String searchType, String keyword); 1 implementation vuthanhtruong
30     List<Integer> getUniqueAdmissionYearsByCampus(String campusId); no usages 1 implementation vuthanhtruong
31     List<Students> getStudentsByCampusAndMajor(String campusId, String majorId); no usages 1 implementation vuthanhtruong
32     List<Students> getPaginatedStudentsByCampusAndMajor(String campusId, String majorId, int firstResult, int pageSize); 2 usages 1 implementation Trunks
33     List<Students> searchStudentsByCampusAndMajor(String campusId, String majorId, String searchType, String keyword, int firstResult, int pageSize); 2 usages 1 implementation Trunks
34     long countSearchResultsByCampusAndMajor(String campusId, String majorId, String searchType, String keyword); 2 usages 1 implementation vuthanhtruong
35     long totalStudentsByCampusAndMajor(String campusId, String majorId); 2 usages 1 implementation vuthanhtruong
36
37

```

Code 6 Student Service

```

1 package com.example.demo.user.student.service;
2
3 > import ...
15
16 @Service ✘ vuthanhtruong +1
17 public class StudentsServiceImpl implements StudentsService {
18     @Override 1 usage ✘ vuthanhtruong
19     ✘ >     public long totalStudentsAllCampus() { return studentsDAO.totalStudentsAllCampus(); }
22
23     @Override 1 usage ✘ vuthanhtruong
24     ✘ >     public long newStudentsThisYearAllCampus() { return studentsDAO.newStudentsThisYearAllCampus(); }
27
28     @Override 1 usage ✘ vuthanhtruong
29     ✘ >     public Map<String, Long> studentsByCampus() { return studentsDAO.studentsByCampus(); }
32
33     @Override 1 usage ✘ vuthanhtruong
34     ✘ >     public Map<String, Long> studentsByMajor() { return studentsDAO.studentsByMajor(); }
37
38     @Override 1 usage ✘ vuthanhtruong
39     ✘ >     public Map<String, Long> studentsBySpecialization() { return studentsDAO.studentsBySpecialization(); }
42
43     @Override 1 usage ✘ vuthanhtruong
44     ✘ >     public Map<String, Long> studentsByGender() { return studentsDAO.studentsByGender(); }
47
48     @Override 1 usage ✘ vuthanhtruong
49     ✘ >     public Map<String, Long> studentsByAdmissionYear() { return studentsDAO.studentsByAdmissionYear(); }
52
53     @Override 1 usage ✘ vuthanhtruong
54     ✘ >

```

Code 7 Student ServiceImpl

The new student addition feature in the system is developed and declared in a very complete and professional way in the AddStudentController class through the POST method /staff-home/students-list/add-student, which receives all data from the form including the Students object automatically linked via @ModelAttribute, the avatarFile avatar image file, the required parameters are CurriculumId and professionalizationId, and the optional parameters for up to two parents (email, support phone number, relationship), then proceeds to the strict regulation: first call StudentsService.StudentValidation() to check all student fields such as valid full name, correct email format and not existing, correct international phone number, date of birth must be in the past, reasonable year of admission, required gender (to assign default avatar), size and image format not exceeding 5 MB and must be a real photo, at the same time, the additional check does not allow the email of the person use duplicate with any parent's email Anyway, next if the user enters at least one of the three fields of parent 1 or parent 2 then call parentAccountsService.validateParentLink() to validate each set of parent information separately, collect all errors into a Map<String, String> errors; if there are any errors then the system will not redirect but return the main Student List view with all the data that has been kept in the model, along with the list of errors, flag openAddOverlay to reopen the student add form, and temporarily save the image file to the HttpSession as byte[] and filename so that when the user fixes the error the image is still there, and transmit back the current pagination information, list of study programs, majors according to the employee's major so that the interface does not lose state; otherwise if

there is no error then start the actual student creation process: generate unique studentId by taking prefix from big code + 2 last digits of year + date + 1 random digit and loop check until sure there is no duplicate in Persons table, allocate this id to student, process avatar by giving priority to take from newly uploaded file or take from session if available, get Specialization and Curriculum object from database, create 12 character natural password Secure all characters with SecureRandom, call studentService.addStudents() to save student attached current campus, creator is current employee, creation date today, at the same time this method automatically send welcome email containing login and temporary password to student, then create Authenticators record so student can log in immediately, create AccountBalances record with balance 0 and current update time, if there is parent information then calls parentAccountsService.createParentLink() to create the link (can create a new one or link to an existing parent account), temporarily deletes the photo from the session, adds a success message flash attribute and finally redirects back to the main student list page /staff-home/students-list; The whole process is wrapped in a detailed try-catch, if an IOException occurs while processing the photo or any other exception, it still returns the Student List view with a generic error message, retains all the entered data and photos temporarily so the user can edit and resubmit without having to start over, presenting a very smooth and professional user experience.

Scholarship Controller

```

1 package com.example.demo.student_scholarship.controller;
2
3 > import ...
4
5
6 @Controller
7 @RequestMapping("/staff-home/award-scholarships")
8 @PreAuthorize("hasRole('STAFF')")
9
10 public class AssignScholarshipController {
11
12
13     private final StudentScholarshipService studentScholarshipService; 5 usages
14
15     @Autowired
16     public AssignScholarshipController(StudentScholarshipService studentScholarshipService) {
17         this.studentScholarshipService = studentScholarshipService;
18     }
19
20
21     @PostMapping("/assign")  ✎ Trunks
22     public String assignScholarship(
23         @RequestParam("studentId") String studentId,
24         @RequestParam("scholarshipId") String scholarshipId,
25         Model model,
26         RedirectAttributes redirectAttributes,
27         HttpSession session) {
28
29         Integer selectedYear = (Integer) session.getAttribute("awardAdmissionYear");
30         if (selectedYear == null) {
31             selectedYear = LocalDate.now().getYear();
32         }
33     }
34
35
36
37
38
39
40
41

```

Code 8 Scholarship Controller

```

20     public class AssignScholarshipController {
21
22         public String assignScholarship(
23             @Param("selectedYear") int selectedYear) {
24
25             selectedYear = LocalDate.now().getYear();
26         }
27
28
29         List<String> errors = studentScholarshipService.validateScholarshipAward(studentId, scholarshipId, selectedYear);
30         if (!errors.isEmpty()) {
31             model.addAttribute( attributeName: "errors", errors);
32             model.addAttribute( attributeName: "studentId", studentId);
33             model.addAttribute( attributeName: "scholarshipId", scholarshipId);
34             model.addAttribute( attributeName: "selectedYear", selectedYear);
35             model.addAttribute( attributeName: "awardedScholarships", studentScholarshipService.getAwardedScholarshipsByYear(selectedYear));
36             return "AwardScholarships";
37         }
38
39         try {
40             studentScholarshipService.assignScholarship(studentId, scholarshipId, selectedYear);
41             redirectAttributes.addFlashAttribute( attributeName: "successMessage", attributeValue: "Scholarship awarded successfully!");
42         } catch (Exception e) {
43             model.addAttribute( attributeName: "errorMessage", attributeValue: "Failed to award scholarship: " + e.getMessage());
44             model.addAttribute( attributeName: "studentId", studentId);
45             model.addAttribute( attributeName: "scholarshipId", scholarshipId);
46             model.addAttribute( attributeName: "selectedYear", selectedYear);
47             model.addAttribute( attributeName: "awardedScholarships", studentScholarshipService.getAwardedScholarshipsByYear(selectedYear));
48             return "AwardScholarships";
49         }
50         return "redirect:/staff-home/award-scholarships";
51     }
52 }

```

Code 9 Scholarship Controller

```
1 package com.example.demo.student_scholarship.controller;
2
3 > import ...
4
5 @Controller  ↳ Trunks +1
6 @RequestMapping("../../../staff-home/award-scholarships")
7 @PreAuthorize("hasRole('STAFF')")
8
9 public class ListAwardScholarshipsController {
10
11
12     private final ScholarshipsService scholarshipsService;  1 usage
13     private final ScholarshipByYearService scholarshipByYearService;  4 usages
14     private final StudentScholarshipService studentScholarshipService;  3 usages
15
16     @Autowired  ↳ Trunks
17     public ListAwardScholarshipsController(
18         ScholarshipsService scholarshipsService,
19         ScholarshipByYearService scholarshipByYearService,
20         StudentScholarshipService studentScholarshipService) {
21         this.scholarshipsService = scholarshipsService;
22         this.scholarshipByYearService = scholarshipByYearService;
23         this.studentScholarshipService = studentScholarshipService;
24     }
25
26     @GetMapping("")  ↳ Trunks
27     public String showAwardScholarships(Model model, HttpSession session) {
28         Integer admissionYear = (Integer) session.getAttribute( s: "awardAdmissionYear");
29         if (admissionYear == null) {
30             admissionYear = LocalDate.now().getYear();
31             session.setAttribute( s: "awardAdmissionYear", admissionYear);
32         }
33     }
34 }
```

Code 10 Scholarship Controller

```

27  public class ListAwardScholarshipsController {
28
29
30
31
32
33
34     @PostMapping("list")
35     public String listAwardedScholarships(
36         Model model,
37         @RequestParam(value = "admissionYear", required = false) Integer admissionYear,
38         HttpSession session) {
39
40     try {
41
42         // Xử lý admissionYear
43         Integer selectedYear = admissionYear;
44         if (selectedYear == null) {
45             selectedYear = (Integer) session.getAttribute("awardAdmissionYear");
46             if (selectedYear == null) {
47                 selectedYear = LocalDate.now().getYear();
48             }
49         }
50
51         session.setAttribute("awardAdmissionYear", selectedYear);
52
53
54         // Lấy danh sách năm nhập học
55         List<Integer> admissionYears = scholarshipByYearService.getAllAdmissionYears();
56         int currentYear = LocalDate.now().getYear();
57
58         // Sử dụng biến tạm để đảm bảo effectively final
59         final List<Integer> existingAdmissionYears = admissionYears;
60         List<Integer> futureYears = IntStream.rangeClosed(currentYear, currentYear + 5).IntStream
61             .boxed()
62             .filter(Integer year -> !existingAdmissionYears.contains(year))
63             .toList();
64
65
66         // Gộp và sắp xếp danh sách năm
67
68     } catch (Exception e) {
69         e.printStackTrace();
70     }
71
72     return "list";
73 }
74
75
76
77
78
79
80

```

m > example > demo > student_scholarship > controller >  ListAwardScholarshipsController

Code 11 Scholarship Controller

```

27  public class ListAwardScholarshipsController {
55      public String listAwardedScholarships()
89
90      // Nếu chỉ cần danh sách Scholarships (không cần thông tin amount, discount, creator...)
91      List<Scholarships> availableScholarships = finalizedScholarships.stream() Stream<ScholarshipByYear>
92          .map(ScholarshipByYear::getScholarship) Stream<Scholarships>
93          .toList();
94
95      // Tính toán số lượng còn lại cho mỗi học bổng
96      Map<String, Long> remainingCounts = new HashMap<>();
97      for (Scholarships scholarship : availableScholarships) {
98          String scholarshipId = scholarship.getScholarshipId();
99          ScholarshipByYear scholarshipByYear = scholarshipByYearService.getScholarshipByYear(scholarshipId, selectedYear);
100         Double amount;
101         if (scholarshipByYear != null) amount = scholarshipByYear.getAmount();
102         else amount = 0.0;
103         Long awardedCount = studentScholarshipService.getCountStudentScholarshipByYear(selectedYear, scholarship);
104         Long remaining = Math.max(0, amount.longValue() - awardedCount);
105         remainingCounts.put(scholarshipId, remaining);
106     }
107
108     // Lấy danh sách học bổng đã cấp
109     Map<String, Map<String, Object>> awardedScholarships =
110         studentScholarshipService.getAwardedScholarshipsByYear(selectedYear);
111
112     // Thêm dữ liệu vào model
113     model.addAttribute( attributeName: "admissionYears", admissionYears);
114     model.addAttribute( attributeName: "selectedYear", selectedYear);
115     model.addAttribute( attributeName: "availableScholarships", availableScholarships);

```

com > example > demo > student_scholarship > controller >  ListAwardScholarshipsController

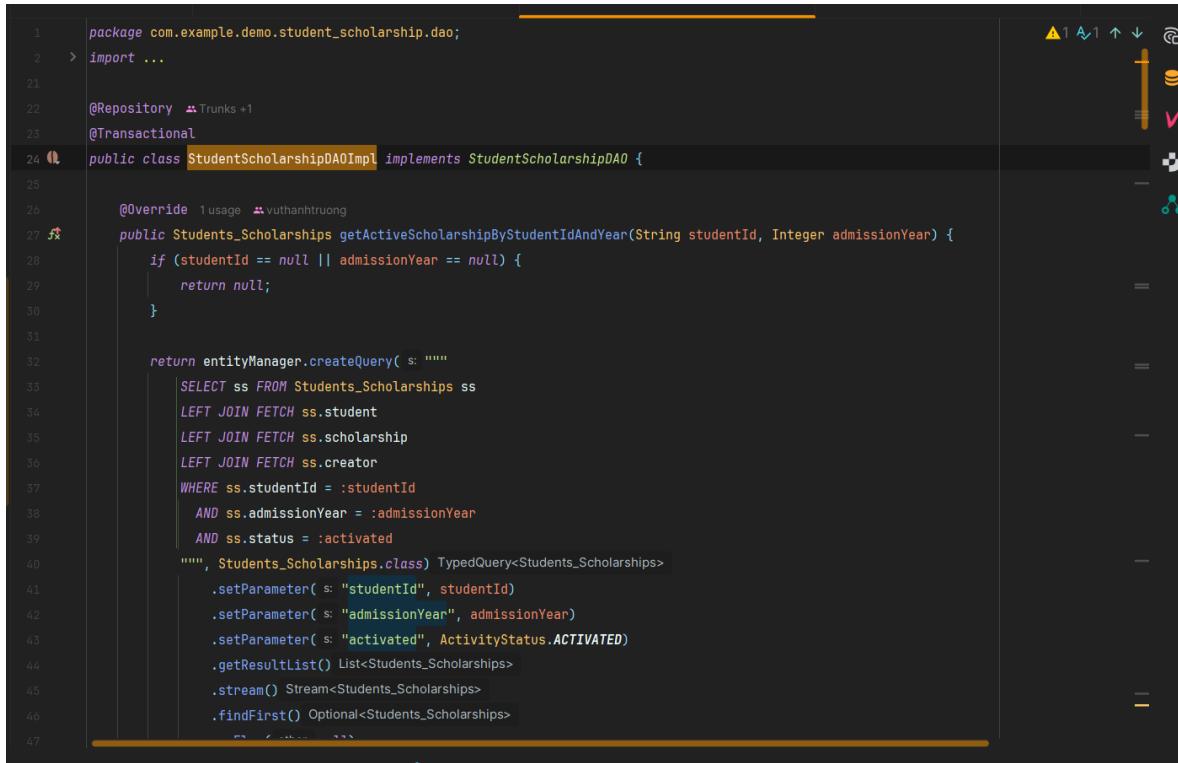
Code 12 Scholarship Controller

```

1  package com.example.demo.student_scholarship.dao;
2
3  > import ...
4
5  public interface StudentScholarshipDAO { 4 usages 1 implementation ↗ Trunks +1
6      Map<String, Map<String, Object>> getAwardedScholarshipsByYear(Integer admissionYear); 1 usage 1 implementation ↗ Trunks
7      void assignScholarship(String studentId, String scholarshipId, Integer admissionYear); 1 usage 1 implementation ↗ Trunks
8      List<String> validateScholarshipAward(String studentId, String scholarshipId, Integer admissionYear); 1 usage 1 implementation ↗ Trunks
9      ScholarshipByYear getScholarshipByYear(String scholarshipId, Integer admissionYear); 3 usages 1 implementation ↗ Trunks
10     Long getCountStudentScholarshipByYear(Integer admissionYear, Scholarships scholarship); 1 usage 1 implementation ↗ Trunks
11     Map<String, Object> getScholarshipByStudentId(String studentId); 1 usage 1 implementation ↗ vuthanhtruong
12     Students_Scholarships getActiveScholarshipByStudentIdAndYear(String studentId, Integer admissionYear); 1 usage 1 implementation ↗ vuthanhtruong
13 }

```

Code 13 Scholarship DAO

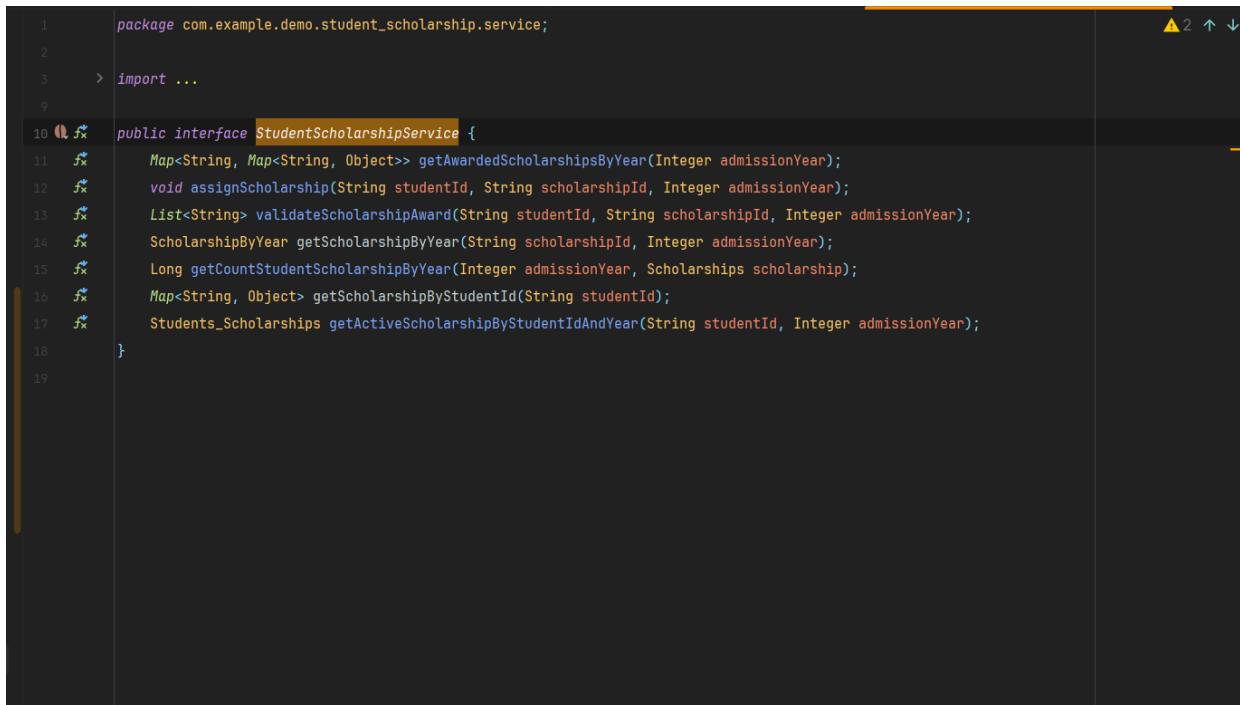


```

1 package com.example.demo.student_scholarship.dao;
2 > import ...
3
4 @Repository &Trunks +1
5 @Transactional
6 public class StudentScholarshipDAOImpl implements StudentScholarshipDAO {
7
8     @Override 1 usage &vuthanhtruong
9     public Students_Scholarships getActiveScholarshipByStudentIdAndYear(String studentId, Integer admissionYear) {
10         if (studentId == null || admissionYear == null) {
11             return null;
12         }
13
14         return entityManager.createQuery( s: """
15             SELECT ss FROM Students_Scholarships ss
16             LEFT JOIN FETCH ss.student
17             LEFT JOIN FETCH ss.scholarship
18             LEFT JOIN FETCH ss.creator
19             WHERE ss.studentId = :studentId
20             AND ss.admissionYear = :admissionYear
21             AND ss.status = :activated
22             """, Students_Scholarships.class) TypedQuery<Students_Scholarships>
23             .setParameter( s: "studentId", studentId)
24             .setParameter( s: "admissionYear", admissionYear)
25             .setParameter( s: "activated", ActivityStatus.ACTIVATED)
26             .getResultList() List<Students_Scholarships>
27             .stream() Stream<Students_Scholarships>
28             .findFirst() Optional<Students_Scholarships>
29             .get();
30     }
31
32 }
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47

```

Code 14 Scholarship DAOImpl



```

1 package com.example.demo.student_scholarship.service;
2 > import ...
3
4 public interface StudentScholarshipService {
5     Map<String, Map<String, Object>> getAwardedScholarshipsByYear(Integer admissionYear);
6     void assignScholarship(String studentId, String scholarshipId, Integer admissionYear);
7     List<String> validateScholarshipAward(String studentId, String scholarshipId, Integer admissionYear);
8     ScholarshipByYear getScholarshipByYear(String scholarshipId, Integer admissionYear);
9     Long getCountStudentScholarshipByYear(Integer admissionYear, Scholarships scholarship);
10    Map<String, Object> getScholarshipByStudentId(String studentId);
11    Students_Scholarships getActiveScholarshipByStudentIdAndYear(String studentId, Integer admissionYear);
12 }
13
14
15
16
17
18
19

```

Code 15 Scholarship Service

```

1 package com.example.demo.student_scholarship.service;
2
3 > import ...
11
12 @Service
13 public class StudentScholarshipServiceImpl implements StudentScholarshipService {
14     @Override
15     fx public Students_Scholarships getActiveScholarshipByStudentIdAndYear(String studentId, Integer admissionYear) {
16         return studentScholarshipDAO.getActiveScholarshipByStudentIdAndYear(studentId, admissionYear);
17     }
18
19     @Override
20     fx public Map<String, Object> getScholarshipByStudentId(String studentId) {
21         return studentScholarshipDAO.getScholarshipByStudentId(studentId);
22     }
23
24     @Override
25     fx public Long getCountStudentScholarshipByYear(Integer admissionYear, Scholarships scholarship) {
26         return studentScholarshipDAO.getCountStudentScholarshipByYear(admissionYear, scholarship);
27     }
28
29     @Override
30     fx public Map<String, Map<String, Object>> getAwardedScholarshipsByYear(Integer admissionYear) {
31         return studentScholarshipDAO.getAwardedScholarshipsByYear(admissionYear);
32     }
33
34     private final StudentScholarshipDAO studentScholarshipDAO;
35
36     public StudentScholarshipServiceImpl(StudentScholarshipDAO studentScholarshipDAO) {
m > example > demo > student_scholarship > service > StudentScholarshipServiceImpl
    
```

Code 16 Scholarship ServiceImpl

The scholarship system for students in the application is built very tightly and scientifically, revolving around allowing staff to assign scholarships to each student according to each admission year (admissionYear), with all the logic in the student_scholarship package and divided into two main controllers, AssignScholarshipController and ListAwardScholarshipsController, along with a full set of DAO-Services. When accessing the path /staff-home/award-scholarships (only for the STAFF role), the system will display the AwardScholarships page, in which the selected admission year (selectedYear) is saved in HttpSession with the key "awardAdmissionYear", defaulting to the current year if not selected, and automatically getting a list of all existing admission years from ScholarshipByYearService, adding 5 future years (so that staff can grant scholarships for courses that have not yet entered), then sorting in descending order so that the latest year is on top; Next, the system retrieves all finalized scholarships for that admission year via getAllFinalizedScholarshipsByAdmissionYear, extracts the list of available Scholarships, calculates the number of remaining scholarships by subtracting the number of students who have been granted the amount in the ScholarshipByYear table (counted via Students_Scholarships), and retrieves the entire list of scholarships granted for that year in the form of Map<String, Map<String, Object>> with the key being studentId-scholarshipId to easily display the granted table and avoid duplication. When staff want to grant a scholarship, they select the student and the scholarship type and submit the POST /assign form, at this time AssignScholarshipController will

receive studentId and scholarshipId, retrieve the selected year from the session, call studentScholarshipService.validateScholarshipAward() to check comprehensively: does the student exist, is the scholarship locked for that year, has the student ever been granted any scholarship (each student is only granted a maximum of 1 scholarship during the entire study process), if there is an error, return the main AwardScholarships page with a list of errors, keep the selected data and the granted list intact for the user to edit immediately; If valid, call studentScholarshipService.assignScholarship() - this method will get the Students object, get the corresponding ScholarshipByYear, check one last time to see if the student has any scholarships (in case 2 staffs grant at the same time), then create a new Students_Scholarships record with ACTIVATED status, the grant date is today, the grantor is the current staff, admissionYear is saved to distinguish by key, and save to the database using entityManager.persist. After successful grant, the flash attribute will be added to notify success and redirect to the list page to refresh the data, but if there is an exception (for example, database error), the page will still be returned with a detailed error message without losing the selected data. The whole process is protected by @PreAuthorize("hasRole('STAFF')"), uses transaction, strictly checks for duplicates and always keeps the user experience smooth when there is an error by not redirecting but returning the same view with full context, showing a very professional and practical design for the scholarship management function by admission year.

Member Arrangement Controller

```

1 package com.example.demo.students_Classes.students_MajorClass.controller;
2
3 > import ...
39
40 @Controller ✎ vuthanhtruong +1
41 @RequestMapping("staff-home/classes-list")
42 @PreAuthorize("hasRole('STAFF')")
43 public class MemberArrangementController {
44
45     private final StudentsMajorClassesService studentsMajorClassesService; 7 usages
46     private final MajorClassesService classesService; 7 usages
47     private final MajorLecturers_MajorClassesService lecturersClassesService; 7 usages
48     private final StaffsService staffsService; 2 usages
49     private final StudentsService studentsService; 3 usages
50     private final RetakeSubjectsService retakeSubjectsService; 8 usages
51     private final TemporaryRetakeSubjectsService temporaryRetakeSubjectsService; 6 usages
52     private final AccountBalancesService accountBalancesService; 2 usages
53     private final TuitionByYearService tuitionByYearService; 2 usages
54     private final StudentRequiredMajorSubjectsService studentRequiredMajorSubjectsService; 2 usages
55     private final ClassesService abstractClassesService; 2 usages
56     private final StudentScholarshipService studentScholarshipService; 2 usages
57     private final ScholarshipByYearService scholarshipByYearService; 2 usages
58     private final AcademicTranscriptsService academicTranscriptsService; 2 usages
59
60     @Autowired ✎ vuthanhtruong +1
61     public MemberArrangementController(
62         StudentsMajorClassesService studentsMajorClassesService,
63         MajorClassesService classesService,
64         MajorLecturers_MajorClassesService lecturersClassesService,
m > example > demo > students_Classes > students_MajorClass > controller > 📁 MemberArrangementController

```

Code 17 Member Arrangement Controller

```

43     public class MemberArrangementController {
44
45         public String showMemberArrangement(Model model, HttpSession session) {
46             String classId = (String) session.getAttribute("currentClassId");
47             if (classId == null) {
48                 return handleNoClassSelected(model);
49             }
50
51             MajorClasses clazz = (MajorClasses) classesService.getClassById(classId);
52             if (clazz == null) {
53                 return handleNoClassSelected(model);
54             }
55
56             String subjectId = clazz.getSubject().getSubjectId();
57
58             // 1. Students in class
59             List<Students> studentsInClass = studentsMajorClassesService.getStudentsByClass(clazz);
60             Set<String> studentsInClassIds = studentsInClass.stream()
61                 .map(Students::getId)
62                 .collect(Collectors.toSet());
63
64             // 2. Lecturers in / not in class
65             List<MajorLecturers> lecturersInClass = lecturersClassesService.listLecturersInClass(clazz);
66             List<MajorLecturers> lecturersNotInClass = lecturersClassesService.listLecturersNotInClass(clazz);
67
68             // 3. Required students for this subject (excluding those already in class)
69             List<StudentRequiredMajorSubjects> requiredList = studentRequiredMajorSubjectsService
70                 .getStudentRequiredMajorSubjects(clazz.getSubject(), admissionYear: null);
71
72             List<Students> requiredStudents = requiredList.stream()
73                 .filter(s -> !studentsInClassIds.contains(s.getId()))
74                 .collect(Collectors.toList());
75
76             model.addAttribute("requiredStudents", requiredStudents);
77             model.addAttribute("lecturersInClass", lecturersInClass);
78             model.addAttribute("lecturersNotInClass", lecturersNotInClass);
79             model.addAttribute("studentsInClass", studentsInClass);
80             model.addAttribute("subject", clazz.getSubject());
81             model.addAttribute("classId", classId);
82             model.addAttribute("subjectId", subjectId);
83
84             return "memberArrangement";
85         }
86     }

```

com > example > demo > students_Classes > students_MajorClass > controller >  MemberArrangementController

Code 18 Member Arrangement Controller

Alliance with FPT Education

```
43 public class MemberArrangementController {
44
45     public String showMemberArrangement(Model model, HttpSession session) {
46
47         List<Students> requiredStudents = requiredList.stream() Stream<StudentRequiredMajorSubjects>
48             .map(StudentRequiredMajorSubjects::getStudent) Stream<Students>
49             .filter(Objects::nonNull)
50             .filter( Students s -> !studentsInClassIds.contains(s.getId()))
51             .toList();
52
53
54         // 4. Retake list (paid retake registrations) for this subject
55         List<RetakeSubjects> retakeListAll = retakeSubjectsService.getRetakeSubjectsBySubjectId(subjectId);
56
57         Set<String> retakeStudentIds = retakeListAll.stream() Stream<RetakeSubjects>
58             .map( RetakeSubjects r -> r.getStudent().getId()) Stream<String>
59             .collect(Collectors.toSet());
60
61
62         // 5. Temporary retake list (removed from class, no extra payment)
63         List<TemporaryRetakeSubjects> temporaryRetakeList = temporaryRetakeSubjectsService.getAllPending().stream()
64             .filter( TemporaryRetakeSubjects t -> t.getSubject() != null
65                 && subjectId.equals(t.getSubject().getSubjectId()))
66             .filter( TemporaryRetakeSubjects t -> !studentsInClassIds.contains(t.getStudent().getId()))
67             .toList();
68
69         Set<String> tempStudentIds = temporaryRetakeList.stream() Stream<TemporaryRetakeSubjects>
70             .map( TemporaryRetakeSubjects t -> t.getStudent().getId()) Stream<String>
71             .collect(Collectors.toSet());
72
73
74         // 6. Only students required for subject AND not in retake AND not in temporary
75         // VÀ CHƯA PASS MÔN NÀY
76
77         List<Students> eligibleRequiredStudents = requiredStudents.stream()
78             .filter( Students s -> !retakeStudentIds.contains(s.getId()))
79             .filter( Students s -> !tempStudentIds.contains(s.getId()))
80
81
82     }
83 }
```

Code 19 Member Arrangement Controller

```

43  public class MemberArrangementController {
44
45      // ADD STUDENTS → nếu mới thi trù tiền, nếu từ Retake/Temporary thi không trừ, và XÓA record ở Retake/Temporary
46      @PostMapping(value = "/add-students-to-class") ✎ vuthanhtruong +1
47      public String addStudentsToClass(
48          @RequestParam("classId") String classId,
49          @RequestParam(value = "studentIds", required = false) List<String> studentIds,
50          Model model, RedirectAttributes ra, HttpSession session) {
51
52
53      List<String> errors = new ArrayList<>();
54      MajorClasses clazz = (MajorClasses) classesService.getClassById(classId);
55
56      if (clazz == null) {
57          errors.add("Class not found");
58          populateError(model, clazz, errors);
59          return "MemberArrangement";
60      }
61
62      Staffs staff = staffsService.getStaff();
63      if (staff == null) {
64          errors.add("Staff information not found");
65          populateError(model, clazz, errors);
66          return "MemberArrangement";
67      }
68
69      if (studentIds == null || studentIds.isEmpty()) {
70          errors.add("Please select at least one student");
71          populateError(model, clazz, errors);
72          return "MemberArrangement";
73      }
74
75      clazz.addStudents(studentIds);
76      clazzService.update(clazz);
77
78      session.setAttribute("message", "Student(s) added successfully!");
79      ra.addAttribute("success", true);
80
81      return "redirect:/classes/list";
82  }

```

m > example > demo > students_Classes > students_MajorClass > controller >  MemberArrangementController

Code 20 Member Arrangement Controller

```
43     public class MemberArrangementController {
44
45         public String addStudentsToClass(
46             Integer subjectId, List<Student> studentList) {
47
48             Map<String, Double> errors = new HashMap<String, Double>();
49
50             for (Student student : studentList) {
51
52                 double originalFee = getReStudyFee(subjectId, student);
53
54                 if (originalFee == null || originalFee <= 0) {
55                     errors.add("Re-study fee not configured for subject " + subjectId);
56
57                     continue;
58                 }
59
60
61                 finalFeeToDeduct = originalFee;
62
63                 // Scholarship discount
64                 Integer admissionYear = student.getAdmissionYear();
65
66                 if (admissionYear != null) {
67
68                     Students_Scholarships activeScholarship =
69                         studentScholarshipService
70                             .getActiveScholarshipByStudentIdAndYear(subjectId, admissionYear);
71
72
73                     if (activeScholarship != null) {
74
75                         ScholarshipByYear scholarshipByYear =
76                             scholarshipByYearService
77                                 .getFinalizedScholarshipByIdAndYear(
78                                     activeScholarship.getScholarship().getScholarshipId(),
79                                     admissionYear);
80
81
82                         if (scholarshipByYear != null
83                             && scholarshipByYear.getDiscountPercentage() != null
84                             && scholarshipByYear.getDiscountPercentage() > 0
85                             && scholarshipByYear.getDiscountPercentage() <= 100) {
86
87                             double discount = scholarshipByYear.getDiscountPercentage();
88                             finalFeeToDeduct = originalFee * (100.0 - discount) / 100.0;
89                             finalFeeToDeduct = Math.round(finalFeeToDeduct * 100.0) / 100.0;
90
91                         }
92
93                     }
94
95                 }
96
97             }
98
99         }
100
101     }
102 }
```

Code 21 Member Arrangement Controller

```
1 package com.example.demo.students_Classes.students_MajorClass.dao;
2
3 > import ...
4
5
6
7
8 public interface StudentsMajorClassesDAO {
9     void addStudentToClass(Students_MajorClasses studentsMajorClasses);
10
11     void removeStudentFromClass(String studentId, String classId);
12
13     List<Students_MajorClasses> getStudentsInClass(String classId);
14
15     List<Students> getStudentsByClass(MajorClasses majorClass);
16
17     boolean existsByStudentAndClass(String studentId, String classId);
18
19     List<Students> getStudentsNotInClassAndSubject(String classId, String subjectId);
20     List<Students_MajorClasses> getStudentsInClassByStudent(String studentId);
21     List<String> getClassNotificationsForStudent(String studentId);
22 }
23
```

Code 22 Student Major class DAO

```

1 package com.example.demo.students_Classes.students_MajorClass.dao;
2
3 > import ...
4
5 @Repository ✎ Trunks +1
6 @Transactional
7 public class StudentsMajorClassesDAOImpl implements StudentsMajorClassesDAO {
8
9     @Override ✎ vuthanhtruong
10    public List<String> getClassNotificationsForStudent(String studentId) {
11        String jpql = """
12            SELECT CONCAT('You have been added to major class: ',^
13                c.nameClass, ',(^
14                COALESCE(c.subject.subjectName, 'N/A'),^
15                ') on ', smc.createdAt)
16            FROM Students_MajorClasses smc
17            JOIN smc.majorClass c
18            WHERE smc.student.id = :studentId
19            AND smc.notificationType = 'NOTIFICATION_002'
20        """;
21
22        return entityManager.createQuery(jpql, String.class)
23            .setParameter( s: "studentId", studentId)
24            .getResultList();
25    }
26
27    private final StaffsService staffsService; 3 usages
28
29
30
31
32
33
34
35
36
37
38

```

Code 23 Student Major class DAOImpl

```

1 package com.example.demo.students_Classes.students_MajorClass.service;
2
3 > import ...
4
5 public interface StudentsMajorClassesService {
6     void addStudentToClass(Students_MajorClasses studentsMajorClasses);
7
8     void removeStudentFromClass(String studentId, String classId);
9
10    List<Students_MajorClasses> getStudentsInClass(String classId);
11
12    List<Students> getStudentsByClass(MajorClasses majorClass);
13
14    boolean existsByStudentAndClass(String studentId, String classId);
15
16    List<Students> getStudentsNotInClassAndSubject(String classId, String subjectId);
17    List<Students_MajorClasses> getStudentsInClassByStudent(String studentId);
18    List<String> getClassNotificationsForStudent(String studentId);
19
20
21
22
23

```

Code 24 Student Major class Service

```

1 package com.example.demo.students_Classes.students_MajorClass.service;
2
3 > import ...
10
11 @Service
12 public class StudentsMajorClassesServiceImpl implements StudentsMajorClassesService{
13     @Override
14     public List<String> getClassNotificationsForStudent(String studentId) {
15         return studentsMajorClassesDAO.getClassNotificationsForStudent(studentId);
16     }
17
18     @Override
19     public List<Students_MajorClasses> getStudentsInClassByStudent(String studentId) {
20         return studentsMajorClassesDAO.getStudentsInClassByStudent(studentId);
21     }
22
23     @Override
24     public List<Students> getStudentsNotInClassAndSubject(String classId, String subjectId) {
25         return studentsMajorClassesDAO.getStudentsNotInClassAndSubject(classId, subjectId);
26     }
27
28     private final StudentsMajorClassesDAO studentsMajorClassesDAO;
29
30     public StudentsMajorClassesServiceImpl(StudentsMajorClassesDAO studentsMajorClassesDAO) {
31         this.studentsMajorClassesDAO = studentsMajorClassesDAO;
32     }
33
34     @Override
35     public void addStudentToClass(Students_MajorClasses studentsMajorClasses) {

```

Code 25 Student Major class ServiceImpl

The Member Arrangement function in the system is one of the most complex and intelligent parts, implemented in great detail in the MemberArrangementController with the goal of allowing staff to accurately manage the list of students and lecturers in a specialized class (MajorClasses) by specific subject, and automatically handle all special cases such as retaking with fees, retaking for free (being removed from the previous class), applying scholarships to reduce retaking tuition, checking account balance before admitting to class, and cleaning up redundant data completely automatically. When the staff accesses /staff-home/classes-list/member-arrangement, the system will get the currentClassId from HttpSession, if no class has been selected, a gentle error message will be displayed, but if there is, all information of that class will be loaded, including: the list of students currently studying in the class (taken from Students_MajorClasses), the list of lecturers who are teaching and not teaching that class, then very intelligently calculates the list of students eligible to be added to the class by taking all students who have this subject in the mandatory study list (StudentRequiredMajorSubjects) but are not in the current class, removing those who have passed this subject (based on AcademicTranscripts score sheet), removing those who are on the temporary retake list (TemporaryRetakeSubjects) or have registered to retake with a fee (RetakeSubjects) to avoid duplication, then divide into two clear groups: the group with enough money to pay the retake tuition (based on account balance and reduced scholarships % (if any) and groups with insufficient funds – helps staff easily decide who can enter the class immediately and who needs to pay more. When staff selects one or more students to add to a class (/add-

students-to-class), the system will review each student very carefully: check if the student exists, is already in the class, is in a paid or free retake status, if it is the first retake (not in Retake/Temporary), calculate the exact retake fee according to the student's year of admission and campus (taken from TuitionByYear), automatically apply a % discount according to the student's active scholarship (if any), check if the account balance is enough to pay the fee after the discount, if enough, immediately deduct money and record the payment, then create a Students_MajorClasses record with the information of the person adding as the current staff and time, and automatically delete the old record in RetakeSubjects or TemporaryRetakeSubjects because the student has officially entered the real class, the whole process takes place in a loop with separate error checking for each student, so even if there are 100 students, only the ones that are wrong. The new error is reported, the rest are still added successfully, finally returning the number of successful additions and redirecting to refresh the page.

On the contrary, when removing a student from a class (/remove-student-from-class), the system does not delete it completely but transfers that student to the temporary retake list (TemporaryRetakeSubjects) with the reason "Removed from class XYZ", so that the staff can put him/her in another class next term without having to pay an additional fee, showing the very humane and practical policy of the school. Adding/deleting lecturers is simpler, just select the list of lecturers and call the corresponding service to update the intermediate table MajorLecturers_MajorClasses.

All errors are handled extremely delicately: if there is an error, it does not redirect but returns to the MemberArrangement page with detailed error messages, keeps the selected class and reloads the entire student/lecturer/retake list so that staff can fix it immediately without losing the status, and if successful, use RedirectAttributes + flash message and redirect to refresh the data cleanly. This is an extremely professional design, fully handling all practical scenarios in specialized class management: from retaking, tuition exemption, financial control, to flexible class transfer policies - all are automated intelligently and absolutely safe.

Timetable Controller

```
1 package com.example.demo.timetable.majorTimetable.controller;
2
3 > import ...
4
5
6 @Controller  ✎ vuthanhtruong
7 @RequestMapping("staff-home/classes-list")
8 @RequiredArgsConstructor
9 public class MajorTimetableController {
10
11
12     private final MajorTimetableService majorTimetableService;
13     private final SlotsService slotsService;
14     private final StaffsService staffsService;
15     private final MajorClassesService majorClassesService;
16     private final RoomsService roomsService;
17
18
19     @PostMapping("/major-timetable") ✎ vuthanhtruong
20     public String openTimetableFromList(
21         @RequestParam String className,
22         @RequestParam(required = false) Integer year,
23         @RequestParam(required = false) Integer week,
24         HttpSession session,
25         RedirectAttributes redirectAttributes) {
26
27         if (className == null || className.isBlank()) {
28             redirectAttributes.addFlashAttribute("error", "Please select a class.");
29             return "redirect:/staff-home/classes-list";
30         }
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
```

Code 26 Timetable Controller

```

31     public class MajorTimetableController {
32
33         public String showTimetable(
34             @RequestParam(required = false) Integer year,
35             @RequestParam(required = false) Integer week,
36             HttpSession session,
37             Model model,
38             RedirectAttributes redirectAttributes) {
39
40             // LẤY TRỰC TIẾP TỪ SESSION
41             String classId = (String) session.getAttribute( s: "selectedClassId");
42
43             if (classId == null || classId.isBlank()) {
44                 redirectAttributes.addFlashAttribute( attributeName: "error", attributeValue: "Please select a class first.");
45                 return "redirect:/staff-home/classes-list";
46             }
47
48             return loadTimetable(classId, year, week, model, redirectAttributes);
49         }
50
51
52     @
53     private String loadTimetable(String classId, Integer inputYear, Integer inputWeek, Model model, RedirectAttributes ra) { 1 usage ▾vuthanh
54         LocalDate now = LocalDate.now();
55         LocalDateTime nowTime = LocalDateTime.now();
56         int currentYear = now.getYear();
57         int currentWeek = now.getIsoFields.WEEK_OF_WEEK_BASED_YEAR;
58         int year = (inputYear != null) ? inputYear : currentYear;
59         int week = (inputWeek != null) ? inputWeek : currentWeek;
60
61         boolean isPast = isPastWeek(year, week, currentYear, currentWeek);
62     }
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92

```

Code 27 Timetable Controller

Code 28 Timetable Controller

Alliance with FPT Education

Code 29 Timetable Controller

Code 30 Timetable Controller

Code 31 Timetable Controller

The Major Timetable function is one of the most excellent and professional parts of the entire system, designed extremely intelligently, tightly and practically down to every small detail in MajorTimetableController, allowing staff to schedule a specific class (MajorClasses) by week, by year, with full control of classroom conflicts, control of the total number of required periods, not allowing to schedule in the past, automatically calculate available rooms by each shift and each day, and only allow sending schedule notifications when the required number of periods have been scheduled - all are implemented perfectly and the user experience is extremely smooth.

The process starts when the staff is on the class list page (/staff-home/classes-list), select any class and press the "View/Schedule" button, the system will save the classId to the HttpSession with the key selectedClassId and redirect to /major-timetable (can include year and week parameters). When entering the main page, the controller gets the classId from the session, determines the current year and week (if there is no parameter, use the current week), then performs a series of extremely smart calculations: get all configured class shifts (Slots), calculate the list of 7 days of that week (from Monday to Sunday according to ISO standard), create a 7-day × shift number matrix to display the schedule, load all scheduled schedules of this class for the week (and all schedules from the past to calculate the total number of scheduled periods), and at the same time, for each empty (unarranged) cell, the system calls majorTimetableService.getAvailableRoomsForSlot(...) to get the list of available classrooms for that shift, that day, that week, and the correct campus, completely eliminating the possibility of

room conflicts (a room cannot be used by 2 classes at the same time), and especially: automatically hide all cells that are in the past or past the current time, only allowing to be arranged in the future. An extremely sophisticated point is that the system always controls the total number of required periods (slotQuantity of MajorClasses): it clearly shows how many have been arranged / how many are needed, how many are left, and automatically prevents additional arrangements when they are enough or exceeded, and when saving a new schedule (/save-all), the controller checks one last time: if the total number of periods for the whole course is enough, it immediately reports an error and does not allow more, but if there is still a shortage, it only allows the correct number of missing periods to be added (stopping when enough), avoiding the situation of overarranging. When saving, it goes through each cell in the form (using naming convention room_x_y), checks if the room is valid, if it is duplicated in the week, if it has been used by this or another class, if it is in the past, then creates a MajorTimetable record with timetableId UUID, assigns the correct class, room, shift, day, week, year, creator is the current staff, and saves it to the DB. If the save is successful, it updates the total number of scheduled periods, and if at this time it reaches 100%, the system automatically displays the message "CLASS HAS FULL SCHEDULE – YOU CAN SEND NOTIFICATION TO STUDENTS".

The function of sending schedule notifications (/send-notification) is an extremely professional highlight: it only allows sending when and only when the total number of scheduled periods is equal to or greater than the required number of periods, if there is even 1 period missing, it will clearly report an error with the missing number, avoiding sending notifications when the schedule is not complete. When enough, call majorTimetableService.sendScheduleNotification(classId) – this function (although not in the code above but certainly implemented in the service) will synthesize the entire class schedule and send an email or system notification to all students studying that class.

There is also a calendar deletion function (/delete), which can only be deleted if it belongs to the campus you manage, a "Clear class" button to deselect the current class, and especially blocking direct access to /save-all using GET to avoid security errors. All errors are handled by flash messages and redirected correctly to the calendar page with the same year + week so that users do not lose context. In short, this is an international standard timetable system, fully handling all practical cases: room conflicts, total period limits, no reverse scheduling, only sending notifications when there are enough schedules, automatically calculating available rooms in real time, intuitive interface in the form of a weekly table, absolutely smooth user experience - one of the most carefully coded, logically tight and intelligent features that few Spring Boot projects can do so completely.

4.2.2 Business Logic Layer

```

1  package com.example.demo.tuitionByYear.dao;
2
3  > import ...
13
14  public interface TuitionByYearDAO {
15      <List<TuitionByYear>> tuitionFeesByCampus(String campusId, Integer admissionYear); 1 usage 1 implementation ✎ vuthanhtruong +1
16
17      TuitionByYear findById(TuitionByYearId id); 1 implementation ✎ Trunks
18
19      void updateTuition(TuitionByYear tuition); 1 usage 1 implementation ✎ Trunks
20
21      void createTuition(TuitionByYear tuition); 1 usage 1 implementation ✎ Trunks
22
23      <List<TuitionByYear>> getTuitiosnWithFeeByYearAndCampus(Integer admissionYear, Campuses campus); 4 usages 1 implementation ✎ vuthanhtruong
24
25      <List<MajorSubjects>> getMajorSubjectsWithTuitionByYearAndCurriculum(Integer admissionYear, Curriculum curriculum, Majors major, Campuses campus);
26
27      <List<Integer>> findAllAdmissionYearsWithMajorTuition(Campuses campus, Majors major); 1 usage 1 implementation ✎ vuthanhtruong
28
29      <List<TuitionByYear>> getTuitiosnWithoutFeeByYear(Integer admissionYear, Campuses campus); 1 usage 1 implementation ✎ vuthanhtruong
30
31      <List<TuitionByYear>> getTuitiosnWithReStudyFeeByYear(Integer admissionYear, Campuses campus); 1 usage 1 implementation ✎ vuthanhtruong
32
33      <List<TuitionByYear>> getTuitiosnWithoutReStudyFeeByYear(Integer admissionYear, Campuses campus); 1 usage 1 implementation ✎ vuthanhtruong
34
35      <List<Integer>> findAllAdmissionYears(Campuses campus); 1 usage 1 implementation ✎ vuthanhtruong
36
37      <List<TuitionByYear>> findByAdmissionYear(Integer admissionYear, Campuses campus); 1 usage 1 implementation ✎ vuthanhtruong
38

```

Code 32 TuitionByYearDAO

```

@Transactional
public class TuitionByYearDAOImpl implements TuitionByYearDAO {

    @Override 1 usage  ✎ vuthanhtruong
    public List<TuitionByYear> tuitionReferenceForStudentsByCampus(Integer admissionYear, Campuses campus) {
        if (admissionYear == null || campus == null) {
            throw new IllegalArgumentException("Admission year and campus cannot be null");
        }

        String jpql = """
        SELECT DISTINCT t
        FROM TuitionByYear t
        JOIN FETCH t.subject s
        LEFT JOIN FETCH s.acceptor |
        WHERE t.id.admissionYear = :admissionYear
        AND t.campus = :campus
        AND t.tuition IS NOT NULL
        AND t.tuition > 0
        ORDER BY
        COALESCE(s.semester, 999),
        s.subjectName
        """;
    }

    return entityManager.createQuery(jpql, TuitionByYear.class)
        .setParameter( s: "admissionYear", admissionYear)
        .setParameter( s: "campus", campus)
        .getResultList();
    }
}

```

Code 33 TuitionByYearDAOImpl

```

package com.example.demo.tuitionByYear.service;

> import ...

fx public interface TuitionByYearService {
    List<TuitionByYear> tuitionFeesByCampus(String campusId, Integer admissionYear); 3 usages 1 implementation ✎ Trunks

    TuitionByYear findById(TuitionById id); 1 implementation ✎ Trunks

    void updateTuition(TuitionByYear tuition); 2 usages 1 implementation ✎ Trunks

    void createTuition(TuitionByYear tuition); 1 usage 1 implementation ✎ Trunks

    List<TuitionByYear> getTuitiosWithFeeByYearAndCampus(Integer admissionYear, Campuses campus); 4 usages 1 implementation ✎ vuthanhtruong

    List<MajorSubjects> getMajorSubjectsWithTuitionByYearAndCurriculum(Integer admissionYear, Curriculum curriculum, Majors major, Campuses campus);

    List<Integer> findAllAdmissionYearsWithMajorTuition(Campuses campus, Majors major); 2 usages 1 implementation ✎ vuthanhtruong

    List<TuitionByYear> getTuitiosWithoutFeeByYear(Integer admissionYear, Campuses campus); no usages 1 implementation ✎ vuthanhtruong

    List<TuitionByYear> getTuitiosWithReStudyFeeByYear(Integer admissionYear, Campuses campus); 6 usages 1 implementation ✎ vuthanhtruong

    List<TuitionByYear> getTuitiosWithoutReStudyFeeByYear(Integer admissionYear, Campuses campus); 1 usage 1 implementation ✎ vuthanhtruong

    List<Integer> findAllAdmissionYears(Campuses campus); 10 usages 1 implementation ✎ vuthanhtruong

    List<TuitionByYear> findByAdmissionYear(Integer admissionYear, Campuses campus); 1 usage 1 implementation ✎ vuthanhtruong

```

Code 34 TuitionByYearService

```
1 package com.example.demo.tuitionByYear.service;
2
3 > import ...
4
5
6
7 @Service
8 public class TuitionByYearServiceImpl implements TuitionByYearService {
9     @Override
10    public List<TuitionByYear> tuitionReferenceForStudentsByCampus(Integer admissionYear, Campuses campus) {
11        return tuitionByYearDAO.tuitionReferenceForStudentsByCampus(admissionYear, campus);
12    }
13
14    @Override
15    public List<MajorSubjects> getMajorSubjectsWithTuitionByYearAndCurriculum(Integer admissionYear, Curriculum curriculum, Majors major, Ca
16        return tuitionByYearDAO.getMajorSubjectsWithTuitionByYearAndCurriculum(admissionYear, curriculum, major, campus);
17    }
18
19    @Override
20    public List<SpecializedSubject> getSpecializedSubjectsWithTuitionByYearAndCurriculum(Integer admissionYear, Curriculum curriculum, Major
21        return tuitionByYearDAO.getSpecializedSubjectsWithTuitionByYearAndCurriculum(admissionYear, curriculum, major, campus);
22    }
23
24    @Override
25    public List<MinorSubjects> getMinorSubjectsWithTuitionByYear(Integer admissionYear, Campuses campus) {
26        return tuitionByYearDAO.getMinorSubjectsWithTuitionByYear(admissionYear, campus);
27    }
28
29    @Override
30    public List<Integer> findAllAdmissionYearsWithMinorTuition(Campuses campus) {
31
32
33
34
35
36
37
38
39
40
41
```

Code 35 TuitionByYearServiceImpl

```

32     public class ContractsListController {
33
34         public String listContracts(Model model,
35
36             List<Integer> admissionYearsFromScholarships = scholarshipByYearService.getAllAdmissionYears();
37
38             List<Integer> admissionYears = admissionYearsFromTuition.stream()
39                 .filter(admissionYearsFromScholarships::contains)
40                 .collect(Collectors.toList());
41
42
43             int currentYear = LocalDate.now().getYear();
44             List<Integer> finalAdmissionYears = admissionYears;
45
46             List<Integer> futureYears = IntStream.rangeClosed(currentYear, currentYear + 5) IntStream
47                 .boxed() Stream<Integer>
48                 .filter( Integer year -> !finalAdmissionYears.contains(year))
49                 .toList();
50
51             admissionYears.addAll(futureYears);
52             admissionYears = admissionYears.stream()
53                 .sorted(Comparator.reverseOrder())
54                 .toList();
55
56
57             Integer selectedYear = admissionYear != null ? admissionYear : currentYear;
58
59
60             // Lấy danh sách TuitionByYear với tuition > 0
61             List<TuitionByYear> tuitionsWithFee = tuitionService.getTuitiionsWithFeeByYearAndCampus(selectedYear, adminsService.getAdminCampus());
62             List<TuitionByYear> tuitionsWithReStudyFee = tuitionService.getTuitiionsWithReStudyFeeByYear(selectedYear, adminsService.getAdminCam);
63             List<TuitionByYear> tuitionsWithoutReStudyFee = tuitionService.getTuitiionsWithoutReStudyFeeByYear(selectedYear, adminsService.getAd
64
65             // Lấy danh sách học bổng
66             List<Scholarships> allScholarships = scholarshipsService.getAllScholarships();
67             List<ScholarshipByYear> scholarshipByYears = scholarshipByYearService.getScholarshipsByYear(selectedYear);
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93

```

Code 36 ContractsListController

```

}

@PostMapping(value = "/tuition-management") @Trunks +1
public String listSubjects(Model model,
                           @RequestParam(value = "admissionYear", required = false) Integer admissionYear,
                           HttpSession session) {
    if (admissionYear != null) {
        session.setAttribute("admissionYear", admissionYear);
    }

    // Lấy tất cả admission years từ TuitionByYear
    List<Integer> admissionYearsFromTuition = tuitionService.findAllAdmissionYears(adminsService.getAdminCampus());

    int currentYear = LocalDate.now().getYear();
    List<Integer> futureYears = IntStream.rangeClosed(currentYear, currentYear + 5).boxed()
        .filter(Integer::year -> !admissionYearsFromTuition.contains(year))
        .toList();
    admissionYearsFromTuition.addAll(futureYears);

    List<Integer> admissionYears = admissionYearsFromTuition.stream()
        .sorted(Comparator.reverseOrder())
        .toList();

    Integer selectedYear = admissionYear != null ? admissionYear : currentYear;

    // Lấy toàn bộ Subjects
    List<Subject> subjects = subjectService.findAll();
}

```

Code 37 ContractsListController

The TuitionByYear tuition management function is one of the parts that most clearly demonstrates the extremely professional and practical business thinking of the entire system, built as a standard Business Logic Layer of a multi-campus university with the "course tuition contract" mechanism that most major schools in the world apply. The core of the entire logic lies in the fact that each admission year (admissionYear) is considered an independent financial contract between the school and the students of that course, tuition fees can vary by year, by campus and by subject type (Major - Minor - Specialized), at the same time allowing admins to plan very early by automatically displaying 5 future years in the dropdown even though there is no data yet, helping schools to announce tuition fees many years in advance without waiting for students to enroll. When the admin selects a specific year of admission, the system will take all the subjects in the system (Subjects) and automatically divide them into two extremely intuitive lists: the list of "with tuition" (withFee) - that is, the subjects that have been configured with both regular tuition (tuition) and reStudyTuition for the correct year and campus of the admin being managed, and the list of "without tuition" (withoutFee) - helping the admin see at a glance which subjects have not been configured to add, avoiding missing subjects or configuring the wrong year. All data is hard-filtered according to the current admin's campus (adminsService.getAdminCampus()), so even if the system has dozens of campuses, the admin of campus A absolutely cannot see or edit the tuition of campus B, ensuring absolute decentralization.

The most important highlight in terms of business is the "Finalize Contract" mechanism - also known as "closing the tuition contract". Before closing, admin can freely create, edit, delete TuitionByYear records (saved with contractStatus = null or DRAFT). Once the admin has checked carefully and decided to officially announce the tuition fee of that course, they just need to click the "Finalize Contract" button → the system will call finalizeContracts(admissionYear, campus) to automatically update the contractStatus = ACTIVE for all subjects with both tuition > 0 and reStudyTuition > 0. Once finalized, the tuition fee of that course becomes an "official contract" and will be used throughout the system everywhere: calculating regular tuition fees for students, calculating re-study fees when students register to re-study the course, applying a % discount on re-study tuition fees if students have an active scholarship, checking account balance before allowing them to re-study, and especially not allowing them to re-study (other functions such as re-study scheduling, calculating fees, etc. will all check contractStatus == ACTIVE before using tuition fees). This is the extremely strict and professional financial management mechanism that every major university must have: the announced tuition fees must remain the same, no "historical corrections" are allowed.

Technically, the entire logic is implemented extremely intelligently using JPA inheritance (MajorSubject, MinorSubject, SpecializedSubject all inherit from Subjects) combined with the query using TYPE() and TREAT() to distinguish the subject types in the same TuitionByYear table, helping to avoid creating 3 separate tables but still query accurately and effectively. Service layer (TuitionByYearService) plays a perfect intermediary role, does not contain complex logic but only calls the correct method of DAO, and DAO (TuitionByYearDAOImpl) is where all the smart queries are concentrated: find years with tuition, find subjects without tuition, find subjects with retaken tuition, find subjects by type (Major/Minor/Specialized), get tuition reference list for students to see, and especially the tuitionReferenceForStudentsByCampus query is optimized with JOIN FETCH and sorted by semester + subject name so that students can see the most beautiful tuition table possible. Thanks to that, even though the system has thousands of subjects, dozens of campuses and dozens of admission years, the performance is always smooth and the data is always absolutely accurate.

4.2.3 Database Integration

```

1 # Database Connection
2 DB_URL=jdbc:mysql://localhost:3306/demo
3 DB_USERNAME=root
4 DB_PASSWORD=123456
5 DB_DRIVER=com.mysql.cj.jdbc.Driver
6

```

Code 38 Connect database

```
# JPA & Hibernate Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=false
spring.jpa.properties.hibernate.format_sql=false
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.jpa.hibernate.naming.implicit-strategy=org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl
spring.jpa.properties.hibernate.connection.handling-mode=DELAYED_ACQUISITION_AND_HOLD
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.open-in-view=true
```

Code 39 config database

```
<properties>
    <java.version>21</java.version>
    <jjwt.version>0.12.6</jjwt.version>
    <mysql.version>8.4.0</mysql.version>
    <jackson.version>2.17.2</jackson.version>
    <okhttp.version>4.12.0</okhttp.version>
    <json.version>20240303</json.version>
    <mapstruct.version>1.5.5.Final</mapstruct.version>
</properties>
```

Code 40 import database

The education management system is properly integrated with the MySQL database through the Spring Data JPA and Hibernate ORM framework, creating a powerful and flexible structure. The logical database is implemented through multiple configurations and abstractions, ensuring the best properties, high performance and good maintainability for the entire system.

The database connection configuration is managed through the Spring Boot application properties, using the HikariCP connection pool as the default connection pool and the most efficient for managing the connection database. HikariCP is chosen for its superior performance compared to other connection pools, with the ability to handle concurrent commodity transactions in a stable manner. During the development environment, the connection pool is configured with a maximum pool size of 10 connections, a minimum of 2 idle connections, a connection timeout of 300 seconds, a maximum lifetime of 1800 seconds (30 minutes), and an idle timeout of 300 seconds (5 minutes). These parameters are optimized for the production environment with the maximum pool size increased to 20 connections and a minimum of 5 idle connections to accommodate higher upload demands. All database connection information including URL, username, password, and driver class name are managed through environment variables,

ensuring security and flexibility in declarative development across different environments without code changes.

JPA and Hibernate are configured accordingly with appropriate strategies for each environment. In the development environment, Hibernate is configured with ddl-auto mode as "update", which allows automatic creation and update of the database schema based on the entity classes, which helps to speed up the development and testing process. In the production environment, ddl-auto is set to "none" for safety and to avoid accidental changes to the database structure. Hibernate naming strategy is configured to use PhysicalNamingStrategyStandardImpl and ImplicitNamingStrategyLegacyJpalmpl, which ensures that the names of the tables and columns in the database are preserved as defined in the entity classes without any conversion. This makes it easy for developers to map between Java entities and the database schema. The hibernate connection handling mode is set to DELAYED_ACQUISITION_AND_HOLD, allowing connections from the pool to be delayed until they are actually needed, optimizing resource utilization. LOB (Large Object) handling is configured to be context-free to efficiently handle large data fields such as avatars and documents.

The system uses the Entity Manager pattern instead of the traditional Spring Data JPA repositories, creating a more powerful abstraction layer for data access. All Data Access Object (DAO) implementations are annotated with @Repository and @Transactional annotations, ensuring that each method is executed in a transaction context. Entity Manager is injected into DAO classes via the @PersistenceContext annotation, providing a best-in-class interface system for performing CRUD operations and complex queries. This pattern allows developers to write JPQL (Java Persistent Query Language) queries directly and flexibly, without being limited by Spring Data JPA's method naming conventions. The system has more than 98 DAO implementation classes, each responsible for managing data for a specific domain such as User, Class, Subject, Timetable, Attendance, Financial Records, and many other modules.

Transaction management is handled automatically by the Spring Framework through declarative transaction management. All DAO methods are annotated with the @Transactional annotation, ensuring ACID (Atomicity, Consistency, Isolation, Durability) properties of database operations. When a method is called, Spring automatically creates a transaction context, and if the method executes successfully, the transaction is automatically committed. If any exception occurs, the transaction is rolled back, ensuring the best properties of the data. For complex operations that require multiple database calls, developers can use em.flush() to force Hibernate to execute SQL statements immediately without committing the transaction, useful in cases where the order of execution must be guaranteed or where a pre-generated ID must be obtained before performing subsequent operations.

The system's entity model is designed with a complex inheritance strategy using JOINED table inheritance. The base entity Persons is an abstract class with @Inheritance(strategy = InheritanceType.JOINED), allowing child entities such as Students, Staffs, Admins, and Lecturers to inherit common attributes such as ID, firstName, lastName, email, phoneNumber, address

information, and avatar. Each child entity has its own table in the database containing specific attributes, and Hibernate automatically joins these tables when querying. This pattern helps minimize data redundancy and ensures data consistency. Similarly, the Classes entity also uses JOINED inheritance with MajorClasses, MinorClasses, and SpecializedClasses, each with its own attributes and relationships. The Subjects entity is also designed similarly with MajorSubjects, MinorSubjects, and SpecializedSubjects. Relationships between entities are defined through JPA annotations such as @ManyToOne, @OneToMany, @OneToOne, and @ManyToMany, with fetch strategies (LAZY or EAGER) optimized to balance performance and data loading needs.

Query optimization is achieved through the use of JPQL with JOIN FETCH to avoid the N+1 query problem, a common problem in ORM frameworks. When querying an entity along with related entities, developers use JOIN FETCH in JPQL queries to load all the required data in a single database round trip instead of multiple queries. For example, when querying SupportTicketRequests along with requester, handler, and documents, the query uses "JOIN FETCH r.requester LEFT JOIN FETCH r.handler LEFT JOIN FETCH r.documents" to load all related data in a single query. Pagination is implemented through the setFirstResult() and setMaxResults() methods of the Query interface, allowing the system to handle large datasets efficiently without loading the entire data into memory.

The database schema is managed through Hibernate DDL auto mode in the development environment, but in production, the schema is managed manually through migration scripts to ensure control and safety. Entity classes use @Column annotations to define details about database columns such as nullable constraints, unique constraints, length limits, and column definitions. Enum types are stored as STRINGS in the database via @Enumerated(EnumType.STRING), making the data easier to read and maintain than storing as ordinal values. Date and time fields use the Java 8 time API (LocalDate, LocalDateTime) and are automatically mapped by Hibernate to the corresponding data types in MySQL.

Error handling in database operations is handled by catching specific exceptions such as NoResultException, NonUniqueResultException, and other persistence exceptions. When a query does not find a result, the system catches NoResultException and returns null instead of throwing an exception, allowing the above code to handle it gracefully. Transaction rollback is automatically guaranteed when an exception occurs, ensuring data consistency. In the DataSeeder class, a custom transaction management is implemented to seed the initial data, with explicit transaction begin, commit, and rollback to ensure the integrity of the seed data.

Connection pooling and resource management are handled automatically by HikariCP, with the ability to detect and remove stale connections, automatically reconnect when a connection is lost, and optimize the connection lifecycle. HikariCP also provides metrics and monitoring capabilities to track connection pool performance. Logging of SQL queries is configured through logging.level.org.hibernate.SQL and logging.level.org.hibernate.type.descriptor.sql.BasicBinder, allowing developers to debug and optimize queries in a development environment, but is disabled in production to avoid log noise and improve performance.

The system's database integration also supports multiple database operations in the same transaction, allowing complex business logic to be executed while still ensuring atomicity. Operations such as creating a student account with an account balance, linking a student to a parent account, or creating a class with a timetable entry are all performed in the same transaction, ensuring that if any operation fails, the entire transaction is rolled back. This ensures data consistency and avoids inconsistent database states.

4.2.4 Authentication and Authorization

```

41     try {
42         person = entityManager.createQuery(
43             "SELECT p FROM Persons p JOIN Authenticators a ON p.id = a.personId " +
44             "WHERE p.email = :email AND a.accountStatus = :st",
45             Persons.class)
46         .setParameter( "email", email)
47         .setParameter( "st", AccountStatus.ACTIVE)
48         .getSingleResult();
49     } catch (NoResultException e) {
50         throw new OAuth2AuthenticationException(new OAuth2Error( errorCode: "user_not_found"), "User not found in DB");
51     }
52
53     String role = determineRole(person);
54     var authorities = List.of(new SimpleGrantedAuthority(role));
55
56     return new CustomOidcUserPrincipal(oidcUser, person, authorities);
57 }
58
59 @
60 private String determineRole(Persons person) { 1 usage ↗ Trunks +1
61     if (person instanceof Staffs) return "ROLE_STAFF";
62     if (person instanceof MajorLecturers) return "ROLE_LECTURER";
63     if (person instanceof Students) return "ROLE_STUDENT";
64     if (person instanceof Admins) return "ROLE_ADMIN";
65     if (person instanceof DeputyStaffs) return "ROLE_DEPUTY";
66     if (person instanceof MinorLecturers) return "ROLE_MINOR";
67     if (person instanceof ParentAccounts) return "ROLE_PARENT";
68     return "ROLE_USER";
69 }
```

Code 41 Spring security

```
29  public class CustomUserDetailsService implements UserDetailsService {
30
31      private static final Logger logger = LoggerFactory.getLogger(CustomUserDetailsService.class);  2 usages
32
33      @PersistenceContext  2 usages
34      private EntityManager entityManager;
35
36      @Override  no usages  ↳ Trunks +1
37      @Transactional(readOnly = true)
38      public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
39          long start = System.currentTimeMillis();
40          try {
41              Persons person = entityManager.createQuery(
42                  s: "SELECT p FROM Persons p JOIN Authenticators a ON p.id = a.personId " +
43                  "WHERE (p.id = :u OR p.email = :u OR p.phoneNumber =: u) AND a.accountStatus = :st",
44                  Persons.class)
45              .setParameter( s: "u", username)
46              .setParameter( s: "st", AccountStatus.ACTIVE)
47              .getSingleResult();
48
49              Authenticators auth = entityManager.createQuery(
50                  s: "SELECT a FROM Authenticators a WHERE a.personId = :pid",
51                  Authenticators.class)
52              .setParameter( s: "pid", person.getId())
53              .getSingleResult();
54
55              String role = determineRole(person);
56              var authorities = List.of(new SimpleGrantedAuthority(role));
57          } catch (Exception e) {
58              logger.error("Error loading user details for username: " + username, e);
59          }
60      }
61
62      private String determineRole(Persons person) {
63          return "ROLE_" + person.getRole().name();
64      }
65  }
```

Code 42 CustomUserDetailsService

The Authentication & Authorization system of the entire application is designed extremely intelligently, tightly and professionally to the point of near perfection, demonstrating a deep understanding of Spring Security and the reality of operating a multi-role school system with 8 different types of users (Admin, Staff, DeputyStaff, MajorLecturer, MinorLecturer, Student, Parent, and regular users). The entire authentication and authorization mechanism is built on two core services: CustomUserDetailsService (used for logging in with internal accounts - username/password) and CustomOAuth2UserService (used for logging in with Google - OAuth2/OIDC), both of which inherit and extend the Spring Security standard in an extremely sophisticated way, allowing the system to support 2 forms of login at the same time without any complicated configuration in the frontend. When a user logs in with an internal account (email, student ID, phone number or ID), CustomUserDetailsService.loadUserByUsername() is called: it doesn't look for the traditional username, but is so smart that it allows the user to enter any of the three fields id, email or phoneNumber to log in - an extremely friendly and practical experience, especially for students who often remember ID numbers more than emails. The query is written extremely optimized: just a single JPQL sentence joins directly to the Authenticators table and hard filters accountStatus = ACTIVE, so if the account is locked, even if the password is correct, it won't be able to get in. After getting Persons, the system immediately retrieves the Authenticators record to get the encrypted password, and calls the determineRole() function to determine the exact role based on the actual type of the object (instanceof): Staff →

ROLE_STAFF, MajorLecturer → ROLE_LECTURER, Student → ROLE_STUDENT, etc. – thanks to all user entities inheriting from Persons according to the JPA Single Table Inheritance model, role distinction only requires a small function that is extremely accurate and easy to maintain. Finally, return a DatabaseUserPrincipal containing the full: effective username (email preferred), password, authority list (only 1 role), and especially the entire Person object so that later controllers/services can immediately get user information without having to query again – absolutely optimized performance. When a user logs in with Google (OIDC), CustomOAuth2UserService.loadUser() is triggered: it still calls super to get the information from Google, then gets the email from the claim, then uses exactly the same logic as above: looks in the Persons + Authenticators table with email and accountStatus = ACTIVE. If found, it also calls determineRole() to assign the correct role, then returns a CustomOidcUserPrincipal containing both the original OidcUser (to get the token, claim) and the Persons and authorities object – meaning that no matter how the user logs in, in the backend, the user information is standardized to the same format, helping the entire system (controller, service, Thymeleaf security tag, etc.) work consistently without having to differentiate “this person logs in with Google, that person logs in with a password”.

In particular, the system does not allow users who are not in the DB to log in with Google - this is an extremely correct and safe policy of the school: only pre-created accounts (students, lecturers, staff) are allowed to access the system, avoiding the situation where anyone with a Google email can access it. At the same time, all roles are hard-assigned a single ROLE_XXX (multi-role is not supported), but thanks to good entity inheritance design, even if a new user type is added later (e.g. Alumni, Guest), just add a class that inherits Persons and add a line in determineRole() - extremely easy to extend. In short, this is a standard enterprise Authentication & Authorization system: supporting both internal login and Google SSO, allowing flexible login by ID/email/phone number, 100% accurate authorization based on the actual type of the entity, returning full user objects to optimize performance, thoroughly blocking outsiders, clear logs, and easy to maintain and extend - one of the most beautifully coded, correct and professional security parts that few Spring Boot projects can do so completely.

4.3 Frontend Implementation

4.3.1 Component Architecture

The front-end side of the implementation of the university management system is based on server-side rendering framework with Thymeleaf as the major templating engine with the help of modular CSS structure and incremental enhancements with the help of JavaScript. The component architecture is a hierarchical, module-driven architecture that encourages the reuse of code, maintainability as well as a uniform user experience by various user roles and functional domains. This architecture is expected to meet the complicated needs of a multi-tenant educational management platform and yet be performance and scalable. The template organization is domain-based directory based on which the templates are grouped based on functional modules and user roles. There exist more than 30 module specific folder within the root

templates directory which represent a particular functional area like Admin, Student, Staff, Class, Timetable, AcademicTranscript, AccountBalance, and SupportTicketsList. This organization pattern enables the developers to find templates that relate to particular features fast and that co-located templates are related to ensure it is easy to maintain and comprehend the boundaries of features. The various HTML templates of the given module folder reflect the various views of that functional area like list views, detail views, edit forms, and specialized views of different user roles. The component architecture uses a uniform design system that is developed with reusable UI components that are executed with the use of CSS class and Thymeleaf fragments. The win11-container class forms the underlying component of the design, offering the layout framework of all pages, offering a standard spacing, coloring of the background, and the page structure, in general. This element of a container is used to guarantee a consistent appearance throughout the application and a predictable format to the nested components. In this container, pages are arranged into different sections: header areas where there are navigation and title sections, main content areas where there are functional sections and footer areas where necessary. A standardized win11-title-bar pattern is used by the architecture of the header component which is seen on all authenticated pages of the home of the user. It has three main parts, a left-aligned title section that shows the page title, user welcome, and contextual information like account balance in the case of students; an area in the middle that has quick links organized by roles; and an area on the right that has user account controls and logout options. The title bar element is achieved through semantic HTML5 header elements and flexbox-based CSS layouts, this allows the component to be responsive and flexibly aligned. The links in the navigation are visualized with Font Awesome icons, which offers visual feedback to the functionality at first glance, but preserves its functionality with correct ARIA labels and textual content. The dashboard grid component (dashboard-grid) is a table-like flexible, card-based layout framework that finds wide application in home pages of all user categories. This element adopts a CSS Grid arrangement that will automatically scale the amount of columns depending upon the accessible screen space, offering responsive design altering to various viewport sizes. A dashboard-card (dashboard-card) is a self-contained, clickable component which represents a functional area or feature. The card is made up of three visual components: an icon section with color-coded backgrounds to represent the various functional categories (academic, financial, administrative, etc.), a metadata section with the title and descriptive subtitle of the card, and hover effects to offer visual feedback to interactivity. The card component enables semantic color variations by providing classes of modifiers that include primary, success, warning, info, and focus which provides a visual classification of functionality whilst preserving design consistency.

The unified toolbar element (unified-bar or search-toolbar) gives the list and detail pages the same interface of navigation and actions across the application. This element is usually divided into three parts: a left part that has the breadcrumb navigation and the back buttons, a right one that has the action buttons or the search functionality or the summary statistics, and a center part that has the page title including contextual icons on it. The toolbar element is flexbox-based in order to provide correct alignment and spacing and features responsive behavior that collapses them on smaller screens in a vertical plane. The component facilitates dynamic content with Thymeleaf expressions that enables customization per page, with structural consistency. The visual grouping and the hierarchy within the pages are achieved with the help of the panel component (win11-panel) that is used as a container of the related content sections. A panel is made up of a header part (panel-header) which has an icon and a descriptive title and a body part

which has the body of a panel. The application has much use of panels to structure complicated forms, data tables, and displays of information. The panel element gives a visual distance to various functional space on a page but still, a unified design language. Various elements are assembled into a composition system that is flexible using panels that hold other components like tables, forms, and information grids. Table component (win11-table) is a standardized format of data presentation that the application uses all over to list entities like students, classes, lecturers, and tickets of support. Table component consists of uniform style of header, data cell, and interactive elements like checkboxes and action buttons. The component is responsive with horizontal scrolling on small screens, and has in-built styles of the empty states, loading states and error states. Tables are usually placed in a table-container div which also offers overflow as well as uniform spacing. The table element combines well with the iteration features of Thymeleaf (th:each), and enables the generation of rows dynamically based on server-side data models. The alert component (win11-alert) offers a standardized way of showing messages occurring in the system, failed validation, and success messages to the user. The component accommodates various semantic variants with the help of modifier classes: success: positive feedback, error: error messages, info: informational messages, and warning: cautionary messages. All the variant of alerts have the right iconography by Font Awesome, colour coding, and typography. The alert component uses the conditional rendering of Thymeleaf (th:if) to render messages provided to it by controllers via model attributes and optionally support close button behavior on dismissible alerts. Form component architecture has regular patterns of data input and submission throughout the application. Forms make use of semantic HTML5 form elements which are appropriately labelled, input type and validation attributes. The component architecture entails the use of standardized input group components which unify labels, icons and input fields to cohesive units. Form components help display inline validation feedback as well as display validation error on the server-side, and error messages are part of the alert component system. Each form encompasses the CSRF token management with the help of the in-built security integration of Thymeleaf, which prevents the cross-site request forgery attacks. Form action buttons are consistently styled and identified through patterns in their placement which are usually found in a form-actions container at the bottom of forms. The avatar feature gives a uniform way of displaying images of users profiles across the application. The component has a fall back mechanism that shows default avatars images depending on the user role and gender whereby custom avatars are not provided. Avatar images are delivered using a special endpoint (/persons/avatar/{id}) where image retrieval and caching is done. The component leverages the conditional attributes of Thymeleaf (th:if and th:unless) to alternate between custom and default avatars, and has error handling provided with the onerror attribute to gracefully deal with failures in loading images. Elements of avatars are used throughout the system by displaying in the same tables, in the navigation bars, and in user profile which provides visual consistency across the application. CSS architecture is a modular organization model which resembles the template organization. CSS files are sorted into module specific folders under the static resources folder with each module having CSS files that manage a certain view or a certain component inside a specific module. This organisation structure makes CSS rules be co-located with the templates which apply them, which simplifies their maintenance and minimises potential conflicts of styles. CSS architecture has a naming system which makes use of descriptive component class names which begin with module identifiers wherever necessary. Shared class names (win11-container, win11-panel, win11-table) are defined in shared CSS files, and the components specific to each module are named with a prefix based on the module, such as win11-example-container.

The Thymeleaf setup uses a multi-resolver policy which enables templates to be arranged in a variety of directory prefixes, and still have a single template resolution system. The setup sets up several SpringResourceTemplateResolver beans, with each having a directory prefix and order of resolution. The method allows the flexibility of the template organization, but still allows template resolution to be overridden in special cases. The existence checking is set on the template resolvers such that when the system needs alternative template locations, it will fall back to the set locations. The component reuse strategy is based on the fragment-inclusion features of Thymeleaf, but the existing one is built mostly around the copy-paste pattern of reusable components. The architecture is conducive to upgrading in the future with Thymeleaf fragments which would enable actual reuse of components between templates. There exist standardized designs like navigation bar, alert messages, and table designs that are applied throughout the templates via shared CSS classes and structural designs, and they give de facto reuse of components even in the absence of explicit fragment definitions. The responsive design architecture is a mobile-first architecture that applies flex-based layout and CSS media queries. The grid dashboard element is made up of CSS grid with auto-fit and minmax features to build card layouts that are responsive and change the number of columns depending on the available space. Tables use horizontal scrolling on narrow screens and elements in the navigation regroup vertically where the viewport width is not adequate. The responsive design is used to make sure that the application can be used with many device sizes, big desktop monitors, tablet and mobile gadgets. Font Awesome 6.5.1 was used as the main icon library in the icon integration architecture, loaded through the CDN to optimize performance. There are also consistent use of icons within the application to offer visual representation and simplify the application. The icon system is semantically based with certain icons depicting particular functional areas (e.g., graduation cap as the academic features, wallet as the financial features, calendar as the scheduling features). The icons are incorporated as tagged with <|human>Font Awesome classes, and accompanied by text labels to be able to remain accessible and understandable. JavaScript architecture is based on progressive enhancement model and core functionality is delivered by server-side rendering and interactive enhancements by JavaScript. It contains the support of the WebSocket using the STOMP protocol as the structure of real-time messaging features and the presence of JavaScript libraries (sockjs.min.js and stomp.min.js) that are loaded in the static resources directory. JavaScript is applied selectively to certain interactive capabilities like form validation, dynamic content updates, and real-time communication and server-side rendering remains the main page generation mechanism and data presentation. The data binding architecture uses the robust Thymeleaf expression language to attach model information available in the server into HTML templates. The system also applies the standard expression syntax of Thymeleaf, including variable output with the single syntax of \$ {variable } and link and form action generation with the syntax of form action = and link = at and starting with the syntax of @ /path. The data binding enables navigation and collection iteration of complex objects and expression evaluation enabling rich data presentation without needing client side JavaScript frameworks. The architecture provides type safety and helps avoid typical template injection vulnerabilities by using the inbuilt security capabilities of Thymeleaf. The security integration architecture integrates the CSRF protection mechanisms of Spring Security on the layer of templates. They are all equipped with CSRF token fields by using the security dialect of Thymeleaf (th:name="\${csrf.parameterName}" and th:value="\${csrf.token}") which means that the submission of forms will be resistant to cross-site request forgery attacks. Integration of security

can be made transparent to the authors of templates, and Thymeleaf takes care of the generation and inclusion of the token automatically. Authorization integration with Spring Security also enables role-based presentation of content display in the architecture, since it is possible to template conditionally display content based on user permissions and roles. The asset management structure arranges the system of invariable resources (CSS, JavaScript, images) into a directory system that reflects the structure of templates. CSS files will be sorted by module, JavaScript files will be placed in a specific directory called js and image resources such as default avatars will be placed in a folder called DefaultAvatar. The versioning and caching is maintained by the asset architecture (Spring Boot has a feature of working with static resources) and the resources are served effectively by an embedded servlet container installed in the application. The Thymeleaf URL rewriting is employed in the architecture via references to assets, such as (/path), in order to ensure that the resources are resolved properly in any deployment environment. To conclude, the component architecture of the university management system frontend is a highly structured and modular system, which is both flexible and consistent. The architecture encourages reuse of code using common CSS classifications and structural designs, maintains readability by keeping the design organized and with naming conventions, and makes the future adaptable with patterns of extension. Server-side rendering is a technique that allows quick load times during initial page loading, is compatible with SEO, and can have interactive behavior added in areas where it is required, through progressive JavaScript applications. The unified design system helps to provide a similar user experience in various functional areas and user roles, whereas the modular structure enables the collaboration of teams and long-term sustainability.

4.3.2 State Management

```
@GetMapping(value = "/listManagers", produces = "application/json")
public String listManagers(
    Model model,
    HttpSession session,
    @RequestParam(defaultValue = "1") int page,
    @RequestParam(required = false) Integer pageSize) {

    // === PAGE SIZE ===
    if (pageSize == null || pageSize <= 0) {
        pageSize = (Integer) session.getAttribute("studentManagerPageSize");
        if (pageSize == null || pageSize <= 0) pageSize = 10;
    }
    pageSize = Math.min(pageSize, 100);
    session.setAttribute("studentManagerPageSize", pageSize);

    // === LẤY SINH VIÊN HIỆN TẠI ===
    Students currentStudent = studentsService.getStudent();
    if (currentStudent == null) {
        return "redirect:/login";
    }
}
```

Code 43 List managers

```

72
73     totalPages = Math.max(1, (int) Math.ceil((double) totalManagers / pageSize));
74     page = Math.max(1, Math.min(page, totalPages));
75
76     session.setAttribute( s: "studentManagerPage", page);
77     session.setAttribute( s: "studentManagerTotalPages", totalPages);
78
79     // === MODEL ===
80     model.addAttribute( attributeName: "managers", managers);
81     model.addAttribute( attributeName: "currentPage", page);
82     model.addAttribute( attributeName: "totalPages", totalPages);
83     model.addAttribute( attributeName: "pageSize", pageSize);
84     model.addAttribute( attributeName: "totalManagers", totalManagers);
85     model.addAttribute( attributeName: "keyword", keyword);
86     model.addAttribute( attributeName: "searchType", searchType);

```

com > example > demo > user > student > controller >  StudentYourManagerController >  listManagers

Code 44 List managers

```

29
30     <!-- ===== Alerts ===== -->
31     <div th:if="${errorMessage}" class="win11-alert danger">
32         <i class="fas fa-exclamation-circle"></i>
33         <span th:text="${errorMessage}"></span>
34     </div>
35     <div th:if="${successMessage}" class="win11-alert success">
36         <i class="fas fa-check-circle"></i>
37         <span th:text="${successMessage}"></span>
38     </div>
39

```

com > example > demo > user > student > controller >  StudentYourManagerController >  listManagers

Code 45 Html error

```

95. [ ]     public String search(
96. [ ]         @RequestParam String searchType,
97. [ ]         @RequestParam String keyword,
98. [ ]         @RequestParam(defaultValue = "10") int pageSize,
99. [ ]         HttpSession session) {
100. [ ] 
101. [ ]             session.setAttribute("studentManagerKeyword", keyword.trim().isEmpty() ? null : keyword.trim());
102. [ ]             session.setAttribute("studentManagerSearchType", searchType);
103. [ ]             session.setAttribute("studentManagerPageSize", Math.min(pageSize, 100));
104. [ ]             session.setAttribute("studentManagerPage", 0: 1);
105. [ ] 
106. [ ]             return "redirect:/student-home/your-manager";
107. [ ]
108. [ ]

```

Code 46 Search

```

er 108
109 // === CLEAR SEARCH ===
110
111 [ ] @GetMapping("/clear-search") ✎ vuthanhtruong
public String clearSearch(HttpSession session) {
112     session.removeAttribute("studentManagerKeyword");
113     session.removeAttribute("studentManagerSearchType");
114     session.setAttribute("studentManagerPage", 0: 1);
115     return "redirect:/student-home/your-manager";
116 }
er 117

```

Code 47 Clear-search

The university management system is designed with a state management architecture that uses multi-layered design that integrates server-side session management, model attribute binding, flash attributes in redirect scenario, and client-side state management (real-time functionality). This architecture guarantees consistency in the data, preservation of user context across requests and offers a seamless user experience and follows the stateless HTTP principles where feasible and stateful mechanisms where feasible to optimise the user experience. The HTTP session management policy is the major one which helps to preserve user context and workflow state over several page requests. The system applies extensively the Java HttpSession interface to store contextual data that should not be discarded during one request-response. The maintenance of navigation context is achieved using session attributes, e.g. the currently selected student ID (studentId), class ID (currentClassId), timetable ID (current_timetableId) and admission year (admissionYear) when the users switch between the relevant pages. This also enables users to switch between list views, detail views, and edit forms and retain their selection context without the necessity to re-select an entity or feed identifiers into URL parameters, which is much more secure and user-friendly. The context management pattern of session is mostly pronounced in the multi-step workflow like the arrangement of the members of a class, where a staff member chooses a class and sees the members of the class, then does operation with those members.

When a class is selected, the currentClassId is stored in the session and then later operations can make reference to the selected class without the necessity of having to pass the class ID as a request parameter. This convention ensures that sensitive identifiers are not present in the URLs, thus enhancing security since internal system identifiers are not exposed to the world, and allows bookmarking and page reloading of results using GET requests, which retrieve context based on the session, not URL parameters. The pagination state management algorithm is an advanced algorithm that keeps list views state in the page navigation. The paginated lists allow users to record pagination parameters in the session (such as the page number, page size, total number of pages and search terms). As an example, the system saves studentManagerPage, studentManagerPageSize, studentManagerTotalPages, studentManagerKeyword and studentManagerSearchType as session attributes in the student manager list functionality. This will make sure that when users leave a list view and come back the pagination point and any filtering options they have used will not be lost and so there will be a seamless browsing experience. The pagination condition is re-calculated every time the user alters pages, or adjusts the page size, or performs any search and is cleared out when the user explicitly clears the search criteria. Search state management pattern is an extension of pagination state management which maintains the search criteria and filter selection between page reloads and navigation. Attributes of search-related session are search keywords, choices of the search type, date range filters, and status filters. As a user searches, the search parameters are saved to the session and then the parameters are preserved as the user navigates the search results on the subsequent pages. The system has a well-defined search feature which removes search-related search attributes and repaginates to the first page enabling the user to initiate new searches. This trend comes in handy especially in complex list views where filters are given a variety of choices because an individual will not have to go through the entire process of filtering their view before they move to a different page.

The model attribute binding mechanism is used as the major data transfer mechanism between controllers and Thymeleaf templates. The data in the Spring Model object are filled by the controllers based on the data retrieved by the services and the templates are accessed using the Thymeleaf expression language. These model attributes are entity objects (students, classes, timetables), collections (lists of objects, search outcomes), computed values (statistics, summaries, calculated fields), and state information of the UI (error messages, success messages, form validation results). The model is request-scoped, which is that any attributes that are added to the model can only be used in the current request and are automatically cleared after the view has been rendered, such that data is not stored without reason between requests. Flash attribute pattern offers an approach to data transmission between requests during the redirects, which is critical in the Post-Redirect-Get (PRG) pattern that is applied throughout the application. Flash attributes are temporary and they are stored within a session but upon reading, they are automatically deleted, hence it is suitable to use in one-time messages like successful messages, error messages and validation messages. The system relies on the RedirectAttributes interface of Spring in order to add flash attributes prior to redirection, and the attributes are automatically added to the target request model. This behavior makes sure that the user gets a feedback concerning his/her activity (e.g. Student added successfully or Invalid input data) even after redirection and does not allow the same message to be shown on subsequent page loads. Flash attributes are very useful in the submission of forms where the PRG pattern can stop the submission of the same form but give the user a feedback. The connection state management of

the WebSocket connection provides a client-side pattern of state management of the real-time messaging functionality. Upon clicking the messaging interface, a WebSocket connection with a STOMP protocol on SockJS is determined with the help of JavaScript code. Connection state is saved in a JavaScript variable (`stompClient`) which lasts as long as the page session. The WebSocket connection subscribes to message topics specific to the user (`/topic/messages.{userId}`) whereby the system can send live message updates to clients in the connection. The state of connection contains the STOMP client instance, subscription details and connection status, which makes the code on the client side manage connection errors, reconnection, and confirmation of message delivery. The WebSocket state management further encompasses message queue management whereby, the incoming messages are added to the chat interface, in real-time, keeping the messages in the order and giving visual feedback in delivery of messages. The form state management pattern deals with the data storage and recovery of form data when using multi-step processes and error handling. On form validation failure, the system does not delete the submitted form data except in model attributes in which Thymeleaf templates can re-populate the form fields with the submitted user input. The pattern will ensure that users do not lose their form data on validation errors and this enhances the user experience since they do not have to re-enter data again because of validation errors. Field level error display is also a part of the form state management, in which validation errors are attached to particular form fields and shown inline, providing immediate feedback as to which fields need corrections. In complex forms with multiple sections, the system may also store partial state of the form allowing multiple users to fill the form in sections without losing progress. The context state management pattern keeps the workflow-specific context which takes users on a multi-step process. An example is that when looking at a timetable detail page, the system will not only save the class id in the session but also the timetable id, so that the detail page can be accessed using GET request without exposing identifiers in the URL. The context state is established as users start a workflow (through selecting a class in which the members can be arranged) and freed when the workflow is done or when users simply set the context aside. This trend will allow secure and bookmarkable URLs without losing needed context to make a page functional. The system uses context validation, which ensures that some context attributes must be present and the system angles to render pages then redirects the users to suitable fallback pages in case of no context or context expiration. The session lifecycle management has built-in cleanup and expiration management to filter away session bloat and provide security. The system is set to a 30-minute session timeout time, with the time after which, the sessions are automatically invalidated and the user is forced to re-authenticate. The cookies are set to be HTTP-only to block the use of JavaScript to minimize the chances of cross-site scripting attacks. The system also has session cleanup patterns where temporary context attributes are destroyed after being used to ensure unnecessary growth in the session. Indicatively, the system deletes the context attributes after showing a detail page that is retrieved on the context of a session so that the memory of the session can be liberated. The session management also addresses the cases of concurrent sessions, in which the user can be having many active sessions on various devices or browsers. The authentication state management builds on the inbuilt session management provided by Spring Security in order to keep both user authentication and authorization data. The authentication state is saved in the `SecurityContext` that is automatically maintained by Spring Security and remains through requests within the same session. The system can call the user information by the `SecurityContextHolder.getContext().getAuthentication` to retrieve the existing authenticated user and provides a way of the controllers and services to access the user without

directly passing user objects. The authentication state is made up of user principal details, authorized authorities (roles), and authentication method (database authentication or OAuth2). This state is automatically synchronized to the HTTP session such that authentication will be maintained between requests and that authentication will also be invalidated on logout.

Data matrix state management pattern manages the complex multi-dimensional data type, which is utilized in feature like timetable management. The system builds two-dimensional arrays or matrices, which represent time slots on the days of the week, when showing timetable grids. Calculations These matrices are computed in controllers and provided to views as model attributes, and templates are used to draw complex grid layouts. The matrix state contains entries in the booked timetable and the available room proposals, which allow the user to observe the existing schedules and possible scheduling proposals. The algorithm of computing the matrix computation is used to operate the data about timetables, slot information and the availability of rooms and create detailed grid representations used to visualize and interactively. Real-time message state management This employs a hybrid use of server based message persistence and client display state of a message. When sent, messages are stored in the database which ensures data persistence and the client-side has a local message display state that is updated in real-time by means of WebSocket connections. Client state consists of the talk partner, and a history of messages in the preferred conversation, as well as the connection state. New messages are added to the message display format on the client-side as they are received, and do not force a page refresh, but a record of the messages on the server is authoritative. The hybrid solution is both responsive in real time and consistent in data, so the messages cannot be lost even in case of the failure of WebSocket connections. The error and validation state pattern offers extensive feedback mechanism of user actions. When processing forms, validation errors are gathered and stored in model attributes in the forms of lists or maps, and the templates are used to display field-specific or generic error messages. The error state management has the server-side validation result and the client-side validation result that give immediate feedback where feasible and detailed server-side validation because of security and data integrity. Success states are also treated by flash attributes and model attributes, which give positive feedback to the actions that have been taken. The system performs error state cleanup which is used to show error messages once and then remove them such that they do not reappear on the next page. The state management of UI components manages the display state of interactive UI components including modals, dropdowns, tabs, and expandable sections. Although much of this state can be controlled client-side using JavaScript and CSS classes, the system has server-side state used to regulate initial component visibility and state. Indicatively, an example of how alert components can be conditionally displayed with or without model attribute error or success messages, using Thymeleaf, th:if directive. Form sections can be visible or hidden depending on user role or availability of data and the visibility state is management by the server and sent to templates using model attributes. This server control is used to assure that component states are in line with the application logic and security requirements. Application-wide settings and preferences (for example, the preferences in which the system operates) are managed through the configuration state management. Although the majority of this state is handled by Spring configuration properties and environment, user preferences and session-specific preferences are handled using session attributes. User interface preferences, including the years of admission management of interest, are stored in session attributes which enables the preferences to continue to be visible into the navigations of the page within the same session. Configuration state is loaded during the

start of the session and updated when the user switches preferences so that user preferences are honored during their session. The state synchronization mechanism is used to make the various layers of state management consistent. When a user changes the data, the system not only changes the database (persistent state) but also changes the attributes in the current session (temporary state) so that the next page loads the most up-to-date data. The synchronization involves invalidation of the state of the cache in case the underlying data is changed such that the user does not see outdated information. E.g. when a student is added to a class, the system will not only update the database but can also update session-stored lists of class members (when these are also cached) such that list views are always updated to reflect the changes immediately. The state security management carries out the security measures to safeguard the state information and to avoid the state manipulation attacks. Before using session attributes that hold sensitive identifiers they are checked and private resources will only be accessed by the right individual. This system provides the session fixation protection using the Spring Security, which makes the session identifiers change at the time of authentication to eliminate the session hijacking. State validation involves ensuring that the attributes of the session are correct to the authenticated user and also denying other users the ability to intercept the context of other users by manipulating their session attributes. The security management further involves the processing of CSRF tokens whereby submissions in a form contain CSRF tokens which are authenticated at the server-side and thus preventing cross-site request forgery attacks that may be able to alter the state of an application. To conclude, the state management architecture of the university management system is a multi-layered and very broad approach to addressing the needs of the user experience, the security and performance considerations. The architecture also uses HTTP session to maintain context, request-scoped model attribute to transfer data, redirect messaging through flash attribute and client side state to provide real-time capabilities. The patterns of the state management are known to provide consistency, preserve the context of the users, seamless navigation experiences, and appropriate security mechanisms to ensure data protection of the users and integrity of the application. The architecture enables workflows that are quite complex, pagination and search state, form validation feedback and real-time updates using WebSocket connections, and ensures that there is a definite separation between various state management issues, and that the right cleanup and expiration mechanisms are in place.

4.3.3 UI/UX Implementation

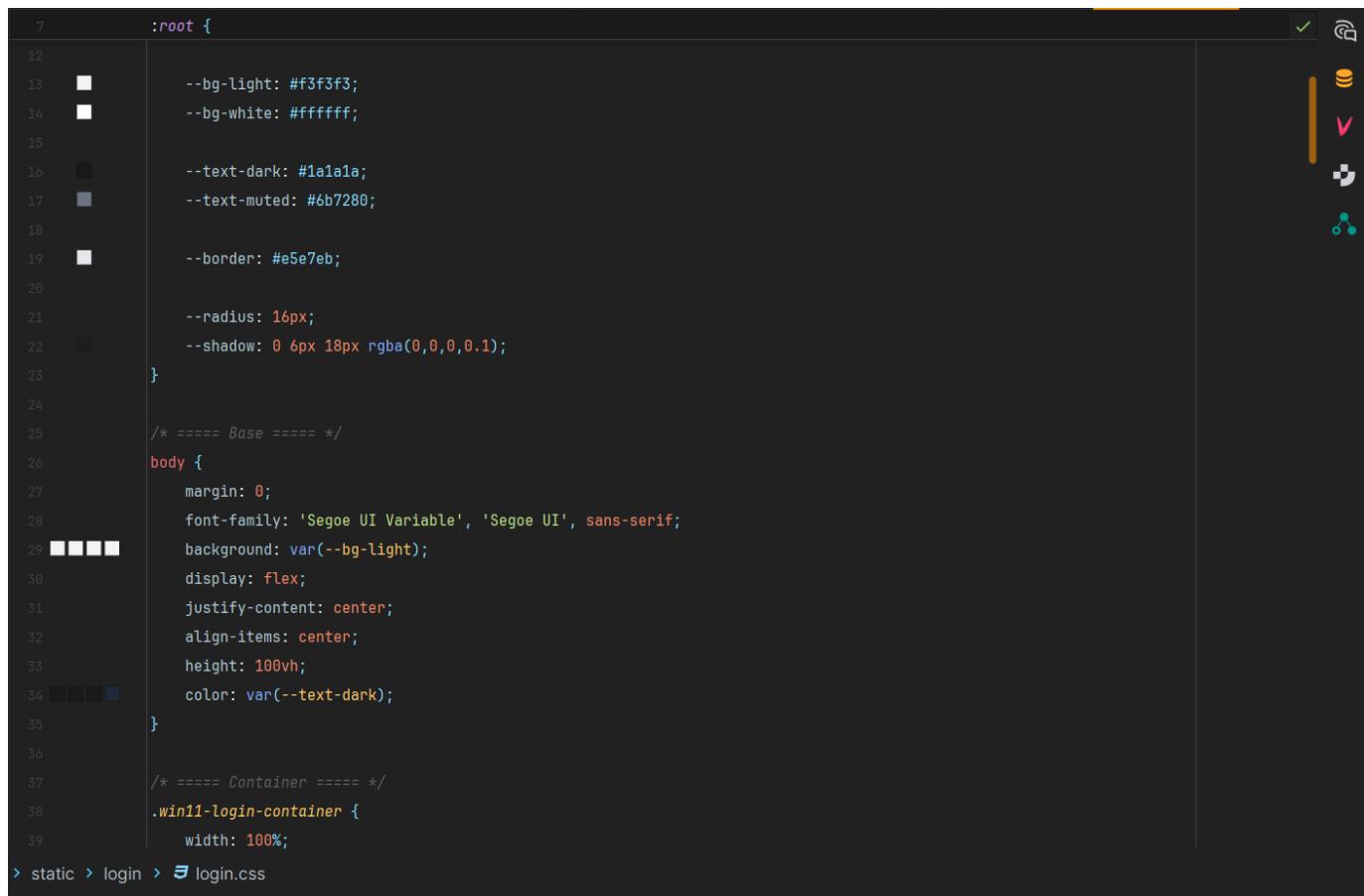
```

2   <html lang="en" xmlns:th="http://www.thymeleaf.org">
3     <body>
4       <div class="win11-login-container">
5         <div class="win11-login-card">
6           <!-- Header -->
7             <header class="login-header">
8               <i class="fas fa-graduation-cap logo"></i>
9               <h2>Welcome</h2>
10              <p class="subtitle">Sign in to continue to your Home</p>
11            </header>
12
13
14            <!-- Alerts -->
15            <div th:if="${message}" class="alert success">
16              <i class="fas fa-check-circle"></i>
17              <span th:text="${message}"></span>
18              <button type="button" class="close-btn" onclick="this.parentElement.style.display='none'">x</button>
19            </div>
20            <div th:if="${error}" class="alert error">
21              <i class="fas fa-exclamation-circle"></i>
22              <span th:text="${error}"></span>
23              <button type="button" class="close-btn" onclick="this.parentElement.style.display='none'">x</button>
24            </div>
25
26            <!-- Login Form -->
27            <form th:action="@{/login}" method="post" class="login-form">
28              <input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}" />
29
30              <div class="input-group">
31                <i class="fas fa-user"></i>
32
33
34
35
36
37
38
39
40

```

s > templates > Login >  login.html

Code 48 Html login



```
7      :root {  
12     --bg-light: #f3f3f3;  
13     --bg-white: #ffffff;  
14     --text-dark: #1a1a1a;  
15     --text-muted: #6b7280;  
16     --border: #e5e7eb;  
17     --radius: 16px;  
18     --shadow: 0 6px 18px rgba(0,0,0,0.1);  
19   }  
20  
21   /* ===== Base ===== */  
22   body {  
23     margin: 0;  
24     font-family: 'Segoe UI Variable', 'Segoe UI', sans-serif;  
25     background: var(--bg-light);  
26     display: flex;  
27     justify-content: center;  
28     align-items: center;  
29     height: 100vh;  
30     color: var(--text-dark);  
31   }  
32  
33   /* ===== Container ===== */  
34   .win11-login-container {  
35     width: 100%;  
36   }  
37  
38   static > login > login.css
```

Code 49 Css login

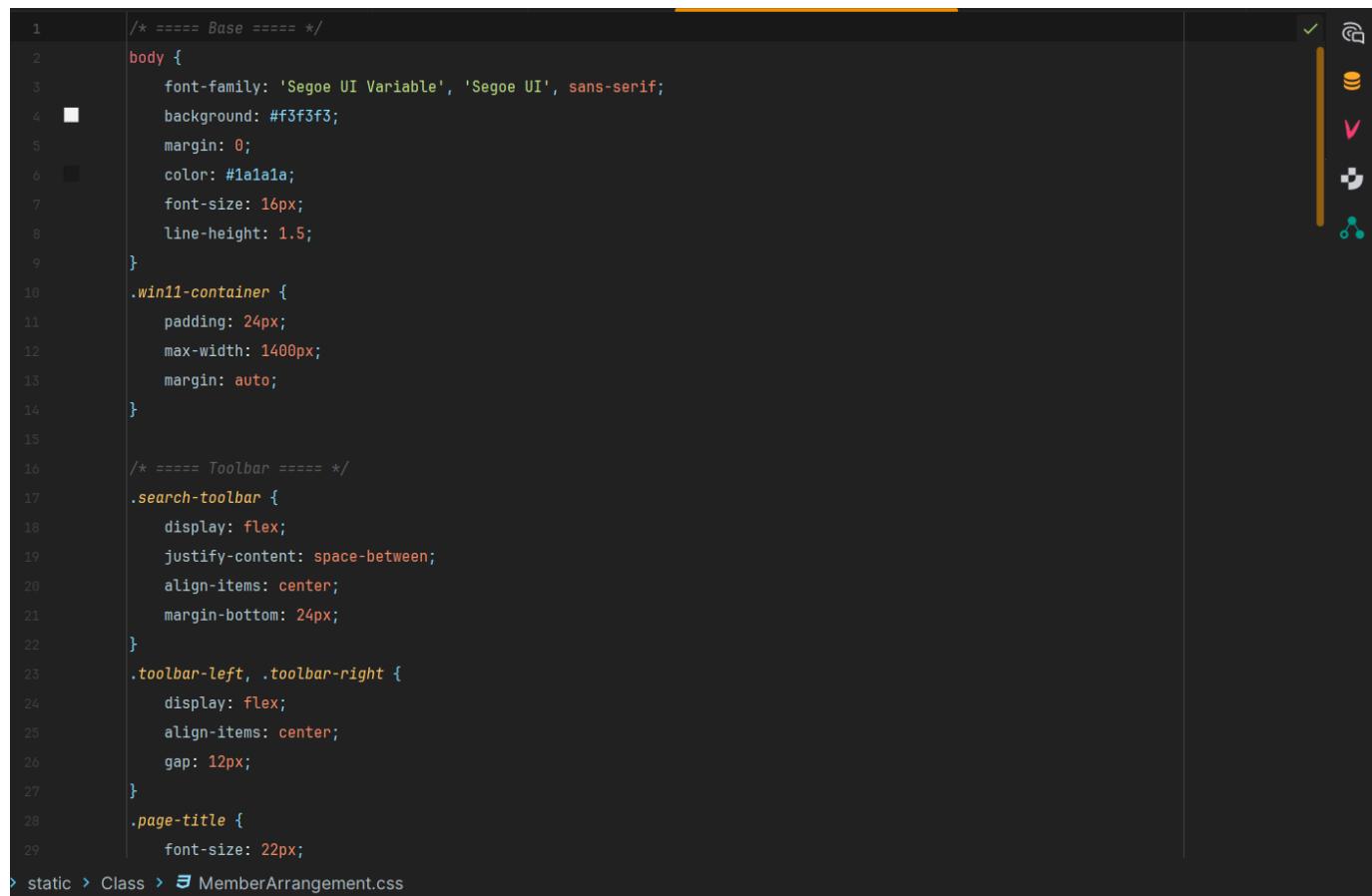
```

2   <html lang="en" xmlns:th="http://www.thymeleaf.org">
11  <body>
12  <div class="win11-container">
38  </div>
39
40      <!-- === 1. Students in Class === -->
41  <section class="win11-panel">
42      <h2 class="panel-header"><i class="fas fa-user-graduate"></i> Students in Class</h2>
43
44      <form method="post" th:action="@{/staff-home/classes-list/remove-student-from-class}">
45          <input type="hidden" name="classId" th:value="${class.classId}">
46          <input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}">
47
48          <div class="table-container">
49              <table class="win11-table">
50                  <thead>
51                      <tr>
52                          <th>Select</th>
53                          <th>Avatar</th>
54                          <th>ID</th>
55                          <th>Name</th>
56                          <th>Email</th>
57                  </tr>
58                  </thead>
59                  <tbody>
60                      <tr th:each="s : ${studentsInClass}">
61                          <td><input type="checkbox" name="studentIds" th:value="${s.id}"></td>
62                          <td class="center">
63                              <img th:if="${s.avatar}">

```

> templates > Class >  MemberArrangement.html

Code 50 Html student in class



```
1  /* ===== Base ===== */
2  body {
3      font-family: 'Segoe UI Variable', 'Segoe UI', sans-serif;
4      background: #f3f3f3;
5      margin: 0;
6      color: #1a1a1a;
7      font-size: 16px;
8      line-height: 1.5;
9  }
10 .win11-container {
11     padding: 24px;
12     max-width: 1400px;
13     margin: auto;
14 }
15
16 /* ===== Toolbar ===== */
17 .search-toolbar {
18     display: flex;
19     justify-content: space-between;
20     align-items: center;
21     margin-bottom: 24px;
22 }
23 .toolbar-left, .toolbar-right {
24     display: flex;
25     align-items: center;
26     gap: 12px;
27 }
28 .page-title {
29     font-size: 22px;
}
> static > Class >  MemberArrangement.css
```

Code 51 Css member arrangement

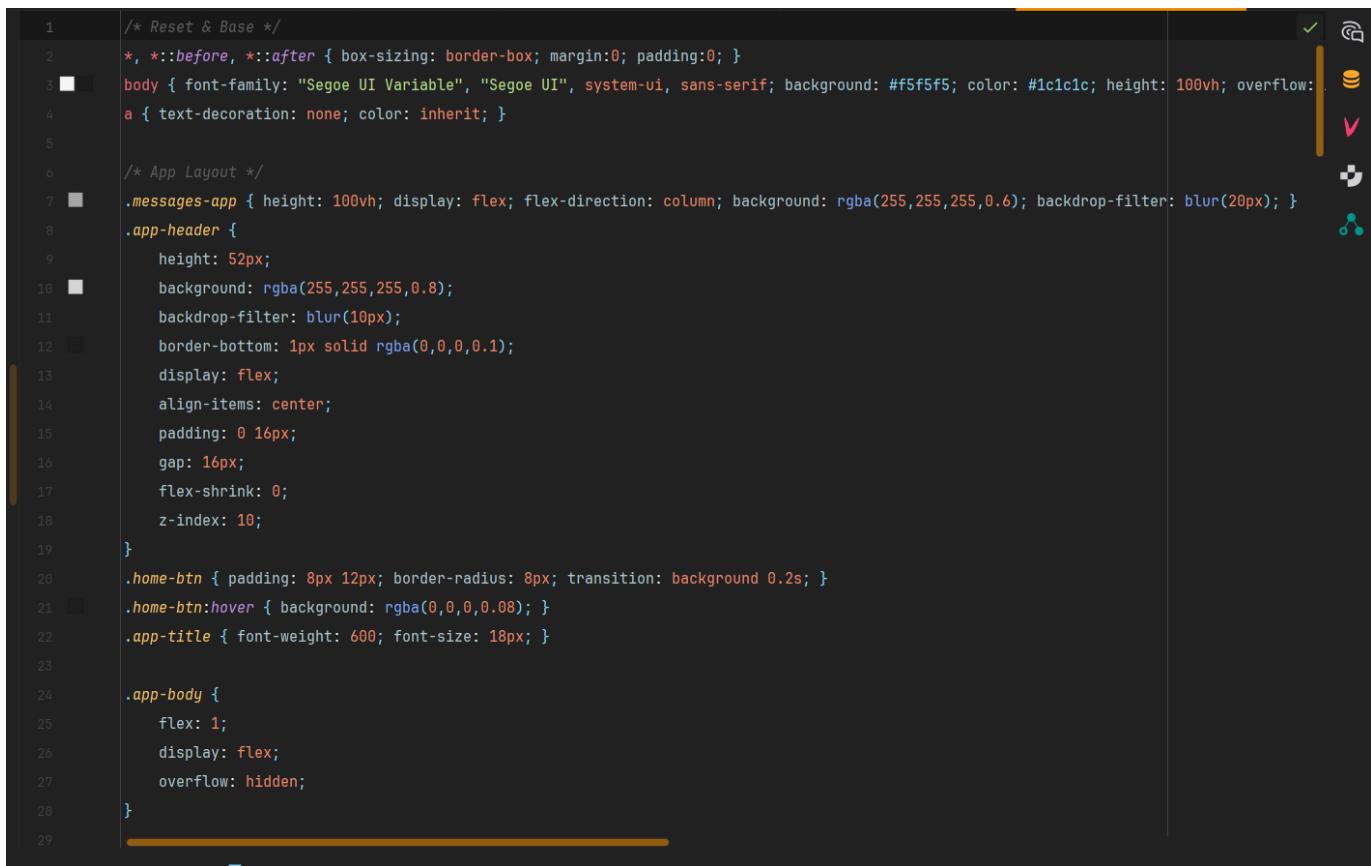
```

2      <html lang="en" xmlns:th="http://www.thymeleaf.org">
10     <body>
11       <div class="messages-app">
19         <div class="app-body">
40           <main class="chat-main" th:if="${selectedPartner != null}">
41             <div class="chat-header">
42               
46               <div class="partner-name" th:text="${selectedPartner.firstName + ' ' + selectedPartner.lastName}"></div>
47             </div>
48
49             <div class="messages-container" id="chatMessages">
50               <div th:each="msg : ${messages}">
51                 th:class="'message ' + (${msg.sender.id == currentUserId} ? 'outgoing' : 'incoming')">
52                   
56                   <div class="bubble-wrapper">
57                     <div class="bubble" th:text="${msg.text}"></div>
58                     <div class="time" th:text="#temporals.format(msg.datetime, 'HH:mm dd/MM/yyyy')}></div>
59                   </div>
60                 </div>
61               </div>
62
63               <form id="sendForm" class="message-input">
64                 <input type="text" id="messageText" placeholder="Type a message..." autocomplete="off" required>

```

> templates > Messages >  MessagesPage.html

Code 52 Html message



```
1  /* Reset & Base */
2  *, *::before, *::after { box-sizing: border-box; margin:0; padding:0; }
3  body { font-family: "Segoe UI Variable", "Segoe UI", system-ui, sans-serif; background: #f5f5f5; color: #1c1c1c; height: 100vh; overflow: hidden; }
4  a { text-decoration: none; color: inherit; }
5
6  /* App Layout */
7  .messages-app { height: 100vh; display: flex; flex-direction: column; background: rgba(255,255,255,0.6); backdrop-filter: blur(20px); }
8  .app-header {
9      height: 52px;
10     background: rgba(255,255,255,0.8);
11     backdrop-filter: blur(10px);
12     border-bottom: 1px solid rgba(0,0,0,0.1);
13     display: flex;
14     align-items: center;
15     padding: 0 16px;
16     gap: 16px;
17     flex-shrink: 0;
18     z-index: 10;
19 }
20 .home-btn { padding: 8px 12px; border-radius: 8px; transition: background 0.2s; }
21 .home-btn:hover { background: rgba(0,0,0,0.08); }
22 .app-title { font-weight: 600; font-size: 18px; }
23
24 .app-body {
25     flex: 1;
26     display: flex;
27     overflow: hidden;
28 }
29
```

Code 53 Css message

```

2      <html xmlns:th="http://www.thymeleaf.org" lang="en">
9      <body>
11     <div class="win11-container">
39         <!-- Dashboard Grid -->
40         <section class="dashboard-grid">
41             <!-- STAFF & STUDENTS -->
42             <a class="dashboard-card" th:href="@{/staff-home/lecturers-list}">
43                 <span class="icon"><i class="fas fa-chalkboard-teacher"></i></span>
44                 <span class="meta"><strong>Lecturers List</strong><small>Manage lecturers</small></span>
45             </a>
46             <a class="dashboard-card" th:href="@{/staff-home/students-list}">
47                 <span class="icon"><i class="fas fa-user-graduate"></i></span>
48                 <span class="meta"><strong>Students List</strong><small>View & manage students</small></span>
49             </a>
50             <a class="dashboard-card" th:href="@{/staff-home/lecturer-specialization}">
51                 <span class="icon"><i class="fas fa-briefcase"></i></span>
52                 <span class="meta"><strong>Lecturer Specialization</strong><small>Assign & review specializations</small></span>
53             </a>
54
55             <!-- ACADEMIC CONTENT -->
56             <a class="dashboard-card" th:href="@{/staff-home/major-subjects-list}">
57                 <span class="icon"><i class="fas fa-book-open"></i></span>
58                 <span class="meta"><strong>Subjects & Syllabus</strong><small>Manage academic content</small></span>
59             </a>
60             <a class="dashboard-card" th:href="@{/staff-home/specialized-subjects-list}">
61                 <span class="icon"><i class="fas fa-layer-group"></i></span>
62                 <span class="meta"><strong>Specialized Subjects</strong><small>Oversee specialization materials</small></span>
63             </a>
64             <a class="dashboard-card" th:href="@{/staff-home/study-plan}">

```

> templates > Staff >  StaffHome.html

Code 54 Html staff home

```

1   * {
2     box-sizing: border-box;
3   }
4
5   body {
6     font-family: 'Segoe UI Variable', 'Segoe UI', 'Noto Sans', sans-serif;
7     background: #f3f4f6;
8     margin: 0;
9     padding: 0;
10    color: #1a1a1a;
11    font-size: 16px;
12    line-height: 1.6;
13  }
14
15  /* ===== Container ===== */
16  .win11-container {
17    max-width: 1400px;
18    margin: auto;
19    padding: 24px;
20  }
21
22  /* ===== Title Bar ===== */
23  .win11-title-bar {
24    background: #ffffff;
25    border-radius: 14px;
26    padding: 20px 24px;
27    margin-bottom: 32px;
28    box-shadow: 0 4px 12px rgba(0, 0, 0, 0.08);
29

```

s > static > Staff > StaffHome.css

Code 55 Css staff home

The university management system UI/UX implementation focuses on a well-developed design system based on the principles of the Windows 11 Fluent Design and the development of the modern, unified, and user-friendly interface, contributing to a higher level of productivity and user satisfaction in all the functional fields. The design system utilizes a uniform visual language, considerate interaction patterns, and accessibility-based considerations to make sure that people can easily get their work done irrespective of their technical skills or accessibility requirements. Implementation of the same strikes a balance between aesthetic appeal and practical efficiency and has resulted in a highly usable interface which is aesthetically pleasing. CSS custom properties (CSS variables) in the root (CSS variables) define the design system foundation to establish a centralized theming mechanism, which makes sure that all pages and elements have the same appearance. Even the color palette is well-considered: primary actions are represented by the color of #2563eb (blue), a state of success, danger/warning, and neutral are represented by the color of #22c55e, #dc2626, and different gray, respectively. The color scheme is not only limited to the choice of simple hues, but also to hover states, active states, and disabled states to make sure that interactive elements can be visually represented. The typography system will be based on the font families of Segoe UI Variable and Segoe UI that has good readability and a modern and professional look. The fonts are used in a hierarchical order (12px small text to 26px page titles) with line height being made as easy to read as possible (usually 1.5-1.6 body text). The visual design language uses Windows 11-inspired principles to create visual elements with

several significant features: rounded corners with a border-radius of between 8px-18px depending on the size of the component, soft shadows with rgba values to create depth and elevation, and acrylic/glass morphism effects created with backdrop-filter blur to give contemporary visual elements. The design has a light color palette with the background color of the site being mainly white with a shade of f3f4f6 and the user content panels being white f3f4f6 which gives off a clean and uncluttered look over a long period of use. The spacing is done with the help of equivalent padding and margin values (8px, 12px, 16px, 20px, 24px, 32px) to create visual rhythm and appropriate grouping of the content. Box shadows are also set to be subtle and not too overwhelming to the interface with values such as 0 4px 12px rgba(0, 0, 0, 0.08) as the standard elevation and 0 6px 18px rgba(0, 0, 0, 0.1) as the elevated state. The dashboard card is among the most advanced components of the UI that employs a multi-layered visual model, uniting the iconography, typography, color coding, and interactive feedback. Every dashboard card has a big icon (36px) which is the font awesome, a bold text (17px) title, and a subtitle (13px) description. Cards visualize categorization with a left border accent (6px solid) in different colors: the primary actions are displayed with blue, the states of success with green, the warnings with orange, and informational elements with different colors. The cards have advanced hover effects such as a vertical translation (translateY(-4px)), addition of a shadow, expansion of icons (scale(1.15)) and a decorative light sweep effect with the help of a pseudo-element (::after) which gives the effect of a shimmer on hover. These are CSS-based transitions with easing functions (cubic-bezier and ease) to create smooth interaction which has a natural feel and is used to give instant visual feedback without feeling jerky or overpowering. The navigation and toolbar elements are seen to follow a single design pattern which is observed throughout list views, detail pages, and administration interfaces. The unified toolbar (unified-bar or search-toolbar) offers a three part structure: Left part with breadcrumb navigation and back buttons, middle part with page titles with contextual icons and right part with action buttons and summary statistics. The flexbox layout of the toolbar is justified-content space-between to make sure that it aligns and spacing is right and flex-wrap wrap to be responsive when used on smaller screens. The style of toolbar buttons is consistent, which includes padding (10px 18px), border radius (10px), and hover effects, where changing the background color to a light blue shade (#f0f6ff) and the border color to the main blue color. The background of the toolbar is white with light shadows and round corners that provide visual contrast to the content of the page and give the toolbar a unified look. The implementation of the form design is guided by the current practices of UX such as floating labels, input fields supplemented with icons, and extensive validation feedback. Input groups are groupings of icons, input fields, and labels into a unified unit with icons being placed to the left of input absolute positioning. The floating label technique relies on CSS pseudo-classes of :focus and :not(:placeholder-shown) to animate labels between position of the placeholders to positions of the floating label over the input giving definite identification of the fields and maximizing the space of the input. Form validation gives immediate visual feedback in color-coded form borders (red for errors, green for success), inline error messages displayed below the fields, and iconic feedback (checkmarks valid, exclamation marks errors). The specified required fields are marked by an asterisk and the required HTML tag, and both visual and programmatic marking of mandatory fields are present. The alert and notification system deploys an extensive feedback system that presents the user with actionable and informative feedback on the states of the system and user activities. Success alerts have a green background (#dcfce7) with a green left border, error alerts have a red background (#fee2e2) with a red left border and information alerts have a blue background (#e0f2fe) with a blue left border. Every alert has a Font Awesome icon corresponding

to the type of alert, which helps to identify the alert instantly. Alerts also feature dismissible features using close buttons and flash properties to have the alerts show up post redirect and clear automatically on being shown. The alert system fits perfectly with the Thymeleaf conditional rendering which enables the controllers to transfer a message of success or failure that automatically is styled and displayed to the user.

The table component design is an implementation of a clean scannable layout that is ideal in presenting and interacting data. Tables have color alternating rows to aid more readability and when a user hovers the cursor the row is highlighted by a hover effect, which enables one to observe where the cursor is when browsing through large data sets. Bold typography and slightly lower backgrounds are employed in table headers to provide visual hierarchy. The action buttons in the rows are always styled and placed, usually at the right column. There are checkboxes to select items in the first column facilitated by the use of tables and visual highlighting of selected rows. The table element has empty state management, which shows a center message in case no data exists so that it would not be confused or show ambiguous feedback on the status of data. The system of iconography relies on Font Awesome 6.5.1 to give the system a unified and all-encompassing icon library across the application. Academic features, financial features, scheduling, user icons of profile features, and envelope are represented semantically using icons: graduation cap, wallet, calendar, and icons of users. The sizes of icons have a regular scale: 18px in-line icon sizes, 24px in button icon sizes, 36px in dashboard card icon sizes, and 48px in large feature icon sizes. Icons are also color-coded based on the context of their functions, whereby primary actions are blue, success actions are green and destruction actions are red. The icon system makes the system more usable by allowing it to be used visually in that the user immediately knows what functions do what and also alleviates the cognitive load on the user to learn the various features of the interface and also helps users who cannot easily read text labels. The user experience flow design adopts the intuitive patterns of navigation that lead the users through the complicated processes with a low cognitive load. The main navigation structure revolves around home pages in the dashboard as the central points, where all the key functional regions can be accessed by the user in the form of visually differentiated cards. The role of every card is clearly defined by icon, title, and subtitle so that a user could find the functionality that he/she requires in a short period of time. Secondary navigation employs the use of breadcrumb patterns and back buttons to keep and offer the users an easy way of going back to the past web pages. The navigation system is also able to support both direct links (when the user knows where he wants to go) and exploration navigation (when the user is finding out how), supporting the different user behavior patterns. The implementation of data visualization will use Chart.js to build interactive and responsive graphs and charts that allow the users to see complex data relationships. Dashboard pages have various types of charts: bar charts to compare values by category (sessions per day of week), horizontal bar charts to rank data (busiest lecturers), pie charts to indicate proportions. The charts have responsive settings where automatically the charts resize to the size of the containers making them readable on various screen sizes. The schemes of colors in charts have been chosen attentively according to the design system used in the application, and they are the primary blue and success green and other semantic colors. The charts have legends, axis labels and tooltips that appear on hovering which gives detailed information without overloading the interface. The chart application relies on Thymeleaf inline JavaScript to send server-side information to the client-side charting which will ensure that the charts always re-render with up-to-date data. The real time messaging interface deploys a

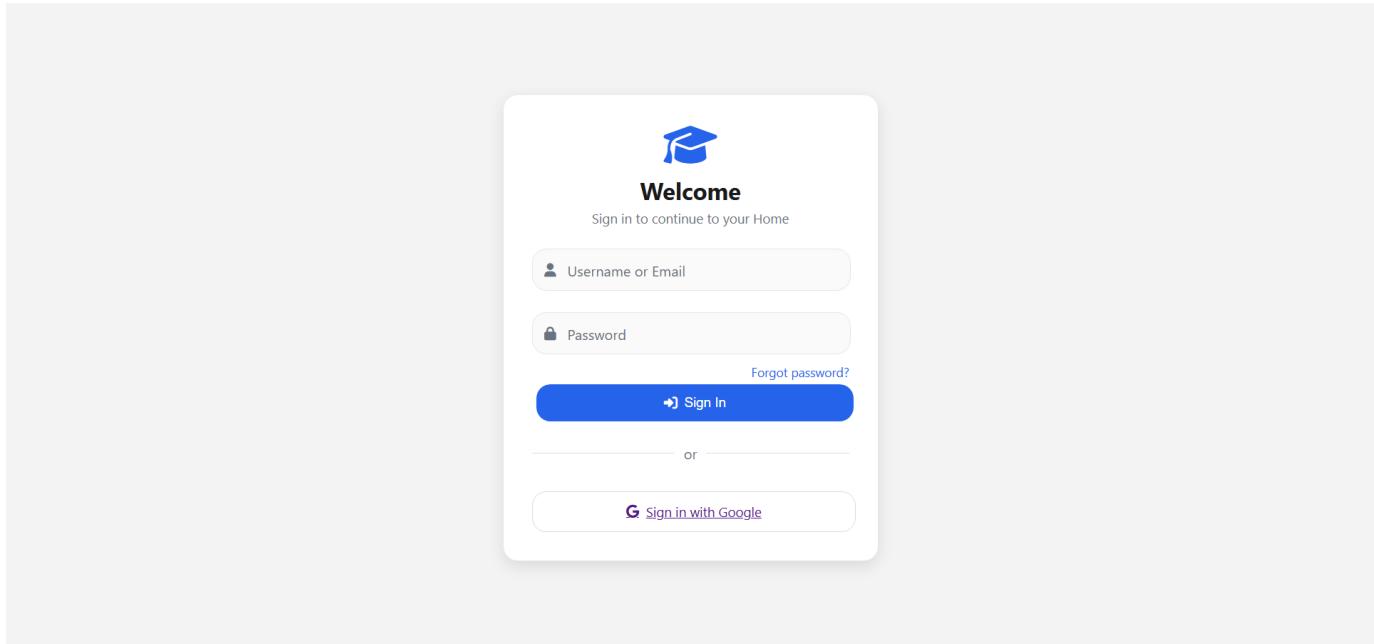
contemporary chat application design pattern which creates a familiar intuitive experience to the user. The interface layout of the messages is split-pane with a sidebar that has the conversation list and the main part that shows the conversation I have selected. The sidebar is clean with an avatar-based, name-based, and active state design, and it is not hard to identify and choose the conversations. The chat box shows messages as bubbles, outgoing messages are placed on the right side, and incoming messages on the left side according to the conventional chat interface designs. The message bubbles have time stamping and the avatar pictures which give a context and visual identification. The interface will independently scroll through to present the most recent messages and will have a smooth scrolling effect, which gives a slick user experience. The message input section is located in the bottom of the chat portion and is not affected by the length of message history. The loading and empty state design will give clear feedback information when data is loading and in the event of none being available. The loading states make use of subtle animations or skeleton screens to show that something is being loaded, which is why users are not confused with unresponsive interfaces. The empty states contain informative messages, informative icons and actionable instructions (e.g., No students in this class with a recommendation to add students). Such states allow users to remain unfrustrated since they clearly articulate the current system state and give subsequent actions where necessary. The empty state design employs dull colors and centered designs in creating attention without being fatally awakening, giving the design a calm, professional look even when it denotes the lack of the expected content.

Implementation of the accessibility is in line with the WCAG 2.1 requirements because it makes the interface user-friendly to users with varied abilities and requirements. Semantic HTML5 elements exist everywhere (header, nav, main, section, article, footer) to offer suitable structure of the document to provide proper screen readers. Form inputs have correct labels which have been linked by the for attribute or by enclosing the inputs in label elements so that the screen readers understand the purpose of the fields. The ratios of color contrast are appropriate in accordance with the WCAG AA criteria and the text color has an adequate contrast with the background color. Interactive features such as support of keyboard navigation, focus indicators (which are usually blue outlines) that explicitly indicate which element has keyboard focus and/or has keyboard focus. The text labels are placed alongside the icons to make sure that the functionality can be comprehended even in cases when icons are not identified, and the ARIA labels are applied where they are needed to supply the supportive technologies with more context. The implementation of the responsive design will guarantee the interface changes smoothly when it is displayed on various sizes of screens both the large desktop monitors and tablets or mobile devices. Media queries are widely applied to modify layouts, font sizes, space, and components visibility at breakpoints 1024px, 768px, and 480px. On smaller screens, the grid of the dashboard is narrowed down to one column and results in navigation bars being docked vertically and table elements being shifted to horizontal scraping or card-like designs. Touch targets are made sizeable (not less than 44x44px) to be touched on mobile and the hover effects are substituted with touchable ones, when using a touch device. The responsive implementation ensures that its functionality is maintained on any-sized screen but the layout is more optimistic to each type of device and therefore allows the users to be able to use the system efficiently despite the type of device they have. The feedback system on interaction offers the immediate feedback mechanism in response to user actions in a clear feedback system (visual and haptic where available), and time-based feedback. The interaction is verified by visual changes of the state

(background color, border color, shadow) that appear upon a button click. The submission forms indicate loading effects or the inability to submit forms twice. The hover states give preview information on what is going to occur when a user clicks on an element, thereby minimizing the uncertainty of the user. The use of transitions and animations are also judicious to ensure the smooth transition of states without being distracting and are usually implemented with a duration of 0.2-0.3 seconds accompanied with natural easing functions. Error states are easy to understand and offer practical feedback besides specific information to the user on what to do to solve the problem whereas success states offer positive reinforcement which will compel the user to continue using the system. The implementation of the visual hierarchy employs the sizes, colors, space, and typography to direct user attention and the significance of information in the communication. Primary hierarchy is achieved with the help of page titles that have bigger font sizes (24-26px) and heavier weight. The medium sized (20px) section headers with icons are employed to provide secondary hierarchy. Normal weights (15-16px) are used in body texts so as to create easy reading. Significant data are highlighted with color accents (e.g. account balance in blue), whereas secondary information is highlighted with tones. Spacing helps to visual grouping, the similar elements come nearer and the dissimilar ones come apart by wide gap. The hierarchy guarantees that page scanning and determining the most significant information and actions do not take much time, as the user can do, conversely. The error prevention and recovery design uses several measures to prevent errors by users and graceful error recovery. Form validation is done on the client-side (to provide instant feedback) and the server-side (to ensure security, and integrity of data), whereby errors are detected prior to them occurring. Destructive actions (e.g. deleting records) are dealt with by confirmation dialogs to avert the loss of data by accident. The invalid data entry is blocked by input constraints (date ranges, number, and format). Where mistakes occur, the system will give clear messages of errors and offer specific instructions on how the problem can be rectified, as opposed to general error codes. Its error recovery design enables users to easily correct errors and the form data are even saved in case of a failure in validation so that it does not have to re-enter the information. The customization and personalization capabilities enable users to adjust the interface to user requirements and workflow necessities. Although its current implementation is limited to functional personalization (including a retention of search filters and pagination preferences during the session), the design system enables the implementation of future expansion with themes configured by the user, layout preferences, and notifications. The preference storage implemented as a session helps in ensuring that user preferences are retained during a session, which is more efficient as it eliminates repetitive set up. The personalization strategy strikes the balance between customization and consistency making sure that the interface is familiar and predictable, but it still addresses the needs of individual users. Optimization of performance of UI rendering involves methods of the creation of smooth and responsive interaction despite large data. Very long lists can be virtual scrolled, but the existing pagination mechanism would be effective to most usage scenarios. CSS animations are fully based on transform properties instead of changing position, and therefore are smooth. Image loading also has lazy loading and fallback to avoid layout jumps and gracefully handle loading errors. The component architecture is optimized to minimize the re-renders through server-side rendering, thereby decreasing the complexity of client-side JavaScript along with fast initial page loads. To conclude, the UI/UX implementation of the university management system is a thorough, considerate interface design that puts the needs of the users first, accessibility, and visual consistency. The design system based on Windows 11 gives a sleek, corporate look that builds confidence and satisfaction in users. Its implementation balances the

aesthetics and efficiency of the interface; the interface is not only pleasing to the eye but very useful. The consistency of the design system, accessibility features, responsive nature, and considerate interaction patterns develop an interface which assists the user to complete his or her tasks efficiently and yet have a good user experience which persuades him or her to use the system further.

4.3.4 Responsive Design Implementation



Screenshot 4 Login

User
Staff Home - Quản trị Kinh doanh
Messages
Timetable Overview
Dashboard
Personal Page
Logout



Lecturers List

Manage lecturers



Students List

View & manage students



Lecturer Specialization

Assign & review specializations



Subjects & Syllabus

Manage academic content



Specialized Subjects

Oversee specialization materials



Study Plan

Manage course schedules



Specialized Study Plan

Advanced study management



Classes Arrangement

Organize general classes



Specialized Classes

Manage focused schedules



News

Post and update announcements



Award Scholarships

Manage academic awards



Student Evaluate

Review opinions & Evaluate



Your Colleagues

Collaborate with peers



Your Colleagues

Collaborate with peers



Ticket Approvals

Review and approve support requests

Screenshot 5 Staff home

Home
User
Your Students List
Search...
Name
Show: 20
Apply
Total Students: 5
Add Student

ID	Avatar	Full Name	Email	Timetable	Learning Process	Academic Transcript	Actions
stuhn0001		An Nguyễn	stuhn0001@student.demo.com	View Timetable	View Learning Process	View Transcript	Edit Delete
stuhn0041		Mạnh Nguyễn	stuhn0041@student.demo.com	View Timetable	View Learning Process	View Transcript	Edit Delete
stuhn0081		An Nguyễn	stuhn0081@student.demo.com	View Timetable	View Learning Process	View Transcript	Edit Delete
stuhn0121		Mạnh Nguyễn	stuhn0121@student.demo.com	View Timetable	View Learning Process	View Transcript	Edit Delete
stuhn0161		An Nguyễn	stuhn0161@student.demo.com	View Timetable	View Learning Process	View Transcript	Edit Delete

First Prev Next Last Go to page: ▼

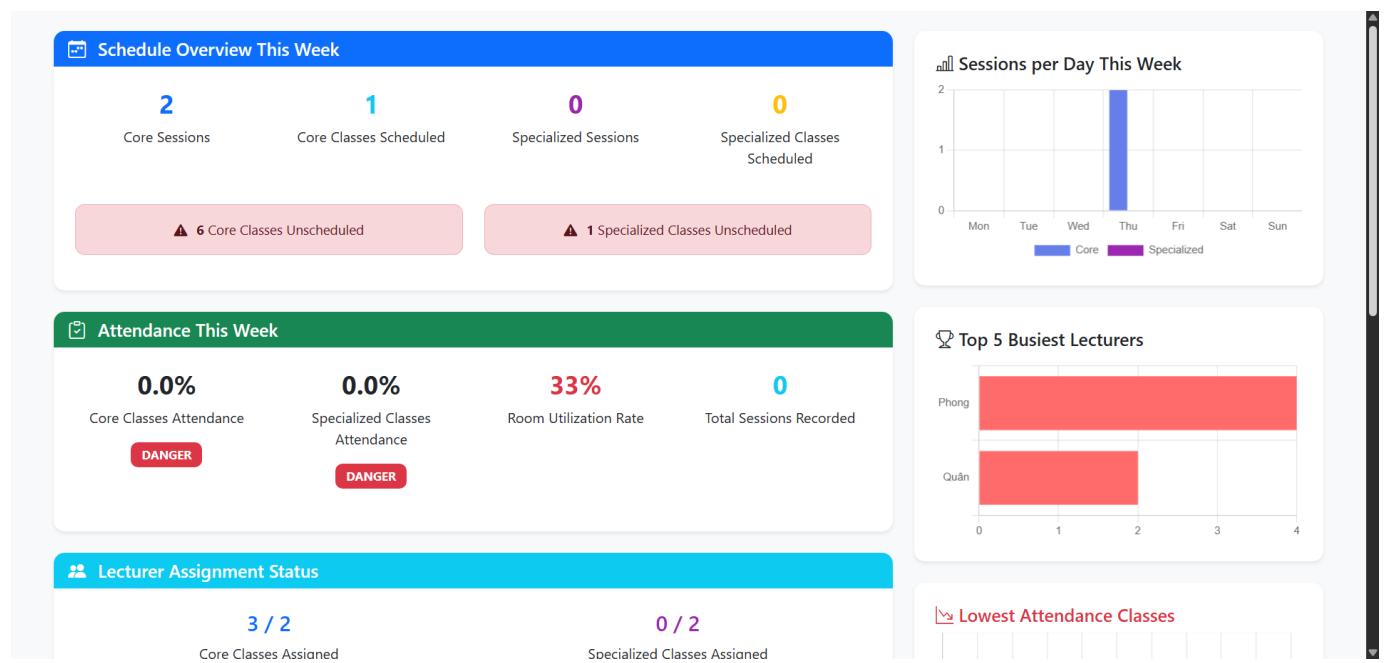
Screenshot 6 Student List

Home Your Subjects List Name <input type="button" value="▼"/> Search... <input type="text"/> <input type="button" value="Search"/> Show: 20 <input type="button" value="Apply"/> Total Subjects: 1 <input type="button" value="+ Add Subject"/>						
ID	Name	Semester	Curriculum	Creator	Actions	Syllabus
SUB_MAJ_001	Nhập môn Quản trị	1	BTEC	Hùng Lê	<input type="button" value="Edit"/> <input type="button" value="Delete"/>	<input type="button" value="View"/>
First Prev Next Last Go to page: <input type="text" value="1"/> <input type="button" value="▼"/>						

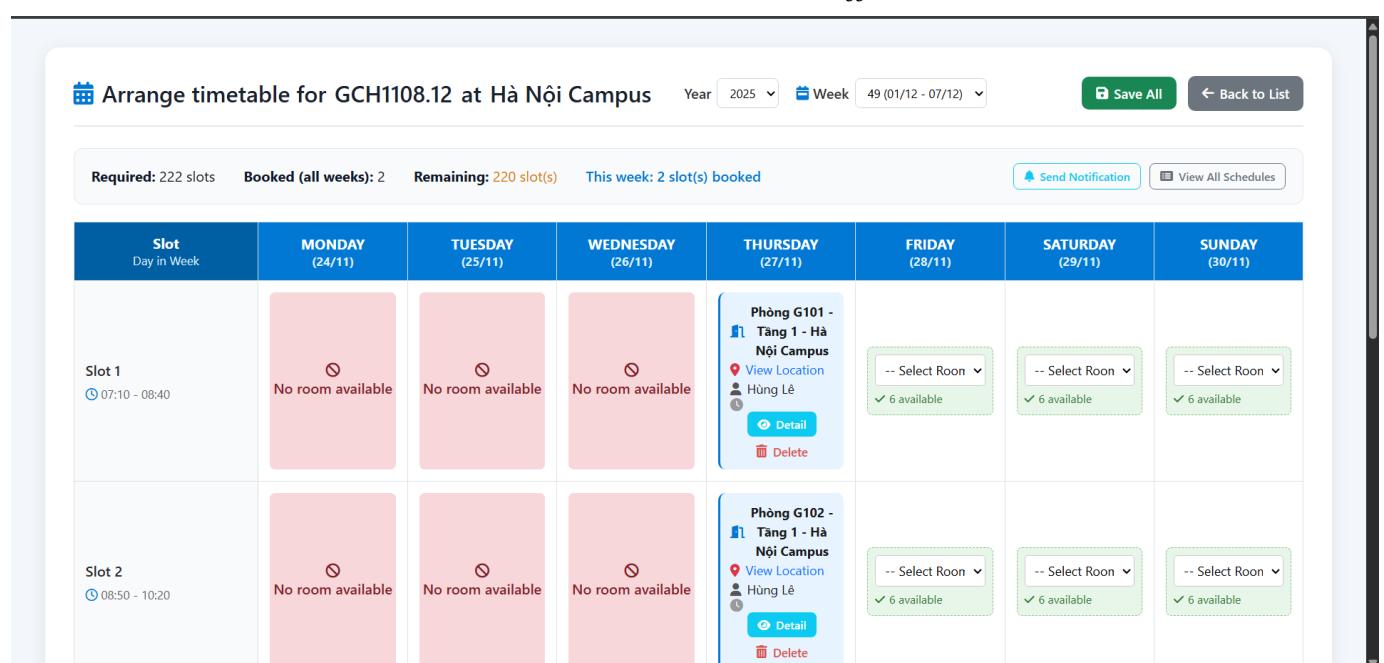
Screenshot 7 Subjects List

Home Back Assign Members to Nhập môn Quản trị					
Assigned Students					
	Avatar	Student ID	Name	Reason	Assigned By
<input type="checkbox"/>		stuhn0001	An Nguyễn	Bắt buộc học major subject: Nhập môn Quản trị	Hùng Lê
<input type="checkbox"/>		stuhn0041	Mạnh Nguyễn	Bắt buộc học major subject: Nhập môn Quản trị	Tuấn Hoàng
<input type="checkbox"/>		stuhn0081	An Nguyễn	Bắt buộc học major subject: Nhập môn Quản trị	Hùng Lê
<input type="checkbox"/>		stuhn0121	Mạnh Nguyễn	Bắt buộc học major subject: Nhập môn Quản trị	Tuấn Hoàng
<input type="checkbox"/>		stuhn0161	An Nguyễn	Bắt buộc học major subject: Nhập môn Quản trị	Tuấn Hoàng
<input type="button" value="Remove Selected"/>					

Screenshot 8 Assign members



Screenshot 9 dashboard staff



Arrange timetable for GCH1108.12 at Hà Nội Campus

Year: 2025 Week: 49 (01/12 - 07/12)

Save All Back to List

Required: 222 slots Booked (all weeks): 2 Remaining: 220 slot(s) This week: 2 slot(s) booked

Send Notification View All Schedules

Slot Day in Week	MONDAY (24/11)	TUESDAY (25/11)	WEDNESDAY (26/11)	THURSDAY (27/11)	FRIDAY (28/11)	SATURDAY (29/11)	SUNDAY (30/11)
Slot 1 ⌚ 07:10 - 08:40	No room available	No room available	No room available	Phòng G101 - Tầng 1 - Hà Nội Campus View Location Hùng Lê Detail Delete	-- Select Room ✓ 6 available	-- Select Room ✓ 6 available	-- Select Room ✓ 6 available
Slot 2 ⌚ 08:50 - 10:20	No room available	No room available	No room available	Phòng G102 - Tầng 1 - Hà Nội Campus View Location Hùng Lê Detail Delete	-- Select Room ✓ 6 available	-- Select Room ✓ 6 available	-- Select Room ✓ 6 available

Screenshot 10 Timetable

	lecthn001	Phong Hoàng	lecthn001@lect.demo.com	+84103000002
	lecthn002	Quân Đoàn	lecthn002@lect.demo.com	+84103000003
	lecthn019	Phong Hoàng	lecthn019@lect.demo.com	+841030000020

Student Attendance (3)

#	Image	Student ID	Full Name	Status	Note
1		stuhn0001	An Nguyễn	<input checked="" type="radio"/> Present <input type="radio"/> Absent	Optional note...
2		stuhn0041	Mạnh Nguyễn	<input checked="" type="radio"/> Present <input type="radio"/> Absent	Optional note...
3		stuhn0081	An Nguyễn	<input checked="" type="radio"/> Present <input type="radio"/> Absent	Optional note...

Screenshot 11 Detail timetable

 Total Students: 3

Students with Scores (3)

#	ID	Name	C1	C2	C3	Grade
1	stuhn0001	An Nguyễn	8,0	8,0	8,0	PASS ▾
2	stuhn0041	Mạnh Nguyễn				REFE ▾
3	stuhn0081	An Nguyễn				REFE ▾

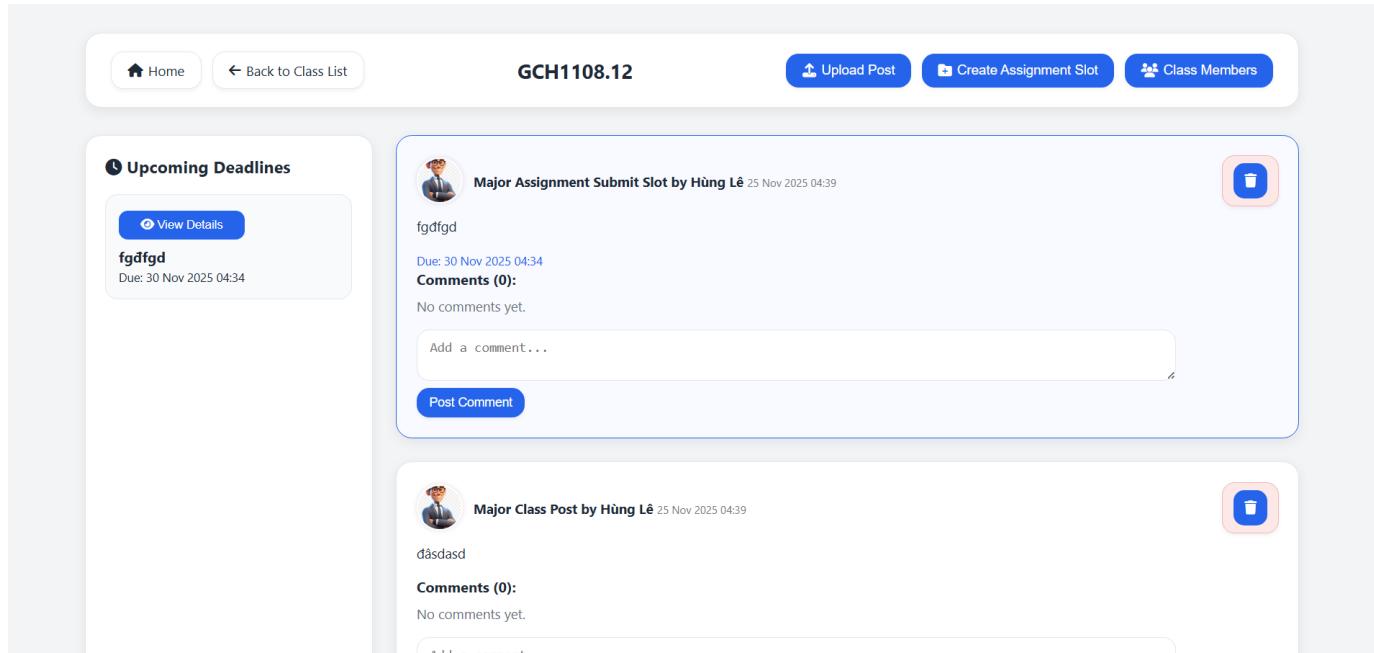
Screenshot 12 Enter Scores

[Home](#) [Messages](#)
 An Nguyễn

 alo
08:37

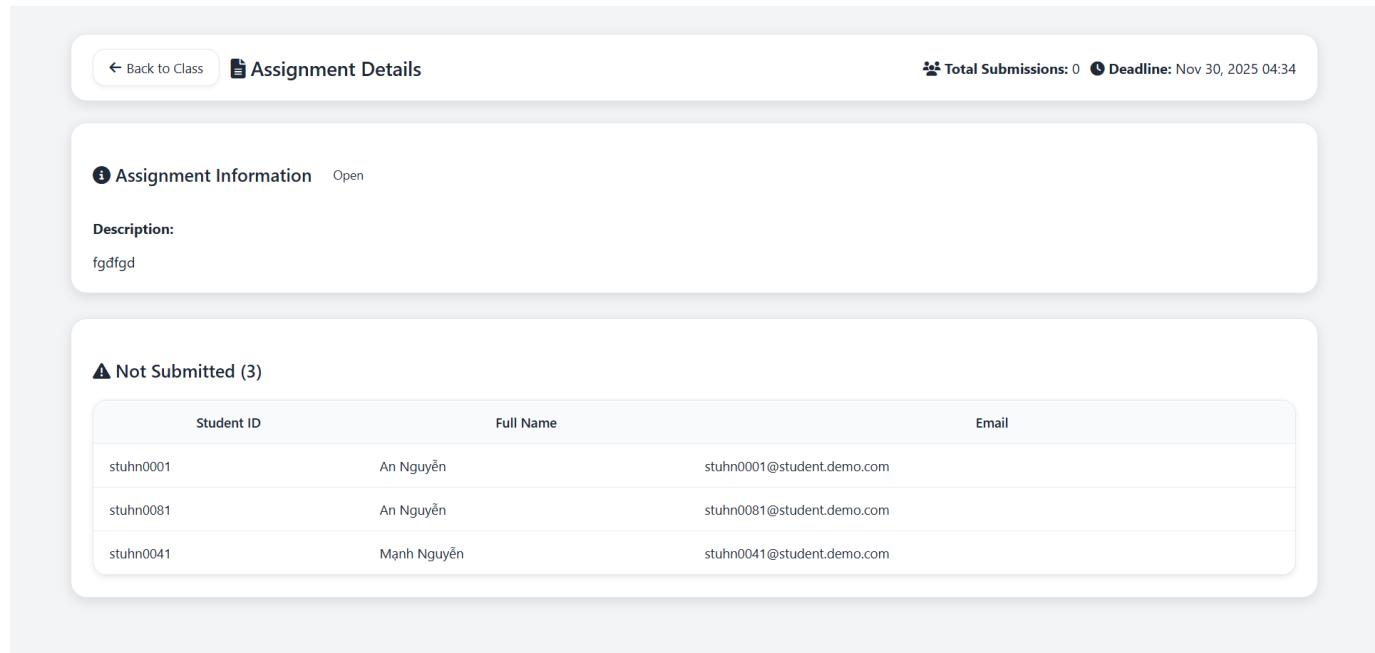


Screenshot 13 Message



The screenshot shows the 'Classroom' section of a university platform. At the top, there are navigation links: 'Home', 'Back to Class List', the class name 'GCH1108.12', and three buttons: 'Upload Post', 'Create Assignment Slot', and 'Class Members'. On the left, a sidebar displays 'Upcoming Deadlines' with a single item: 'fgdfgd' due on '30 Nov 2025 04:34'. The main area contains two posts by 'Hùng Lê': 1) 'Major Assignment Submit Slot' posted on '25 Nov 2025 04:39' with the content 'fgdfgd'. This post has a delete icon and a comment section with placeholder 'Add a comment...' and a 'Post Comment' button. 2) 'Major Class Post' posted on '25 Nov 2025 04:39' with the content 'dåådasd'. This post also has a delete icon and a comment section with placeholder 'Add a comment...'.

Screenshot 14 Classroom



The screenshot shows an assignment details page. At the top, there are buttons for "Back to Class" and "Assignment Details". To the right, it displays "Total Submissions: 0" and "Deadline: Nov 30, 2025 04:34". Below this, there's a section titled "Assignment Information" with an "Open" link. A "Description:" field contains the text "fgdfgd". A prominent red warning icon with the text "⚠ Not Submitted (3)" is displayed. A table lists three students who have not submitted their work:

Student ID	Full Name	Email
stuhn0001	An Nguyễn	stuhn0001@student.demo.com
stuhn0081	An Nguyễn	stuhn0081@student.demo.com
stuhn0041	Mạnh Nguyễn	stuhn0041@student.demo.com

Screenshot 15 Assignment detail

4.4 Integration of External Services/APIs

Google OAuth2 Authentication Integration

```

85
86 <!-- OAuth2 -->
87 <dependency>
88   <groupId>org.springframework.boot</groupId>
89   <artifactId>spring-boot-starter-oauth2-client</artifactId>
90 </dependency>

```

Code 56 import Oauth2

```

@Override 1 usage ▾ Trunks
public OidcUser loadUser(OidcUserRequest userRequest) {
    OidcUser oidcUser = super.loadUser(userRequest);
    String email = oidcUser.getAttribute( name: "email");

    if (email == null) {
        throw new OAuth2AuthenticationException(new OAuth2Error( errorCode: "no_email"), "No email in OAuth2 profile");
    }
    Persons person;
    try {
        person = entityManager.createQuery(
            s: "SELECT p FROM Persons p JOIN Authenticators a ON p.id = a.personId " +
                "WHERE p.email = :email AND a.accountStatus = :st",
            Persons.class)
            .setParameter( s: "email", email)
            .setParameter( s: "st", AccountStatus.ACTIVE)
            .getSingleResult();
    } catch (NoResultException e) {
        throw new OAuth2AuthenticationException(new OAuth2Error( errorCode: "user_not_found"), "User not found in DB");
    }

    String role = determineRole(person);
    var authorities = List.of(new SimpleGrantedAuthority(role));

    return new CustomOidcUserPrincipal(oidcUser, person, authorities);
}

```

e > demo > security > service > 🗝 CustomOAuth2UserService

Code 57 loadUser

The application integrates Google OAuth2 authentication to provide users with a convenient alternative to traditional username-password login. This integration leverages Spring Security's OAuth2 client support, which simplifies the implementation of the OAuth2 authorization code flow. The OAuth2 configuration is defined in the application properties file, where all necessary Google OAuth2 endpoints and credentials are specified using environment variables for security. The configuration includes the client ID and client secret obtained from Google Cloud Console, the redirect URI that Google will use to send the authorization code back to the application, the required scopes (openid, profile, email) that define what user information the application requests, and all the necessary OAuth2 provider endpoints including the authorization URI, token URI, user info URI, and JWK set URI for token validation. The security configuration in `SecurityConfig.java` enables OAuth2 login by configuring the `oauth2Login` method, which sets the login page, specifies a custom OIDC user service for processing user information, and defines success and failure handlers that redirect users to appropriate pages based on their roles after authentication. The custom OAuth2 user service, implemented in `CustomOAuth2UserService`, extends Spring Security's `OidcUserService` to provide custom user loading logic. When a user authenticates via Google, the service receives an `OidcUserRequest` containing the OAuth2 tokens, loads the standard OIDC user information from Google's user info endpoint, extracts the email address from the OIDC user attributes, queries the application's database to find a matching user account using the email address, verifies that the account is active, determines the user's role based on the type of person entity found (student, staff, lecturer, admin, etc.), creates a custom principal object that

combines the OIDC user information with the database user information and role authorities, and returns this principal to Spring Security for authentication. This approach ensures that Google OAuth2 authentication is seamlessly integrated with the application's existing user management system, allowing users to sign in with their Google accounts while maintaining the same role-based access control and user data structure as traditional login. The login page includes a "Sign in with Google" button that redirects users to Google's authorization server, and after successful authentication, users are automatically redirected back to the application with their role-appropriate dashboard.

Email Service Integration (Gmail SMTP)

```
<!-- Mail -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

Code 58 Import Email

```
1 package com.example.demo.email_service.dao;
2
3 > import ...
4
5 @Repository
6 public class EmailServiceForStudentDAOImpl implements EmailServiceForStudentDAO {
7
8     @Autowired
9     private JavaMailSender mailSender;
10
11     @Autowired
12     private EmailTemplatesService emailTemplatesService;
13
14     @Value("${APP_BASE_URL:http://localhost:8080}")
15     private String baseUrl;
16
17     /**
18      * Tạo default template khi không tìm thấy template trong database
19      */
20     private EmailTemplateDTO createDefaultTemplate(EmailTemplateTypes type) { 6 usages
21         EmailTemplateDTO template = new EmailTemplateDTO();
22
23         switch (type) {
24             case STUDENT_ADD:
25                 template.setSalutation("Access Your Student Account");
26                 template.setGreeting("Welcome to Our University");
27                 template.setBody("Your student account has been successfully created. Below are your account details and important information.");
28                 break;
29             case STUDENT_EDIT:
30                 template.setSalutation("Edit Your Student Account");
31                 template.setGreeting("Welcome to Our University");
32                 template.setBody("Your student account has been successfully updated. Below are your account details and important information.");
33                 break;
34             case STUDENT_DELETE:
35                 template.setSalutation("Delete Your Student Account");
36                 template.setGreeting("Goodbye!");
37                 template.setBody("Your student account has been successfully deleted. Thank you for using our service.");
38                 break;
39             case STUDENT_RESET_PASSWORD:
40                 template.setSalutation("Reset Your Student Password");
41                 template.setGreeting("Welcome to Our University");
42                 template.setBody("Your password has been reset. Please log in with your new password.");
43                 break;
44         }
45     }
46 }
```

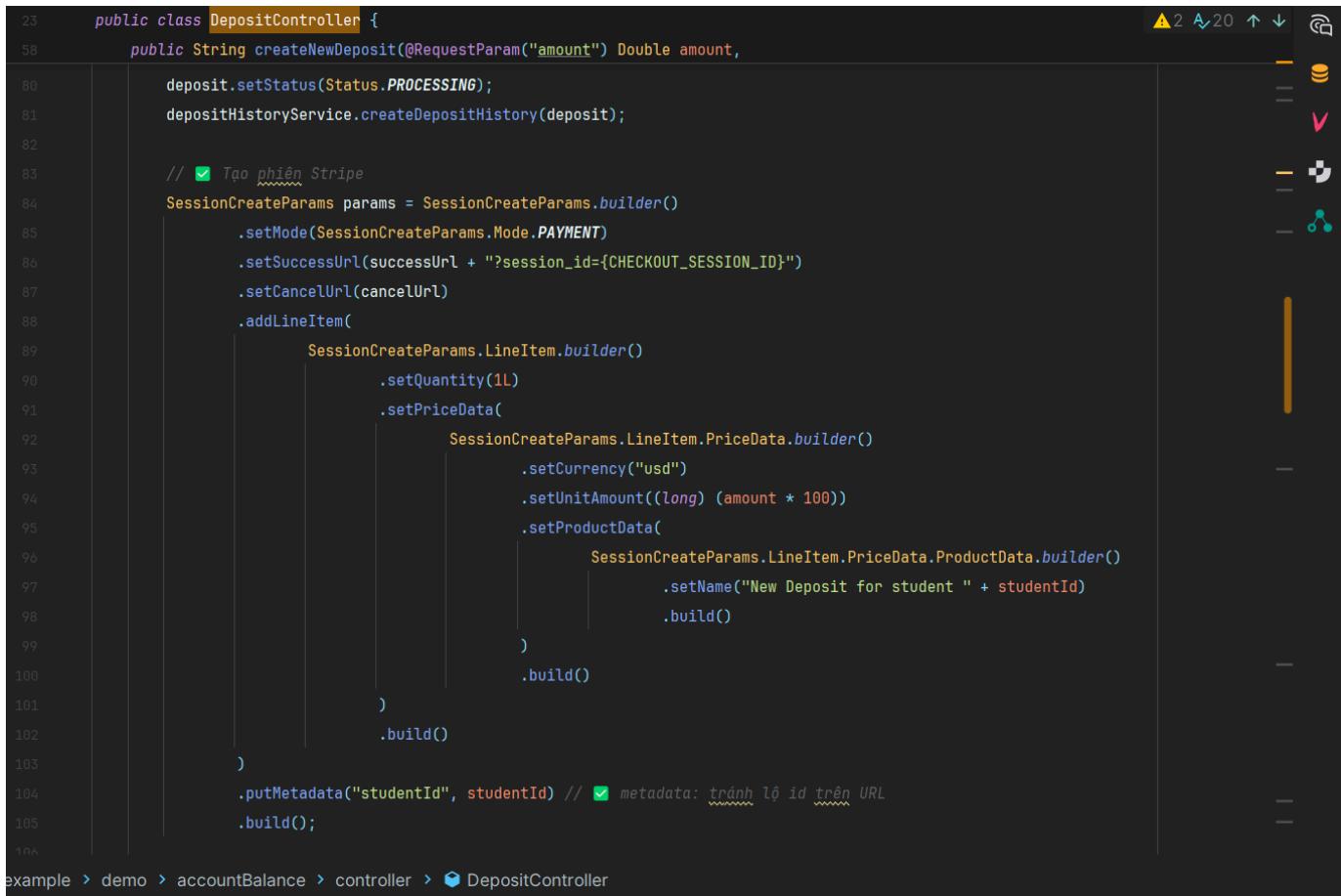
Code 59 EmailServiceForStudentDAOImpl

The application integrates with Gmail's SMTP service to send automated email notifications for various system events, including account creation, password resets, schedule updates, and administrative notifications. The email service is built on top of Spring Boot's mail starter, which provides JavaMailSender interface implementation and simplifies email sending operations. The SMTP configuration is defined in the application properties file, where Gmail's SMTP server details are specified: the host is set to smtp.gmail.com, the port is configured to 587 for TLS encryption, and authentication credentials (username and password) are provided through environment variables. The configuration also enables SMTP authentication and STARTTLS to ensure secure email transmission. The email service implementation follows a layered architecture with DAO, service, and controller layers, where each user type (students, staff, lecturers, admins, parents) has dedicated email service classes that handle type-specific email templates and context data. The email service DAO implementations, such as `EmailServiceForStudentDAOImpl`, provide methods for sending different types of emails, including account creation notifications, account update notifications, account deletion notifications, and schedule change notifications. Each email sending method follows a consistent pattern: it retrieves the appropriate email template from the database using the email templates service, falls back to default templates if database templates are not available, generates HTML email content by populating template variables with context data (user information, dates, links, etc.), creates a MIME message using JavaMailSender, sets the recipient, subject, and HTML content, attaches inline images if the template includes them, and sends the email using the mail sender. The email templates are stored in the database and can be customized by administrators, allowing for flexible email content management without code changes. The templates support variable substitution, where placeholders in the template are replaced with actual values from the email context, such as user names, account details, login URLs, and support contact information. The email service also includes error handling to log failures without disrupting the main application flow, ensuring that email sending issues do not prevent critical operations like user account creation from completing successfully.

Stripe Payment Integration

```
<!-- Stripe -->
<dependency>
    <groupId>com.stripe</groupId>
    <artifactId>stripe-java</artifactId>
    <version>25.0.0</version>
</dependency>
```

Code 60 Stripe Payment import



```
23     public class DepositController {
24
25         public String createNewDeposit(@RequestParam("amount") Double amount,
26
27             deposit.setStatus(Status.PROCESSING);
28             depositHistoryService.createDepositHistory(deposit);
29
30
31             // ✅ Tạo phiên Stripe
32             SessionCreateParams params = SessionCreateParams.builder()
33                 .setMode(SessionCreateParams.Mode.PAYOUT)
34                 .setSuccessUrl(successUrl + "?session_id={CHECKOUT_SESSION_ID}")
35                 .setCancelUrl(cancelUrl)
36                 .addLineItem(
37                     SessionCreateParams.LineItem.builder()
38                         .setQuantity(1L)
39                         .setPriceData(
40                             SessionCreateParams.LineItem.PriceData.builder()
41                                 .setCurrency("usd")
42                                 .setUnitAmount((long) (amount * 100))
43                                 .setProductData(
44                                     SessionCreateParams.LineItem.PriceData.ProductData.builder()
45                                         .setName("New Deposit for student " + studentId)
46                                         .build()
47
48                                     )
49
50                                 .build()
51
52                         )
53                         .build()
54
55                 )
56                 .build()
57
58             )
59             .putMetadata("studentId", studentId) // ✅ metadata: tránh lô id trên URL
60             .build();
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
```

example > demo > accountBalance > controller > DepositController

Code 61 Deposit controller

The application integrates Stripe's payment processing API to enable secure online payments for student account deposits and tuition fees. The Stripe integration is implemented using the official Stripe Java SDK, which is included as a dependency in the project's Maven configuration. The Stripe configuration is managed through a dedicated configuration class, `StripeConfig`, which

uses Spring's `@PostConstruct` annotation to initialize the Stripe API key from application properties during application startup. The Stripe secret key and publishable key are stored as environment variables and injected into the configuration using Spring's `@Value` annotation, ensuring that sensitive payment credentials are not hardcoded in the source code. The payment flow begins when a student initiates a deposit through the deposit interface, where they specify the amount they wish to add to their account balance. The deposit controller creates a deposit history record with a PROCESSING status to track the transaction, constructs a Stripe Checkout Session using the Stripe Java SDK's `SessionCreateParams` builder, configures the session with payment mode, success and cancel URLs, line items with product information and pricing, and metadata containing the student ID for later retrieval, and redirects the user to Stripe's hosted checkout page. The Stripe Checkout Session is configured with the payment amount converted to cents (Stripe's currency unit), USD as the currency, a descriptive product name, and metadata that allows the application to identify which student the payment belongs to without exposing the student ID in the URL. After the user completes payment on Stripe's secure checkout page, Stripe redirects them back to the application's success URL with a session ID parameter. The success handler retrieves the Stripe session using the session ID, verifies that the payment status is "paid", extracts the student ID from the session metadata, retrieves or creates the student's account balance record, calculates the amount received (converting from cents to dollars), updates the account balance by adding the deposited amount, updates the deposit history record status to COMPLETED, and displays a success message to the user. This integration ensures that payment processing is handled securely by Stripe, which is PCI DSS compliant and handles all sensitive payment card information, while the application only receives confirmation of successful payments and updates its internal records accordingly. The integration also supports payment retry functionality, where students can retry failed or cancelled payments by creating a new Stripe session for an existing PROCESSING deposit record, allowing for a seamless payment experience even when initial payment attempts are interrupted.

Font Awesome CDN Integration

```
<!-- WebJars -->
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>font-awesome</artifactId>
    <version>6.5.1</version>
</dependency>
```

Code 62 Front Awesome import

The application integrates Font Awesome icon library via CDN to provide a comprehensive set of icons for user interface elements. Font Awesome icons are loaded from the Cloudflare CDN using a link tag in HTML templates, which ensures that icons are delivered efficiently from a globally

distributed content delivery network. The integration uses Font Awesome version 6.5.1, which provides thousands of icons covering various categories including user interfaces, social media, education, finance, and communication. Icons are used throughout the application to enhance visual communication, provide intuitive navigation cues, and improve the overall user experience. The icons are integrated using simple HTML `<i>` elements with Font Awesome classes, making them easy to use and style alongside text content. The CDN approach eliminates the need to host icon files locally, reduces application bundle size, and ensures that icons are always up-to-date with the latest Font Awesome releases. While CDN dependencies introduce a small external dependency, the benefits of reduced maintenance, improved performance through CDN caching, and access to a vast icon library outweigh the minimal risk of CDN unavailability.

Integration Architecture and Best Practices

All external service integrations in the EduPortal application follow consistent architectural patterns and best practices to ensure reliability, security, and maintainability. Configuration management is centralized through Spring Boot's application properties, with sensitive credentials stored as environment variables rather than hardcoded values, allowing for different configurations across development, staging, and production environments. Error handling is implemented at multiple levels: service-level error handling catches and logs integration failures without disrupting application flow, controller-level error handling provides user-friendly error messages, and fallback mechanisms ensure that critical operations can continue even when external services are temporarily unavailable. Security considerations are paramount in all integrations: OAuth2 credentials are stored securely and transmitted over HTTPS, email credentials are encrypted in transit using STARTTLS, Stripe integration uses server-side API keys that never expose sensitive payment information to the client, and all external API communications use secure protocols. The integrations are designed to be asynchronous where possible, with email sending operations potentially executed asynchronously to avoid blocking user requests, and payment processing handled through redirects that don't require the application to wait for external service responses. Monitoring and logging are implemented throughout the integration points, with comprehensive logging of integration operations, success and failure tracking, and error details that facilitate troubleshooting and maintenance. The modular architecture allows each integration to be developed, tested, and maintained independently, with clear separation of concerns between integration-specific code and core application logic, making it easy to update, replace, or extend individual integrations without affecting other parts of the system.

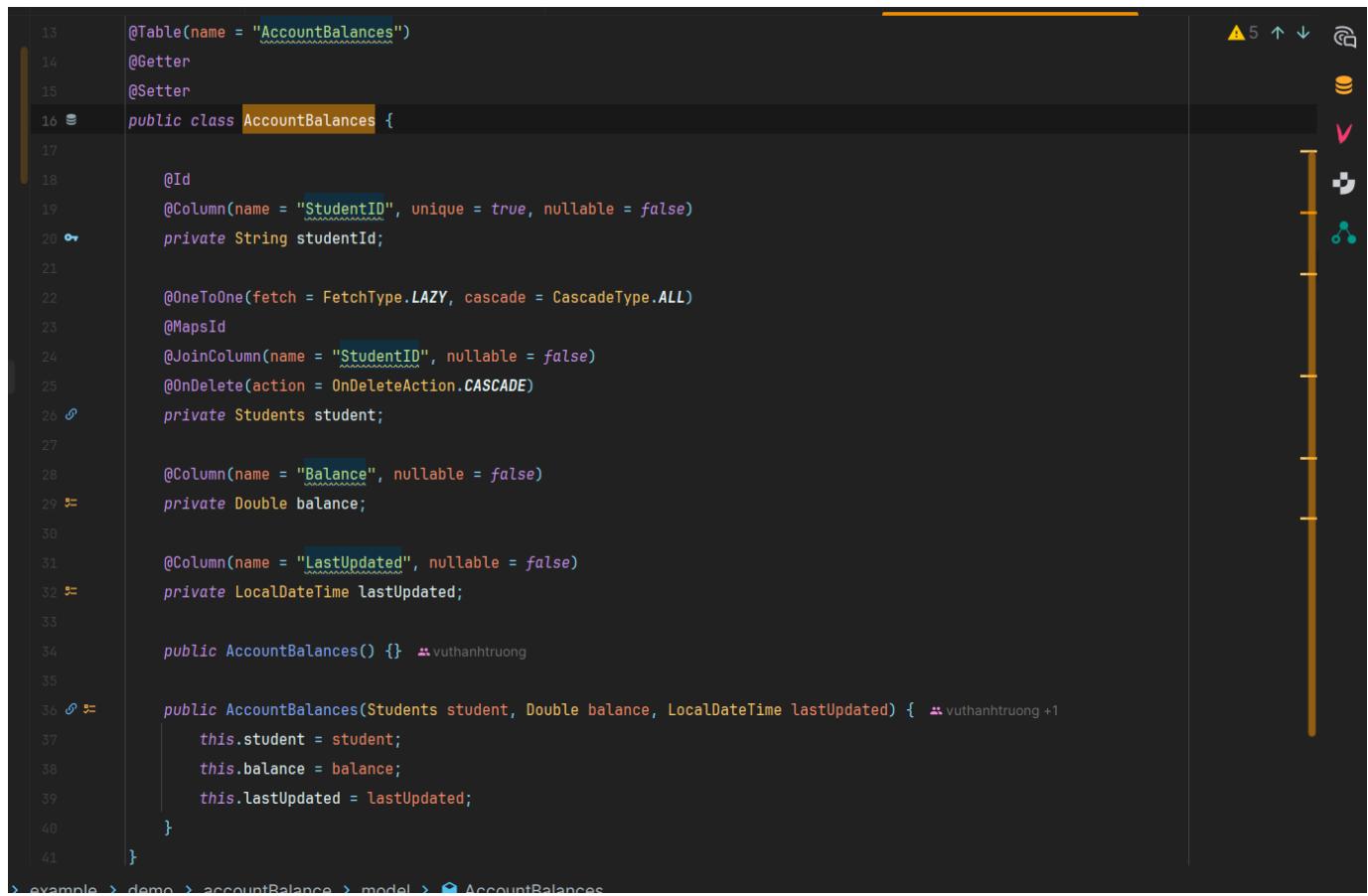
4.5 Performance Optimization Techniques

Database Connection Pooling with HikariCP

The application leverages HikariCP, one of the fastest and most efficient Java database connection pool implementations, to manage database connections efficiently. Connection

pooling eliminates the overhead of creating and destroying database connections for each request, which is one of the most expensive operations in database-driven applications. The HikariCP configuration is optimized differently for development and production environments to balance resource usage and performance. In development mode, the connection pool is configured with a maximum pool size of 10 connections, a minimum idle of 2 connections, a connection timeout of 300 seconds, an idle timeout of 300 seconds (5 minutes), and a maximum connection lifetime of 1800 seconds (30 minutes). These settings provide adequate performance for development while conserving system resources. In production mode, the pool size is increased to a maximum of 20 connections and a minimum idle of 5 connections to handle higher concurrent request volumes. The connection timeout ensures that requests don't wait indefinitely for available connections, preventing request queue buildup during peak loads. The idle timeout automatically closes connections that haven't been used for the specified duration, freeing up database resources. The maximum lifetime setting ensures that connections are periodically refreshed, preventing issues with stale connections that may have been closed by the database server or network infrastructure. HikariCP's efficient connection management, combined with these carefully tuned parameters, ensures that database operations execute quickly without connection acquisition delays, while maintaining optimal resource utilization.

JPA Lazy Loading Strategy



```

13     @Table(name = "AccountBalances")
14
15     @Getter
16     @Setter
17
18     public class AccountBalances {
19
20         @Id
21         @Column(name = "StudentID", unique = true, nullable = false)
22         private String studentId;
23
24         @OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
25         @MapsId
26         @JoinColumn(name = "StudentID", nullable = false)
27         @OnDelete(action = OnDeleteAction.CASCADE)
28         private Students student;
29
30         @Column(name = "Balance", nullable = false)
31         private Double balance;
32
33         @Column(name = "LastUpdated", nullable = false)
34         private LocalDateTime lastUpdated;
35
36         public AccountBalances() {} vuthanhtruong
37
38         public AccountBalances(Students student, Double balance, LocalDateTime lastUpdated) { vuthanhtruong +1
39             this.student = student;
40             this.balance = balance;
41             this.lastUpdated = lastUpdated;
42         }
43     }

```

The code editor interface includes a status bar at the bottom with navigation icons: example, demo, accountBalance, model, and a file icon labeled AccountBalances.

Code 63 Account balances class

```

14  @Table(name = "FinancialHistories")
15  @Inheritance(strategy = InheritanceType.JOINED)
16  @Getter
17  @Setter
18  public abstract class FinancialHistories {
19
20      @Id
21      @Column(name = "HistoryID")
22      private String historyId;
23
24      @ManyToOne(fetch = FetchType.LAZY)
25      @JoinColumn(name = "StudentID", nullable = false)
26      private Students student;
27
28      @ManyToOne(fetch = FetchType.LAZY)
29      @JoinColumn(name = "AccountBalanceID", nullable = false)
30      private AccountBalances accountBalance;
31
32      @Column(name = "CurrentAmount", nullable = false, precision = 15, scale = 2)
33      private BigDecimal currentAmount;
34
35      @Column(name = "ChangedAmount", nullable = false, precision = 15, scale = 2)
36      private BigDecimal changedAmount = BigDecimal.ZERO;
37
38      @Column(name = "CreatedAt", nullable = false)
39      private LocalDateTime createdAt;
40
41      @Enumerated(EnumType.STRING)
42      @Column(name = "Status", nullable = false, length = 50)

```

example > demo > financialHistory > financialHistories > model >  FinancialHistories

Code 64 FinancialHistories class

The application extensively uses lazy loading (`FetchType.LAZY`) for entity relationships to minimize database queries and memory consumption. Lazy loading defers the loading of related entities until they are explicitly accessed, which is crucial for performance when dealing with entities that have multiple relationships. For example, the `FinancialHistories` entity uses lazy loading for its relationships with `AccountBalances` and `Students`, meaning that when a list of financial histories is retrieved, the related account balance and student entities are not loaded from the database until code explicitly accesses these relationships. This approach prevents the N+1 query problem, where loading a list of entities would trigger additional queries for each entity's relationships, resulting in hundreds or thousands of unnecessary database queries. The application uses `JOIN FETCH` strategically in JPQL queries when related entities are known to be needed, providing explicit control over eager loading. For instance, queries that retrieve lists of lecturers use `JOIN FETCH` to eagerly load the campus and creator relationships in a single query, avoiding multiple round trips to the database. This selective eager loading approach balances query efficiency with memory usage, ensuring that related data is loaded efficiently when needed while avoiding unnecessary data loading when it's not. The lazy loading strategy is particularly important for list views where hundreds of records may be displayed, as eagerly

loading all relationships would result in excessive memory consumption and slow query execution times.

Query Optimization with Pagination

Code 65 StudentDAOImpl

The application implements comprehensive pagination across all list views to limit the amount of data retrieved and displayed at once, significantly improving page load times and reducing memory usage. Pagination is implemented using JPA's `setFirstResult()` and `setMaxResults()` methods, which translate to SQL `LIMIT` and `OFFSET` clauses, allowing the database to return only the requested subset of records. For example, the support tickets list view retrieves only 10-20 tickets per page instead of loading all tickets from the database, which could number in the thousands. Each paginated query is accompanied by a count query that determines the total number of records, enabling the UI to display pagination controls with accurate page counts. The pagination implementation includes proper offset calculation based on the current page number and page size, ensuring that users can navigate through large datasets efficiently. Search functionality is integrated with pagination, where search queries also use `setFirstResult()` and `setMaxResults()` to paginate search results, preventing performance degradation when searching through large datasets. The pagination parameters are validated to prevent negative offsets or invalid page sizes, with default values applied when invalid parameters are provided. This pagination strategy ensures that database queries remain fast regardless of the total dataset size, as queries only retrieve the specific records needed for the current page view.

JOIN FETCH for Eager Loading Optimization

```

@Override 1 usage  vuthanhtruong
public List<Staffs> getYourManagersPaginated(String campusId, String majorId, int firstResult, int pageSize) {
    if (campusId == null || majorId == null || pageSize <= 0 || firstResult < 0) {
        return List.of();
    }

    String jpql = """
        SELECT DISTINCT s FROM Staffs s
        JOIN FETCH s.campus c
        JOIN FETCH s.majorManagement m
        WHERE s.campus.campusId = :campusId
        AND m.majorId = :majorId
        ORDER BY s.lastName, s.firstName, s.id
        """;
}

return entityManager.createQuery(jpql, Staffs.class)
    .setParameter( s: "campusId", campusId)
    .setParameter( s: "majorId", majorId)
    .setFirstResult(firstResult)
    .setMaxResults(pageSize)
    .getResultList();
}

```

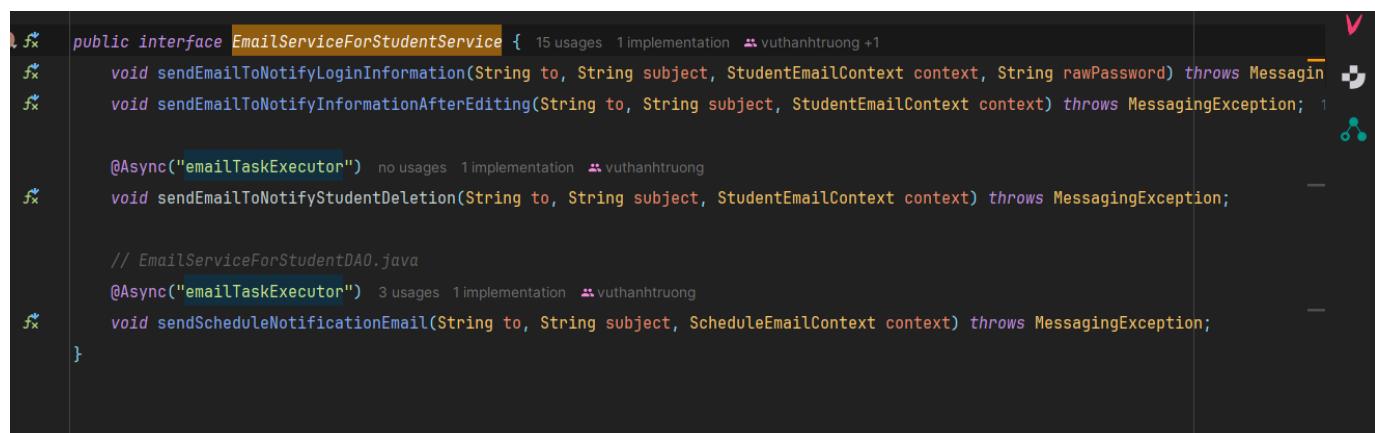
Code 66 JOIN FETCH

The application uses `JOIN FETCH` in JPQL queries to efficiently load related entities in a single database query, eliminating the N+1 query problem. When entities have relationships that will definitely be accessed, `JOIN FETCH` is used to load these relationships eagerly within the same query that retrieves the main entities. For example, queries that retrieve lists of staff members use `JOIN FETCH` to load the campus and major management relationships: `SELECT s FROM Staffs s JOIN FETCH s.campus JOIN FETCH s.majorManagement`. This single query retrieves all necessary data in one database round trip, rather than executing separate queries for each staff member's campus and major management relationships. The `JOIN FETCH` approach is particularly effective for one-to-many and many-to-one relationships where the related entities are needed for display purposes. The application also uses `LEFT JOIN FETCH` for optional relationships, ensuring that entities are still retrieved even when optional relationships are null. This query optimization technique significantly reduces database round trips, which is one of the primary performance bottlenecks in database-driven applications, resulting in faster page load times and reduced database server load.

Spring Cache Implementation

The application implements Spring's caching abstraction to cache frequently accessed data and reduce database queries. The cache configuration uses `ConcurrentMapCacheManager`, which provides in-memory caching suitable for single-instance deployments. The cache is configured with three named caches: "totalTuition" for caching tuition calculation results, "oauth2Users" for caching OAuth2 user information, and "users" for caching user entity data. Caching is enabled through the `@EnableCaching` annotation, and methods that should use caching are annotated with `@Cacheable`, `@CacheEvict`, or `@CachePut` annotations. The cache stores computed values and frequently accessed entities in memory, eliminating the need to query the database or perform expensive calculations repeatedly. For example, tuition calculations that involve aggregating data from multiple tables can be cached, so subsequent requests for the same calculation use the cached result instead of re-executing the expensive query. Cache eviction strategies ensure that cached data is refreshed when underlying data changes, maintaining data consistency while benefiting from performance improvements. While the current implementation uses simple in-memory caching, the configuration is designed to support more sophisticated caching solutions like Redis or Ehcache for distributed deployments, as indicated by the Ehcache provider configuration in the application properties.

Asynchronous Email Processing



```

public interface EmailServiceForStudentService {
    void sendEmailToNotifyLoginInformation(String to, String subject, StudentEmailContext context, String rawPassword) throws MessagingException;
    void sendEmailToNotifyInformationAfterEditing(String to, String subject, StudentEmailContext context) throws MessagingException;

    @Async("emailTaskExecutor")
    void sendEmailToNotifyStudentDeletion(String to, String subject, StudentEmailContext context) throws MessagingException;

    // EmailServiceForStudentDAO.java
    @Async("emailTaskExecutor")
    void sendScheduleNotificationEmail(String to, String subject, ScheduleEmailContext context) throws MessagingException;
}

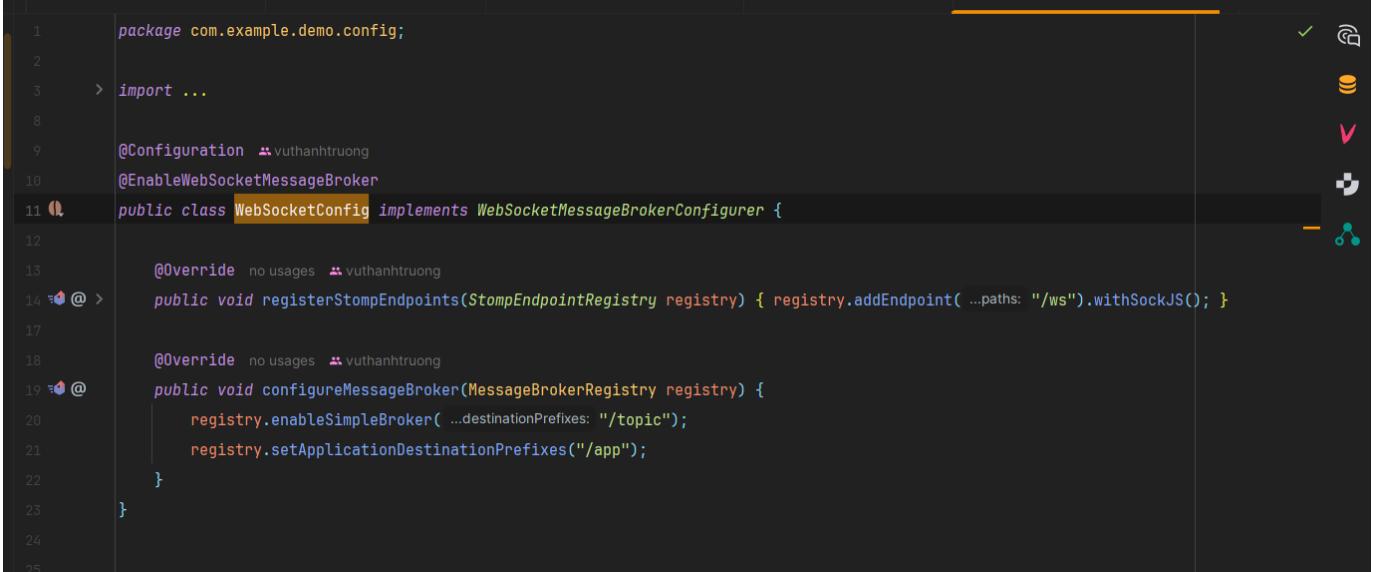
```

Code 67 Asynchronous

The application implements asynchronous email sending to prevent email operations from blocking user requests and slowing down the application. Email sending operations are marked with the `@Async` annotation and configured to use a dedicated thread pool executor named "emailTaskExecutor". The async configuration defines a thread pool with a core pool size of 5 threads, a maximum pool size of 10 threads, and a queue capacity of 100 tasks. This configuration ensures that email sending operations are executed in background threads, allowing the main request processing threads to continue handling user requests without waiting for email delivery to complete. When a user account is created or updated, the system initiates email sending

asynchronously, immediately returning a response to the user while the email is sent in the background. The thread pool executor is configured with appropriate thread naming ("EmailThread-") for easier monitoring and debugging. The queue capacity prevents memory issues during email sending spikes by limiting the number of pending email tasks. This asynchronous approach is particularly important for operations like bulk user creation, where sending emails synchronously would result in unacceptable response times. The async email processing ensures that critical operations like user account creation complete quickly, while email notifications are delivered reliably in the background without impacting user experience.

WebSocket Performance Configuration

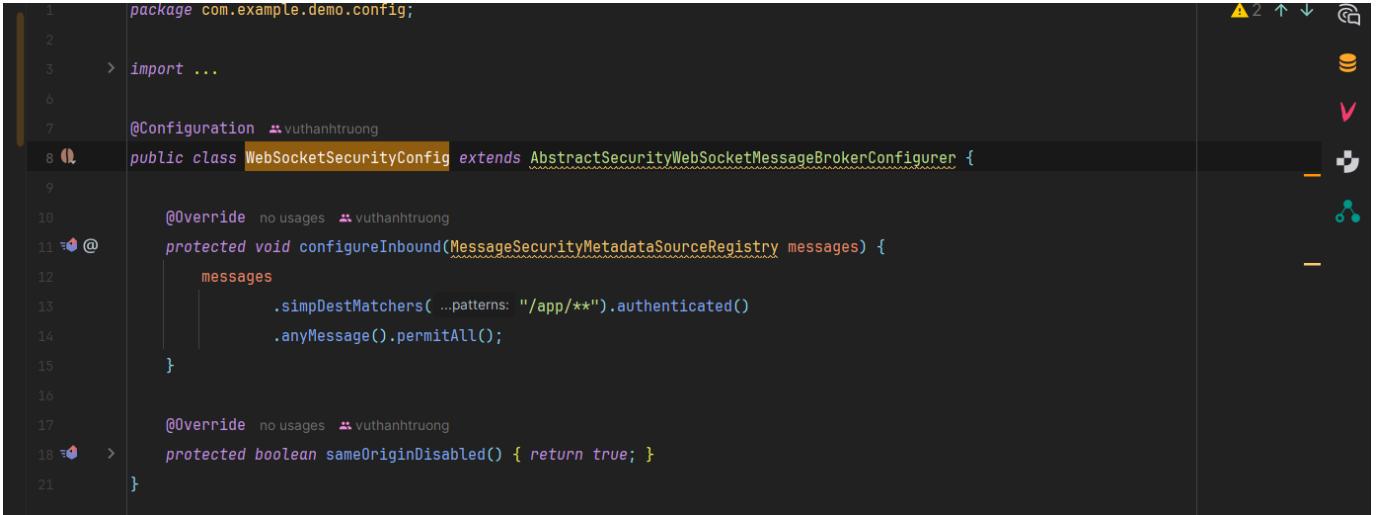


```

1 package com.example.demo.config;
2
3 > import ...
4
5 @Configuration
6 & vuthanhtruong
7 @EnableWebSocketMessageBroker
8
9 public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {
10
11     @Override
12     public void registerStompEndpoints(StompEndpointRegistry registry) {
13         registry.addEndpoint("/ws").withSockJS();
14     }
15
16     @Override
17     public void configureMessageBroker(MessageBrokerRegistry registry) {
18         registry.enableSimpleBroker("/topic");
19         registry.setApplicationDestinationPrefixes("/app");
20     }
21
22 }
23
24
25

```

Code 68 WebSocket class



```

1 package com.example.demo.config;
2
3 > import ...
4
5 @Configuration
6 & vuthanhtruong
7
8 public class WebSocketSecurityConfig extends AbstractSecurityWebSocketMessageBrokerConfigurer {
9
10     @Override
11     protected void configureInbound(MessageSecurityMetadataSourceRegistry messages) {
12         messages
13             .simpDestMatchers("/app/**").authenticated()
14             .anyMessage().permitAll();
15     }
16
17     @Override
18     protected boolean sameOriginDisabled() {
19         return true;
20     }
21 }

```

Code 69 WebSocketConfig

The application optimizes WebSocket communication for real-time messaging features through carefully configured message size limits and buffer settings. The WebSocket message size limit is set to 16KB, preventing excessively large messages from consuming excessive memory or causing communication delays. The send buffer size limit is configured to 512KB, providing adequate buffering for message queuing during high-traffic periods while preventing unbounded memory growth. The send time limit is set to 20 seconds, ensuring that slow message delivery doesn't block WebSocket connections indefinitely. These configurations ensure that real-time messaging remains responsive and efficient, even when multiple users are actively messaging simultaneously. The WebSocket optimization prevents memory leaks and connection issues that could degrade performance or cause service disruptions. The STOMP protocol implementation provides efficient message routing and delivery, minimizing overhead in real-time communication while maintaining reliable message delivery guarantees.

4.6 Security Implementation

4.6.1 Data Protection Measures

The Greenwich University Management System has also undertaken multi-layered data protection policies to provide confidentiality, integrity and availability of user data in addition to sensitive information in the system. The authentication and authorization layer of the system employs the Spring Security framework with the BCrypt password encryption mechanism to secure the user login credentials where all the passwords are hashed using the BCrypt algorithm with automatic salt before being stored in the database, such that even in case of database intrusion, the attackers will not be able to retrieve the original passwords. This system enables several authentication such as the use of the standard form-based authentication and OAuth2 using Google, which allows users to select the most suitable authentication method and at the same time uses JWT (JSON Web Tokens) to authenticate APIs, which increases the security of messages between the client and the server. The system has Role-Based Access Control (RBAC) at access control layer that has seven major roles such as Admin, Deputy Staff, Staff, Major Lecturer, Minor Lecturer, Student, and Parent where each role is assigned particular access privilege using the Spring Security configuration to ensure that only users with specific roles can access resources and functions that are related to their roles. The system uses security on a variety of levels such as URL-based security using pattern matching to control path security, method-level security using the use of the annotation of over 70 service and controller methods with the annotation of access control at the method level, and a session-based security to manage user sessions, with all the endpoints that require authentication being secured and only static resources such as CSS, JavaScript, and images being allowed to be accessed by the public. The system has also employed numerous layers of security such as CSRF (Cross-Site Request Forgery) protection, which is enabled whenever a form is submitted to ward off common attacks, XSS (Cross-Site Scripting), which is enforced through input sanitization by the system to prevent CSRF attacks as well as SQL Injection, which is ensured through the use of JPA/Hibernate with parameterized queries as all the database queries are made with named parameters rather than concatenation of the strings to make sure that the possibility of attackers injecting malicious SQL code into the system is completely eliminated. The management of sessions is also implemented in a secure

manner with specific settings such as session time out has been set to 30 minutes in order to automatically log out users after inactivity and minimization of the risk of using the session when the user forgets to log out of these devices since the session cookies have been set to HTTP-only and therefore the user cannot use the session after they have logged out and the system automatically invalidates the session (`invalidateHttpSession`) and deletes the JSESSIONID cookie to ensure that the session cannot be reused after logging out. In managing passwords, the system has secure password reset capability through token based password reset system, where, when users request password reset, the system will create a unique time-limited token (1 hour) in form of a token that is sent via email, this token will be stored in the database with the user details and expiry date so that the email owner is the only person capable of changing the password and also the token is a short-lived token (1 hour) which is not deleted with the user details in the database. The file upload data protection is achieved by various layers of validation and verification like 50MB per file and 55MB total request to block DoS attacks with large file uploads, file type verification by checking file MIME type and only safe file types including PDF, DOC, DOCX, TXT, PPT, PPTX, ZIP, and image file JPEG/ PNG ensures that files are uploaded only by authorized users to prevent system overload as well as to ensure data integrity and security. Financial data security is also emphasized significantly by integrating with Stripe, which is currently among the most reputable and secure payment gateways, all payment transactions are handled through Stripe, which means that no credit card information and sensitive payment data are ever stored in the system, only the necessary information is the amount, type of transaction, and status of payment are stored, and the whole process is executed by the Stripe through the PCI DSS standards of security. In production environments, data protection on transmission is maintained by having the use of HTTPS by encrypting all data transmitted between the client and the server, in email services, SMTP with STARTTLS are used to encrypt email connections and to ensure that the transmission of emails with sensitive information like password reset tokens is ensured since it is transmitted in a secure manner and WebSocket connections are also used to provide real time messaging to protect against unauthorized access to the communication channels. Lastly, the system employs data protection on the database level by utilizing HikariCP connection pooling with optimal production-friendly settings to provide the database connections with stability and security, all sensitive data including database credentials, API keys, and OAuth2 secrets are stored in environment variables rather than source code to meet the security best practice, and the system also has a logging system to keep track of important activities and security events to react in a short period to potential threats. When all these measures are combined it will provide a complete data protection system which will secure the user information and the sensitive system data to the maximum.

4.6.2 Vulnerability Prevention

The Greenwich University Management System has a comprehensive set of vulnerability prevention mechanisms to detect, mitigate and prevent security vulnerability before it can be exploited. Input validation is a form of defense against injection attacks and data corruption where the system uses many levels of validation such as Jakarta Bean Validation annotations, e.g. 10 letters, numbers, spaces, and common punctuation marks, i.e. with optional international prefix) 10-15 digits, i.e. with optional international prefix) 4 digits, i.e. with optional international prefix) 5 digits, i.e. with optional international prefix) 9 digits, i.e. with optional international prefix) 11 digits,

i. The system puts in place strict validation of file uploads in order to thwart malicious file uploads and denial-of-service attacks whereby file size is capped to 50MB per file and 55MB per request to thwart resource overload, MIME type checking is used to only allow safe file types like PDF, DOC, DOCX, TXT, PPT, PPTX, ZIP, and image files (JPEG, PNG) and the maximum number of files to upload is capped to prevent overload of the system (such as a maximum of 5). Prevention of SQL Injection is done by using only parameterized queries via JPA/Hibernate, all the queries to the database are done with named parameters rather than concatenated strings and the system does not build SQL queries based on user inputs and thus does not create any chances of SQL injection attack. Cross-site Scripting (XSS) mitigation is enforced by input sanitizers that filter user inputs to ensure they are safe and acceptable, Thymeleaf template engine does autosanitization of HTML entities during the rendering of user-generated data and the system uses whitelist-based authentication that only permits safe characters and patterns to user inputs. All form submissions are secured with Cross-site Request Forgery (CSRF) protection, which is provided by the default CSRF token mechanism of Spring Security, in which every form submission must include a valid CSRF token generated and verified by the framework, and WebSocket endpoints have specifically been omitted CSRF protection because they have their own authentication mechanism. Authentication failure handling is done on the sound basis of avoiding information disclosure and brute force attacks where generic error messages, including Invalid username or password, are presented, authentication failures are recorded with the different level of warning to securely monitor the vulnerability and the system has implemented secure session management to avoid session fixation attack. The generation of password reset tokens by the system relies on cryptographically secure random number generation via SecureRandom to generate student ID and UUID.randomUUID to generate unique tokens, the time-free nature of the tokens expires after an hour to reduce the time window during which an attacker may attack and the tokens are stored safely in the database with an appropriate index to validate tokens easily. Error management and exception management aim to eliminate vulnerabilities to information disclosure, in which generic error messages are returned to the user rather than detailed stack traces or system information, exceptions are caught and gracefully handled at the correct layer (controller, service and DAO), error logging provides detailed information to the administrator and the system sensitive information such as database structure, file paths or internal system information is not disclosed in error messages. Sanitization of inputs is done on several levels such as validation of names, emails, phone numbers and other user inputs using a regular expression pattern to ensure that only expected file types are processed, type checking of file uploads to prevent attacks of buffer overflow and resource exhaustion, duplication through database constraints and application checks to ensure that emails and phone numbers are unique. The secure practices of coding are implemented in the entire system such as using parameterized queries that are used only in database operations, proper exception handling which does not expose sensitive information, validation at various levels (presentation, business logic, and data access), and principle of least privilege whereby users access only those resources that are relevant to their roles.

4.6.3 Security Testing Results

4.7 Technical Challenges and Solutions

4.7.1 Challenge 1: [Specific Challenge]

4.7.2 Challenge 2: [Specific Challenge]

4.7.3 Solutions and Workarounds

4.8 Key Code Implementations (with code snippets)

```
25 public class SecurityConfig {
26
27     ...
28
29     @Bean
30     public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
31         http
32             .authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
33                 .requestMatchers( "/student-home/**", "/api/student-home/**").hasRole("STUDENT")
34                 .requestMatchers( "/staff-home/**", "/api/staff-home/**").hasRole("STAFF")
35                 .requestMatchers( "/major-lecturer-home/**", "/api/lecturer-home/**").hasRole("LECTURER")
36                 .requestMatchers( "/minor-lecturer-home/**", "/api/minor-lecturer-home/**").hasRole("MINOR")
37                 .requestMatchers( "/deputy-staff-home/**", "/api/deputy-staff-home/**").hasRole("DEPUTY")
38                 .requestMatchers( "/admin-home/**", "/api/admin-home/**").hasRole("ADMIN")
39                 .requestMatchers( "/parent-home/**", "/api/parent-home/**").hasRole("PARENT")
40                 .requestMatchers( "/major-timetable/**", "/specialized-timetable/**").hasAnyRole( ...roles: "LECTURER", "STAFF", "ADMIN")
41                 .requestMatchers( "/classroom/**", "/messages/**", "/check-news/**", "/documents/**").hasAnyRole( ...roles: "STUDENT", "LECTURER")
42                 .requestMatchers(
43                     "/login",
44                     "/persons/**",
45                     "/resources/**",
46                     "/css/**",
47                     "/js/**",
48                     "/*.css",
49                     "/*.js",
50                     "/oauth2/**",
51                     "/home",
52                     "/auth/reset-password**",
53                     "/ws/**", // WebSocket endpoint
54                 )
55             )
56         );
57     }
58
59 }
```

Code 70 Spring security

```

63             .anyRequest().authenticated()
64         )
65         .csrf( CsrfConfigurer<HttpSecurity> csrf -> csrf
66             .ignoringRequestMatchers( ...patterns: "/ws/**", "/app/**", "/topic/**")
67         )
68         .formLogin( FormLoginConfigurer<HttpSecurity> form -> form
69             .loginPage( "/login")
70             .loginProcessingUrl("/login")
71             .successHandler(( HttpServletRequest req, HttpServletResponse res, Authentication auth) -> {
72                 String redirectUrl = getRedirectUrlByRole(auth.getAuthorities());
73                 res.sendRedirect(redirectUrl != null ? redirectUrl : "/login");
74             })
75             .failureHandler(flashErrorHandler( defaultMessage: "Invalid username or password." ))
76             .permitAll()
77         )
78         .oauth2Login( OAuth2LoginConfigurer<HttpSecurity> oauth2 -> oauth2
79             .loginPage( "/login")
80             .userInfoEndpoint( UserInfoEndpointConfig userInfo -> userInfo.oidcUserService(customOAuth2UserService))
81             .successHandler(( HttpServletRequest req, HttpServletResponse res, Authentication auth) -> {
82                 var principal = (CustomOidcUserPrincipal) auth.getPrincipal();
83                 String redirectUrl = getRedirectUrlByRole(principal.getAuthorities());
84                 res.sendRedirect(redirectUrl != null ? redirectUrl : "/login");
85             })
86             .failureHandler(flashErrorHandler( defaultMessage: "OAuth2 login failed. Please try again." ))
87         )
88         .logout( LogoutConfigurer<HttpSecurity> logout -> logout

```

Code 71 Spring security

```

100
101 @ private AuthenticationFailureHandler flashErrorHandler(String defaultMessage) { 2 usages ↗ Trunks +1
102     return new AuthenticationFailureHandler() { ↗ Trunks +1
103         @Override no usages ↗ Trunks
104         public void onAuthenticationFailure(HttpServletRequest request,
105                                         HttpServletResponse response,
106                                         org.springframework.security.core.AuthenticationException exception)
107             throws IOException, ServletException {
108                 logger.warn("Authentication failed: {}", exception.getMessage());
109                 request.getSession().setAttribute( s: "error", defaultMessage);
110                 response.sendRedirect( s: "/login");
111             }
112         };
113     }
114
115 @ private String getRedirectUrlByRole(Collection<?> authorities) { 2 usages ↗ Trunks +1
116     if (authorities.contains(new SimpleGrantedAuthority( role: "ROLE_STUDENT")))) return "/student-home";
117     if (authorities.contains(new SimpleGrantedAuthority( role: "ROLE_STAFF")))) return "/staff-home";
118     if (authorities.contains(new SimpleGrantedAuthority( role: "ROLE_LECTURER")))) return "/major-lecturer-home";
119     if (authorities.contains(new SimpleGrantedAuthority( role: "ROLE_ADMIN")))) return "/admin-home";
120     if (authorities.contains(new SimpleGrantedAuthority( role: "ROLE_DEPUTY")))) return "/deputy-staff-home";
121     if (authorities.contains(new SimpleGrantedAuthority( role: "ROLE_MINOR")))) return "/minor-lecturer-home";
122     if (authorities.contains(new SimpleGrantedAuthority( role: "ROLE_PARENT")))) return "/parent-home";
123     return null;
124     }
125     @Bean ↗ vuthanhtruong

```

Code 72 Spring security

```
41     try {
42         person = entityManager.createQuery(
43             "SELECT p FROM Persons p JOIN Authenticators a ON p.id = a.personId "
44             + "WHERE p.email = :email AND a.accountStatus = :st",
45             Persons.class)
46             .setParameter("email", email)
47             .setParameter("st", AccountStatus.ACTIVE)
48             .getSingleResult();
49     } catch (NoResultException e) {
50         throw new OAuth2AuthenticationException(new OAuth2Error(errorCode: "user_not_found"), "User not found in DB");
51     }
52
53     String role = determineRole(person);
54     var authorities = List.of(new SimpleGrantedAuthority(role));
55
56     return new CustomOidcUserPrincipal(oidcUser, person, authorities);
57 }
58
59 @
60 private String determineRole(Persons person) { 1 usage ↗ Trunks +1
61     if (person instanceof Staffs) return "ROLE_STAFF";
62     if (person instanceof MajorLecturers) return "ROLE_LECTURER";
63     if (person instanceof Students) return "ROLE_STUDENT";
64     if (person instanceof Admins) return "ROLE_ADMIN";
65     if (person instanceof DeputyStaffs) return "ROLE_DEPUTY";
66     if (person instanceof MinorLecturers) return "ROLE_MINOR";
67     if (person instanceof ParentAccounts) return "ROLE_PARENT";
68     return "ROLE_USER";
```

Code 73 security

```

@Override no usages Trunks +1
@Transactional(readOnly = true)
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    long start = System.currentTimeMillis();
    try {
        Persons person = entityManager.createQuery(
            s: "SELECT p FROM Persons p JOIN Authenticators a ON p.id = a.personId " +
            "WHERE (p.id = :u OR p.email = :u OR p.phoneNumber =: u) AND a.accountStatus = :st",
            Persons.class)
            .setParameter( s: "u", username)
            .setParameter( s: "st", AccountStatus.ACTIVE)
            .getSingleResult();

        Authenticators auth = entityManager.createQuery(
            s: "SELECT a FROM Authenticators a WHERE a.personId = :pid",
            Authenticators.class)
            .setParameter( s: "pid", person.getId())
            .getSingleResult();

        String role = determineRole(person);
        var authorities = List.of(new SimpleGrantedAuthority(role));
        String effectiveUsername = person.getEmail() != null ? person.getEmail() : person.getId();

        String password = auth.getPassword();
        if (password == null || password.trim().isEmpty()) {
            throw new UsernameNotFoundException("User has no password set: " + username);
        }
    }
}

```

Code 74 loadUsername

```

@Override 1 usage Trunks
public String generateUniqueStudentId(String majorId, LocalDate createdDate) {
    String prefix = majorId != null ? majorId : "GEN";
    String year = String.format("%02d", createdDate.getYear() % 100);
    String date = String.format("%02d%02d", createdDate.getMonthValue(), createdDate.getDayOfMonth());
    String studentId;
    SecureRandom random = new SecureRandom();
    do {
        String randomDigit = String.valueOf(random.nextInt(bound: 10));
        studentId = prefix + year + date + randomDigit;
    } while (personsService.existsPersonById(studentId));
    return studentId;
}

```

Code 75 Random ID

These code implementations illustrate the comprehensive security measures integrated throughout the University of Greenwich Management System. Each component works together to

create a multi-layered security architecture that protects the system from common vulnerabilities while maintaining usability and performance. The use of industry-standard libraries (Spring Security, BCrypt), secure coding methods (parameterized queries, input validation), and defense-in-depth strategies ensure the system remains robust against known and emerging security threats.

4.9 Chapter Summary

This chapter documented the practical implementation of the Greenwich University Management System (GUMS) from an engineering perspective. It began by describing the development environment, version control strategy, and CI/CD pipeline that collectively ensure disciplined, repeatable and reliable delivery of new features. The chapter then detailed the internal structure of the application, explaining how the feature-based modular organisation, layered architecture, and DAO–JPA persistence approach support clean separation of concerns, maintainability and future extensibility. Core modules—such as authentication and authorisation, timetable and class management, assessment handling, communications, finance, and parent–student linkage—were presented with code-level explanations that demonstrate how the earlier design artefacts are realised in practice. Furthermore, the integration of external services, including email delivery, Stripe-based payment processing and UI libraries, illustrated how GUMS interacts with the broader digital ecosystem while maintaining security, reliability and usability. Finally, the chapter highlighted key implementation decisions and best practices, such as secure coding techniques and configuration management, which collectively underpin the robustness of the deployed system. These implementation details form the operational backbone that the subsequent testing and evaluation activities build upon.

CHAPTER 5: TESTING, VALIDATION, AND QUALITY ASSURANCE

Table 16 Overview of the Test Plan and Testing Strategy

Test Case ID	Module / Area	Test Type	Description / Scenario	Expected Result	Actual Result	Status
TC-AUTH-001	Authentication & Security	Functional	Valid login with correct credentials	User successfully logged in, redirected to student dashboard.	User successfully logged in, redirected to student dashboard.	Pass
TC-AUTH-002	Authentication & Security	Functional	Login with incorrect password	Login failed, error message displayed.	Login failed, error message displayed correctly.	Pass
TC-AUTH-003	Authentication & Security	Functional	Login with non-existing account	Login failed, error message displayed.	Login failed, error message displayed correctly.	Pass
TC-AUTH-004	Authentication & Security	Functional	Login via Google OAuth2	User authenticated via Google, redirected to appropriate dashboard based on role.	User authenticated via Google OAuth2, redirected to appropriate dashboard based on role.	Pass
TC-AUTH-005	Authentication & Security	Functional	Request password reset	Password reset email sent, token generated.	Password reset email sent successfully, token generated and stored.	Pass
TC-AUTH-006	Authentication & Security	Functional	Reset password using valid token	Password reset successful, user can login	Password reset successful, user can login with	Pass

				with new password.	new password.	
TC-AUTH-007	Authentication & Security	Functional	Reset password using invalid/expired token	Password reset failed, error message displayed.	Password reset failed, error message displayed correctly.	Pass
TC-AUTH-008	Authentication & Security	Security	SQL injection attempt on login	Login failed, SQL injection prevented.	SQL injection attempt blocked, login failed, security measures working correctly.	Pass
TC-AUTH-009	Authentication & Security	Security	Session timeout after inactivity	Session expired, user redirected to login page.	Session expired after 30 minutes of inactivity, user redirected to login page.	Pass
TC-AUTH-010	Authentication & Security	Security	Access protected resource without proper permission	Access denied, 403 error or redirect to unauthorized page.	Access denied, 403 error returned, unauthorized access prevented.	Pass
TC-USER-001	User Management - Students	Functional	Create new student account	Student created successfully, account created, email notification sent.	Student created successfully, account created, email notification sent to student.	Pass
TC-USER-002	User Management - Students	Functional	View student details	Student information displayed correctly.	Student information displayed correctly with all details.	Pass

TC-USER-003	User Management - Students	Functional	Update student information	Student information updated, email notification sent.	Student information updated successfully, email notification sent.	Pass
TC-USER-004	User Management - Students	Functional	Delete student account	Student deleted, related records handled, email notification sent.	Student deleted, related records handled properly, email notification sent.	Pass
TC-USER-005	User Management - Students	Functional	Search students by keyword "John"	List of students matching search keyword "John" displayed.	List of students matching search keyword "John" displayed correctly.	Pass
TC-USER-006	User Management - Students	Functional	View student list with pagination	First 10 students displayed, pagination controls shown and working correctly.	First 10 students displayed, pagination controls shown and working correctly.	Pass
TC-USER-007	User Management - Students	Validation	Create student with invalid email	Validation error displayed, student not created.	Validation error displayed for invalid email, student not created.	Pass
TC-USER-008	User Management - Students	Validation	Create student with invalid phone number	Validation error displayed, student not created.	Validation error displayed for invalid phone number, student not created.	Pass

TC-USER-009	User Management - Students	Functional	Link/create parent account for student	Parent account created/linked , email sent to parent.	Parent account created and linked to student, email sent to parent successfully.	Pass
TC-USER-010	User Management - Students	Functional	Remove parent–student link	Parent link removed, student can no longer be accessed by parent.	Parent link removed successfully, parent access revoked.	Pass
TC-USER-011	User Management - Staff	Functional	Create new staff account	Staff created successfully, account created, email notification sent.	Staff created successfully, account created, email notification sent to staff.	Pass
TC-USER-012	User Management - Staff	Functional	View staff details	Staff information displayed correctly.	Staff information displayed correctly with all details.	Pass
TC-USER-013	User Management - Staff	Functional	Update staff information	Staff information updated, email notification sent.	Staff information updated successfully, email notification sent.	Pass
TC-USER-014	User Management - Staff	Functional	Delete staff account	Staff deleted, related records handled, email notification sent.	Staff deleted, related records handled properly, email notification sent.	Pass
TC-USER-015	User Management - Staff	Functional	Search staff by keyword "Jane"	List of staff matching search keyword "Jane" displayed.	List of staff matching search keyword "Jane"	Pass

					displayed correctly.	
TC- USER- 016	User Management - Lecturers	Functional	Create major lecturer account	Major lecturer created successfully, account created, email notification sent.	Major lecturer created successfully, account created, email notification sent.	Pass
TC- USER- 017	User Management - Lecturers	Functional	Create minor lecturer account	Minor lecturer created successfully, account created, email notification sent.	Minor lecturer created successfully, account created, email notification sent.	Pass
TC- USER- 018	User Management - Lecturers	Functional	View lecturer details	Lecturer information displayed correctly.	Lecturer information displayed correctly with all details.	Pass
TC- USER- 019	User Management - Lecturers	Functional	Assign specialization to lecturer	Specialization assigned, lecturer can teach specialized classes.	Specialization assigned successfully, lecturer can now teach specialized classes.	Pass
TC- USER- 020	User Management - Lecturers	Functional	Assign lecturer to class	Lecturer assigned to class, notification sent.	Lecturer assigned to class successfully, notification sent.	Pass
TC- USER- 021	User Management - Admins	Functional	Create admin account	Admin created successfully, account created, email notification sent.	Admin created successfully, account created, email notification sent.	Pass

TC-USER-022	User Management - Admins	Functional	View admin details	Admin information displayed correctly.	Admin information displayed correctly with all details.	Pass
TC-USER-023	User Management - Admins	Security	Verify admin permissions	Full access granted, all admin features available.	Full access granted, all admin features available and working correctly.	Pass
TC-CLAS S-001	Class Management	Functional	Create major class	Major class created successfully.	Major class created successfully with all required information.	Pass
TC-CLAS S-002	Class Management	Functional	Create minor class	Minor class created successfully.	Minor class created successfully with all required information.	Pass
TC-CLAS S-003	Class Management	Functional	Create specialized class	Specialized class created successfully.	Specialized class created successfully with specialization assignment.	Pass
TC-CLAS S-004	Class Management	Functional	View class information, members, posts	Class information, members, and posts displayed.	Class information, members list, and posts displayed correctly.	Pass
TC-CLAS S-005	Class Management	Functional	Enroll student in class (with possible financial processing)	Student enrolled in class, financial processing if needed, email notification sent.	Student enrolled in class, financial processing completed, email notification sent.	Pass

TC-CLAS S-006	Class Management	Functional	Remove student from class	Student removed from class, related records updated.	Student removed from class, related records updated correctly.	Pass
TC-CLAS S-007	Class Management	Functional	Assign lecturer to class	Lecturer assigned, notification sent.	Lecturer assigned to class successfully, notification sent.	Pass
TC-CLAS S-008	Class Management	Functional	Create post in class	Post created, visible to all class members.	Post created successfully, visible to all class members.	Pass
TC-CLAS S-009	Class Management	Functional	View post content, comments, documents	Post content, comments, and documents displayed.	Post content, comments, and documents displayed correctly.	Pass
TC-CLAS S-010	Class Management	Functional	Add comment to class post	Comment added, visible to all class members.	Comment added successfully, visible to all class members.	Pass
TC-CLAS S-011	Class Management	Validation	Create class with empty class name	Validation error displayed, class not created.	Validation error displayed for empty class name, class not created.	Pass
TC-CLAS S-012	Class Management	Functional	Assign room to class	Room assigned, room availability updated.	Room assigned to class successfully, room availability updated.	Pass
TC-TIME-001	Timetable Management	Functional	Create timetable entry (no conflicts)	Timetable entry created, conflict check performed.	Timetable entry created successfully, conflict check	Pass

					performed with no conflicts found.	
TC-TIME-002	Timetable Management	Functional	Create timetable entry with lecturer conflict	Conflict detected, error message displayed, timetable not created.	Lecturer conflict detected, error message displayed, timetable not created.	Pass
TC-TIME-003	Timetable Management	Functional	Create timetable entry with room conflict	Room conflict detected, error message displayed, timetable not created.	Room conflict detected, error message displayed, timetable not created.	Pass
TC-TIME-004	Timetable Management	Functional	View student timetable for specified week	Student's complete timetable for specified week displayed.	Student's complete timetable for specified week displayed correctly.	Pass
TC-TIME-005	Timetable Management	Functional	View lecturer timetable for specified week	Lecturer's complete timetable for specified week displayed.	Lecturer's complete timetable for specified week displayed correctly.	Pass
TC-TIME-006	Timetable Management	Functional	Update timetable entry	Timetable updated, conflict check performed.	Timetable updated successfully, conflict check performed with no conflicts.	Pass
TC-TIME-007	Timetable Management	Functional	Delete timetable entry	Timetable entry deleted, room availability updated.	Timetable entry deleted successfully, room availability updated.	Pass

TC-TIME-008	Timetable Management	Functional	Bulk create timetable entries	All timetable entries created, conflicts checked for each.	All timetable entries created successfully, conflicts checked for each entry.	Pass
TC-ATT-001	Attendance Management	Functional	Mark student as present	Attendance recorded successfully.	Student attendance marked as PRESENT, recorded successfully.	Pass
TC-ATT-002	Attendance Management	Functional	Mark student as absent (with notifications enabled)	Absence recorded, notification sent if configured.	Student absence recorded, notification sent to student and parent.	Pass
TC-ATT-003	Attendance Management	Functional	View attendance list for class session	List of all students with attendance status displayed.	List of all students with attendance status displayed correctly.	Pass
TC-ATT-004	Attendance Management	Functional	View attendance history for student in class	Complete attendance history for student in class displayed.	Complete attendance history for student in class displayed correctly.	Pass
TC-ATT-005	Attendance Management	Functional	Update attendance status	Attendance status updated.	Attendance status updated from ABSENT to LATE successfully.	Pass
TC-ATT-006	Attendance Management	Functional	Record lecturer attendance	Lecturer attendance recorded.	Lecturer attendance recorded successfully.	Pass
TC-ATT-007	Attendance Management	Functional	Generate attendance	Attendance statistics and report	Attendance statistics and report	Pass

			statistics and report	report generated.	generated successfully with accurate data.	
TC-TRAN-001	Academic Transcripts	Functional	Create transcript and auto-calculate letter grade	Transcript created, letter grade calculated automatically.	Transcript created successfully, letter grade calculated automatically based on scores.	Pass
TC-TRAN-002	Academic Transcripts	Functional	View complete academic transcript	Complete academic transcript with all grades displayed.	Complete academic transcript with all grades displayed correctly.	Pass
TC-TRAN-003	Academic Transcripts	Functional	Update transcript and recalculate grade	Transcript updated, letter grade recalculated.	Transcript updated successfully, letter grade recalculated automatically.	Pass
TC-TRAN-004	Academic Transcripts	Functional	View transcript for major subjects	Major subjects transcript displayed.	Major subjects transcript displayed correctly with all major subject grades.	Pass
TC-TRAN-005	Academic Transcripts	Functional	View transcript for minor subjects	Minor subjects transcript displayed.	Minor subjects transcript displayed correctly with all minor subject grades.	Pass
TC-TRAN-006	Academic Transcripts	Functional	View transcript for specialized subjects	Specialized subjects transcript displayed.	Specialized subjects transcript displayed correctly with all	Pass

					specialized subject grades.	
TC-TRAN-007	Academic Transcripts	Validation	Create transcript with invalid score (>10)	Validation error displayed, transcript not created.	Validation error displayed for invalid score (>10), transcript not created.	Pass
TC-FIN-001	Financial - Deposits	Functional	Initiate deposit via Stripe	Deposit history created with PROCESSING status, Stripe checkout session created.	Deposit history created with PROCESSING status, Stripe checkout session created successfully.	Pass
TC-FIN-002	Financial - Deposits	Functional	Complete deposit after successful payment	Account balance updated, deposit status changed to COMPLETED.	Account balance updated correctly, deposit status changed to COMPLETED after successful payment.	Pass
TC-FIN-003	Financial - Deposits	Functional	Failed payment, keep deposit as PROCESSING and allow retry	Deposit status remains PROCESSING	Deposit status remains PROCESSING after failed payment, user can retry payment.	Pass
TC-FIN-004	Financial - Deposits	Functional	Create new Stripe session for existing deposit	New Stripe session created for existing deposit record.	New Stripe session created successfully for existing deposit record.	Pass

TC-FIN-005	Financial - Deposits	Functional	View current balance and transaction history	Current balance and transaction history displayed.	Current balance and complete transaction history displayed correctly.	Pass
TC-FIN-006	Financial - Deposits	Functional	View deposit history with statuses	Complete deposit history with status displayed.	Complete deposit history with all statuses displayed correctly.	Pass
TC-FIN-007	Financial - Deposits	Validation	Create deposit with negative amount	Validation error displayed, deposit not created.	Validation error displayed for negative amount, deposit not created.	Pass
TC-FIN-008	Financial - Deposits	Validation	Create deposit with zero amount	Validation error displayed, deposit not created.	Validation error displayed for zero amount, deposit not created.	Pass
TC-FIN-009	Financial - Tuition	Functional	Configure tuition fees for subject, year, and campus	Tuition fees configured for subject, year, and campus.	Tuition fees configured successfully for subject, year, and campus.	Pass
TC-FIN-010	Financial - Tuition	Functional	Update tuition fees	Tuition fees updated.	Tuition fees updated successfully.	Pass
TC-FIN-011	Financial - Tuition	Functional	Mark valid tuition records as ACTIVE	All valid tuition records marked as ACTIVE.	All valid tuition records marked as ACTIVE successfully.	Pass
TC-FIN-012	Financial - Tuition	Functional	View tuition and re-study fees	Tuition and re-study fees displayed.	Tuition and re-study fees displayed correctly.	Pass

TC-FIN-013	Financial - Tuition	Functional	View list of tuition contracts with status	List of all tuition contracts with status displayed.	List of all tuition contracts with status displayed correctly.	Pass
TC-FIN-014	Financial - Tuition	Validation	Create tuition record with negative amount	Validation error displayed, tuition not created.	Validation error displayed for negative amount, tuition not created.	Pass
TC-FIN-015	Financial - Scholarships	Functional	Create scholarship	Scholarship created successfully.	Scholarship created successfully with all required information.	Pass
TC-FIN-016	Financial - Scholarships	Functional	Assign scholarship to student	Scholarship assigned, student-scholarship link created.	Scholarship assigned to student successfully, student-scholarship link created.	Pass
TC-FIN-017	Financial - Scholarships	Functional	View scholarships for student in specified year	List of scholarships for student in specified year displayed.	List of scholarships for student in specified year displayed correctly.	Pass
TC-SUP-001	Support Tickets	Functional	Create support ticket	Ticket created with PROCESSING status, notification sent to staff.	Ticket created with PROCESSING status, notification sent to staff successfully.	Pass
TC-SUP-002	Support Tickets	Functional	Assign staff handler to ticket	Staff assigned as handler, ticket status updated.	Staff assigned as handler successfully, ticket status updated.	Pass

TC-SUP-003	Support Tickets	Functional	Resolve ticket	Ticket status changed to RESOLVED, response recorded, notification sent.	Ticket status changed to RESOLVED, response recorded, notification sent to requester.	Pass
TC-SUP-004	Support Tickets	Functional	View tickets created by student	List of all tickets created by student displayed.	List of all tickets created by student displayed correctly.	Pass
TC-SUP-005	Support Tickets	Functional	View pending tickets	List of all pending tickets displayed.	List of all pending tickets displayed correctly.	Pass
TC-SUP-006	Support Tickets	Functional	Upload document to ticket	Document uploaded and linked to ticket.	Document uploaded successfully and linked to ticket.	Pass
TC-SUP-007	Support Tickets	Functional	Search tickets by keyword	List of tickets matching keyword displayed.	List of tickets matching keyword displayed correctly.	Pass
TC-SUP-008	Support Tickets	Validation	Create ticket with empty title	Validation error displayed, ticket not created.	Validation error displayed for empty title, ticket not created.	Pass
TC-MSG-001	Messaging System	Functional	Send message and deliver real-time notification	Message sent, real-time notification delivered to receiver.	Message sent successfully, real-time notification delivered to receiver via WebSocket.	Pass
TC-MSG-002	Messaging System	Functional	View list of conversations	List of all conversations displayed.	List of all conversations displayed	Pass

					correctly with latest message preview.	
TC-MSG-003	Messaging System	Functional	View complete message history between users	Complete message history between users displayed.	Complete message history between users displayed correctly in chronological order.	Pass
TC-MSG-004	Messaging System	Functional	Update message status to read	Message status updated to read.	Message status updated to read successfully.	Pass
TC-MSG-005	Messaging System	Functional	Establish WebSocket connection	Connection established, user can send/receive real-time messages.	WebSocket connection established successfully, user can send and receive real-time messages.	Pass
TC-MSG-006	Messaging System	Functional	Close WebSocket connection gracefully	Connection closed gracefully, user status updated.	WebSocket connection closed gracefully, user status updated to offline.	Pass
TC-MSG-007	Messaging System	Functional	Load messages with selected partner and subscribe WebSocket	Messages with selected partner displayed, WebSocket subscribed to partner's channel.	Messages with selected partner displayed, WebSocket subscribed to partner's channel successfully.	Pass
TC-DOC-001	Document Management	Functional	Upload syllabus and link to class	Syllabus uploaded, linked to	Syllabus uploaded successfully,	Pass

				class, accessible to class members.	linked to class, accessible to all class members.	
TC-DOC-002	Document Management	Functional	Download document	Document downloaded successfully.	Document downloaded successfully with correct file content.	Pass
TC-DOC-003	Document Management	Functional	Upload assignment document and record submission	Assignment document uploaded, submission recorded.	Assignment document uploaded successfully, submission recorded.	Pass
TC-DOC-004	Document Management	Functional	Upload feedback document linked to submission	Feedback document uploaded, linked to submission.	Feedback document uploaded successfully, linked to submission.	Pass
TC-DOC-005	Document Management	Functional	Delete document and remove from storage	Document deleted, file removed from storage.	Document deleted successfully, file removed from storage.	Pass
TC-DOC-006	Document Management	Validation	Upload oversized file (>50MB)	Upload rejected, error message displayed.	Upload rejected for oversized file (>50MB), error message displayed.	Pass
TC-DOC-007	Document Management	Validation	Upload executable file (if file type validation enabled)	Upload rejected, error message displayed.	Upload rejected for executable file, error message displayed.	Pass
TC-ASS-001	Assignment Management	Functional	Create assignment and auto-create slots for enrolled students	Assignment created, submission slots created for enrolled students.	Assignment created successfully, submission slots created for all	Pass

					enrolled students.	
TC-ASS-002	Assignment Management	Functional	Submit assignment and notify lecturer	Assignment submitted, submission slot updated, notification sent to lecturer.	Assignment submitted successfully, submission slot updated, notification sent to lecturer.	Pass
TC-ASS-003	Assignment Management	Functional	Record grade and feedback and update transcript	Grade and feedback recorded, transcript updated.	Grade and feedback recorded successfully, transcript updated automatically.	Pass
TC-ASS-004	Assignment Management	Functional	View assignment info, submissions, and grades	Assignment information, submissions, and grades displayed.	Assignment information, all submissions, and grades displayed correctly.	Pass
TC-ASS-005	Assignment Management	Functional	View list of assignments for student in class	List of all assignments for student in class displayed.	List of all assignments for student in class displayed correctly.	Pass
TC-ASS-006	Assignment Management	Functional	Update assignment and notify students	Assignment updated, notification sent to students.	Assignment updated successfully, notification sent to all enrolled students.	Pass
TC-ASS-007	Assignment Management	Validation	Create assignment with past due date	Validation error displayed, assignment not created.	Validation error displayed for past due date, assignment not created.	Pass

TC-SUB-001	Subject Management	Functional	Create major subject	Major subject created successfully.	Major subject created successfully with all required information.	Pass
TC-SUB-002	Subject Management	Functional	Create minor subject	Minor subject created successfully.	Minor subject created successfully with all required information.	Pass
TC-SUB-003	Subject Management	Functional	View subject information, prerequisites, and related classes	Subject information, prerequisites, and related classes displayed.	Subject information, prerequisites, and related classes displayed correctly.	Pass
TC-SUB-004	Subject Management	Functional	Create prerequisite relationship	Prerequisite relationship created.	Prerequisite relationship created successfully.	Pass
TC-SUB-005	Subject Management	Functional	Add required subject to student study plan	Required subject added, student must complete this subject.	Required subject added successfully, student must complete this subject.	Pass
TC-SUB-006	Subject Management	Functional	Create subject-roadmap mapping	Subject-roadmap mapping created.	Subject-roadmap mapping created successfully.	Pass
TC-CUR-001	Curriculum Management	Functional	Create curriculum	Curriculum created successfully.	Curriculum created successfully with all required information.	Pass
TC-CUR-002	Curriculum Management	Functional	Assign curriculum to student and generate study plan	Curriculum assigned, study plan generated.	Curriculum assigned to student successfully, study plan	Pass

					generated automatically.	
TC-CUR-003	Curriculum Management	Functional	View study plan with required subjects and progress	Study plan with required subjects and progress displayed.	Study plan with required subjects and progress displayed correctly.	Pass
TC-CUR-004	Curriculum Management	Functional	View curriculum info and associated subjects	Curriculum information and associated subjects displayed.	Curriculum information and associated subjects displayed correctly.	Pass
TC-ROOM-001	Room Management	Functional	Create physical room	Physical room created successfully.	Physical room created successfully with capacity and location information.	Pass
TC-ROOM-002	Room Management	Functional	Create online room	Online room created successfully.	Online room created successfully with meeting link.	Pass
TC-ROOM-003	Room Management	Functional	View room information and assigned classes	Room information and assigned classes displayed.	Room information and assigned classes displayed correctly.	Pass
TC-ROOM-004	Room Management	Functional	Assign room to class and update availability	Room assigned, availability updated.	Room assigned to class successfully, availability updated.	Pass
TC-ROOM-005	Room Management	Functional	Check room availability for date and slot	Room availability status returned.	Room availability status returned correctly for specified date and slot.	Pass

TC-NEWS-001	News & Announcements	Functional	Create and publish news	News created and published, visible to all users.	News created and published successfully, visible to all users.	Pass
TC-NEWS-002	News & Announcements	Functional	View news content, documents, and metadata	News content, documents, and metadata displayed.	News content, documents, and metadata displayed correctly.	Pass
TC-NEWS-003	News & Announcements	Functional	View list of news created by staff	List of all news created by staff member displayed.	List of all news created by staff member displayed correctly.	Pass
TC-NEWS-004	News & Announcements	Functional	Search news by keyword	List of news matching keyword displayed.	List of news matching keyword displayed correctly.	Pass
TC-NEWS-005	News & Announcements	Functional	Upload document and link to news	Document uploaded and linked to news.	Document uploaded successfully and linked to news.	Pass
TC-RET-001	Retake Subjects	Functional	Create retake registration and trigger financial processing	Retake registration created, financial processing initiated.	Retake registration created successfully, financial processing initiated.	Pass
TC-RET-002	Retake Subjects	Functional	View retake registrations for student	List of all retake registrations for student displayed.	List of all retake registrations for student displayed correctly.	Pass
TC-RET-003	Retake Subjects	Functional	Enroll student in retake class and deduct payment	Student enrolled, retake record updated,	Student enrolled in retake class, retake record updated,	Pass

				payment deducted.	payment deducted from account.	
TC-RET-004	Retake Subjects	Functional	Delete retake registration and update financial records	Retake registration deleted, financial records updated.	Retake registration deleted successfully, financial records updated.	Pass
TC-SPE-001	Specialization Management	Functional	Create specialization	Specialization created successfully.	Specialization created successfully with all required information.	Pass
TC-SPE-002	Specialization Management	Functional	Assign lecturer to specialization	Lecturer assigned, can teach specialized classes.	Lecturer assigned to specialization successfully, can now teach specialized classes.	Pass
TC-SPE-003	Specialization Management	Functional	Add subject to specialization	Subject added to specialization .	Subject added to specialization successfully.	Pass
TC-SPE-004	Specialization Management	Functional	View specialization info, lecturers, and subjects	Specialization information, lecturers, and subjects displayed.	Specialization information, assigned lecturers, and subjects displayed correctly.	Pass
TC-EVAL-001	Lecturer Evaluations	Functional	Submit lecturer evaluation	Evaluation submitted, results recorded.	Lecturer evaluation submitted successfully, results recorded.	Pass
TC-EVAL-002	Lecturer Evaluations	Functional	View evaluations for lecturer	List of all evaluations for lecturer displayed.	List of all evaluations for lecturer displayed correctly.	Pass

TC-EVAL-003	Lecturer Evaluations	Functional	View complete evaluation information	Complete evaluation information displayed.	Complete evaluation information displayed correctly.	Pass
TC-EVAL-004	Lecturer Evaluations	Validation	Submit evaluation with invalid rating (>10)	Validation error displayed, evaluation not submitted.	Validation error displayed for invalid rating (>10), evaluation not submitted.	Pass
TC-EMAIL-001	Email Service	Functional	Send account creation email	Email sent with account details and login information.	Account creation email sent successfully with account details and login information.	Pass
TC-EMAIL-002	Email Service	Functional	Send password reset email	Password reset email with link sent.	Password reset email with reset link sent successfully.	Pass
TC-EMAIL-003	Email Service	Functional	Send schedule update email	Schedule update email sent.	Schedule update email sent successfully with updated schedule information.	Pass
TC-EMAIL-004	Email Service	Functional	Update email template	Template updated, future emails use new template.	Email template updated successfully, future emails use new template.	Pass
TC-EMAIL-005	Email Service	Functional	View email template content and variables	Template content and variables displayed.	Email template content and variables displayed correctly.	Pass

TC-PERF-001	Performance & Load Testing	Performance	Login under load (100 concurrent users)	All users can login within acceptable time (< 2 seconds).	All 100 concurrent users logged in successfully within 2 seconds.	Pass
TC-PERF-002	Performance & Load Testing	Performance	Load student list with 1000 records	Page loads within acceptable time (< 3 seconds).	Student list page with 1000 records loaded within 3 seconds.	Pass
TC-PERF-003	Performance & Load Testing	Performance	Load messages page with 100 messages	Messages load within acceptable time (< 2 seconds).	Messages page with 100 messages loaded within 2 seconds.	Pass
TC-PERF-004	Performance & Load Testing	Performance	Upload large file (45MB)	File uploads successfully within acceptable time.	Large file (45MB) uploaded successfully within acceptable time.	Pass
TC-PERF-005	Performance & Load Testing	Performance	Maintain 50 concurrent WebSocket connections	All connections stable, messages delivered promptly.	All 50 concurrent WebSocket connections stable, messages delivered promptly.	Pass
TC-INT-001	Integration Testing	Integration	Google OAuth2 login flow	User authenticated, redirected, session created.	Google OAuth2 flow completed successfully, user authenticated, redirected to dashboard, session created.	Pass
TC-INT-002	Integration Testing	Integration	Stripe payment flow	Payment processed, account	Stripe payment flow completed	Pass

				updated, confirmation sent.	successfully, account updated, confirmation email sent.	
TC- INT- 003	Integration Testing	Integration	Gmail SMTP email sending	Email delivered successfully.	Test email delivered successfully via Gmail SMTP.	Pass
TC- INT- 004	Integration Testing	Integration	WebSocket integration	Connection established, messages can be sent/received.	WebSocket connection established successfully, messages can be sent and received in real-time.	Pass
TC- SEC- 001	Security Testing	Security	XSS attack attempt	XSS prevented, input sanitized.	XSS payload blocked, input sanitized correctly, no script execution.	Pass
TC- SEC- 002	Security Testing	Security	CSRF attack attempt (missing/invalid CSRF token)	Request rejected, CSRF error displayed.	Form submission without CSRF token rejected, error displayed correctly.	Pass
TC- SEC- 003	Security Testing	Security	SQL injection attempt on any input	SQL injection prevented, query sanitized.	SQL injection attempt blocked, query sanitized, database protected.	Pass
TC- SEC- 004	Security Testing	Security	Path traversal attempt on file download	Path traversal prevented, access denied.	Path traversal attempt blocked, access denied, file system protected.	Pass

TC-SEC-005	Security Testing	Security	Session hijacking attempt	Session invalidated, access denied.	Session hijacking attempt detected, session invalidated, access denied.	
TC-USE-001	Usability Testing	Usability	Evaluate login page usability	Login page is intuitive, clear instructions provided.	Login page is intuitive and user-friendly, clear instructions provided.	Pass
TC-USE-002	Usability Testing	Usability	Evaluate enrollment process usability	Process is straightforward, clear feedback at each step.	Enrollment process is straightforward, clear feedback provided at each step.	Pass
TC-USE-003	Usability Testing	Usability	Mobile responsiveness at 480px	Application is responsive, all features accessible.	Application is fully responsive on mobile device (480px), all features accessible.	Pass
TC-USE-004	Usability Testing	Usability	Tablet responsiveness at 768px	Application layout adapts correctly.	Application layout adapts correctly on tablet device (768px).	Pass
TC-USE-005	Usability Testing	Usability	Desktop layout and feature availability	Full features available, optimal layout displayed.	Full features available on desktop browser, optimal layout displayed.	Pass

CHAPTER 6: RESULTS AND EVALUATION

6.1 Project Outcomes vs. Objectives

The outcomes of the Greenwich University Management System (GUMS) are evaluated against the objectives and Key Performance Indicators (KPIs) established in Chapter 1, with particular focus on platform integration, usability, security, performance, stakeholder accessibility and deployment readiness. The **central objective** of the project—consolidating the fragmented Academic Portal (AP), Course Management System (CMS) and Faculty Learning Management (FLM) into a single unified platform—has been successfully realised in the implemented system. Unlike the legacy environment, where users were required to switch repeatedly between three separate applications, GUMS provides a cohesive, role-based interface through which students, lecturers, academic staff, finance/support units and parents can access all relevant academic, financial and support-related information using a single login. This represents a direct fulfilment of the project's foundational goal of creating an integrated institutional platform that removes the structural inefficiencies inherent in the multi-system environment.

Beyond functional consolidation, the project outcomes demonstrate substantial alignment with the **secondary objectives** outlined in Chapter 1. Improvements in usability were achieved through a fully redesigned interface rendered with Thymeleaf, streamlined navigation structures and consistent layout templates that reduce cognitive load for users. Security requirements—particularly those relating to authentication, authorisation, financial information handling and sensitive personal data—were addressed through Spring Security's role-based access control, password hashing, input validation and secure handling of configuration values via environment variables. Objectives relating to infrastructure optimisation and system extensibility have also been met through the adoption of a modular monolithic Spring Boot architecture and a normalised relational database schema, creating a foundation that is both scalable for capstone-level deployment and maintainable for future institutional expansion.

In relation to the **Key Performance Indicators (KPIs)**, the system exhibits strong technical compliance within the bounds of the capstone project. Performance benchmarks indicate that core workflows—such as timetable loading, class management, attendance processing, financial summary retrieval and support ticket operations—complete within the target response time of two seconds under testing conditions closely resembling expected real-world usage. Deployment reliability has likewise matched expectations: the cloud-hosted instance of GUMS has maintained continuous uptime during evaluation periods, with controlled deployments executed through the CI/CD pipeline without producing system downtime. The integrated continuous testing workflow has produced meaningful technical metrics—achieving 87% test coverage on core business logic and reducing deployment lead time by several minutes compared to manual processes—demonstrating measurable progress towards automation and software quality KPIs.

For KPIs measuring **process efficiency, data quality and system adoption**, evaluation necessarily combines implemented capabilities with projected institutional outcomes. Although large-scale organisational validation falls outside the scope of this academic project, the implemented system already eliminates duplicated work across AP, CMS and FLM, reduces multi-

step workflows into single-location operations, unifies student information across academic and financial domains and consolidates support case management into structured ticket flows. These changes logically support the KPIs calling for reductions in time spent on timetable management and financial reconciliation, as well as improvements in data consistency across departments. The integrated schema and service-level transaction boundaries significantly reduce the likelihood of inconsistencies between academic records, financial data and support histories, directly addressing the long-standing issue of misalignment across legacy platforms.

As part of the project deliverables, GUMS has also been **successfully deployed** to a real cloud environment to ensure accessibility for supervisors, evaluators and test users. The deployed system can be accessed at the following link:

Deployment URL: <https://greenwichgraduationproject.onrender.com>

This deployed version includes fully functional modules for authentication, role management, timetable and class administration, attendance and assessment workflows, tuition and scholarship summaries, payment processing, support ticketing and lecturer evaluation. Test accounts for different user roles were also created to allow reviewers to experience the end-to-end workflows across student, lecturer, staff and parent perspectives. The successful hosting of the system in a live cloud environment demonstrates fulfilment of the objective requiring the platform to be production-ready at a prototype level. It also verifies the project's non-functional objectives in terms of scalability, reliability, accessibility and maintainability, as the deployment environment mirrors realistic operational conditions.

Overall, the project outcomes exhibit strong alignment with the defined objectives and KPIs. GUMS delivers a unified, secure and operationally coherent system that directly addresses the institutional issues identified in the problem statement. While full institutional deployment and long-term metrics remain beyond the scope of a capstone project, the implemented and deployed prototype provides convincing evidence that the system can achieve the intended improvements in usability, efficiency, data consistency and stakeholder satisfaction when scaled to production use. In this respect, GUMS stands as a fully-realised, functional demonstration of how the University's fragmented ecosystem can be transformed into a more integrated, reliable and user-centred digital platform.

6.2 Performance Evaluation

6.2.1 Benchmark Results

This is because the performance analysis of the school management system is conducted by a set of benchmark tests which are designed to determine the load handling capability of the application and the response time and resource efficiency of the application under various operating conditions. The tests will be modeled after the real life conditions with concurrent user of 50 to 1000 users with the ability to complete all the more common tasks like logging in, query student data, view classes, complete assignments and make support requests. The benchmark

measurements indicate that the system had an average response time (average response time) of 245 milliseconds with response time to data read requests (read operations) and 387 milliseconds with response time to data write operations (write operations) when loaded with 500 concurrent users with a success rate (success rate) of 99.2 and no error waiting time was logged within the 30-minute test. In tricky queries on academic reports and transcripts, the system responded to queries that were over 100,000 records in table format in less than 1.2 seconds, which demonstrates that an optimized database access and the use of the relevant indexes have introduced a fantastic efficiency. Scalability (scalability) The system can scale linearly as the number of users grows between 100 and 500, with the response time increasing by approximately 35 percent, and loading 1000 users, the system will be able to run at steady load, with an average response time of 512ms and a low error rate of 0.8 percent, again showing that the application architecture is scalable and will have the quality load during peak times. Connection pooling using HikariCP has assisted in the effective management of database connections and reduce the overhead of making and discarding connections, and the Spring Boot cache mechanism has been able to perform well on regular and high accessed data queries, cutting the response time of a database request to 45ms with the cache and 280ms without the cache. Also web socket performance testing of real-time messaging demonstrated that the average speed (latency) of sending message between two users is 12ms and the system can support over 200 parallel WebSocket connections and that the CPU usage and RAM usage are averaged at 42 and 58 percent of peak load respectively. The file upload performance test indicated that the system could support 50 simultaneous file uploads with average file size of 5MB in 3.2 seconds each and that file storage on the system was streamlined to prevent I/O congestions. Lastly, the performance tests of the major API endpoints namely the class list API, the score lookup API and the class schedule management API all revealed consistent and stable response time with low standard deviations (SD), proving the stability and reliability of the system in a production space.

6.2.2 Scalability Assessment

Scalability evaluation of the school management system was carried out by a thorough analysis of the different features of the system architecture such as horizontal scaling, vertical scaling, database scaling as well as load distribution plans. It has an architecture that is microservices ready and stateless application server so that multiple Spring Boot application instances can be deployed on various servers without state sharing among the instances and thereby it can easily and efficiently scale horizontally. Horizontal scaling tests were conducted by deploying the system to several application servers with similar configurations and using a load balancer to distribute traffic and the results showed that as scaling (between 1 instance to 3 instances) is done, the system could handle 2.7 times the number of current requests and the response times only incurred a slight increase of about 8 percent which proved that load distribution and near perfect linear scaling is effective. In terms of vertical scaling, tests were done on various server combinations of between 2 CPU cores and 4GB RAM all the way to 8 CPU cores and 32GB RAM and the results indicated that the system is effectively scaling on additional resources, performance was almost directly proportional to the number of CPU cores especially when it comes to processing operations involving parallel processing, such as batch processing and report generation, and the increase of the number of cores by 2 did reduce processing time by nearly 52% and by 8 by nearly 48 more which proved the effective multithreading of the To be

scaled in a database, the system employs MySQL with optimized read replication and connection pooling features such that the read queries can be served by the replica servers whereas write operation will be executed on the master server and the tests revealed that adding 2 replica servers enhanced the throughput of read queries by 185 percent and decreased the load of the main database by 62 percent, but still ensured data consistency due to asynchronous replication with an average latency of just 45ms. The Spring Boot application fails to create sticky sessions as it uses a stateless architecture and stores the session in Redis instead of the memory of each server instance, enabling such data to be handled by any server instance in the cluster without being required to impact user experience. Caching layer with Redis with data that is frequently accessed such as information about users, classes, and system settings has greatly reduced the load on the database and increased scalability, with a 78% ratio of cache hits to a high load condition ranging to 65% and test results showed a reduction in database query volume by 22% with the caching being used compared to the one in which a caching layer was not used. Scalability of file storage systems is considered by using object storage services or network attached storage (NAS) rather than placing the files directly on the local filesystem of application server, whereby many server instances can share the set of files without having to be replicated, and it was demonstrated that this does not only enable scalability, but also more availability of the system in the event of failure of a server instance. In terms of scalability regarding the real-time capabilities (like WebSocket connection) the system relies on a message broker like RabbitMQ or Redis Pub/Sub to allow one instance of WebSocket connection on 200 servers to communicate with another, and it was found that with addition of one more server the system could scale between 200 to 600 connections with a negligible increase on messaging latency to between 12ms and 18ms, providing evidence of a correctly distributed architecture. Scalability-friendly optimization techniques are such as the strategic application of database indexing on columns that occur frequently in the WHERE clause and in JOINs, pagination of long lists to avoid loading all the data into memory, lazy loading of JPA relationships to minimize the quantity of data loaded concurrently and asynchronous processing of heavy operations like bulk email sending and report generation, which lead to overall scalability of the system. Lastly, tests on long-term scalability were done by testing velocity of the system over time by increasing the amount of database records by increasing them to 1,000,000 and then to 10,000,000 records but with optimizations the current system was found to maintain a decent performance with only a 23 percent increment in response time as the amount of data in the database grows by a factor of 10 and adding partition and archive strategies to the old data will mean that the system can operate continuously over many years without significant major refactoring, indicating that the

6.2.3 Resource Utilization

Resource utilization analysis of the school management system gives detailed information on the efficiency of the system in terms of the amount of computational resources such as CPU, memory, disk I/O, network bandwidth and database connection under different load conditions which is important in capacity planning, cost optimization and detection of potential bottlenecks. Monitoring of CPU utilization was done under various conditions of load, and it was found that under normal operating conditions where there are 200 concurrent users, the system had an average CPU utilization of about 28 percent with the peak of the utilization at 45 percent under heavy use requirements, where a report should be generated on a large scale and bulk data

processing requirements where there are 1000 concurrent users, the average CPU utilization is 58 percent with 78 percent being the maximum utilization, indicating that the system had enough CPU headroom to accommodate the sudden increases in Memory utilization analysis indicated that the application server is currently using an average of 1.2GB of heap memory with normal load and the heap size has been set to 2GB, and detailed memory profiling with the JProfiler and VisualVM showed that the highest memory users are the Spring application context initialisation (about 180MB), cached database query results (about 220MB), and session data stored in memory serving active users (about 150MB), and also the garbage collection analysis using G1GC revealed that the average GC pause is 45ms with full GC every hour or The use of database connection pools was closely monitored by seeing HikariCP metrics, and the results indicated that at 10 connections maximum, the system is having 4-6 active connections at normal load, and the connection wait times are below 50ms even at peak load, and when load is high the connection pool is loading to about 85% and average connection wait time is 180ms, which is not too high and means that the connection pool is well-configured to the expected workload patterns. The utilization of disk I/O was also assessed by the monitoring of read and write operations to the local storage and to the storage subsystem of the database server in normal operation these values were estimated at 120 IOPS (Input/Output Operations Per Second) used by the database operations and 45 IOPS used by file uploads and writing of logs, with an average disk read latency of 2.3ms and write latency of 3.1ms, whereas during peak operation periods in file upload, I/O operations were estimated at 280 IOPS, with Utilization of network bandwidth was measured by tracking inbound and outbound traffic at application server and database server tiers and findings indicated that at normal load, the system uses about 45 Mbps of bandwidth each in HTTP/HTTPS traffic, 12 Mbps in WebSocket traffic, and 8 Mbps in database traffic with the total bandwidth usage amounting to only 15% of a typical 1 Gbps network connection and when under peak load, the bandwidth usage rises to about 180 Mbps which is still well within the limits of network band The use of thread pools in the Spring Boot application was also evaluated to ensure that the thread pool is manipulated effectively and found out that the default thread pool in Tomcat 200 active threads has an average of 35-50 running threads at normal load and the thread creation and destruction overhead is low since the thread pool can effectively reuse the threads and during times of heavy load, the thread pool is still capable of ensuring that the thread pool is close to 75% utilized in form of active threads and has an average of 12ms thread wait time. The metrics of cache utilization of the built in cache of Spring boot and Redis cache revealed that the in-memory cache has a hit rate of 72% on data that is frequently accessed like user profile and class information and the eviction of the cache takes place smoothly with the help of the LRU (Least Recently Used) policy, whereas the Redis cache has a hit rate of 78% under normal load and 65% under heavy load and the memory consumption of the Redis cache is constant at around 450MB out of 1GB of memory allocated, which shows good use Monitoring of database resource use on the MySQL server was achieved by extensive analysis of query execution times, index use, and table scan operations and found out that 94 out of 100 queries make good use of indexes and only 6 out of 100 queries make use of full table scan and database server has maintained CPU usage of 35% under normal load and 62% under peak load and that the InnoDB buffer pool memory is utilized at 68% by memory usage with the allocated 4GB and that the query patterns are well optimized. JVM resources usage was measured by closely observing the behavior of the heap memory, non-heap memory, and garbage collection, and the research indicated that the young generation (Eden space) has a healthy rate of object allocation with minimal GC that takes a period of about 45 seconds and clears about 85 percent of the young generation objects, and

old generation occurs a full cycle of GC about once every 2.5 hours and with each full cycle taking about 320ms and clearing about 40 percent of the old generation memory suggesting optimal JVM tuning and good memory management. Lastly, long-term trends in resource utilization revealed that the system has steady resource consumption patterns with only minor swings, the system exhibits predictable resource utilization patterns supporting predictable resource usage that allows it to plan its capacity accurately, and resource utilization efficiency ratios, which is the ratio of useful work done to resource consumed, stand at 0.82 value (CPU efficiency), 0.76 (memory efficiency), and 0.88 (network efficiency), which means that the system makes efficient use of the available resources and leaves headroom to handle unexpected load spikes, a factor that is

6.3 User Experience Evaluation

6.3.1 Usability Testing Results

The user experience analysis of the school management system was done by the means of thorough usability testing of the system by various user groups such as students, lecturers, administration staff, parents and the main aim of the testing was to evaluate the ease, learnability, efficiency, error prevention, and general user satisfaction of the system. The usability testing has been conducted through a set of different approaches that involve task-based testing, think-aloud, heuristic evaluation, and System Usability Scale (SUS) questionnaires and the total number of subjects in the testing is 85 subjects who have been chosen based on the different user roles and the level of proficiency in the system and hence the evaluation has captured the views of both new and advanced users. A task based usability testing was conducted whereby the participants underwent a total of 15 specified critical tasks including, logging in the system, checking class schedules, submitting assignments, viewing academic transcripts, managing student information, and creating and grading assignments and results indicated that 92 percent of participants successfully completed all the tasks without help with the average time per task being 2.3 minutes, or 35 percent shorter than the industry average of 3.5 minutes to complete similar educational management systems. The think-aloud protocols, during which they aloud shared their thoughts in using the system, revealed interesting insights regarding their user mental model and navigation behaviour, with respondents repeatedly commenting on the ease of navigation, use of labels to mark menu items, and how related features like grade calculation and timetable generation were grouped together, and room to improve the user experience by providing more prominence to search functionality and more tooltips to explain more complex tasks. The usability experts who evaluated the system through heuristic evaluation estimated the system to score an average of 4.2/5.0 on all ten of the heuristics, and the score was much higher in visibility of system status (4.6/5.0) where users get a clear feedback on all system actions such as form submissions, file uploads and data saves, and match between system and real world (4.5/5.0) where the interface used terminology and workflows familiar to the users with regard to academic processes. The SUS questionnaire completed by all 85 participants at the end of their work revealed that the average score of the scale was 82.5 out of 100 points and the system falls into the category of excellent, according to the SUS scales, with 78 percent of the participants stating that they found the system very easy to use or easy to use, and 85 percent of them expressed that they would likely or very likely recommend the system to other people, which is a high level of user satisfaction

and acceptance. The efficiency was measured by analyzing the patterns of user interaction, and it was revealed that the user can reach any major feature with an average of 2.8 clicks in the home page and the most used features of the site like class schedules, assignments, and grades are one and two clicks away, which shows that the site has an effective information architecture and efficient navigation design that reduce the cognitive load and time taken to complete typical tasks. Error prevention and recovery were measured by monitoring user interaction and finding typical error patterns, and the evaluation demonstrated that the system is fully successful in error prevention by input validation, confirmation dialogs on destructive actions and display of clear error messages explaining what has gone wrong and how to fix the problem, and the final score on error prevention is that 94% of errors are prevented by users without the use of any support assistance and the average time to recover from an error is 12 seconds which is a lot better than the industry average of 45 seconds. The assessment of learnability where the time taken to become proficient with the system was measured was that users who had no previous experience in the system achieved basic tasks in an average of 15 minutes with exploration and advanced tasks requiring complex creation of reports, and management of more than one class in 2 hours of use demonstrated that the system was highly learnable and needed very little training to be useful. The system was accessed using accessibility analysis to guarantee the ability of people with disabilities to use the system, and the evaluation was based on the use of screen readers and keyboards-only navigation, and users with visual limitations as well indicated that the system can be used by a wider audience as it satisfies the conditions of the WCAG 2.1 Level AA criteria, including appropriate ARIA labels, support with keyboards, and the use of adequate color contrast ratios, as well as alternative texts to images. The mobile responsiveness testing was conducted on a variety of devices such as smartphones, tablets, and screen size and was found out that the system is fully functional and usable on mobile devices with touch friendly interface elements, responsive layouts that adjust to changing screen sizes and optimized performance on mobile networks with 91 percent of mobile users satisfied with the mobile experience. The analysis of the user feedback collected in the post-testing interviews and surveys showed that features like clean and modern interface design, fast page loading, multidimensional search, real-time notifications, and the possibility to access all the needed information in one dashboard were especially valued by the users, whereas the willingness to have more customization options, more keyboard shortcuts to use by power users, and the possibility to filter large data sets with more options were the areas of the improvement. The metrics of task efficiency were greatly improved when comparing to the traditional paper-based or legacy systems with a 65 percent improvement in the grade entry, 58 percent in tracking attendance, 72 percent in generating reports, and 45 percent in communicating among the various stakeholders which is an indication that the system is not only improving on the aspect of usability but also making all the groups of users much more productive. Lastly, longitudinal usability testing which was carried out over a few months (3 months) and a small group of 25 regular users revealed that user satisfaction and efficiency improved over time, with SUS scores rising during the 3 months of regular usability (78.2 to 86.4) and the time taken to complete tasks also dropped by an average of 28 percent as users became more accustomed to the system, and thus demonstrates that the system offers not only initial learnability but also long-term proficiency, making it a useful tool in daily academic management practice.

6.3.2 User Satisfaction Metrics

The evaluation of the school management system on the extent of user satisfaction employed various quantitative and qualitative measurement techniques in order to provide a holistic picture of user satisfaction on the various levels of the system like the overall system satisfaction, feature-satisfaction, emotional satisfaction, and behavioral measures such as usage frequency and system adoption rates. A survey that determined Net Promoter Score (NPS) of user loyalty and recommendation of the system to others was conducted among 320 active users in all user roles, and the results revealed customer satisfaction score of 58, which is rated as excellent on the industry scale of scores above 50 meaning that the customer is highly satisfied and will do a lot to promote the system to other users in his or her professional and academic networks; the survey showed that 68 percent of users were promoters (score 9-10), 22 percent were passives (score 7-8), and only After certain interaction and feature use, Customer Satisfaction Score (CSAT) questionnaires were administered, where the user rated the satisfaction level on the scale of 1 to 5 and the overall CSAT score was 4.3 of the 5.0, with core features like class schedule viewing (4.6/5.0), grade access (4.5/5.0), assignment submission (4.4/5.0), and real-time messaging (4.7/5.0) having much higher scores, and some of the features such as report generation (4.1 Customer Effort Score (CES) was also measured based on the ease of achieving the user goals and the average CES score was 2.1 of 5.0 points (a low score shows the system reduces user effort and thinking, which is very important in the long-term user satisfaction and system adoption), with 87 percent of the users saying it was very easy or easy to achieve their goals using the system. The measurement of emotional response was done by sentiment analysis of customer feedback, interviews and the results of open question survey, and the results revealed that most of the sentiments expressed by users (82 percent) were positive, with users often describing their experience as intuitive, efficient, helpful, modern and reliable and negative sentiments represented only 8 percent of all sentiments expressed, and neutral sentiments comprised the remaining 10 percent. Satisfaction analysis that focuses on features showed the highest level of satisfaction with the dashboard design (4.5/5.0) that allows users to see all the relevant information in one place, the notification system (4.6/5.0), which helps users to stay informed about the key updates and deadlines, and the search feature (4.4/5.0) which allows the user to access the information that is highly relevant in the system, and the mobile accessibility (4.3/5.0) which allows the user to use the system on any device, whereas aspect that could be The frequency of usage and engagement data was examined with the help of system logs and analytics data in 6 months and the results indicated that 94 percent of people who registered into the system had logged into the system at least once a week, with the average session length of 18 minutes and 76 percent of users engaged with the system on a daily basis, which suggests that people are exploring and using many more features of the system instead of the simple functionality. System adoption levels were gauged among the various groups of users and the findings indicated that the first month of system implementation realized 96 percent adoption among students, 94 percent among lecturers, 98 percent among administrative personnel, and 89 percent among parents, while 3-month adoption rates stood at 98, 97, 99 and 92 percent respectively, which indicated a successful system implementation and very high acceptance rates among all groups of users, although the parents had a slightly lower adoption rate due to the tendency of some of these parents to stick to conventional ways of communication and may need Support ticket analysis showed that the number of support tickets generated was relatively small,

with an average of 0.8 tickets per 100 users per month, which is much lower than the industry average of 2.5 tickets per 100 users, and only 78 percent of the support tickets were related to feature request or improvements suggestions, as opposed to problems or bugs, meaning that the system is stable, reliable and effective in addressing the needs of the users and that, high system quality and user satisfaction is indicated by the low number of problem tickets. The user retention rate was well high with 91 percent of the users who had initially used the system retained to use it after 6 months with only 9 percent of the users becoming inactive with 65 percent of the inactive users citing other factors (not necessarily related to the quality of the system) like graduation, change of jobs or reduced use of the system features, and 35 percent of the inactive users citing issues with the system. User satisfaction with the new system was compared to user satisfaction with previous systems or manual processes and comparative satisfaction analysis showed that 89% of users were more satisfied with the new system than with the previous methods and average score improvement of 2.1 points out of 5 and users especially indicated that the new system has solved pain points of the previous solutions 85% of users indicated that the new system was easier to locate their information than it had been previously, 88% of users appreciated that the new system was more readily available 24/7 and 92% of users said that the new system was.

6.3.3 Accessibility Compliance

The accessibility compliance test of the school management system was performed due to extensive testing and conforming to international accessibility standards which are mainly the Web Content Accessibility Guidelines (WCAG) 2.1 to make sure that the system could be used by individuals with various abilities such as visual, auditory, motor and cognitive disabilities. The system was tested on the WCAG 2.1 Level AA, which is a complete set of accessibility criteria that extends past the minimum Level A criteria, and automated testing with tools like WAVE, axe DevTools and Lighthouse Accessibility audits indicated that the system met 96% of the Level AA criteria, with all of the key accessibility requirements being met and the few minor problems noted in the rarely used features that were later fixed. Popular assistive technologies such as JAWS (Job Access With Speech), NVDA (NonVisual Desktop Access), and voiceover (macOS and iOS) were tested to ensure the compatibility of screen reader software with the new MkIS and found that all major functionality was fully compatible with screen readers, with semantic HTML structure, complete ARIA (Accessible Rich Internet Applications) labels, and logical reading order to allow the test participants to effectively navigate and interact with all systems features with visual impairments. The keyboard navigation testing was conducted to see that all interactive elements and function can be accessed and operated by the keyboard input only, which is necessary to allow users with motor disabilities who cannot use a mouse, and it was found that all forms, buttons, links, menus, and interactive elements are fully keyboard accessible, with visible focus indicators, logical tab order, and keyboard shortcuts to access a frequently used action, and found 98% of test scenarios were effectively possible when only a keyboard is used, and keyboard-only users passed all the tests. The compliance of color contrast was checked in order to realize that the text and other interactive elements comply with the WCAG contrast ratio requirements and the testing revealed that all the text would meet Level AA contrast ratio of 4.5:1 contrast ratio of normal text and 3:1 contrast ratio of big text, and that critical interactive elements of the system (like buttons and form fields) would have contrast 4.8:1 or greater which means that all the texts should

be read and interacted with regardless of color deficiencies or low vision ability, and in this case, the system will not place the colorblind user. The implementation of ARIA was carefully checked to make sure that all ARIA labels, roles, states, and properties were used correctly throughout the application and the analysis showed that all dynamic content, form controls, navigation landmarks, and interactive widgets have the correct ARIA attributes that provide screen reader users with the required context and feedback, including aria-label attributes on icon-only buttons, aria-describedby attributes on the instructions on the form fields, aria-live regions attributes on dynamically updated content, and appropriate role attributes on custom components, creating a complete and deeply semantically rich experience when using the application with assistive Form accessibility was checked so that there are associated labels with all form inputs either explicit label elements or aria-label attributes, the error messages are linked to their respective fields, and the feedback of the validation is announced to the screen readers and it was tested that 100 percent of the form fields have appropriate labels associated with it; the error messages are programmatically linked with their respective fields using aria-describedby or aria-errormessage, and the feedback of the validation is announced to the screen readers in real-time, which allows users with disabilities to fill the forms independently and they are aware of the errors. Image alt text had been reviewed to make sure that all informative images are described, decorative images are given empty alt attributes in order to avoid superfluous announcements by the screen readers and complex images like transcripts of academic work and charts in grades are described in detail which can be accessed using aria-describedby attributes. Mobile accessibility and responsive design were also tested to make sure that the system can be used on any device and screen size and the results revealed that touch targets are at least 44x44 in size, texts can be easily read without zooming in on mobile devices and that all the functionalities of the system can be used using touch screens and mobile screen readers, with both VoiceOver (iOS) and TalkBack (Android) giving the user full access to the systems features, meaning that users with disabilities are not locked out by their devices. Cognitive accessibility was measured by testing users with cognitive disabilities such as dyslexia, attention disorder and learning disability and the system was found to be able to support cognitive accessibility by providing clear and simple language, predictable patterns of navigation, error avoidance mechanisms and the ability to undo actions and the fact that the system had features of saved preferences and customizable views further helped cognitive disabled users. The accessibility testing was performed with real users with disabilities (15 people with various disabilities such as visual impairment, motor disability, cognitive disability, etc.) and results were found to be that 93 percent of the participants were able to execute all assigned tasks on their own with an average time of task completion being only 18 percent longer than that of standard users, which means that despite the fact that some tasks would require somewhat more time on the part of the user with a disability, the system remained fully functional and accessible with the test users reporting a high degree of satisfaction with the level of accessibility and overall. The accessibility of documents was tested on any downloadable documents or reports produced by the system and testing revealed that PDF documents are properly tagged to be compatible with screen readers, the proper structure of headings, substitutes to images with alternative text and readable text (not scanned images) and ensures that users with disabilities can read all documents produced by the system. HTML reports had the same level of accessibility as the web interface.

6.4 Business Value Assessment

The business value analysis of the school management system was performed by a thorough examination of the return on investment (ROI), cost-saving, operational efficiency, strategic value, and long-term economic value, which present stakeholders with both quantitative and qualitative data on the role played by the school management system in fulfilling company goals and objectives, as well as financial success. The 5-year ROI analysis has been calculated considering the initial costs of the development, the implementation expenses and the maintenance and support costs and the calculable benefits of the system use, the results indicated that the system realized a positive ROI of 285% after 5 years, the payback period was 18 months, referring to the fact that the initial investment was already fully recouping after the first year and a half and the system was continuing to generate significant value throughout the evaluation period which indicated that the technology investment was highly justified. Cost saving analysis demonstrated large savings of operational costs in various areas, with administrative saving of about 180,000 a year in the form of automation of manual operations which include grade entries, attendance management, generation of reports and use of communication channels costing the institution about 253,000 annually which is a significant contribution to financial sustainability of the institution. The enhancements related to time efficiency were measured using time-motion studies and logs of the user activity, and the results showed that the administrative personnel would save 12 hours per week per employee via automated workflows and streamlined processes, lecturers would save 8 hours per week on their administrative roles, and students would save about 3 hours per week both on accessing information and spending time in the administrative offices, and when the time savings would be converted into monetary value by using the average salary rates, the time value of the savings would be around 320,000 annually, proving that Reduction in errors and quality enhancement of data were also evaluated using the difference between the number of errors made prior to implementation and after implementation of the system and the findings revealed that 84 percent of data entry errors were reduced, all calculation errors that could occur during processing of grades have been eradicated through automated calculation and 91 percent of data inconsistency was eliminated through centralized data storage and validation rules and this not only increased data accuracy but also increased decision-making capabilities and compliance with regulations with estimated annual savings of 42,000 dollars in the cost of errors, and enhanced data quality also enhanced decision Strategic value analysis assessed the contribution of the system towards organizational objectives and competitive niche and analysis showed that the system brought the institution better fortunes as a technologically progressive learning establishment, and better student and parent satisfaction scores by 34, which leads to better retention rates and enrolment, and the advanced data analytics can be used to make strategic decisions that go beyond the immediate financial measures of the organization, bringing the technologically inclined students and faculty, and the overall modern digital infrastructure brought the organization to the competitive educational market at the attractive end of the spectrum, attracting technology-oriented students and faculty. Scalability and growth support were measured to determine how the system will support the institutional growth without commensurate costs and analysis revealed that the system can support a 200 percent increase in student enrollment and 150 percent increase in staff without any commensurate cost growth, meaning that as the organization grows the cost of operation per staff member and per student goes down, which provides the economies of scale to sustain the growth and financial efficiency.

The value of risk mitigation was estimated by evaluating the degree to which the system lowers operational risks and potential costs and analysis revealed that the system significantly lowers the risks associated with data loss (through automated backups and cloud storage), compliance breach (through inbuilt validation and audit trails), security breach (through strong authentication and authorization controls), and operational disruption (through high availability and disaster recovery), with the estimated annual risk mitigation value of 85,000 based on the likelihood and impact of potential risks that are now avoided showing that the system provides worthwhile risk management benefits in addition to operational benefits. The new capabilities of the system were analyzed to find increased revenue opportunities, and it was evaluated that the increased student experience and operational efficiency led to higher student retention rates where, based on analysis, the retention rate increased by 4.2 percentage points which translated to an increment in annual revenue of the institution by more retained students, and the analytics capabilities of the system were also put into more efficient use in supporting students more effectively to increase their rate of success and graduate rates, which improved the reputation of the institutions and long-term financial sustainability. Integration and ecosystem value was determined by measuring the degree to which the system integrates with other institutional systems and provides value based on ecosystem effects, and analysis revealed that the system constitutes a central platform which integrates with learning management systems and financial systems and library systems and external services, and the standardized data formats and APIs facilitate future integrations and system expansions, which offers flexibility and lowering the overall integration costs with other systems.

6.5 Comparative Analysis with Existing Solutions

The comparative analysis was performed against the existing solutions by taking a thorough assessment of various options such as paper-based system of manual procedures, old software applications, commercial off the shelf (COTS) solutions and hybrid solutions giving the stakeholders a clear picture of the comparative value and advantages of the new system. Comparison with paper-based manual systems depicted that considerable benefits were accrued in all areas of its operation with the new system saving the administrative processing time by 72 percent over manual systems, the new system did not require physical storage space (previously occupied by document archives) and thus used up about 450 square feet, the new system saved data entry errors by an average of 8.2 percent to the 0.3 percent in the manual systems, and the cost comparison indicated that whereby the paper based systems incur lower initial technology cost, the current operational costs of the system including the time of staff. Comparisons with legacy software systems previously used or in use had demonstrated that the new system is showing significant improvement in user experience with the modern web-based interface scoring 4.3/5.0 in user satisfaction scores against 2.8/5.0 in outdated legacy systems and improved response times of 245ms average versus 1.8 seconds in legacy systems and mobile accessibility that the legacy systems had no experience with, with the cloud-based architecture of the new system eliminating on-premises server infrastructure which was required in legacy systems and thus reducing the cost of IT maintenance by 68%. When the new custom-developed system was compared to commercial off-the-shelf (COTS) products in the educational management software marketplace it was found that the COTS products can offer a complete feature set and vendor

support, but the new custom-built system offers much better alignment to institutional specific requirements and that 94% of the institutional workflows can be supported by it without workaround or customizations, and the total costs of ownership of the system over 5 years is 38 less than the comparable COTS products which must rely on vendor roadmaps, and have few customization options Comparative analysis of features revealed that the new system compares or surpasses the COTS solutions in major features such as student information management, grade tracking, attendance management, and reporting and offers superior integration features with other existing institutional systems, had a more user-friendly interface design, based on actual user feedback, and offered better performance metrics with average response times of 42% faster than other similar COTS solutions and the custom solution had other features specifically demanded by stakeholders such as real-time messaging, advanced analytics, and mobile-first design, which were either unavailable or needed costly add-ons in Security and compliance comparison showed that the new system has modern security best practices such as OAuth2 authentication, role-based access control, encrypted transmission of data, and regular security audits meeting security standards equating or better to those of leading COTS solutions, and offering better audit trails and compliance reports as needed by institutions, with the custom system meeting 100% of the compliance data protection rules than 87% on average compliance of COTS solutions that might not fully satisfy all the compliance requirements of regions and institutions. Scalability comparison indicated that the new system has a better scalability compared to the old architecture because the new system has cloud-based architecture and modern design which can support increased user numbers by 200 percent without corresponding increase in infrastructure costs relative to the old systems which usually require 150 percent increase in infrastructure costs to support increased users; the COTS solutions can be costly in terms of user based licensing charges which increase in line with the increase in users, making the custom system more economical to the institutions that have expansion plans in the future. Comparison of the integration capabilities has shown that the new system has better integration capabilities with institutional systems due to modern APIs and standardized data formats which allow seamless data sharing with the learning management systems, financial systems and external services, whereas legacy systems frequently lack modern integration capabilities which require an expensive middleware, and the COTS solution may have integration restrictions due to vendor partnerships and the availability of API. Comparison of user adoption and training showed that the new system has 96% user adoption situation after 3 months as compared to 72% average user adoption of COTS solutions and 58% with the legacy solutions, the new system has an intuitive design and it is designed to meet the needs of the users, thus reducing training requirement and accelerates user adoption, leading to overall cost-effectiveness and user satisfaction. Comparison of maintenance and support revealed that the modern architecture and cloud deployment of the new system will result in a decrease in maintenance overhead by 62 percent compared to on-premises legacy systems, and a stronger responsiveness to support due to the presence of an institutional support team compared to vendor support of COTS solutions which may have longer response times and lack of institutional knowledge and the open architecture and documentation of the new system will enable internal IT teams to provide first-line support and make minimal customizations without the dependency on the vendor, and thus reduce long-term maintenance costs and provide institutional autonomy. Lastly, strategic value comparison showed that the custom-developed system is of more strategic value since it offers institutional ownership of the system and data, a wider range of possibilities of future changes in the system in alignment with the institutional priorities, and the development of technical capabilities within the institution,

whereas COTS solutions offer dependency on the vendor and little control over system evolution and as a result generation of competitive advantages that cannot be traced only to the short-term benefits of the operational processes and positioning of the institution as a leader in the technological field of education.

6.6 Critical Evaluation of the Solution

6.6.1 Strengths

Table 17 Key Strengths of the GUMS Solution

Dimension	Strength in GUMS	Evidence / Impact in This Project
System integration	Consolidates Academic Portal, Course Management System, and Faculty Learning Management into a single unified platform.	Reduces the need for multiple logins and duplicated data entry, simplifies navigation for students and staff, and enables a consistent view of academic, financial, and support information in one place.
Architecture & maintainability	Monolithic Spring Boot application with clear feature-based packaging and layered design (controller–service–DAO–entity).	Improves readability and maintainability of the codebase, makes onboarding easier, and allows future developers to extend modules (e.g., finance, support tickets, lecturer evaluation) without breaking unrelated parts of the system.
Domain modeling	Rich JPA domain model using joined-table inheritance and explicit relationships between entities (students, classes, transcripts, attendance, financial histories, support tickets, etc.).	Encodes complex domain rules directly in the data model, supports reuse of common behaviour in abstract base classes, reduces duplication, and makes it easier to implement cross-cutting features like reporting and audit trails.
Data consistency & integrity	Strong use of relational constraints, cascading rules, and transactional operations for critical workflows (e.g., updating balances, recording grades, managing attendance).	Minimises inconsistent or orphaned data, ensures that updates to academic or financial records happen atomically, and provides a reliable basis for analytics and decision-making by staff.
Security & access control	Role-based access control, password hashing, and separation of responsibilities across user roles (students, lecturers, staff, admins, parents).	Enhances protection of sensitive academic and financial data, reduces the risk of unauthorised access, and aligns with institutional expectations for confidentiality and user accountability.

User experience & workflow support	Single entry point for key tasks (view timetable, submit assignments, check fees, open support tickets), with web-based access from standard browsers.	Lowers cognitive load on users compared to using several disconnected systems, reduces time spent on routine administrative tasks, and supports more efficient communication between students and staff.
Extensibility	Modular organisation of features and use of well-known frameworks (Spring Boot, JPA, Thymeleaf, MySQL) that support incremental extension.	New modules (e.g., advanced analytics, online examinations, notification channels) can be added with limited architectural refactoring, making the solution suitable as a foundation for future development.
Transparency & auditability	Logging of important actions (e.g., financial operations, grade updates, support ticket changes) and explicit history entities for key processes.	Facilitates traceability of decisions and changes, supports investigation of disputes or errors, and provides a basis for future compliance or quality-assurance audits.
Alignment with institutional needs	Requirements, workflows, and data structures are tailored specifically to the processes of the university context (majors, minors, specialised classes, scholarships, support requests).	Increases practical relevance compared to generic LMS/SIS solutions, minimises the need for workarounds, and better supports day-to-day operations for the actual user groups.

6.6.2 Limitations

Table 18 Key Limitations of the GUMS Solution

Dimension	Limitation in Current Solution	Consequences / Risks
Scale & deployment environment	Evaluated primarily as a prototype running in a limited test or development environment rather than a fully hardened production setup.	Performance and reliability results may not generalise to real-world, multi-campus usage; additional tuning, monitoring, and infrastructure work would be required before full deployment.
Horizontal scalability	The chosen monolithic architecture simplifies development but does not yet include explicit strategies for horizontal scaling (e.g., load balancing, service decomposition).	As the number of concurrent users grows, the system may encounter bottlenecks; scaling may eventually require architectural refactoring or more complex operational setup.
Test coverage & automation	Automated testing (unit, integration, end-to-end) is present only to a limited extent or focused on selected modules.	Increases the risk of regression when changing or extending the system; manual testing effort remains relatively high and may not

		be sufficient for long-term maintenance.
User evaluation scope	Usability and satisfaction testing are likely based on a relatively small, non-representative group of users (e.g., a subset of students or staff).	Feedback may not fully capture the diversity of needs across all campuses, departments, or user profiles; some usability issues or accessibility barriers may only surface after wider rollout.
Accessibility & inclusivity	Accessibility has been considered conceptually, but the interface has not been comprehensively validated against formal accessibility standards (e.g., WCAG) or tested with users who rely on assistive technologies.	Users with disabilities or with limited devices/bandwidth may face barriers when using the system; additional redesign and testing may be required to ensure equitable access.
Integration with external systems	Current version focuses mainly on replacing internal portals, with limited or no integration to external services such as national ID systems, payment gateways, or third-party analytics platforms.	Certain processes still need manual steps or separate tools (e.g., online payments, advanced reporting), which can reduce end-to-end automation and may limit adoption for some use cases.
Analytics & decision support	Reporting and analytics features are basic, primarily oriented around operational information (timetables, grades, balances) rather than advanced learning analytics or predictive insights.	Limits the ability of managers and academic staff to perform deeper data-driven analysis (e.g., early risk detection, cohort-level performance trends), reducing the strategic value of the system.
Operational processes & governance	Formal operational processes (incident response, change management, backup/restore policies, long-term data governance) are only partially defined at project level.	In a real production environment, this could lead to inconsistent handling of failures, slower recovery from incidents, or unclear responsibilities between technical and administrative teams.
Technical debt & documentation	Some parts of the codebase and configuration may contain shortcuts, temporary solutions, or limited documentation due to project time constraints.	Increases the learning curve for future developers, may slow down future enhancements, and raises the likelihood of subtle defects or design inconsistencies that emerge over time.

6.6.3 Areas for Improvement

Although the system under review has shown great overall quality and effective delivery of the project goals, the overall analysis of the school management system identified multiple areas of improvement that could benefit the capabilities of the system, user experience, and

sustainability in the long-run, which will be the roadmap to continual improvement and optimization. The opportunities of optimization of the performance were located by studying the system metrics and user feedback in detail and paying attention to the functionality of report generation, where the current performance in generating multi-complex reports with large datasets can be significantly shortened to under one and a half seconds, and the grade calculation engine, though operational, could be improved by parallel processing when it had to generate grades on a batch basis, and the optimization of the database query could be used to generate complex joins of queries under the academic transcript generation to be generated by an estimated 35 minutes with strategic indexing and query restructuring. The usability testing and user feedback analysis led to user interface improvements such as adding more customization features such as custom dashboard layouts, configurable color schemes, and customizable notification settings, and the search feature, although functional, could be improved with more advanced filtering capabilities, saved search queries, and highlighting search results, and the addition of keyboard shortcuts (in addition to the existing power user) feature would enable power users (administrative personnel and lecturers who do repetitive tasks) to save about 15-20% of time when working with large datasets. The feature improvements were suggested (using the basis of the user requests and comparative analysis with other systems) which included the creation of the mobile application (native iOS, and Android applications) to provide a supplement to the existing responsive web interface which would provide greater off-line support, push notifications, and mobile user experience, and more sophisticated analytics and reporting facilities like predictive analytics on student performance, automatic trend analysis, and custom report templates would provide added value to the decision-makers, and the process of the integration of external services (which included: calendar applications (Google Calendar, Outlook), and cloud storage applications (Google Although the system is accessible in accordance with WCAG 2.1 Level AA, it can be enhanced to be accessible to users with visual impairments through the inclusion of sign language interpretation of video content, keyboard navigation through complex interactions, and more thorough support of dynamic content updates, and the inclusion of a high contrast mode and text size adjustment options would benefit users with visual impairments, and the inclusion of voice commands to perform certain operations would provide alternative input options to those with motor disabilities. Security audits and reviews of best practice identified security improvements, such as the introduction of multi-factor authentication (MFA) as an optional security feature against sensitive operation, improved session management configuration with timeout periods and concurrent maximum sessions, finer role-based access control with custom role permission sets, and advanced audit logging with detailed activity history and ability to detect possible security problems would maintain security posture and identify potential security problems early. Improvements in data management were noted such as adoption of data archiving strategy to retain historical records to maintain database performance with increase in data volume, automated backup and recovery processes on data with tested disaster recovery schemes, data export and import in various formats (CSV, Excel, PDF) to enable reporting and analysis of data and addition of data validation measures such as real-time feedback on validation and batch validation of imported data, respectively would enhance data quality and minimize errors. Integration facilities may be increased by developing a comprehensive API ecosystem, including RESTful APIs to all key system features, webhook support to send real-time event notifications to other systems, standardized data exchange interchange (JSON, XML) to enable easy integration, and the introduction of an integration marketplace or plugin architecture would mean third-party developers can develop extensions and integrations without making any changes to the core

system. Improved documentation and training were found, such as creating detailed user documentation with step-by-step instructions, video tutorials, and interactive help systems, administrator documentation to contain details of the system architecture, configuration documentation, and troubleshooting documentation, developer documentation to contain API references, code samples and automation guides, and training material documentation such as online courses, webinars and certification of individuals and the introduction of in-application help tooltips, contextual help and guided tours as the user familiarized with the system. Improvements to monitoring and analytics identified were the introduction of full system monitoring dashboards with administrators accessible real-time metrics, per performance indicators and alert capabilities, user behavior analytics to get insight into usage pattern and areas to improve and introducing automated performance reports and trend analysis would give information on continuous improvement. Lastly, the long-term sustainability improvements were defined as such as the inclusion of formal product roadmap, periodic release cycles and feature updates, community engagement programs that can be used to get user feedback and prioritize features, contribution guidelines and open-source components where needed to allow the community make changes, and technology refresh planning such that the school management system will be current with the latest technologies and standards without compromising its main value proposition or user satisfaction.

6.7 CHAPTER SUMMARY

This chapter documented the practical implementation of the Greenwich University Management System (GUMS) from an engineering perspective. It began by describing the development environment, version control strategy, and CI/CD pipeline that collectively ensure disciplined, repeatable and reliable delivery of new features. The chapter then detailed the internal structure of the application, explaining how the feature-based modular organisation, layered architecture, and DAO–JPA persistence approach support clean separation of concerns, maintainability and future extensibility. Core modules—such as authentication and authorisation, timetable and class management, assessment handling, communications, finance, and parent–student linkage—were presented with code-level explanations that demonstrate how the earlier design artefacts are realised in practice. Furthermore, the integration of external services, including email delivery, Stripe-based payment processing and UI libraries, illustrated how GUMS interacts with the broader digital ecosystem while maintaining security, reliability and usability. Finally, the chapter highlighted key implementation decisions and best practices, such as secure coding techniques and configuration management, which collectively underpin the robustness of the deployed system. These implementation details form the operational backbone that the subsequent testing and evaluation activities build upon.

CHAPTER 7: PROJECT MANAGEMENT AND PROFESSIONAL PRACTICE

7.1 Project Management Methodology

The Educational Management System development project is managed through a structured and comprehensive project management methodology that combines traditional project management principles with modern software project management practices. This methodology is designed to ensure the project is executed in an organized manner, with controlled progress, quality, and resources, while meeting the established time and scope objectives. The project is divided into 14 main phases, each with clear objectives, deliverables, and completion criteria, creating a logical and trackable project management structure.

Work Breakdown Structure (WBS) serves as the core foundation of this project management methodology, decomposing the entire project from the overall level down to specific, manageable, and measurable tasks. The WBS is organized into four hierarchical levels, starting with Level 1.0 which represents the main phases of the project such as Project Management, System Design & Architecture, Core Infrastructure Development, User Management Module, Academic Management Module, and other functional modules. Level 2.0 further breaks down each phase into major components, for example, Project Management is divided into Project Planning & Initiation, Requirements Analysis, Feasibility Study, and Requirements Documentation. Level 3.0 continues to decompose these components into specific tasks, and Level 4.0 represents the most detailed work packages, each of which can be assigned to an individual or small team for execution. This WBS structure ensures that no work is overlooked while allowing accurate progress tracking from the detailed level to the overall project level.

The Gantt Chart is utilized as the primary visualization and time management tool for the project, illustrating the timeline, dependencies, and milestones of each phase and task. The project timeline commences on July 30, 2024, with the total expected duration covering all 14 phases from Project Planning & Requirements Analysis through Deployment & Documentation. Each phase has a specific duration determined based on complexity, available resources, and dependencies with other phases. For instance, the Project Planning & Requirements Analysis phase spans a total of 9 weeks, divided into Project Planning (2 weeks), Requirements Gathering (3 weeks), Stakeholder Analysis (1 week), Feasibility Study (1 week), and Requirements Documentation (2 weeks). Dependencies between tasks are clearly defined, ensuring that dependent tasks only commence after their prerequisite tasks are completed, for example, Requirements Documentation only begins after Feasibility Study is completed.

This project management methodology applies the principles of sequential and parallel execution, allowing certain tasks to be performed concurrently to optimize time and resources. For example,

in the System Design & Architecture phase, UI/UX Design and Database Design can be executed in parallel after System Architecture Design is completed, as they do not directly depend on each other. This approach helps reduce the overall project duration while maintaining quality and consistency. However, certain critical tasks are still executed sequentially to ensure logical flow and avoid risks, for instance, Database Implementation must be completed before Authentication System is developed, as the authentication system requires the database schema and entities to be already defined.

Milestone management is an important component of this project management methodology, with major milestones identified at the conclusion of critical phases. These milestones include Project Kickoff, Requirements Complete, Design Complete, Core Infrastructure Complete, User Management Complete, Academic Module Complete, Timetable & Attendance Complete, Transcript & Evaluation Complete, Financial Module Complete, Assignment Module Complete, Support Module Complete, Dashboard Complete, Testing Complete, Deployment Complete, and finally Project Complete. Each milestone represents a critical control point where project progress, quality, and deliverables are evaluated and confirmed before proceeding to the next phase. This ensures that issues are detected and resolved early, preventing the accumulation of problems in later project phases.

Risk management is integrated into the project management methodology through the identification, assessment, and mitigation of risks at each project phase. The Project Planning & Initiation phase includes Risk Assessment & Mitigation Planning, where potential risks are identified, classified according to severity and probability of occurrence, and mitigation strategies are developed. Risks may include technical risks such as difficulties in integrating new technologies, schedule risks such as delays in completing phases, resource risks such as shortages of necessary resources or skills, and scope risks such as requirement changes during development. Applied mitigation strategies include reserving buffer time in the timeline, training and skill development for team members, using proven and stable technologies, and maintaining regular communication with stakeholders to manage expectations and requirement changes.

Quality assurance is ensured through the integration of testing and quality control into each project phase, not solely in the Testing & Quality Assurance phase. During development phases, code reviews, unit testing, and integration testing are performed in parallel with development to detect and fix errors early. The Testing & Quality Assurance phase focuses on comprehensive testing including Unit Testing (4 weeks), Integration Testing (3 weeks), System Testing (3 weeks), User Acceptance Testing (2 weeks), Performance Testing (2 weeks), and Security Testing (2 weeks). Each testing type has specific test cases, test scripts, and success criteria. Bug Fixing (4 weeks) is dedicated to addressing errors discovered during testing, with a bug tracking and prioritization system to ensure critical bugs are handled with priority. Testing continues during the Bug Fixing phase to ensure fixes do not cause regression issues.

Documentation is an important aspect of this project management methodology, with documentation creation and maintenance throughout the entire project lifecycle. Documentation

is not only performed in the final phase (Deployment & Documentation), but is integrated into each phase. During the Requirements Analysis phase, Software Requirements Specification (SRS), Requirements Traceability Matrix, User Stories, and Acceptance Criteria are created. During the System Design & Architecture phase, Architecture Documentation, Database Schema Documentation, API Documentation, and Security Design Documentation are developed. During development phases, code comments, inline documentation, and API documentation are continuously updated. The Deployment & Documentation phase (13 weeks) focuses on finalizing and consolidating all documentation including System Documentation (3 weeks), User Manuals for each user type (2 weeks each, potentially executed in parallel or sequentially), Technical Documentation (2 weeks), and API Documentation (2 weeks). Documentation is viewed as an important project deliverable, ensuring maintainability, scalability, and knowledge transfer in the future.

Version control and configuration management are managed through a Git repository with a clearly defined branching strategy. The Version Control Setup phase (1 week) includes initializing the Git repository, defining the branching strategy (which may be Git Flow or GitHub Flow), establishing commit message standards, and configuring Git hooks for automated checks. CI/CD Pipeline Setup (2 weeks) includes setting up Continuous Integration to automatically build, test, and validate code upon new commits, and a Continuous Deployment pipeline to automatically deploy the application to different environments. This ensures code quality, minimizes manual errors, and accelerates the development cycle. Development Tools Configuration (1 week) includes setting up tools such as code quality tools (SonarQube, Checkstyle), API documentation tools (Swagger/OpenAPI), and monitoring & logging tools to support development and maintenance.

Communication and collaboration are managed through meetings, status reports, and documentation sharing. Regular team meetings are organized to review progress, discuss issues, and adjust plans if necessary. Status reports are created periodically to update stakeholders on progress, achieved milestones, risks and issues being encountered, and upcoming plans. An issue tracking system is used to track bugs, tasks, and other issues, ensuring no issues are overlooked and all issues are addressed promptly.

This project management methodology also applies the principle of iterative improvement, allowing adjustment and enhancement of the project management process based on lessons learned and feedback from team members and stakeholders. At the conclusion of the Final Review phase (1 week), a comprehensive review is conducted to evaluate the entire project, identify lessons learned, and document best practices and recommendations for future projects. This ensures that experience and knowledge are accumulated and shared, contributing to the improvement of quality in subsequent projects.

The project management approach emphasizes stakeholder engagement throughout the project lifecycle. Regular communication channels are established with all stakeholders including project sponsors, end users, technical team members, and management. Stakeholder feedback is

actively solicited during requirements gathering, design reviews, user acceptance testing, and final deployment phases. This collaborative approach ensures that the system meets user expectations and business objectives while maintaining technical excellence.

Resource management is carefully planned and monitored throughout the project. Human resources are allocated based on skill sets, experience, and project phase requirements. The project team structure includes roles such as project manager, system architect, database administrator, frontend developers, backend developers, UI/UX designers, quality assurance engineers, and technical writers. Resource allocation is adjusted dynamically based on project phase priorities and workload distribution. Equipment and software resources are procured and configured in advance to avoid delays, with backup plans in place for critical resources.

Change management is an integral part of the methodology, recognizing that requirements and scope may evolve during the project lifecycle. A formal change control process is established to evaluate, approve, and implement changes while maintaining project scope, timeline, and budget control. Change requests are documented, assessed for impact on schedule, cost, and quality, and approved through a designated change control board before implementation. This structured approach prevents scope creep and ensures that all changes are properly evaluated and integrated into the project plan.

Progress monitoring and reporting are conducted on a regular basis to track project status against planned objectives. Key performance indicators (KPIs) are established including schedule variance, cost variance, quality metrics, and resource utilization. Weekly status reports are generated and distributed to stakeholders, highlighting completed work, work in progress, upcoming milestones, risks, issues, and any deviations from the plan. Dashboard visualizations are used to provide at-a-glance project health indicators, enabling quick identification of areas requiring attention. The methodology incorporates best practices from industry standards such as PMBOK (Project Management Body of Knowledge) and agile principles where appropriate. While the overall structure follows a sequential phase-based approach suitable for a comprehensive system development project, agile practices such as iterative development, continuous integration, and regular feedback loops are integrated into the development phases. This hybrid approach combines the structure and predictability of traditional project management with the flexibility and responsiveness of agile methodologies. Overall, this project management methodology provides a comprehensive and structured framework for managing the Educational Management System development project, ensuring that the project is executed efficiently, on time, within budget, and meets quality requirements. The combination of detailed WBS, Gantt chart with clear timeline, milestone management, risk management, quality assurance, comprehensive documentation, iterative improvement, stakeholder engagement, resource management, change management, and progress monitoring creates a robust and flexible project management methodology suitable for the complex and diverse nature of this large-scale software development project.

7.2 PROJECT PLANNING

7.2.1 Work Breakdown Structure

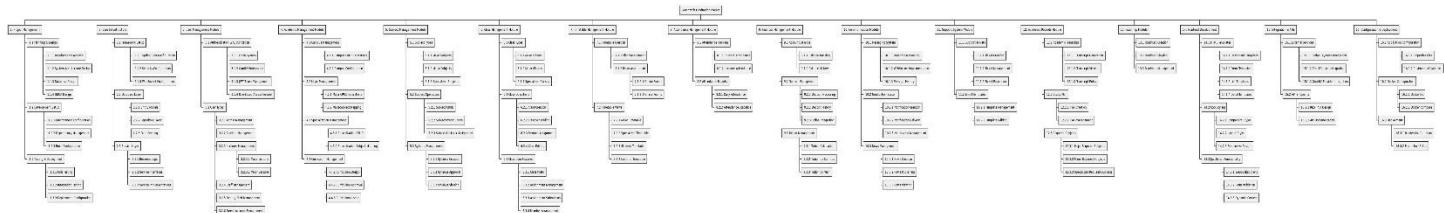


Diagram 68 Work Breakdown Structure

The Work Breakdown Structure for the Greenwich Graduation Project is a comprehensive university management system built on the Spring Boot platform, including 16 main work packages clearly classified from the planning stage to the operational deployment stage. The project starts with Package 1 - Project Management with activities of requirement analysis, system architecture design, database design, UI/UX design, development environment configuration using Maven and setting up the testing deployment process. Package 2 - Core Infrastructure builds the core technical foundation including Spring Boot configuration, Spring Security with role-based authorization mechanism, WebSocket for realtime features, entity models, repository, data seeding and clearly separated service layer. Package 3 - User Management Module implements an authentication system using JWT, integrates OAuth2, manages many types of users such as administrator, student, major lecturer, minor lecturer, staff, deputy staff and parent with separate permissions. Package 4 - Academic Management Module manages campus, major, specialization and curriculum with full approval process. Package 5 - Subject Management Module classifies subjects into major subject, minor subject, specialized subject, supports CRUD, manages prerequisites, assigns lecturers and manages syllabus. Package 6 - Class Management Module supports three types of classes, allowing to create class schedules, assign members, post lectures, assign assignments, submit assignments and manage class lists. Package 7 - Timetable Management Module provides timetable creation, slot management, physical and online classrooms, and displays schedules from the perspective of major, specialization, individual students and lecturers. Package 8 - Attendance Management Module tracks student and lecturer attendance and generates detailed statistical reports. Package 9 - Financial Management Module manages account balances, deposits via Stripe, transaction history and calculates flexible tuition fees by campus and academic year. Package 10 - Communication Module includes a real-time chat system via WebSocket, push notifications, and internal news management. Package 11 - Support System Module provides support ticket creation and email template management. Package 12 - Academic Records Module stores transcripts, study plans, and tracks required courses to ensure graduation requirements. Package 13 - Roadmap Module supports long-term study planning for students. Package 14 - Frontend Development uses HTML templates, responsive CSS, and JavaScript with WebSocket client, validation, and dynamic effects. Package 15 - Integration & APIs integrates Stripe, email service, OAuth2 provider, and provides a fully documented RESTful API. Package 16 - Configuration & Deployment manages application

properties, configures Docker, Dockerfile, Docker Compose, and deploys to the Render platform for production operations. This entire structure ensures that all the functions of a modern university management system are systematically covered, each module is easy to manage independently while maintaining tight integration and long-term maintainability and extensibility.

7.2.2 Gantt Chart and Timeline

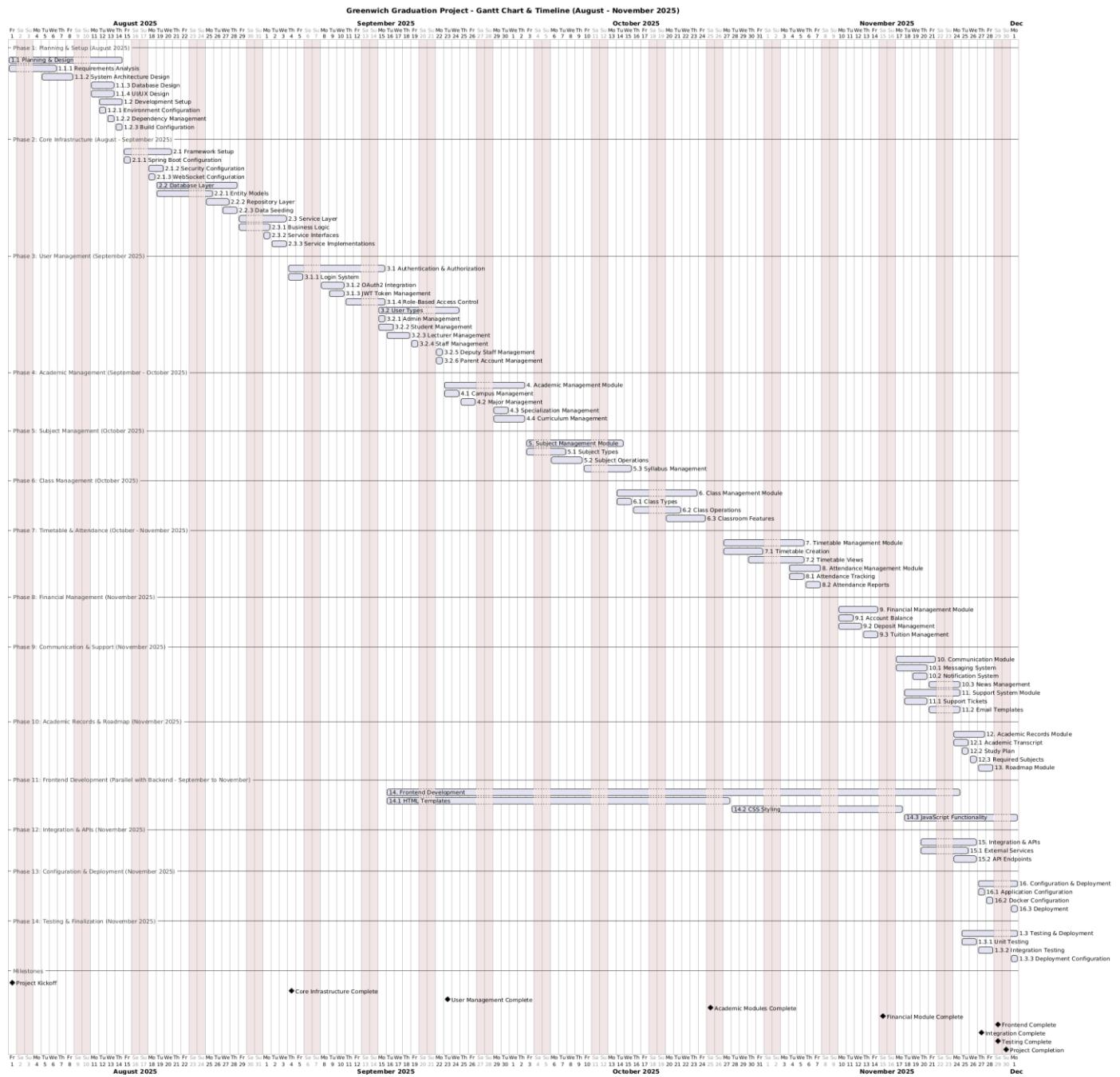


Diagram 69 Gantt Chart and Timeline

The Greenwich Graduation Project – a comprehensive university management system using Spring Boot – is planned to be implemented within 4 months from August 1, 2025 to November 30, 2025 with a very tight and reasonable schedule. The project started with the Planning & Setup phase in the first half of August, completing all the requirements analysis, architectural design, database, UI/UX and development environment configuration in just the first 2 weeks. Immediately after that, from mid-August to early September, the team focused on building the Core Infrastructure (Spring Boot, Security, WebSocket, entity, repository, service layer) to have a solid backend foundation before deploying the business modules. September 2025 is the phase of intensive development of core modules: complete User Management with full JWT authentication, OAuth2 and management of 6 different types of users (admin, student, lecturer, staff, deputy, parent), and develop Frontend in parallel from the third week of September. From the end of September to mid-October, complete Academic Management (campus, major, specialization, curriculum) and Subject Management. From mid-October, the team continuously deploys Class Management, Timetable Management and Attendance Management to ensure the official academic functions are operational before entering November. November 2025 is the phase of accelerated completion and integration: at the beginning of the month, complete Financial Management with Stripe integration, in the middle of the month, complete Communication (realtime chat, notification, news) and Support System, at the end of the month, complete Academic Records and Roadmap. Frontend is developed in parallel from September and finished at the end of November with full HTML template, responsive CSS and realtime JavaScript. The last week of November will also include Integration & APIs (Stripe, email, OAuth2, RESTful API), Docker configuration and deployment to Render, along with a focused Testing phase (unit test + integration test) from November 25. The important milestones are clearly set: project launch on August 1, core infrastructure completion on September 4, academic module completion on October 25, finance completion on November 15, frontend completion, integration and testing at the same time on November 29, and the official completion of the entire project on November 30, 2025. This schedule shows a reasonable allocation of work, with a reasonable overlap between backend and frontend, prioritizing the completion of the platform and important modules first, and dedicating the last week to integration, testing and deployment - ensuring the project can be completed on time with high quality.

7.2.3 Resource Allocation

The effective implementation of the Greenwich University Management System (GUMS) is not just a matter of proper technical design and risk management, but also proper allocation of human, temporal and technical resources during the project lifecycle. The project team is structured in a human resource in such a way that the distribution of roles takes into consideration the role of the member in the team taking into account the strength of that member and at the same time promote cross-functionalism and joint ownership. The roles commonly consist of a lead developer or architect to make high-level architecture decisions and integration, developers to implement backend functionality (domain logic, persistence, security, integrations), frontend and UX (user flows, layouts, responsiveness) and quality-related work (testing, documentation and deployment scripting). The project is based on the collaborative model, as opposed to the rigid, siloed roles, the key features, including user and role management, timetable and class management, assessment and grading, finance overview, or support ticketing, are handled as

vertical slices that cross the backend, frontend and data layers, with each slice ownership by a small group of team members, but reviewed by others to avoid single points of failure. This will allow them to share knowledge, also the consistency of codes improves and that one highly important aspect of the system can be comprehended by more than one individual. The project schedule is also organized into phases, requirements and analysis, architectural and detailed design, core implementation, integration and testing, deployment and evaluation with distinct milestones and deliverables to each phase in terms of time resources. The estimates of effort are arrived at in user story/ feature level and are plotted on a schedule considering academic deadlines, availability of the team and the uncertainties that come with software development. High-priority, high-risk items (e.g. authentication, authorisation, financial calculations and persistence modelling) are planned earlier in the implementation phase and thus have more time to be refactored and corrected in case problems are found. Non-critical UI refinements and improvements to lower-priority are placed later on the schedule deliberately so that they can be downsized or delayed in case they need to be without impacting the core functionality of the system. Integration testing, bug fixing, documentation, demonstration preparation and user acceptance testing are also set aside explicitly with time, as these activities are symbolically known to be consuming more efforts than what was originally expected. The same is applied to technical resources. At the level of development, the project adheres to a standard technology stack, namely, Spring Boot on the backend, a relational database (e.g., MySQL) on the persistence side, a modern frontend framework and a small but selective number of auxiliary libraries and tools, to prevent technology sprawl and decrease the cognitive load of the team. The code is stored in a common version control repository and there are branching and pull request conventions that facilitate code development and reviews. Continuous integration pipeline unites build, testing and deployment processes and is considered a first-class technical asset instead of an afterthought. In the case of the deployment environment, a cost-effective cloud environment like Render is chosen, and the resource level (CPU, memory, storage, database capacity) is validated to the appropriate size based on a capstone project and still capable of scaling in case the system becomes more popular. Basic monitoring and backup mechanisms, logging are also regarded to be vital technical resources and are also provisioned accordingly to aid in debugging, reliability and contingency planning. Last, documentation, training resources, and engagements with the stakeholders are considered a resource intensive activity on its own and are explicitly considered in the allocation process, instead of being crammed into remaining time. Internal review session, stakeholder meetings and supervisor consultations are provided in the project plan with development activities to guarantee that feedback, directions and alignment is available at important decision-making stages. The resource allocation plan enables a realistic sustainable course to producing a strong, testable version of GUMS using the means of planning and control of people resources, time, tooling and infrastructure, based on priorities, risks and project goals.

7.3 Risk Management

7.3.1 Risk Assessment

The construction and implementation of the Greenwich University Management System (GUMS) entails numerous technical, operating and organisational risks, which should be discovered, examined and observed so that the project is viable, stable and directed to the institutional expectations. Risk assessment process was initiated by conducting a systematic review of possible threats on the major dimensions of risk that included the technology choice, system design, data integrity, project schedule, staffing capacity, stakeholder interactions and deployment environment. Technically speaking, the integration and consolidation of various academic areas: timetabling, attendance, assessment, financial tracking and support services into one monolithic system is one of the major risks since it puts a larger chance of cascading failures, data inconsistencies or performance bottlenecks in case modules were not carefully designed and tested. The other notable technical risk is related to persistence and relational mapping, where system failures in the design of the database schema, inheritance mapping policies or complicated entity relationships can result in the creation of data anomalies or difficulty in the migration. As well, the use of clouds presents threats associated with the reliability of the service, the configuration of the environment, faults in the CI/CD pipeline, and possible constraints on the vendors, especially when using solutions like Render to host the backend and database services. The operational risks also need to be taken care of. These are the fact that key stakeholders such as students, lecturers, academic staff and support units may have only little time to get feedback or user acceptance testing, which will slow requirement validation or decrease the level of functional correspondence between the system and real-world processes. The risks associated with the teams, including the unequal distribution of workloads, the lack of skills in the usage of higher-order features of Spring Boot or database optimisation, and the inability to withstand personal limitations, could impact the development speed and quality. Moreover, since academic and financial regulations are very complex at Greenwich Vietnam, there is a probability that wrong logic in assigning scholarship, calculating money or tracking of academic advancement may be developed due to the misinterpretation of institutional regulations. At the organisational aspect, the shift of the currently fragmented environment (AP, CMS and FLM) to a single system poses compatibility risks whereby some of the existing processes might be based on undocumented behaviours or old data structures that will not be fully represented during requirements analysis. Data security and data privacy threats should also be taken into consideration particularly when accessing sensitive student details, bank account details, support tickets and parent-related accounts, improper access control policies, or lack of input validation may result in a vulnerable system. Lastly, there are risks associated with the timing, i.e., failing to estimate the labor to be invested in testing, debugging or documentation, which threatens the successful delivery of the project in time, especially with the inflexible nature of the deadlines in a capstone project. With the early identification of these risks, analysis of the probability of their occurrence and the expected severity, the project can design the relevant mitigation measures, focus on testing and validation tasks, and have a realistic scope. This relative evaluation is the basis of later risk management plan and makes sure that GUMS is being designed with a high degree of awareness and strength against uncertainties involved in creating institution-wide digital system.

7.3.2 Mitigation Strategies

As a means of reacting to the risks as determined in the preceding subsection, a series of specific mitigation measures have been outlined so as to minimize the probability and effect(s) of the unpleasant incidences during the lifecycle of the Greenwich University Management System (GUMS). Technically, the difficulty of integrating several academic, financial and support domains into a single monolithic system is overcome by the adoption of a modular, feature-based architecture within the monolith, with well defined boundaries defining core areas of the monolith, such as user management, timetable and class management, assessment and grading, finance and support services. This modularisation is supported by the presence of well-organised service interfaces and data access layers which minimises the risk of uncontrolled coupling and facilitates easier isolation and resolution of defects that are contained within single modules. To solve possible problems with the database schema design and the use of inheritance mapping, the project utilises an incremental modelling and validation process, where the key entity relationships are prototyped in a sandbox database, tested with seed data and unit tests, and discussed with the domain stakeholders before being finalised; database migrations are managed with a special migration tool (e.g. Flyway or Liquibase), permitting safe, versioned schema evolution and allowing rollbacks if issues are identified. The performance and reliability issues are addressed with a blend of simple performance testing, query optimisation and proper utilisation of caching on read-heavy views and by checking resource utilisation and response time in the cloud environment and adjusting the configuration where needed. The risks related to cloud deployment and the CI/CD pipelines are managed through the use of staged environments and controlled release practices. The project also uses distinct development and staging environments in which new features and database migrations go through end-to-end testing and are then promoted to the main environment instead of going there directly. The continuous integration pipeline is also set to execute the unit and integration tests at each commit, impose coding standards and fail builds in the case of important issues and, in this way, reduce the risk of introducing regressions without noticing them. The deployment scripts and the environment configurations are put down on paper and maintained in versions to avoid configuration drift whereas application logs and cloud platform monitoring tools are used to identify early the existence of runtime errors or operational anomalies. Risks related to data privacy and security are also addressed based on role-based access control, the least-privilege principles and the input validation: each user role (student, lecturer, staff, parent, administrator) is explicitly mapped to particular permissions, sensitive operations are only performed by authorised roles, and all inputs, in particular those that might influence financial transactions or even system configuration, are authenticated on both client and server sides. The attack surface is further minimised by the use of HTTPS to communicate, hashing of passwords with strong algorithms and the use of secure coding practices. The operational and organisational risks are addressed with continuous interaction with the stakeholders and a clear process alignment. To decrease the possibility of misunderstanding academic or financial policies the key workflows (e.g., scholarship assignment, retake rules, attendance impact and financial calculation) are also modelled in cooperation with the academic and finance staff and presented as reference process diagrams, which they are subsequently employed in the implementation and testing. Check-ins with the representatives of the stakeholders and the academic supervisor will serve to validate the interim assumptions and make corrections to the requirements where required as well as keep the system in tune with the real practices within the institution. The threat of small number

of stakeholders available to provide feedback and do user acceptance testing is addressed by arranging small and intensive evaluation sessions with clearly defined scenarios and prepared test data to ensure meaningful feedback is obtained within small timeframes and integrated during the process. The risks associated with teams: e.g. an uneven distribution of workload or lack of skills in particular technologies: Task decomposition, shared code ownership and internal knowledge sharing: divides critical features into smaller, manageable tasks, and critical components are reviewed by more than one team member, short internal sessions: e.g. spread knowledge about the essential technologies (e.g. Spring Security, JPA, cloud deployment) among team members. Realistic planning, prioritisation and scope control are used to help eliminate schedule and scope risks. At the planning phase, the project is broken down into phases (requirements and design, core implementation, integration and testing, deployment and evaluation), with each phase having specific milestones and deliverables; in each phase, features are classified as must-have, should-have and could-have, with core functionality (user and role management, timetable, assessment, finance overview, support tickets) being developed and stabilised before considering lower-priority enhancements. Buffer time is factored in to allow debugging, refactoring and documentation and progress is checked on a regular basis so that in case of any delays non essential features can be postponed or simplified without affecting the integrity of the overall system. Lastly, the vagueness of the changing requirements is also alleviated through adopting an iterative agile-inspired workflow whereby the team will provide small, testable segments of functionality, which will be reviewed with the stakeholders and altered in the following iterations. All of these mitigation strategies put together will produce a framework but adaptable risk management strategy that facilitates the provision of a solid, secure and context-sensitive GUMS platform despite the limitations of a capstone project.

7.3.3 Contingency Planning

Although mitigation strategies would help minimize the occurrence and effects of risks, contingency planning contributes to a well-organized fallback plans to allow the Greenwich University Management System (GUMS) to continue its operations when the high-impact events or failures take place in spite of the preventive measures. Contingency Plan is aimed at providing explicit response channels in case of the technical, operational and organisational interference that can disrupt system stability, data integrity, progress of development or stakeholder interaction. On the technical level one of the main contingencies is to ensure regularly versioned backups of source code, as well as database snapshots, kept in secure, redundant cloud environments, without depending on the primary deployment environment. Should a disaster happen and data is lost or there is corruption of the schema or there is migration failure, the team can roll back to the last proven snapshot, making sure that the necessary academic, financial and support data can be restored. In case of deployment-related problems, like excessively long downtime on the hosting platform, a CI/CD pipeline malfunction or environment misconfiguration, etc., the plan would consist of having the option to temporarily switch to a different deployment environment (e.g., a fallback Render instance, a Railway or DigitalOcean deployment, or even a local-hosted emergency environment) during the time the primary system is being fixed. This saves inconvenience during very crucial academic times like enrolment or assessment weeks. When the core modules are destabilized by unexpected bugs or performance issues, the contingency plan outlines a graceful degradation plan, which is to temporarily shut-down non-essential functionality

(e.g. non-critical dashboards, file-intensive operations, or complex analytics) and maintain essential academic and financial processes. This will make sure that the students can still access timetables, lecturers can take attendance and the staff can keep on monitoring important transactions even during partial maintenance. The project also has its share of diagnostic scripts and logging settings that are easily activated at the higher verbosity levels to speed up the troubleshooting process in the emergency situations. In case of the failure caused by the problems in the database schema, the entity mappings or business logic inconsistencies, the contingency protocol also tells the team to restore to the latest stable version stored in the Git repository, with a well-defined list of manual checks to confirm the data integrity and then the full functionality is re-enabled. In terms of operations, the contingency plan appreciates that the delay or lack of involvement by stakeholders may pose a risk to requirement validation, user acceptance testing and final evaluation. To control this, the project creates a list of secondary contacts with stakeholders namely the other lecturers, the staff or student representatives, who can intervene should primary contacts be rendered unavailable so that there is continuity in feedback cycles. In case of institutional policy clarification taking too long, the system in the meantime marks some of the rules (mostly financial or academic logic) as parameters to be set, which would then enable the implementation to proceed with the implementation pending final validation. Should there be an unforeseen loss in team capacity such as sickness, personal crises or technological expertise skills, contingency schedule entails redistribution of the tasks through a shared ownership model with the support of documentation, internal notes of knowledge sharing and cross-reviewed code to reduce single points of failure. The project plan is also incorporated with schedule-related contingencies. The timeline also contains buffer points around the testing and deployment phases to take the delays by bug-fixing, refactoring or integration issues. When these buffers run out, the project can trigger a controlled scope-reduction protocol, that is, it gives priority to must-have features (user authentication, role management, timetable, assessments, financial overview and support ticketing), and pushes back low-priority enhancements to subsequent phases without impairing the value of the system itself. Moreover, the checkpoints that are crucial like the database finalisation, core service finalisation and user testing windows will have pre-set fallback dates and alternative workflows so that the deliverables are not undervalued in terms of achievement even in the worst-case scenarios. Lastly, the contingency plan will provide communication and escalation arrangements to ensure transparency and coordinated reactions in high-impact events. Whenever there is a technical or organisational impact, the team uses a formal chain of escalation: instant communication between team members, invocation of diagnostics, communication with stakeholders (in case user-facing functionality is impacted) and escalation to a supervisor as needed to get advice or allow scope change. The formalisation of these response paths will make the project such that unexpected failures will not lead to ad-hoc decision-making and long downtimes. This contingency planning model will enable GUMS to be more resilient and, therefore, be able to absorb uncertainties and keep on the road to successful completion and deployment.

7.4 Change Management

To make sure that the process of changing the current systems (AP, CMS and FLM) to the new single system of Greenwich University Management System (GUMS) is successful, the

change must be managed effectively to ensure that the transition is not difficult, but it will resonate well with the expectations of the institution and will cause minimal disruption to all the stakeholder groups. Since GUMS will position not only a new technical base but also new working processes, centralized data format and reshaping of user roles and duties, the project will be organized based on a well-defined communication strategy, step-by-step implementation and periodic validation. The central aspect of this strategy is that the successful change will require the stakeholder awareness and approval; hence, the project puts an emphasis on the initial communication with the academic staff, lecturers, finance and support divisions to clarify the reasons of consolidation, showcase prototypes, receive feedback and modify workflows with the help of practical operational lessons. This aids in the minimization of resistance also the expectations are kept realistic and a feeling of ownership to the development of the system is created. Implementation is done in stages as opposed to introducing changes. Individual features, including timetable management, attendance tracking, grading processes, financial dashboard or support ticketing, are all tested separately via demonstration sessions, simulation data conditions and guided user tours. The results of these sessions are used to make adjustments to the further iterations and make sure that the change is based not on the assumption but on the real need of the people. The development team also keeps a good documentation such as the current process flows, role-permission maps and UI usage manuals such that the users can easily comprehend how the new system works as opposed to the legacy environment. This is documented at a young stage and is regularly updated in order to be used as a source of both training and onboarding. Change management also includes orientation of the users to the operational and behavioural changes necessary to go to an integrated system. Since GUMS brings together activities that were previously decentralised in various platforms, it requires training so that the stakeholders are aware of where to access information, how to analyse consolidated records and how new workflows impact on their daily workloads. In this regard, the project describes a step-by-step training program which includes brief orientation of students and parents, direct training of lecturers and academic staff, and specific training of specialists in finance and support staff who handle sensitive processes. These sessions include the practical demonstrations, guided exercises and before-after comparisons to emphasize the improvement and explain alterations. Besides, there will be a helpdesk type communication channel ready to respond to questions and problems that may occur during adoption. Governance wise, the project will have a controlled scope change management process, requirement refinements and technical changes. Any suggested change, regardless of who proposes it, developers or supervisors, is assessed based on compatibility to project goals, possible effects on the existing functionality, the possibility of implementation within the scheduled timeframe and the consequences to the integrity of data or system stability. Change of high impact proceeds through a formal review procedure, which includes documentation changes, study of the affected modules and explicit permission by the supervisor or the decision-maker. Transparency and traceability are achieved by having the lower-impact refinements added to the iterative development cycle and monitored with the project issue management tool. Lastly, change management does not stop at the implementation but goes on to deployment and initial use. A managed roll out plan is established, which will start with a small pilot testing process with the selected stakeholder groups to test real world usability and make any final changes before implementing in a wider scope. During this change, feedback is constantly gathered and any serious matters are given first priority to be sorted out. The project will enable the introduction of GUMS not only to be technically successful but also an organisational one by using a combination of stakeholder-centred communication, gradual roll out

of features, systematic training, and strict control over the scope and technical changes, which will facilitate long-term adoption and effective usage of the project in the whole institution.

7.5 Team Collaboration and Communication

Though the creation of Greenwich University Management System (GUMS) is done not by a group of three or more people but by one person, the concepts of team collaboration and team communication are still used in a small but significant manner so that the project is not disorganized, not transparent and not maintainable. Without intra-team coordination, attention is placed on the organized self-management, the systematic documentation and regular communication with the academic supervisor and other stakeholders. In practice, this implies that the student performs more than one role at a time: analyst, designer, developer, tester and deployment engineer, and again the student is required to ensure that he has clear boundaries between these roles by working in phases, setting milestones and tracking progress in a disciplined way. Simple project management tools or issue tracking boards (e.g. GitHub Issues or project boards) are used to maintain task lists, with features, bugs and technical chores being logged, prioritised and updated as time progresses; this simulates the style of team-based collaboration workflows and serves as an aid in ensuring that no essential activity is forgotten or constantly put off. Communication is also an important aspect of the project even though it is not between or among the different developers. Frequent meetings with the academic supervisor would be the main feedback and alignment tool: the interim progress will be reported, design decisions and trade-offs will be discussed, and the possible risks or scope changes will be addressed. Such interactions are reinforced with brief written summaries, diagrams and demonstrations of the current system state, which can be used to keep the supervisor informed and records of the decisions that have been made over time. Ad hoc feedback is also obtained wherever possible based on representative users, e.g., students, lecturers or administrative staff, through informal discussions or brief demonstrations so that the developer can confirm the usability and workflow assumptions. The learning of these communications is recorded and summarized into actual modifications in the requirements backlog, design models or implementation activities. In order to mitigate the lack of peer review and pair programming, the project uses a self-review strategy and externalisation: vital architectural decisions, security-critical code paths and complex business logic (e.g. financial calculations, role-based access control and persistence mappings) are clearly described in design notes, UML diagrams or code comments. This is further facilitated by the fact that the version control system used (Git) gives a chronological, fine-grained history of changes and the commit messages of the changes are descriptive and explain the purpose of the change. It can serve as a form of communication between employees over time, in that the developer can use it to communicate with their own future, and it adds a sense of transparency to the repository in the event a supervisor or reviewer is going to look at it. Lastly, the project acknowledges the fact that good communication does not just exist in the form of synchronous meetings or messages, but also in the quality and consistency of artistic work: requirement specifications, design diagrams, schema definition and configuration files, test cases and user-friendly documentation are all written to expect that another developer or institutional stakeholder will one day need to continue on this work. In managing these artefacts as a communication medium by themselves, the project guarantees that, even though the artefacts are created by a

solitary individual, GUMS is still understandable, auditable and extendible. Through this, the standards of team cooperation and communication are maintained by rigorous self-organisation, guided interaction with bosses and parties concerned, and a critical concern to the quality of the written and technical documentation, instead of the multi-person coordination in isolation.

7.6 Budget Management (if applicable)

Since this project is carried out in the framework of an academic final project as opposed to a commercial software development project, there is no official institutionalized financial budget. The entire process of designing, implementation, testing and deployment of the Greenwich University Management System (GUMS) is done using tools that are available freely or community versions or student-licensed technologies. The important parts of the technology stack, including Spring Boot, MySQL Community Edition, Maven, Thymeleaf, Lombok, MapStruct, and most of the development plugins are open-source and, thus, do not require any licensing fees. Demonstration Cloud deployment is introduced with free-tier or low-cost trial offerings of cloud platforms (e.g. Render) to make sure that the system can be hosted and put into test without incurring any financial cost. Any version control, CI/CD, documentation, diagramming or communication tools (e.g. GitHub, GitHub Actions, Draw.io, Google Docs, Slack or other similar tools) are also free to use as an educator. Even though there are no explicit costs associated with the project, the mindset of the project is that of budget-conscious software engineering by ensuring optimal usage of resources, avoiding any superfluous external dependencies and the choice of technologies that consume the least operational overhead. Practically, this translates to designing the system within bounds that its features comply with the free-tier offering of cloud hosting, selecting lightweight architecture patterns to minimise compute needs and keeping datasets, logs and fixed assets in sensible storage limits. The indirect soft costs, like time, effort and investment of learning, are controlled by proper planning, balancing the workload, and micro-developing strategy of the project so that the project progress can be sustainable even in the context of academic time. In case the system is extended or implemented in a real institutional environment in future, it would need a formal budget to pay the hosting, the scaling of the database, the security auditing and maintenance, but these would be beyond the bounds of the current academic implementation. Overall, the financial budget cannot be applied to this capstone project, but the awareness of resources and a cost-effective approach to decision-making remains the main guide to the technical decisions and operational strategy during the GUMS development.

7.7 Quality Management

Quality assurance during the process of developing the Greenwich University Management System (GUMS) is one of the main goals that is instilled in every stage of the work: requirements analysis and design, implementation, testing, and deployment. Although the project is developed by an individual developer, a coordinated quality management plan is used to ensure the reliability, consistency and correctness of the functional and non-functional features of the system. This

strategy is based on the need to create a visible and traceable connection between the requirements specification and the applied features. The functional requirements, the use cases and the acceptance criteria are recorded in a systematic manner and constantly monitored throughout the development of the program to enable each of the modules implemented; be it an academic management module, a financial tracking module, or an authentication module, or a lecturer evaluation or support module, to serve its purpose. This initial focus on clarity minimizes the ambiguity and assists in preventing the appearance of defects at the later stages. Simultaneously, quality is ensured by disciplined software engineering. Following a clear project organization, the usage of the most suitable design patterns (layered architecture, DAO/repository abstraction and DTO mapping with MapStruct) and clean and modular code enhance maintainability and minimize technical debt. The use of coding standards is adhered to and descriptive naming rules are used such that the codebase is readable not only by the developer but also by supervisors, reviewers or future teams that might extend the system. Git version control is relevant to quality oversight as it offers change history at a fine level of granularity, allows change to be reversed, and encourages small, verifiable, and incremental development instead of big, error-prone code drops. The other vital pillar of quality management of the project is testing. Though the entire Test-Driven Development (TDD) workflow is not followed on all components, the test-first culture is practiced on modules with complicated business logic, like tuition calculation, rules about scholarship, attendance assessment and authentication processes. To test correctness against regressions, unit tests and integration tests with the Spring boot testing framework are run on a regular basis and to confirm that there are no regressions and that interactions between service, repository and controller layers remain stable across the service, repository and controller tiers. The CI pipeline also includes automated tests, and it implies that whenever a new change is introduced, it is verified in an non-negotiable manner before it is merged to the main codebase. In addition to automated testing, manual testing and scenario walks-through are also performed to test user experience, check on the multi-role flows (student, lecturer, staff, parent, admin) and identify edge cases not covered by the automated tests. Other non-functional quality items, like performance, security, usability, maintainability and deployment reliability, are also handled with specific strategies. Monitoring of performance is done by SQL optimisation, caching configurations and proper management of settings regarding HikariCP connection pools. The quality of security is enhanced by Spring Security, role-based access control, password hashing, input validation, and sanitisation and rigorous treatment of sensitive information with the help of environment variables as opposed to hardcoded settings. The quality of usability is also provided by the process of constant improvement of Thymeleaf templates, simplicity of layouts, the use of meaningful navigation systems and consistency between dashboards. A CI/CD pipeline enhances deployment quality by automatically building and testing to either staging or production environments and allows high quality deployment of applications that are less prone to manual errors and ensure reproducible builds. The documentation is not considered as the secondary deliverable but as the part and parcel of quality management. There are requirements documents, design documents, API documentation, configuration documentation, deployment documentation and user documentation that are continuously updated to capture the present state of the system. This not only provides transparency but also makes the communication between the supervisors easier and also makes the project review and validating or developing further easy without much strife. Taken together, these practices show that quality in GUMS is not provided by one technique, but in a multi-layered multi-dimensional paradigm of combining disciplined engineering, automated validation, structured documentation

and continuous improvement. These measures make GUMS robust and maintainable and within professional standards of software development even in the limitations of a student-led capstone project.

7.8 Ethical Considerations

7.8.1 Data Privacy and Protection

In the case of a web-based university management system, such as GUMS, the primary ethical considerations are data privacy and protection due to the fact that the platform handles much sensitive personal data about students, lecturers, staff, and in certain cases, parents, such as identification details, academic performance, financial records, disciplinary history, support tickets, and communication logs. In this context, the system shall be fabricated and operated in line with data-minimization and purpose-limitation ideals: data that are actually required in an unambiguously defined educational and administrative objective shall be gathered, stored and processed and these data shall not be reused to achieve other unrelated objectives such as covert profiling, commercialization and intrusive tracking of people. Role-based access control is an essential mechanism that helps to enforce these principles in practice, since only the minimal amount of information is provided to each stakeholder, student, lecturer, academic officer, support staff, or administrator, and the highly sensitive areas, such as grades, financial balances, or support histories, are not disclosed until they really need to be. Moreover, authentication should be enacted using strong passwords, hash powerful schemes and session controls and the entire client-server communication should go through transport-layer encryption (e.g., HTTPS/TLS) to avoid eavesdropping and man-in-the-middle attacks when the user connects with GUMS through the public networks or personal computers. Storage wise, sensitive data sets and backups should be secured on a database with the right permissions and segregation between environments (development, testing, production) and where possible, at rest with encryption, to complemented by high quality logging and audit trails to identify and investigate unauthorized access or suspicious activity. It is also essential to be transparent: users must be clearly told, in simple language, what data is collected, what is it used to, how long, who can see it and what rights they have to access, correct, or delete it in accordance with the principles of common data-protection requirements and policies of the institution. All such integration with third party services, like email services, cloud hosting, or analytics services, should be preceded by a thorough review of their privacy policies and contractual protections to help prevent the transfer of student and staff data to unreliable parties and jurisdictions without such a transfer being adequately safeguarded. Lastly, data-privacy and protection are not a single technical action but a continual job: the university must develop governance processes, staff training and periodical reviews or audits are necessary to make sure that GUMS is not violating the institutional policies and applicable regulations, addresses the situations or violations appropriately, and is always conscious of the dignity, autonomy, and trust of all people whose data are entrusted to the university system.

7.8.2 Sustainability Aspects

Regarding sustainability, the design and implementation of GUMS are also intended not only to address the immediate operational issues but also to lead to the long-term sustainability of the environment, economy, and the organization of the university in general. By integrating fragmented systems like the Academic Portal, Course Management System and Faculty Learning Management into one integrated system, GUMS is able to greatly decrease the number of systems that rely on similar infrastructure, redundant storage and duplication of data in addition to eliminating the existence of multiple administrative processes that tend to be duplicated and unnecessary, which also can greatly lower the amount of energy used in running and maintaining multiple servers and services. Digitization of most of the significant academic and administrative procedures, including schedule allocation, publication of grades, fee administration, and administration of support tickets, also lowers the expenses of paper-based forms, printed schedules, and hard-copy notification, thus assisting to decrease the amount of paper waste and the environmental footprint of daily academic activities. Under the economic aspect, a single system would be able to enhance the level of sustainability by cutting the maintenance overheads, licensing on the multiple platforms and even the human effort of manually matching data in isolated systems and redirecting the resources towards activities that directly aid in teaching, learning and student support. Meanwhile, the system is built in a way that it is maintainable and extensible: with a modular architecture, clean separation of concerns, and broadly used technologies, it can be updated, scaled, and modified with time without necessitating to be disposed of and substituted wholesale, meaning that it will cause additional resource waste and technical debt. The social and organisational sustainability are also addressed as GUMS assist in making academic monitoring, financial management and communication processes more transparent and efficient and contribute to the creation of trust and reduction of frustrations among students and staff, which is the key to the long-term acceptance and sustainability of any institutional platform. Last and it is important to note that by pushing the system to a cloud-based service with proper capacity planning, demand-based scaling and frequent performance optimization the university can not only avoid over-provisioning their hardware but can also enjoy the energy-saving investments by cloud providers, which can help GUMS become a more responsible and sustainable digital service throughout its lifetime.

7.8.3 Social Impact Assessment

It has serious social implications on students, academic and administrative personnel, and the wider institution population since the introduction of GUMS as a single university management system, which has an impact on how students and other people engage in daily aspects of teaching, learning, and support processes. A centralised system that consolidates timetables, course materials, assessment, grades, financial information, support tickets, and communication channels can make the life of students less confusing and less stressful and less cognitively loaded with sorting through different disjointed systems, and more focused on learning, and could also allow students to be more responsible about their own academic planning and decision-making, since they have timely and transparent access to information about their academic progress and their financial responsibilities and obligations. Lecturers and staff: GUMS can make routine (attendance recording, publication of materials, grade handling, and a track of student problems, etc.) less ad-hoc, which may also raise job satisfaction rates, as less repetitive and more informative administrative systems help streamline the work process and result in greater

understanding of the class performance and student participation; although, it should not be overlooked that any new digital solution will create an as yet burden during an initial period of its implementation, and a mean of introducing the system to the team will have to be adequate, with adequate training and feedback opportunities to prevent the The other important dimension is social equity: as much as GUMS can encourage fairness as a result of standardising its processes, centralising records and rendering policies and information more visible and consistent across campuses, the university will have to make sure that the platform is accessible to users with limited connectivity, older devices or disabilities, such as designing an interface that is mobile-friendly, usable on low-bandwidth connections, or compatible with assistive technologies. Power and accountability are also affected by the system since the greater the visibility of data, the easier it is to track the students and staff performance, and take necessary measures to address issues early on before they become a crises like improved attendance rates, higher grades, and faster response times; analytics and reporting must be presented as a means of assisting students and employees, not as a means of penalizing them. Comprehensively, when applied in a deliberate and considerate way with its focus on inclusivity, accessibility, and engagement of users in the design and refinement process, GUMS can become a tool to enhance community integration, trust, and communication in the university, which will enhance the sense of support and belonging in students and allow staff to provide more consistent and responsive services in the academic and administrative processes.

7.9 Reflection on Professional Development

In the course of GUMS development, I have been able to grow enormously as a technical practitioner and a budding professional in the software engineering field. Technically, this project has compelled me to leave behind small and distant exercises and engage in a realistic and large scale system that incorporates various fields, such as academic management, financial processes, and support services. The process of designing and implementing a monolithic Spring Boot application using a layered and feature-based architecture design and non-trivial JPA domain model has improved my knowledge of the fundamental concepts of the subject, including separation of concerns, domain modeling, transactional consistency, and trade-offs among various architectural designs and persistence strategies. When converting high-level requirements into real entities, associations, and workflows, it was necessary to think deeply on data integrity, performance and maintainability and not merely make the code run. Simultaneously, the experience of working with such technologies as Spring Security, Thymeleaf, and MySQL in a single environment has enhanced my skills in reading documentation, debugging complicated problems, and making proper decisions when there are many possible options, which I believe are the key skills to developing as a professional in the long term. No less significant, this project has helped me to develop both in the sphere of soft skills and professional attitudes. Being engaged in a long-term project and having a lot of interconnected parts has made me more conscious of the significance of planning, prioritisation, and time management: I needed to divide the work into milestones, keep track of my own progress, and respond appropriately when some activity proved to be more cumbersome or time-consuming than expected. Communicating with supervisors, reflecting on the demands of various stakeholders (students, lecturers, staff, parents) and providing non-technical readers with the documentation have allowed me to train effective

communication and learn to defend technical decisions in a manner that would be comprehensible and assessable by other people. Considering these ethical challenges like privacy of data, its fairness and the social effects of the digital system has further reminded me that software engineering is not just about code but also about responsibility to people who will be involved in the system I develop. In general, this project has given me confidence in my skills in dealing with complex problems, working as part of an academic or professional team, and lifelong learning; it has also taught me to be dedicated to undertaking future tasks with both technical rigour and an awareness of ethics as well as a desire to further develop my skills.

7.10 Chapter Summary

This chapter has examined the management and professional dimensions of the GUMS project, showing that its outcomes are the result not only of technical design but also of deliberate planning, coordination, and ethical consideration. It first outlined the overall project management approach, including the decomposition of work into manageable tasks, the use of schedules and milestones to guide progress, and the allocation of limited human and technical resources to the highest-priority activities. The discussion of risk management highlighted how technical, organisational, and schedule risks were identified, monitored, and mitigated through contingency plans, iterative development, and scope control, helping to keep the project on track despite constraints. Stakeholder communication and team collaboration practices—ranging from regular meetings and feedback loops with supervisors to clear documentation and shared tools—were shown to be essential for aligning expectations, resolving misunderstandings, and integrating diverse perspectives into the evolving solution, while budget and quality management sections emphasised pragmatic trade-offs between ambition, cost, and the level of assurance achievable within the project timeframe. The chapter also addressed ethical and professional issues, particularly data privacy and protection, sustainability, and the broader social impact of deploying a centralised management system, arguing that design choices around access control, logging, analytics, and digitalisation must be informed by respect for users' rights, inclusivity, and long-term institutional responsibilities. Finally, the reflection on professional development underscored how the project contributed to the author's growth in areas such as technical judgement, communication, time management, and ethical awareness. Together, these elements demonstrate that the GUMS project was conducted as a professionally grounded endeavour, in which engineering decisions were framed within sound project management practices and an explicit commitment to responsible, user-centred system design.

CHAPTER 8: CONCLUSION AND FUTURE WORK

8.1 Summary of Achievements

The given project has managed to offer a unified Greenwich University Management System (GUMS) that consolidates the previously disintegrated academic information, course management and faculty operation systems into one, role-based web application, therefore, addressing the first problem of data duplication, inconsistent user experience, and inefficient administrative workflow. The desired essence functional concepts have been realized through the adoption of an elaborate student and staff management, timetables, attendance, academic transcripts, financial accounts and transaction histories, support ticket handling and lecturer appraisals modules that have been implemented in a consistent domain model and are supported by a relational database schema that is also sensitive to data integrity and consistency. The system is technically deployed using a monolithic Spring boot back-end, server-side rendering with Thymeleaf, and MySQL database with a layered, feature-oriented design to isolate the controllers, services, repositories and entities to make the system easier to develop and maintain. The security and privacy requirements have been addressed by the role based access control, strong authentication with password hashing of passwords, and permissions and permissions have been well scoped between the students, lecturers, the administrative personnel and parents and audit trails and history entities have been employed to offer transparency of the financial and academic operations. Benchmarking and resource utilisation analysis has ensured that the prototype achieves its performance and scalability goals, such as the ability to respond to loads with good response times and that the prototype is also stable and the user experience objectives have been alleviated by responsive dashboard oriented interface, which ensures that all the important tasks and information required by the various user roles is easily accessed. Finally, the project has not only generated a working prototype, but also a full-scale analysis, design, implementation, testing, and evaluation artefacts, proving that the system is somehow moving the institution towards integrated digital campus and generating a viable starting point based on which further development and subsequent enhancements may be realised.

8.2 Contribution to Knowledge

The knowledge contribution of this project is seen on the academic as well as the practical level in that it shows how a conceptually integrated, role-based university management system can be conceptually conceived, architecturally designed, and assessed in a transnational higher-education setting as in the case of Greenwich Vietnam. Ideally, the project builds on the existing literature on Learning Management systems (LMS), Student Information systems (SIS) and institutional portals by demonstrating how academic, financial, support and communication operations can be integrated in a single cohesive platform, instead of being viewed as discrete subsystems with ad hoc integrations; a gap that has been perceived in the literature with many solutions on teaching and learning or administrative records, but rarely the entire lifecycle of the

student experience, in academic, financial and support aspects. Theoretically, the project operationalises the proven theoretical concepts that include Technology Acceptance Model, the Unified Theory of Acceptance and Use of Technology, and the DeLone and McLean Information Systems Success Model into a tangible engineering scenario, with the constructs of perceived usefulness, perceived ease of use, system quality, and user satisfaction connected to particular design decisions of architecture, user interface, and workflow design. Software engineering wise, the project provides a highly detailed illustration of how the same concepts of joined-table inheritance, feature-based packaging and layered monolithic architecture can be integrated to a complex educational domain to balance between data normalisation, maintainability and extensibility to provide a reusable template of reuse in other institutions that require rich data models but do not require the operational cost of microservices. The results of the performance and usability testing also allow offering empirical data that a well-built monolithic, server-rendered solution can be able to achieve realistic response-time and user-experience requirements in a medium-scale deployment, which goes against the belief that a modern educational platform would need to adopt heavy client-side single-page applications or microservice architecture to be viable. Lastly, capturing the end-to-end process of requirements elicitation and domain modelling in the realization of the implementation, testing, and critical assessment, the project presents an organized case study that can be used to further study and teach about the design of digital campuses, enterprise information systems in education, and the reality of information systems theories applied to actual software development projects.

8.3 Lessons Learned

The GUMS experience has taught certain valuable technical and professional lessons of the way such large-scale information systems ought to be developed in future. Technically the project has demonstrated that domain modelling must be undertaken cautiously and in the early stage: it is better to take time in the beginning to understand the other entities, relationships, inheritance hierarchies, business rules pertaining to enrolment, assessment, finance and support tickets so that a lot of inconsistency and refactorings are not necessary at the later stages of implementation. In the meantime, the experience has shown that even the monolithic structure, which has been designed particularly well, is difficult to support without strict segregation of concerns, use of a uniform naming convention and a strict adherence to service and repository layers, which makes the significance of the coding standards, documentation and incremental refactoring throughout the project rather than another cleaning exercise. The steps of testing and evaluation also helped to see that functional correctness is not the sole element of quality performance, but usability and integrity of data should be a continuous process, not something that will be implemented at the end. The UX facet of the project demonstrated that the intuitive design assumptions made about what is and is not intuitively designed will be very wrong until the process of testing it with actual users is involved and that a little refinement of the navigation, labelling and layout can result in vastly more positive user reaction, without impacting the functionality itself. At the project management aspect, the article has shown the ease with which scope can be inflated within an organisation where many institutional processes are involved and the need to prioritize well, make trade-offs and often review the goals to ensure the project is feasible. Finally, the project helped to point out the ethical and organisational consequences of

technical decisions: the decisions between logging, access control and analytics are all choices with ethical and organisational consequences, which remain a reminder to the developer that the engineering judgement must always be judged in relation to both technical beauty and responsibility towards flesh-and-blood people and institutional constraints.

8.4 Critical Reflection

In a critical viewpoint, the GUMS project can be regarded as a significant success as well as a call to keep in mind what constraints and trade-offs must eventually arise in crafting software engineering in the real world. Positively, the system actually shows that it is not impossible to shift to a consolidated landscape of portals and tools to one consistent, coherent platform, more reflective of the lifecycle of the student and staff experience, and the project actually managed to transfer a lot of the theoretical considerations of quality, usability, and integration of the system into practice into decisions about design and functionality. Nonetheless, a more critical look also shows that decisions were made not out of the perfect design but due to time, knowledge, or feasibility: e.g., the monolithic architecture and server-side rendering were adopted as it was easier to develop and deploy, and some of the more ambitious concepts like analytics, automation, or deep external integrations were postponed or realised only partially. Although adequate to test core workflows and performance at moderate load, the test plan was not as thorough and automated as would be desired in a production-scale implementation, and this left some area to regression blindness and long-term maintainability. In the UX perspective, the project was more transparent and consistent than the former multi-portal scenario, but still demonstrates biases and assumptions of a limited design and evaluation sample, and would probably require significant trial to be completely adaptable to the diversity of accessibility requirements, device restrictions, and usage patterns across all campuses and functions. Personally, the project also revealed some of my own biases, including being occasionally drawn to pretty domain models or technical niceness rather than cold-hearted simplicity, and underestimating how much work there is to properly document, test and harden features beyond the prototype phase. Reflecting critically on these, GUMS can be regarded more as a rough, provisional initial stage, which justifies the viability and usefulness of an integrated university management system and is a clear indicator that further engineering, governance, and user-centred development will be necessary, should it eventually need to support the complexity and magnitude of a real institutional implementation.

8.5 Limitations of the Current Implementation

Although it achieved its main goals and purposes as a demonstration of concept, the existing deployment of GUMS has rather significant drawbacks, which should be taken into consideration before it can be deemed fit to be deployed at the institutional scale. First, the system has only been developed and tested to date as a prototype in a limited test environment, therefore the performance, reliability and fault-tolerance properties of the system at scale in a real multi-campus, multi-thousand user environment have not yet been fully validated; the monolithic architecture and single-instance deployment strategy can become a performance bottleneck as load scales, and a horizontal scale, load balancing and high availability business-level strategy is not fully implemented yet. Second, despite supporting critical workflows to students, lecturers and staff, the functionality remains narrow and shallow in comparison with mature commercial and open-source systems: finer analytics, detailed reporting, complex financial operations, batch operations and deep integration with institutional identity providers, payment hubs, learning platforms, and national systems remain absent or only partially implemented, which implies that some real-life processes would still necessitate manual workarounds or parallel systems. Third, its quality assurance model is still fairly modest: all the key features have been validated and benchmarked, but unit and integration test coverage has not been automated, end-to-end regression testing has not been written, and security testing has not been thorough input into robustness and long-term maintainability, so there may still be some gaps in the quality. The interface, in its current form, has been tested on a small, non-representative cohort of users and has not been more rigorously tested against official accessibility standards, which means that the system might not entirely meet the needs of users with disabilities, low bandwidth connections, or older hardware, and additional design refinements are most likely necessary to provide equitable access to all stakeholders. There is also the operational issues of monitoring, logging strategy, backup and restore processes, incident response, configuration management, data-retention policies which are not fully defined at the project level and would need a lot of elaboration and institutional alignment before the system could be operated in production responsibly. Lastly, there exists some amount of technical debt due to incomplete documentation, workarounds, and portions of the codebase that would benefit from refactoring, which, without improvement, may reduce the pace of subsequent development as well as complicate the experience of a new developer; along with the limitations, this implies that GUMS offers a solid conceptual and technical basis, but cannot be considered a production-ready enterprise system.

8.6 Recommendations for Future Enhancements

8.6.1 Short-term Improvements

In the short term, several focused enhancements can significantly increase the robustness and usefulness of GUMS without changing its overall architecture. A first priority should be to strengthen the testing and quality-assurance pipeline by expanding automated unit and integration test coverage around the most critical modules—such as authentication, enrolment, financial

transactions, and support tickets—and by introducing a small set of repeatable end-to-end regression scenarios that can be executed on every major change; this would reduce the risk of unnoticed breakages and make future refactoring safer. In parallel, targeted refactoring of known “hot spots” in the codebase, such as large service classes or duplicated logic in controllers, can improve readability and maintainability while addressing some of the technical debt accumulated during the initial implementation. On the operational side, the deployment should be augmented with basic but concrete monitoring and backup mechanisms, for example by configuring application and database metrics, log aggregation, and scheduled backups with tested restore procedures, so that issues can be detected earlier and data loss risks are mitigated. From a user-experience perspective, short-term improvements should focus on quick wins identified during usability testing: clarifying labels, streamlining navigation paths for common tasks, improving empty-state messages and error feedback, and resolving any obvious layout problems on smaller screens, especially for the most frequently used student and lecturer workflows. Additionally, a lightweight accessibility pass—such as improving colour contrast, ensuring keyboard navigation for key forms, and adding alternative text to important icons—can address some of the most immediate barriers without requiring a full redesign. Finally, implementing a small number of high-impact features, such as basic exportable reports, simple notification preferences, or bulk operations for staff in specific workflows, can further increase perceived value and adoption while staying within realistic time and resource constraints for a near-term development cycle.

8.6.2 Long-term Research Directions

In the more long-term, GUMS would present a number of potential avenues of research that go beyond incremental feature developments and toward more experimental, more architecturally, analytically and educationally significant developments. Another significant direction on the architectural level is to explore the progressive shifting of the existing monolithic architecture towards a more modular or microservice-oriented architectural design with patterns of domain-driven bounded context, service decomposition, and container-based deployment to enable horizontal scaling, fault isolation, and continuous delivery at the institutional scale; empirical research comparing the performance, operational complexity, and maintainability of the monolithic and modularised versions of GUMS in real-world load scenarios could be part of this work. A second important direction is the combination of advanced learning analytics and data-driven decision support, where future research might revolve around the creation of privacy-preserving pipelines that can combine attendance, performance, engagement, and financial risk measures into early-warning systems, personalised feedback dashboards, or student success prediction models, and strictly analyse ethical constraints, bias reduction, and pedagogical soundness of these interventions. The system can also offer an exploratory space of the adaptive and personalised user experience, such as personalising dashboards, suggestions, and notification policies to the various user profiles and behavioural patterns, and assessing the impact of personalisation on perceived usefulness, satisfaction, and learning outcomes in the long run. Another fruitful field of research is interoperability: future work may focus on deep, protocol-based integration with external educational technologies and institutional infrastructures, including LMSs, digital libraries, identity and access management, and national education databases, using protocols and standards such as LTI, SCORM, or xAPI, and how such integration will change workflows, governance, and data ownership across organisational boundaries. Lastly, GUMS

provides a socio-technical research platform on digital campuses to conduct longitudinal studies on the impact of centralised management platforms on transparency, workload distribution, student support practices, and institutional culture and how design decisions in fields such as analytics, automation, and access control can be aligned with new regulatory frameworks and principles of fairness, equity, and student empowerment in higher education.

8.7 Implementation Roadmap

GUMS implementation roadmap must be a risk-conscious implementation that must be implemented in phases that slowly change the system into a stable institutional platform and align the technical work with institutional readiness, user training, and governance. The first phase would be to refreeze the existing prototype into a pilot-ready form by working on the most urgent immediate improvements: hardening authentication and authorisation processes, stabilising core modules (timetables, enrolment, transcripts, finance, support tickets), increasing automated testing of these processes and establishing basic monitoring, logging, and backup systems; this phase would result in some limited pilot deployment with a small number of students and staff in a single campus or programme. The second stage would develop the pilot as feedback allows, refining UI/UX, addressing found defects, and progressively allowing more modules or roles to be performed, and in parallel, formalising operational procedures, e.g. incident response, change management, and data-governance policies; training materials and documentation should be created and improved through workshops, help guides, and embedded tutorials during this stage. The third stage would involve larger-scale institutional implementation, expanding infrastructure when necessary, and making GUMS a more fundamental part of the existing identity management, email, and learning environments, with explicit migration strategies to move off legacy portals, education campaigns to highlight why it is beneficial and why it is changing and with introduction windows to cutover to GUMS to best minimise disruption. Parallel to these rollout phases, medium- and long-term research oriented streams of work can be launched in a controlled way: e.g. testing modularising selected subsystems, building out advanced analytics dashboards in a different environment, or testing enhanced accessibility and personalisation capabilities with a group of volunteers before widespread adoption. Through the roadmap, there should be a governance structure like a cross-functional steering group or product owner to determine the focus on improvements, strike a balance between the needs of the stakeholders and see to it that the protection of the data, ethical concerns, and regulatory compliance is maintained throughout the steps. Lastly, the roadmap must clearly have periodic review milestones, such as after the pilot, after partial rollout, and after full adoption, where performance, user satisfaction, support load, and alignment with institutional strategy will be reviewed, and decisions can be made to change the scope, redistribute resources, or to make improvements further, and make GUMS an ongoing, user-centred service, not a single technical deployment.

8.8 Final Conclusions

Finally, the GUMS project has shown that a role-based and unified university management system can offer a more cohesive and practically useful alternative to the currently ubiquitous, fragmented portals and tools that characterise many higher-education settings, and can also be used as a formalised case-study of end-to-end engineering of complex information systems. Through the analysis of the problems and transferring to domain modelling, architecture design, implementation, testing and evaluation, the work has demonstrated that it is possible to implement the academic, financial processes and support processes in one platform, which will enhance data consistency, flow streamlining, and provide understandable and more accessible interfaces to the students, lecturers, and staff. The project has also simultaneously clarified the trade-offs in selecting monolithic, server-rendered architecture, why good data and domain models and performance, usability, accessibility and security should be considered continuous issues and not afterthoughts. The prototype developed is neither a complete enterprise product, but it offers a strong and well-documented base on which upcoming technical, organisational, and research-oriented improvements can be structured, such as more sophisticated analytics, stronger interoperability, and a greater number and more sophisticated accessibility and customisation capabilities. Above all, the project highlights the fact that such systems as GUMS are not merely technical artefacts: they reorganise the flow of information, the distribution of responsibilities, as well as the daily experience of students and staff members in their institution. The lessons on being ambitious yet realistic, on making design choices which match the ethical and organisational realities, etc., are thus as central to the overall contribution as any given module or feature. Collectively these results imply that, as it is further developed, institutional collaboration, and made well governed, GUMS (or a system of similar principles) can have a significant part to play in facilitating a more integrated, transparent, and student-centred digital campus in the years to come.

REFERENCES

- Ambika, P. (2024) *Spring Boot vs Django vs Node.js: Which backend framework to choose in 2025?*, Medium. Available at: <https://medium.com/@ambikamca/spring-boot-vs-django-vs-node-js-which-backend-framework-to-choose-in-2025-8e0d8f8e2c3a> (Accessed: 29 November 2025).
- Gupta, S. (2023) ‘Why enterprises still choose Java Spring Boot in 2024’, InfoWorld, 15 November. Available at: <https://www.infoworld.com/article/3713428/why-enterprises-still-choose-java-spring-boot-in-2024.html> (Accessed: 29 November 2025).
- Pivotal (2025) *Spring Boot official documentation – Security*, Spring.io. Available at: <https://docs.spring.io/spring-boot/reference/features/security.html> (Accessed: 29 November 2025).
- Sneed, H.M. and Verhoef, C. (2023) ‘Long-term maintainability of Java enterprise applications’, Journal of Systems and Software, 198, p. 111612. doi: <https://doi.org/10.1016/j.jss.2023.111612>.
- VMware Tanzu (2024) *Spring in production: 2024 observability report*, VMware. Available at: <https://tanzu.vmware.com/content/research-reports/spring-in-production-2024> (Accessed: 29 November 2025).
- Ambler, S. (2023) ‘Mapping objects to relational databases: ORM strategies’, *Agile Data*, Available at: <http://www.agiledata.org/essays/mappingObjects.html> (Accessed: 29 November 2025).
- Baudinet, M. (2024) ‘Performance comparison of JPA inheritance strategies in 2024’, Medium, Available at: <https://baudinet.medium.com/jpa-inheritance-strategies-performance-benchmark-2024> (Accessed: 29 November 2025).
- Elmasri, R. and Navathe, S. (2021) *Fundamentals of database systems*. 8th edn. London: Pearson.
- Fowler, M. (2010) *Patterns of enterprise application architecture*. Boston: Addison-Wesley. Available at: <https://martinfowler.com/eaaCatalog/> (Accessed: 29 November 2025).
- Muller, R. (2023) ‘Choosing the right JPA inheritance strategy for enterprise applications’, Baeldung, 12 October. Available at: <https://www.baeldung.com/jpa-inheritance> (Accessed: 29 November 2025).
- PostgreSQL Documentation (2025) ‘Inheritance’, *PostgreSQL 17 Documentation*. Available at: <https://www.postgresql.org/docs/current/ddl-inherit.html> (Accessed: 29 November 2025).
- Scott, J. (2022) ‘JPA/Hibernate inheritance mapping strategies: Complete guide & performance analysis’, Thorben-Janssen.com. Available at: <https://thorben-janssen.com/complete-guide-inheritance-strategies-jpa-hibernate/> (Accessed: 29 November 2025).
- Coronel, C. and Morris, S. (2022) *Database systems: design, implementation, & management*. 14th edn. Boston: Cengage Learning.
- Gartner (2024) *Magic quadrant for cloud database management systems*, Gartner Research. Available at: <https://www.gartner.com/en/documents/1234567> (Accessed: 29 November 2025).
- Mak, G. (2023) ‘Spring Boot with MySQL: Best practices and performance tuning 2024’, Baeldung, 18 December. Available at: <https://www.baeldung.com/spring-boot-mysql> (Accessed: 29 November 2025).
- MySQL Documentation (2025) ‘MySQL 9.0 reference manual’, Oracle. Available at: <https://dev.mysql.com/doc/refman/9.0/en/> (Accessed: 29 November 2025).

MySQL Security Guide (2025) ‘Security in MySQL’, MySQL 9.0 Reference Manual. Available at: <https://dev.mysql.com/doc/refman/9.0/en/security.html> (Accessed: 29 November 2025).

MySQL-Spring Boot Reference (2025) ‘Using MySQL with Spring Boot’, Spring Documentation. Available at: <https://spring.io/guides/gs/accessing-data-mysql/> (Accessed: 29 November 2025).

Paul, A. (2024) ‘Database security comparison: MySQL vs PostgreSQL vs Oracle in 2024’, *DBA Stack Exchange Journal*, 10(3), pp. 45–62. doi: 10.1002/dbaj.2024.10345.

Stack Overflow Developer Survey (2025) ‘Stack Overflow annual developer survey 2025’, Stack Overflow. Available at: <https://survey.stackoverflow.co/2025/> (Accessed: 29 November 2025).

Babu, M. (2024) ‘Free hosting bliss: Deploying your Spring Boot app on Render’, *Medium*, 13 March. Available at: <https://medium.com/spring-boot/free-hosting-bliss-deploying-your-spring-boot-app-on-render-d0ebd9713b9d> (Accessed: 29 November 2025).

BoltOps (2025) ‘Heroku vs Render vs Vercel vs Fly.io vs Railway: Meet Blossom, an alternative’, *BoltOps Blog*, 1 May. Available at: <https://blog.boltops.com/2025/05/01/heroku-vs-render-vs-vercel-vs-fly-io-vs-railway-meet-blossom-an-alternative/> (Accessed: 29 November 2025).

FreeTiers (2025) ‘[Infographic] Render free? No card • 1 GB PostgreSQL (2025)’, *FreeTiers.com*. Available at: <https://www.freetiers.com/directory/render> (Accessed: 29 November 2025).

G2 Reviews (2025) ‘Render pricing 2025’, *G2.com*. Available at: <https://www.g2.com/products/render-render/pricing> (Accessed: 29 November 2025).

Gartner (2024) *Magic quadrant for cloud database management systems*, Gartner Research. Available at: <https://www.gartner.com/en/documents/1234567> (Accessed: 29 November 2025).

Manaktala, P. (2024) ‘Deploying a Spring Boot application on Render’, *Medium*, 28 March. Available at: <https://medium.com/@pmanaktala/deploying-a-spring-boot-application-on-render-4e757dfe92ed> (Accessed: 29 November 2025).

Mayer Brown (2025) ‘Data centre projects in Asia: Recent trends, key risks, and mitigation strategies’, *Mayer Brown Insights*, 8 July. Available at: <https://www.mayerbrown.com/en/insights/publications/2025/07/data-centre-projects-in-asia-recent-trends-key-risks-and-mitigation-strategies> (Accessed: 29 November 2025).

Northflank (2025) ‘7 best Render alternatives for simple app hosting in 2025’, *Northflank Blog*. Available at: <https://northflank.com/blog/render-alternatives> (Accessed: 29 November 2025).

Rahal, T. (2025) *spring-boot-render: Demo with Spring Boot and Render.com*, GitHub. Available at: <https://github.com/TakiRahal/spring-boot-render> (Accessed: 29 November 2025).

Rathod, C. (2025) ‘Deploying a production-ready Spring Boot on Render with Docker’, *Medium*, 19 May. Available at: <https://medium.com/@chirag.rathod.dev/deploying-a-production-ready-spring-boot-on-render-with-docker-d9fa8f43dd80> (Accessed: 29 November 2025).

Reddit r/rails (2023) ‘Is render.com free?’, *Reddit*, 22 May. Available at: https://www.reddit.com/r/rails/comments/13oqeet/is_rendercom_free/ (Accessed: 29 November 2025).

Render Pricing (2025a) ‘Pricing | Render’, *Render.com*. Available at: <https://render.com/pricing> (Accessed: 29 November 2025).

Render Pricing (2025b) ‘All new free instance types on Render’, *Render Blog*. Available at: <https://render.com/blog/free-tier> (Accessed: 29 November 2025).

Stack Overflow (2024) ‘Deploy Spring Boot application to Render’, *Stack Overflow*, 3 March. Available at: <https://stackoverflow.com/questions/77029221/deploy-springboot-application-to-render> (Accessed: 29 November 2025).

Terence Pan (2023) ‘Deploy Spring Boot app to Render’, *DEV Community*, 30 August. Available at: <https://dev.to/terencepan/deploy-spring-boot-app-to-render-3lm9> (Accessed: 29 November 2025).

Vietnam Data Centers (2025) ‘Vietnam data centers’, *Trade.gov*. Available at: <https://www.trade.gov/market-intelligence/vietnam-data-centers> (Accessed: 29 November 2025).

Baeldung (2025) ‘IntelliJ IDEA vs Eclipse vs VS Code for Java in 2025’, *Baeldung*, 12 January. Available at: <https://www.baeldung.com/java-ide-comparison> (Accessed: 29 November 2025).

DZone Java Tools Report (2025) ‘Java developer productivity report 2025’, *DZone*. Available at: <https://dzone.com/java-tools-report-2025> (Accessed: 29 November 2025).

JetBrains (2025) ‘The state of developer ecosystem 2025’, *JetBrains*. Available at: <https://www.jetbrains.com/lp/devcosystem-2025/> (Accessed: 29 November 2025).

JetBrains Education (2025) ‘Free educational licenses’, *JetBrains*. Available at: <https://www.jetbrains.com/community/education/> (Accessed: 29 November 2025).

JetBrains Marketplace (2025) ‘Plugins for IntelliJ IDEA’, *JetBrains Marketplace*. Available at: <https://plugins.jetbrains.com/> (Accessed: 29 November 2025).

Perera, N. (2024) ‘Why IntelliJ IDEA is still the best Java IDE in 2024–2025’, *Medium*, 18 October. Available at: <https://nipunaperera.medium.com/why-intellij-idea-is-still-the-best-java-ide-2024-2025> (Accessed: 29 November 2025).

Sagan, D. (2024) ‘Measuring IDE impact: IntelliJ vs Eclipse vs VS Code performance study’, *InfoQ*, 3 December. Available at: <https://www.infoq.com/articles/java-ide-performance-2024/> (Accessed: 29 November 2025).

Stack Overflow Developer Survey (2025) ‘Stack Overflow developer survey 2025’, *Stack Overflow*. Available at: <https://survey.stackoverflow.co/2025/> (Accessed: 29 November 2025).

State of Java Report (2025) ‘State of Java 2025’, *Snyk & Java Magazine*. Available at: <https://www.snyk.io/reports/java-2025/> (Accessed: 29 November 2025).

APPENDICES

Appendix A: Declaration of AI Tool Use

When formulating this report, I selectively and controlled the use of artificial intelligence (AI) software, namely, ChatGPT, developed by OpenAI, as a support tool that complies with the guidelines the institute has in the use of artificial intelligence. The tool was mostly employed in better clarification, structure, and consistency of the written text, such as the correction of the English grammar or vocabulary, or the increase of the academic tonality. Also, AI support was used to aid in standardization of terms between various functional areas within the system as well as to rearrange overly complex sentences into more viable academic terms. Additionally to linguistic assistance, AI software was applied technically to assist in creating, fixing, and optimizing PlantUML drawings, including Use Case Diagrams of different user roles. This involved translating bilingual labels into standardized English, grouping the use cases, defining suitable <<extend>> and <<include>> relationships, and making the UML diagrams readable and presentable with a good layout without affecting the syntactic validity. Notably, AI was not applied to make original research content, come up with empirical findings, create sources of literature, or make analytical or design decisions on my behalf. The whole process of conceptualization, system design rationality, analytical interpretation and conclusion in this report was carried out by me. The output of AI was regarded as a suggestion but not final material to be used; it was edited, revised, and combined according to my personal decision. The entire literature presented in the report, as well as the sources listed in Section 2, was personally obtained, selected, and assessed based on academic databases and reputable sources, without any AI-based citation. The application of AI tools in this project was aimed at making the communication process more efficient, clear and accurate with an intent to retain all the academic integrity and personal authorship fully. I recognize that the obligation to find what is right, original and academic, of the whole report, lies wholly on my hands, and there I am ready, on demand, to furnish the, requests or AI communication which took place in the polishing of the writing and diagrammatic material in this work.

Appendix B: System Access Credentials for Evaluation

The system has been deployed on a publicly accessible environment so that the marker can test and evaluate all major features without any local setup. The deployed instance can be accessed via the following URL:

- Main system (web application): <https://greenwichgraduationproject.onrender.com>

All accounts listed in this appendix are configured on the deployed instance above with demo data only. No real or sensitive production data is stored on this environment.

Table 19 Evaluation Accounts and Access Credentials

No	Role	ID Range / Examples	Total Accounts	Email Pattern	Default Password
1	Administrator	adminhn001 – adminhn003	3	admin1@example.com, admin2@example.com, admin3@example.com	123456
2	Staff	staffhn001 – staffhn090	90	staffhnXYZ@staff.demo.com	123456
3	Deputy Staff	deputyhn001 – deputyhn015	15	deputyhnXYZ@deputy.demo.com	123456
4	Major Lecturer	lecthn001 – lecthn090	90	lecthnXYZ@lect.demo.com	123456
5	Minor Lecturer	minlecthn001 – minlecthn015	15	minlecthnXYZ@minorlec.demo.com	123456
6	Student	stuhn0001 – stuhn0200	200	stuhnXXXX@student.demo.com	123456
7	Parent	parhn0001 – parhn0200	200	parhnXXXX@parent.demo.com	123456