VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Software Engineering

Assignment 3:

Tutor Support System at HCMUT - Phase 3

Advisor(s):	Trần Trương Tuấn Phát	
Group:	7	
Student(s):	Võ Hoàng Sơn	2353057
	Trần Hải Đăng	2352254
	Lâm Minh Tùng Quân	2352991
	Phan Quốc Đại Sơn	2353053
	Trương Gia Kỳ Nam	2352787
	Nguyễn Hữu Phúc	2352938
	Châu Anh Nhật	2352856

HO CHI MINH CITY, DECEMBER 2025

Contents

Name	Task	% Complete
Võ Hoàng Sơn	Design activity diagram for Tutor, Department Chair, Program Admin. Design sequence diagram for Tutor Course Feedback. Design state chart of Assignment Lifecycle	100%
Trần Hải Đăng	Design activity diagram for Student, Coordinator. Design sequence diagram for Handling feedback and Reports, Searching and Importing Libary References. Design statechart for Session Lifecycle	100%
Lâm Minh Tùng Quân	Design sequence diagram for Deadline and Assignemenet management, Session Management, Session request handling, Chat and Notification System, Tutor Evaluation and Feedback.	100%
Phan Quốc Đại Sơn	Design sequence diagram for Sign In process, Course access and Resource Retrieval, Course Matching and Assignment Process, Learning Material Management, AI matching module.	100%
Trương Gia Kỳ Nam	Design user interface - Login, Calendar, Course Dashboard and Integrated chat, deployment view and implementation view.	100%
Nguyễn Hữu Phúc	Design use cases, user interface - Course detail and submission management, My Course overview and Report Module, Tutor Registration and Matching Module	100%
Châu Anh Nhật	Design use cases, UI figma for user interface - System Dashboard, HCMUT library, Evaluation and report module, Code evaluation for Students, tutors.	100%

1 Introduction

The implementation of the Tutor Support System at Ho Chi Minh University of Technology (HCMUT) is driven by the increasing demand to improve academic support for students in a competitive higher education environment. As the curriculum becomes more complex, many students struggle with adapting to university-level coursework or developing essential academic skills. Therefore, establishing a structured tutor and mentor program is essential to support students' learning and personal growth.

This system centralizes the management of tutoring and mentoring activities, ensuring efficient resource usage while addressing each student's needs. By integrating processes such as tutor–student matching, classroom workspace, session scheduling, feedback collection, and performance tracking, the system reduces administrative workload and enhances the tutoring experience for all participants.

2 Project Detail Specification

2.1 Context Description

Ho Chi Minh University of Technology (HCMUT) has a diverse academic curriculum comprising not only challenging foundation courses, such as calculus and physics, but also demanding specialized subjects for upper-year students. Many first-year students struggle to achieve satisfactory results due to the gap between high school and university teaching methods, while seniors encounter difficulties with complex major-specific modules. To address this, HCMUT introduces the **Tutor Support System** — a digital platform that connects students with tutors in their courses, tracks session progress, and facilitates personalized academic guidance. The system also provides coordinators with analytical tools to ensure quality improvement and efficient management.
The system serves multiple stakeholders:

2.2 Stakeholders and User story

There are three Stakeholders in the system :

- **Students**: Participants who register for the program to receive academic support and skill development. They can be matched with tutors, attend in-person or online

sessions, access learning resources, and evaluate session quality.

- **Tutors**: Faculty members or high-achieving students responsible for setting availability, organizing advising sessions, and sharing official learning materials. They also track and record students' progress to ensure effective academic guidance.
- **Coordinator**: The operational manager of the Tutor program who oversees the tutor-student matching process, monitors program effectiveness through analytical reports, and ensures optimal resource allocation for academic support.

It is important to note that the roles of **Student** and **Tutor** are flexible and context-dependent rather than mutually exclusive. A single user account can simultaneously hold both roles; for instance, a high-achieving student may serve as a Tutor for a course in which they have demonstrated proficiency, while remaining a Student in other courses they are currently attending. The system manages these permissions at the course level, allowing users to seamlessly switch between learning and teaching contexts within the same account. The user story will be introduced in the section below.

2.2.1 Student

- To register for tutoring programs based on personal academic needs and support requirements.
- To receive the personalized academic support through learning materials and supplementary exercises shared by tutors.
- To receive automated notifications for upcoming schedules and participate in tutoring sessions (in-person or online).
- To directly access and retrieve official textbooks and references via the integrated HCMUT_LIBRARY system.
- To provide feedback on course quality and evaluate tutor performance after completion.

2.2.2 Tutor

- To register for the program by specifying preferred teaching subjects and professional capabilities.

- To upload learning materials and create assignments for students within the course workspace.
- To organize teaching sessions (online or in-person) and record meeting minutes or session notes.
- To track and record the learning progress of assigned students throughout the course.
- To access the HCMUT_LIBRARY to search for and share official academic resources.
- To provide feedback on course quality and student performance upon course completion.

2.2.3 Coordinator

- To manage registration flows by viewing capacity overviews and student waiting lists, and to facilitate course assignment via manual selection or AI matching monitoring.
- To oversee course quality by monitoring student feedback and reviews, and handling complaints to ensure effective academic support.
- To aggregate data and export operational reports with specific metrics for resource optimization.

2.2.4 Integrated system

- **HCMUT_SSO** : Authenticate users and grant unified login sessions
- **HCMUT_DATACORE** : Synchronize personal data and roles to assign permissions in the system
- **HCMUT_LIBRARY** : Grant permission to access/share learning materials for sessions.
- **Notification Service** : Send automatic notifications for class, session, and general announcements.
- **Chat Service**: Enable real-time communication and file sharing between users.
- **AI Matching Service** : Recommend optimal pairings based on profile and needs

2.3 Objectives

The Tutor Support System aims to enhance the effectiveness, transparency, and scalability of academic tutoring and mentoring at Ho Chi Minh University of Technology (HCMUT). The system provides integrated tools for students, tutors, and coordinators, ensuring efficient communication, centralized data management, and continuous improvement of academic support.

- **Facilitate Effective Tutoring:** Simplify tutor–student registration, course assignment, and session management to improve accessibility and reduce administrative workload.
- **Provide Structured Academic Support:** Offer students a centralized platform to register for tutoring programs, request sessions, and monitor learning progress throughout their courses.
- **Empower Tutors:** Allow tutors to manage both online and in-class sessions, schedule and record attendance, share materials, and provide feedback to students.
- **Optimize Coordination:** Equip coordinators with tools to assign tutors, manage session plans, detect scheduling conflicts, and monitor course-level performance.
- **Integrate Academic Resources:** Connect the system with HCMUT_LIBRARY to provide shared learning materials and course references accessible to both students and tutors.
- **Ensure Data Integration and Security:** Utilize centralized authentication via HCMUT_SSO and maintain consistent user and course data synchronization through HCMUT_DATACORE.
- **Promote Continuous Improvement:** Enable evaluation and feedback mechanisms for both tutors and students to assess session quality and enhance program outcomes.
- **Scalability and Modernization:** Design the system to be extensible for future improvements, including AI-based tutor matching, data-driven analytics, and personalized learning support.
- **Promote Accountability and Transparency:** Maintain ethical use of academic data and AI tools, ensuring fairness, privacy, and integrity across all system operations.

In summary, the system aims to create a unified, data-driven platform that supports the entire tutoring process—from student registration to tutor assignment, session tracking, and performance evaluation—while maintaining transparency, scalability, and integration with HCMUT's digital infrastructure.

2.4 Scope

The system will handle:

- Tutor and student profile management, including subjects, experience, and learning goals.
- Student registration and tutor assignment (manual or AI-driven tutor matching).
- Support for non-academic mentoring programs.
- Provision of personalized academic support, content delivery, and supplementary exercises within the virtual classroom.
- Scheduling, cancellation, and notifications for tutoring sessions (online or in-class).
- Record keeping for session details and attendance.
- Feedback collection from both students and tutors after course.
- Report generation and analytics for coordinators.
- Secure access and single sign-on via HCMUT_SSO.
- Data synchronization with HCMUT_DATACORE and integration with HCMUT_LIBRARY for course resources.

Out of scope:

- Personalized learning support with AI integration.
- Community interaction features such as tutor discussion boards.

3 Functional Requirements:

3.1 Use case whole System

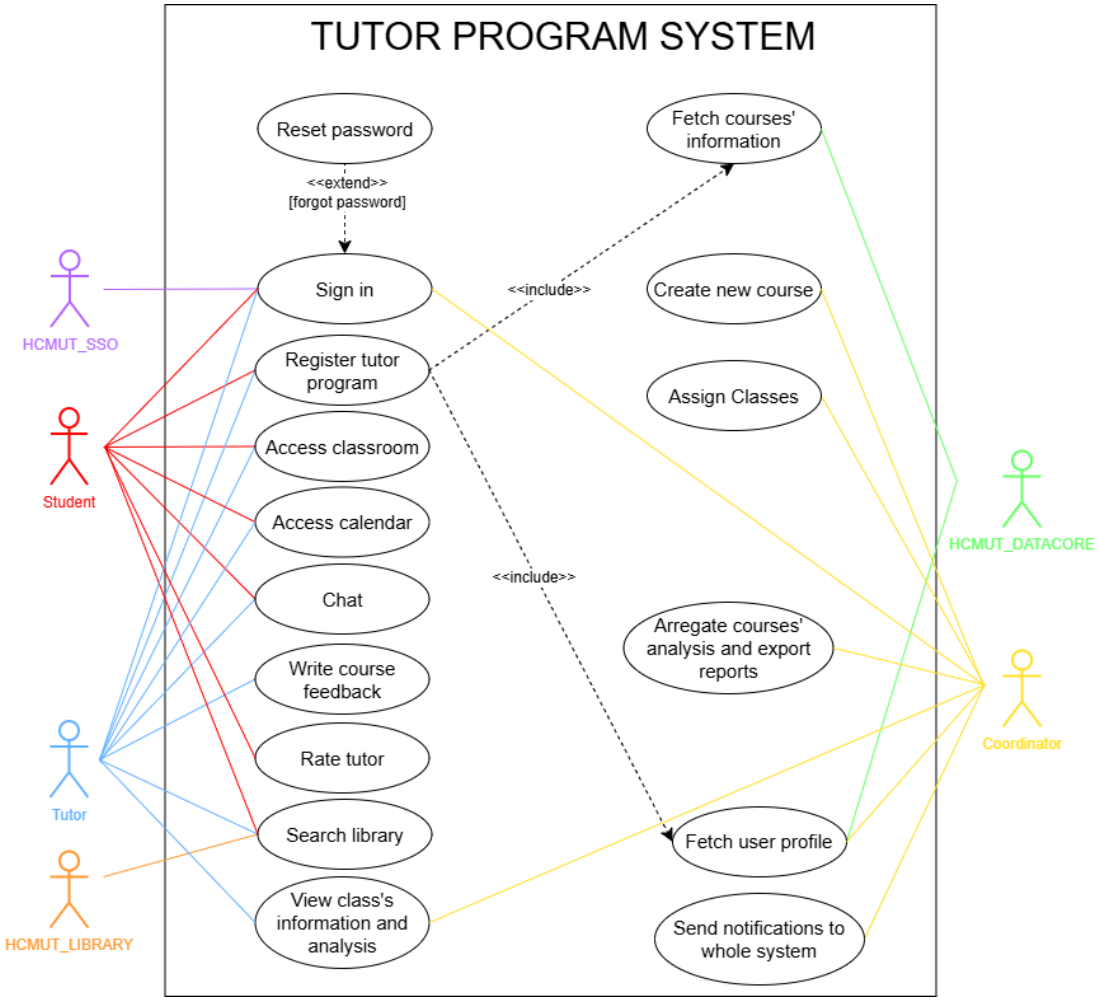


Figure 3.1: Whole system Use case

3.2 Use-case details

3.2.1 Sign in

Name	Sign in
Actors	Student, Tutor, Coordinator; HCMUT_SSO (External Service)

Description	Users log into the Tutor Program System using the HCMUT_SSO authentication service. The system identifies each user's role and loads the correct dashboard and permissions.
Preconditions	The user has a valid HCMUT_SSO account and the service is available.
Trigger	User accesses to the Tutor Program website.
Normal Flow	1. User enters their credentials on the sign-in page. 2. User clicks on "Đăng nhập" 3. HCMUT_SSO verifies the credentials and returns an authentication token. 4. System receives the token, identifies the user's role, and opens the corresponding dashboard.
Alternative Flow	A1) If a valid SSO token already exists, the system signs in automatically. A2) The user ticks "Nhớ thiết bị này" on the sign-in page. The system stores a cookie about your sign-in information to your device for next time. A3) The user clicks "Quên mật khẩu?" on the sign-in page. The system redirects them to the HCMUT_SSO password recovery service.
Exceptional Flow	E1) Invalid username or password format — the system displays format error. E2) Invalid credentials — SSO denies access and shows "Sai tên đăng nhập hoặc mật khẩu."
Postconditions	User is authenticated and the system shows the main dashboard based on their role.

3.2.2 Register Tutor Program

Name	Register Tutor Program
Actors	Student, Tutor
Description	Students and tutors register for the Tutor Program.

Preconditions	1. User is signed in and the registration period is open. 2. The system has successfully fetched course data from HCMUT_DATACORE and synchronized any additional courses created by the Coordinator.
Trigger	User selects "Đăng ký môn học" from the sidebar.
Normal Flow	1. User opens registration page. 2. User chooses in subject, language, method(online only or both online and offline sessions), selects prefered location if offline, some optional special requirements, students can declare their aim while tutors can declare their records. 3. System saves registration draft every minute. 4. User submits the form. 5. System validates and saves the registration, clear the draft, redirects user to registration history.
Alternative Flow	A1) A registration draft exists and the page load the draft. A2) User clicks "Hủy bỏ" to clear the draft registration. A3) User deletes registration in history.
Exceptional Flow	E1) Registration period closed. System shows "Registration closed".
Postconditions	Registration is saved and visible for coordinator to assign classes.

3.2.3 Create New Course

Name	Create New Course
Actors	Coordinator
Description	The Coordinator manually creates a new course entity (subject) in the system. This allows the addition of non-academic or supplementary courses not available in HCMUT_DATACORE for registration. The Coordinator can also view and manage (delete) these manually created courses.

Preconditions	1. Coordinator is signed in successfully. 2. The "Course Management" module is accessible.
Trigger	Coordinator selects "Quản lý môn học" (Course Management) from the sidebar.
Normal Flow	1. Coordinator accesses the Course Management page. 2. System displays a list (history) of courses previously created by the Coordinator. 3. Coordinator clicks on the "Tạo môn học mới" (Create New Course) button. 4. System displays the course creation form. 5. Coordinator inputs the required information: Course Name, Course Code (Must be unique), select Language, description 6. Coordinator clicks "Tạo môn học" (Create) to submit. 7. System validates the course code uniqueness and required fields. 8. System saves the new course, adds it to the available course pool, and redirects the Coordinator back to the list view with a success message.
Alternative Flow	A1) Coordinator selects a course to delete and confirms; the system deletes it if no active classes exist.
Exceptional Flow	E1) The entered course code already exists. System shows error "Mã môn học đã tồn tại". E2) Coordinator tries to delete a course with active classes or registrations. System shows error message.
Postconditions	The new course is added to the system database and becomes available in the "Register Tutor Program" list for students and tutors to select.

3.2.4 Assign Classes

Name	Assign Classes
Actors	Coordinator

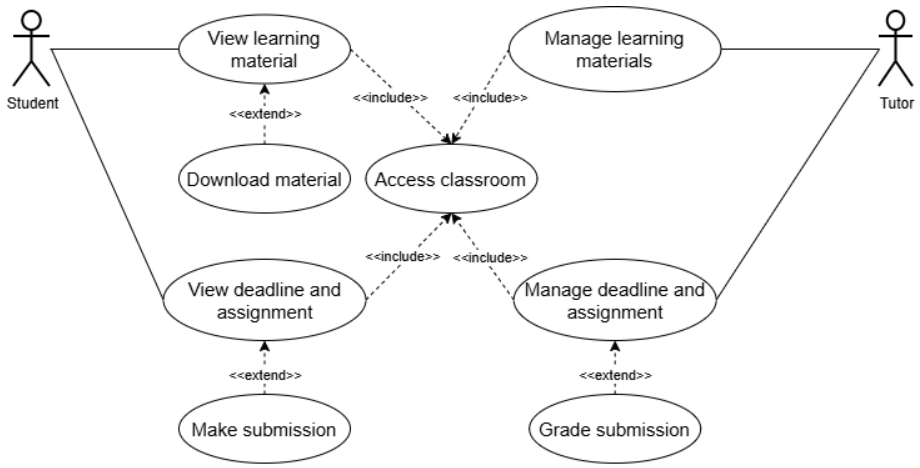
Description	Coordinator assigns tutors and students to classes (a group of students with 1 tutor) based on registration data such as subject, language, method, and some special requirements. The system can also use AI-based matching to optimize assigning tutor to classes.
Preconditions	Coordinator is signed in, tutors and students have completed registration.
Trigger	Coordinator selects "Quản lý đăng ký" from the sidebar.
Normal Flow	1. Coordinator opens assignment page. 2. System lists all tutors' and students' registrations. 3. Coordinator can manually assign students to tutor's class. 4. System suggests class assignments using AI matching. 5. Coordinator confirms assignments. 6. System saves classes' information and notifies tutors and students for new classes.
Alternative Flow	E1) Coordinator can search or filter. E2) Remain registrations not assign to class, coordinator decides to assign manually or confirm and send reject notification to them.
Postconditions	Class assignments are stored and notifications sent.

3.2.5 Access Classroom

Name	Access Classroom
Actors	Student, Tutor
Description	Users enter the online classroom to view materials, assignments. Tutors can upload content and manage assignments.
Preconditions	User is signed in and enrolled in a class.
Trigger	User selects a class from dashboard.

Normal Flow	1. System displays list of enrolled classes. 2. User opens a class. 3. System displays classroom page with materials and assignments.
Postconditions	Classroom accessed successfully.

3.2.6 Manage Learning Materials



Name	Manage Learning Materials
Actors	Tutor
Description	Tutors manage learning references and course materials after entering the classroom. Tutors can upload, update, or delete materials such as lecture notes or reading lists, then students can view and download them. The goal is to keep all course materials organized and easily accessible.
Preconditions	- User is signed in and has already accessed the classroom. - The course repository for learning materials exists.
Trigger	Tutor clicks "Chỉnh sửa" from the classroom menu.

Normal Flow	1. The system displays a list of existing learning materials and references for the course in editor mode. 2. Tutor can upload new reference files by creating a new tab with the type of material or updates/deletes existing ones. 3. System validates the file format and saves it. 4. Students can view or download available materials.
Exceptional Flow	E1) File is corrupted or exceeds size limit, shows alert
Postconditions	New or updated materials are stored successfully.

3.2.7 Manage Deadline and Assignment

Name	Manage Deadline and Assignment
Actors	Tutor
Description	Tutor manages course assignments by creating, editing, or deleting tasks and setting submission deadlines. Tutor can also check student submissions and give grades or feedback.
Preconditions	- User is signed in and has already accessed the classroom. - The course repository for learning materials exists.
Trigger	Tutor clicks "Chỉnh sửa" from the classroom menu.
Normal Flow	1. Tutor views classroom page in editor mode. 2. Tutor creates a new tab with type assignment, fills title, description, and deadline. 3. System saves the new assignment. 4. Students are notified of the new assignment. 5. Tutor later reviews submissions upon clickinig "Xem bài nộp" on a assignment tab and gives grades or comments.
Alternative Flow	A1) Tutor edits an existing assignment or extends its deadline.
Exceptional Flow	E1) Missing required fields when creating new assignment, alerts "Please fill in all fields."

Postconditions	Assignment information is saved and students are notified of all updates.

3.2.8 Make Submission

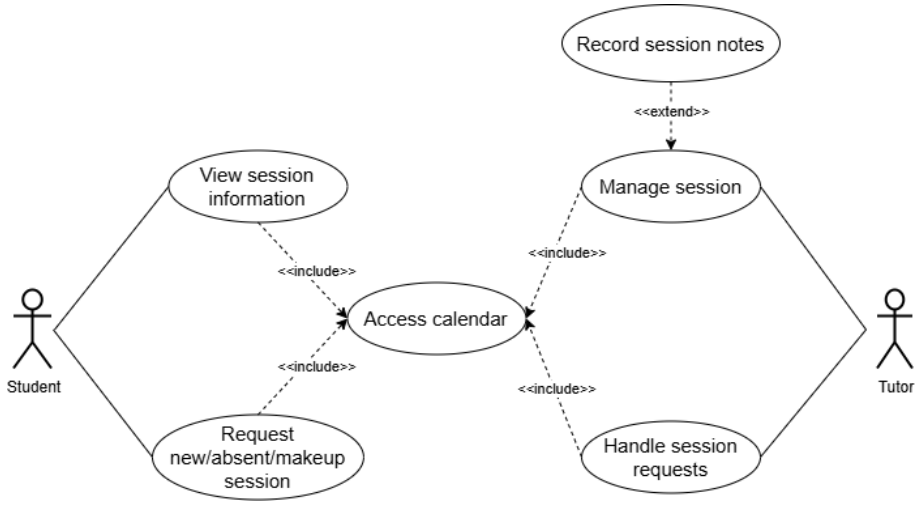
Name	Make Submission
Actors	Student
Description	Student checks active assignments, views details and deadlines, submits files before the due date, and later reviews tutor feedback and grades.
Preconditions	Student is in classroom page and at least one assignment is available.
Trigger	Student selects "Thêm bài nộp" on a assignment tab.
Normal Flow	3. Student selects files to submit before the due date. 4. System saves the submission and confirms completion. 5. Student later checks feedback and grades.
Alternative Flow	A1) Student replaces an earlier submission before the deadline.
Exceptional Flow	E1) Submission after deadline, alerts "Deadline passed." E2) Invalid file type, alerts "Unsupported file format."
Postconditions	Submission is stored and marked as completed. Grades and feedback will appear after tutor review.

3.2.9 Access Calendar

Name	Access Calendar
Actors	Student, Tutor
Description	Users access their personal calendar to view their schedule, including upcoming and past learning sessions. This interface serves as the central hub for managing time, tracking session status, and navigating to specific session details.

Preconditions	User is signed in.
Trigger	User selects "Lịch học" from the navigation sidebar.
Normal Flow	1. User clicks on the Calendar tab in the sidebar. 2. System retrieves the user's scheduled sessions (time, course, status) from the database. 3. System displays the calendar interface (defaulting to the current month or week) populated with session blocks.
Postconditions	The personal calendar is displayed, allowing the user to view the schedule or select specific sessions for more details.

3.2.10 Manage Session



Name	Manage Session
Actors	Tutor
Description	Tutor creates, updates, or deletes sessions for the course. Each session contains topic, time, and meeting details. The system checks for schedule conflicts and alerts tutors when updates are saved.
Preconditions	Tutor is assigned to at least one course and in calendar page.
Trigger	Tutor selects "Tạo buổi học".

Normal Flow	1. Tutor opens the create session page. 2. Tutor creates a new session by entering topic, date, time, and choose type: online or offline with some descriptions. 3. System saves the session and alerts "New session created." 4. Tutor views the list of existing sessions from calendar.
Alternative Flow	A1) Tutor may edit or delete existing sessions.
Exceptional Flow	E1) Invalid or overlapping schedule, alerts "Time conflict detected." E2) Tutor tries to delete a past session, alerts "Cannot delete completed session."
Postconditions	Sessions are stored and visible to both tutor and students. Notifications are sent automatically for new or updated sessions.

3.2.11 Handle Session Requests

Name	Handle Session Requests
Actors	Student, Tutor
Description	Students request new sessions or absence for a existing session. Tutors review these requests, approve or reject them, if approve then the tutor need to create or edit the session, then the system updates the course schedule accordingly.
Preconditions	Student and tutor belong to the same course.
Trigger	Student clicks on "Yêu cầu buổi học" in the calendar page.

Normal Flow	1. Student opens the request page. 2. Student chooses "Tạo yêu cầu mới". 3. Student selects course and type of request, fills in details such as title, reason and session mode (online or classroom). 4. Student sends request upon clicking "Gửi yêu cầu". 5. Request is sent to the tutor. 6. Tutor receives notification of the new request. 7. Tutor opens the request list and reviews details. 8. Tutor approves or rejects the request. 9. If approved, tutor may create new session or record student's absence the system updates the schedule and alerts both tutor and student. 10. If rejected, student is notified with the tutor's reason.
Alternative Flow	A1) Student withdraws a request before tutor reviews it.
Postconditions	Approved or rejected requests are recorded. The course schedule and notifications are updated accordingly.

3.2.12 Record Session Notes

Name	Record Session Notes
Actors	Tutor
Description	Tutor records notes and attendance for each completed session. Notes include learning topics, student participation, and comments. Students can later view these notes to track their progress.
Preconditions	The session has been completed and exists in the course schedule. Tutor has permission to edit notes.
Trigger	Tutor selects "Chỉnh sửa" on the session after class completion.

Normal Flow	1. Tutor opens a completed session. 3. Tutor checks attendance and upload meeting minutes for the session. 4. System saves the notes and alerts "Notes saved successfully." 5. Student receives a notification that session notes are available. 6. Student views notes and attendance in the calendar page.
Alternative Flow	A1) Tutor attaches supporting files such as slides or images. A2) Tutor updates notes later and system replaces the old version.
Exceptional Flow	E1) Tutor leaves required fields blank or attendance check, alerts "Incomplete note details".
Postconditions	Session notes are stored and visible to both tutor and student. Data is used for progress tracking and report analysis.

3.2.13 Chat

Name	Chat
Actors	User
Description	Users exchange messages or share files through the in-system chat. The chat allows real-time communication and keeps message history within each class. Group chat is supported.
Preconditions	Both tutor and student have joined the same class and group chat feature is enabled for that class. Users can also chat individually between each other.
Trigger	User opens the chat tab via clicking chat icon.

Normal Flow	1. User opens the chat window. 2. User chooses a chat. 3. User types and sends a message. 4. Message is delivered instantly to the recipient and notify them. 5. Receiver reads and replies through the same channel. 6. Messages are saved in the chat history for future reference.
Alternative Flow	A1) User attaches a file or image in the chat.
Exceptional Flow	E1) Message delivery fails due to network issue, alerts "Tin chưa được gửi"
Postconditions	Messages are exchanged successfully and stored in chat history. Users can reopen the chat to view past conversations at any time.

3.2.14 Rating Tutor

Name	Rating Tutor
Actors	Student
Description	After completing a course, the student rates the tutor's performance using a star scale and optional comments. These ratings are collected for program quality analysis and for coordinators to review tutor performance.
Preconditions	The course is completed and the student has participated in a class of that course.
Trigger	Student selects "Đánh giá" from the classroom page.
Normal Flow	1. Student opens the rating interface. 2. Student selects a star rating (from 1 to 5) for many criteria. 3. Student optionally writes a comment. 4. System automatically saves the rating draft every minute. 5. Student submits the rating. 6. System saves the rating and alerts "Đánh giá thành công".

Alternative Flow	A1) A draft exists: System loads the previously saved rating draft. A2) Student edits rating within a short time window and re-submits.
Exceptional Flow	E1) Students don't complete rating star when submit, alerts "Vui lòng đánh giá hết các tiêu chí".
Postconditions	Rating and comment are stored in the system and linked to the tutor's performance record. Coordinator can view these results in feedback reports.

3.2.15 Write Class Feedback

Name	Write Class Feedback
Actors	Tutor
Description	After finishing a course, the tutor writes a short feedback report including overall class performance, student engagement, and any teaching difficulties. This helps coordinators review course quality and make improvements.
Preconditions	The course is completed and the tutor is assigned to a class of that course.
Trigger	Tutor selects "Đánh giá" from classroom page.
Normal Flow	1. Tutor opens the feedback form for a completed course. 2. Tutor enters comments about class results and student performance. 3. Tutor may add suggestions for improvement. 4. System automatically saves the feedback draft every minute. 5. Tutor submits the feedback. 6. System saves it and alerts "Feedback submitted."
Alternative Flow	A1) A draft exists: System loads the previously saved feedback draft.

Exceptional Flow	E1) Tutors don't complete giving feedbacks when submit, alerts "Vui lòng điền đầy đủ thông tin".
Postconditions	Tutor feedback is stored and becomes visible to the coordinator in the course report section.

3.2.16 Search Library

Name	Search Library
Actors	Student, Tutor, HCMUT_LIBRARY(external service)
Description	Users search for academic references, research materials, or documents from the integrated HCMUT digital library, then view, download or share the resource.
Preconditions	User is signed in and the HCMUT Library service is available.
Trigger	User selects "Thư viện" from sidebar.
Normal Flow	1. User opens the library search interface. 2. User enters a keyword or title. 3. System returns matching results with brief descriptions. 4. User clicks a reference tab to view, download, or share document.
Exceptional Flow	E1) No results found, alerts "Không tìm thấy kết quả"
Postconditions	Selected materials are accessed successfully and logged in the system.

3.2.17 Fetch User Profile

Name	Fetch User Profile
Actors	Coordinator, HCMUT_DATACORE (External Service)

Description	The Coordinator retrieves detailed profile information of a specific user (student or tutor) directly from the university's centralized database (HCMUT_DATACORE). This ensures that the academic and personal data used for course assignment and management is accurate and up-to-date.
Preconditions	1. Coordinator is successfully signed in. 2. Connection to HCMUT_DATACORE service is active.
Trigger	Coordinator selects a user profile to view or searches for a specific user ID.
Normal Flow	1. Coordinator requests to view a user's details (by clicking on a user in a list or entering a User ID). 2. System sends a data request containing the User ID to the HCMUT_DATACORE adapter. 3. HCMUT_DATACORE processes the request and returns the user's profile data (Full Name, Student/Staff ID, Faculty, Academic Status, etc.). 4. System maps the external data to the local user model. 5. System displays the detailed user profile to the Coordinator.
Alternative Flow	A1) System automatically updates the local record if it is outdated compared to DATACORE.
Exceptional Flow	E1) HCMUT_DATACORE returns "Not Found"; system displays error "Không tìm thấy dữ liệu". E2) Connection to HCMUT_DATACORE fails; system displays connection error message.
Postconditions	The user's profile information is successfully retrieved and displayed for the Coordinator.

3.2.18 View Class's Information and Analysis

Name	View Class's Information and Analysis
Actors	Coordinator, Tutor

Description	Users access a comprehensive dashboard for a specific class to monitor operational performance, including aggregated metrics, course details, and individual student progress.
Preconditions	User is signed in and authorized to view the specific class.
Trigger	User selects the "Tổng quan" (Overview) tab from the specific classroom interface.
Normal Flow	1. User accesses the specific class workspace. 2. User clicks on the "Tổng quan" tab. 3. System aggregates operational data including session logs, assignment submissions, and attendance records. 4. System calculates summary metrics: number of completed sessions, assignment submission rates, and attendance percentages. 5. System displays the dashboard containing class metrics, course info, tutor details, and student progress/attendance list.
Exceptional Flow	E1) System fails to calculate metrics; displays "Data unavailable" message.
Postconditions	The class performance data and student progress are displayed.

3.2.19 Arregate courses' analysis and export reports

Name	Arregate courses' analysis and export reports
Actors	Coordinator
Description	Coordinator reviews and combines data from multiple classes of a course under their management. The system aggregates information such as tutor performance, student participation, and course results for summary reports.
Preconditions	All course data and feedback are available in the system. Coordinator has access rights to the target courses.
Trigger	Coordinator selects "Thống kê" from sidebar.

Normal Flow	1. System lists all courses managed by the coordinator. 2. Coordinator selects a course. 3. Coordinator reviews summary metrics and trends. 4. Coordinator saves or exports the aggregated report.
Alternative Flow	A1) Coordinator filters courses by semester or department.
Exceptional Flow	E1) Export error, alerts "Failed to export report."
Postconditions	Aggregated data and reports are stored and ready for department analysis or submission to the Department Chair.

3.2.20 Send Notification to Whole System

Name	Send Notification to Whole System
Actors	Program Administrator
Description	Administrator sends announcements or important notifications to users such as tutors, students, or coordinators. Notifications can include event reminders, policy updates, or maintenance alerts.
Preconditions	Administrator is signed in and notification service is active.
Trigger	Administrator selects “Send Notification” from the dashboard.
Normal Flow	1. Administrator opens the notification interface. 2. Administrator writes the message content. 3. Administrator selects target users or groups. 4. Administrator confirms and sends the message. 5. System delivers the message and alerts "Notification sent."
Alternative Flow	A1) Administrator schedules a notification to be sent later. A2) Administrator chooses only one group, such as “Tutors” or “Students.”
Exceptional Flow	E1) Notification service error, alerts "Unable to send, please retry."

Postconditions	Notification is delivered to all selected users and stored in the system log for tracking.

3.3 Non-interactive Functional Requirements

The following requirements describe system functions that operate automatically without direct user interaction. These ensure smooth background operations, data consistency, and reliable communication across all user roles.

- The system shall automatically synchronize user authentication and profile data with **HCMUT_SSO** and **HCMUT_DATACORE** to maintain consistency across institutional systems.
- The system shall automatically deliver notifications and reminders to users regarding upcoming sessions, cancellations, or schedule adjustments.
- The system shall periodically back up essential data, including session records, feedback, and reports, to ensure data integrity and recovery in case of failure.
- The system shall record and maintain logs for all critical operations—such as sign-in, registration, assignment, and deletion—to support auditing and accountability.
- The system shall automatically update tutor and student availability based on completed or cancelled sessions, ensuring accurate scheduling data.
- The system shall generate and distribute periodic performance and participation reports to coordinators and department chairs.
- The system shall continuously monitor tutor capacity and issue alerts to coordinators when a class approaches or exceeds its enrollment limit.
- The system shall automatically close the registration portal and update the program status to 'Closed' precisely when the configured deadline is reached, ensuring data integrity for the matching process.
- The system shall automatically archive completed classes and sessions at the end of each semester to maintain system performance while preserving historical data for analytics.
- The system shall maintain automated integration with **HCMUT_LIBRARY** services to provide seamless access to shared learning materials.

4 Non-Functional Requirements

- **Security & Authentication:** The system must enforce strict authentication via **HCMUT_SSO** and implement Role-Based Access Control (RBAC) to ensure users only access data authorized for their specific roles (Student, Tutor, Coordinator).
- **Data Privacy:** All sensitive information (passwords, personal identification) must be encrypted both in transit (SSL/TLS) and at rest to comply with university data protection standards.
- **Performance:** The system should support a high volume of concurrent users (e.g., during course registration periods) with a response time of less than 3 seconds for standard transactions.
- **Availability:** The system ensures 99.9% uptime during critical academic periods (e.g., registration weeks, exam seasons) to support continuous learning activities.
- **Maintainability & Scalability:** The architecture must follow modular design principles (e.g., Layered Architecture) to facilitate easy maintenance, debugging, and future expansion like AI-matching integration.
- **Usability:** The user interface must be intuitive, responsive across devices (Desktop, Tablet, Mobile), and support bilingual display (Vietnamese/English) to accommodate the diverse academic community.

5 User Interface (UI) Design: Mockup

5.1 Overview

The User Interface (UI) mockup of the **Tutor Support System (TSS)** is designed to provide an intuitive, accessible, and visually consistent experience for all user roles, including Students, Tutors, Coordinators, Department Chairs, and Program Administrators. Following the system objectives defined in Phase 1, the mockup emphasizes functionality, clarity, and integration with HCMUT's digital infrastructure such as *HCMUT_SSO*, *HCMUT_DATACORE*, and *HCMUT_LIBRARY*.

Developed with a human-centered design approach, the mockup aims to support users in accomplishing academic and administrative tasks efficiently while maintaining a familiar and cohesive visual language aligned with HCMUT's identity. Each UI element is crafted to ensure smooth task flows, reduce cognitive load, and enhance usability across both desktop and mobile devices.

5.2 Key UI screens

5.2.1 Login Interface – HCMUT SSO Authentication

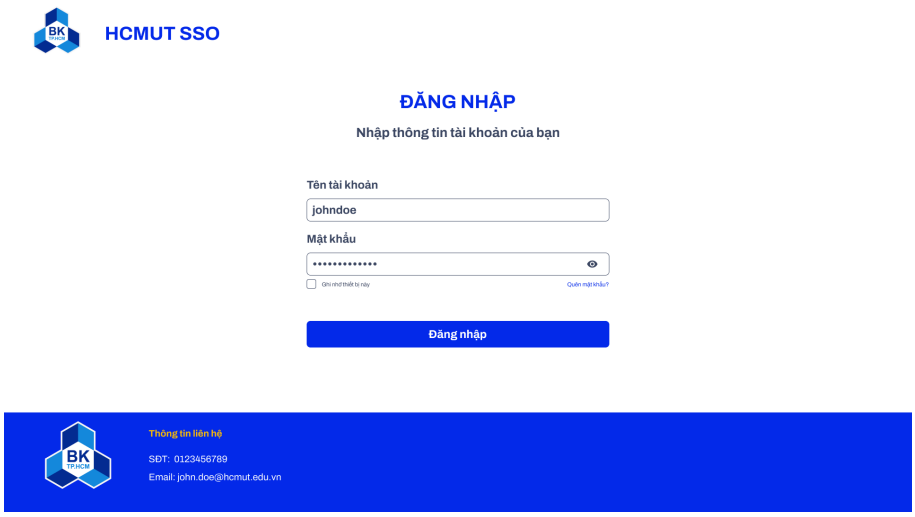


Figure 5.1: Login Interface – Standard State

Purpose: The Login Interface serves as the secure entry point for all user roles within the Tutor Support System (TSS). It integrates with the *HCMUT_SSO* service to ensure unified authentication and centralized credential management across the university's digital ecosystem.

Features:

- **Visual Design:** A minimal, professional interface that uses the university's signature blue palette.
- **Branding Consistency:** The BK TPHCM hexagonal logo is incorporated to reinforce institutional identity.
- **Focused Layout:** The login form is centered on a clean white background to maximize clarity and reduce distractions.
- **Localized Fields:** *Username* and *Password* inputs are clearly labeled in Vietnamese, aligning with the university's bilingual guidelines.

User Flow: When a user enters credentials, the system forwards them to *HCMUT_SSO* for verification. If valid, a session token is issued and the user is redirected to their corresponding dashboard (Student, Tutor, Coordinator, etc.). If authentication fails, inline red validation messages are displayed without page reload, preserving usability and minimizing user frustration.

5.2.2 Calendar Interface

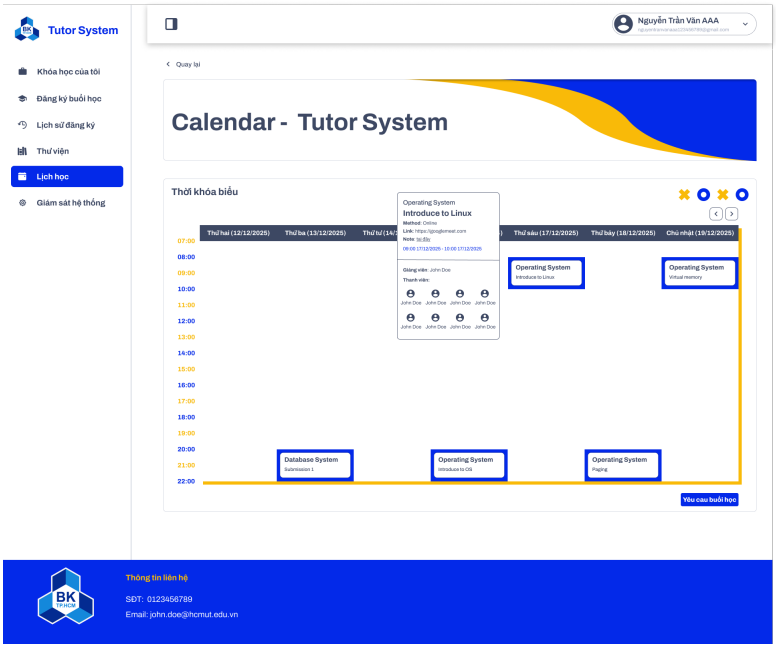


Figure 5.2: Students Calendar Interface

Purpose: The Calendar Interface is the central workspace where tutors and students can view, manage, and interact with their deadlines and tutoring sessions.

Features:

- **Weekly Overview:** Displays all deadlines and tutoring sessions for the selected week, with quick navigation to previous and next weeks.
- **Session Requests:** Allows students to request new tutoring sessions directly from the calendar when additional support is needed.

User Flow:

- Students view their personal calendar. By default, it shows the current week; navigation arrows move between weeks.
- Students click a session block to join the class or access its materials.
- Tutors create sessions by completing a form. After saving, the session appears on the student's calendar.

5.2.3 Course Dashboard and Integrated Chat Interface

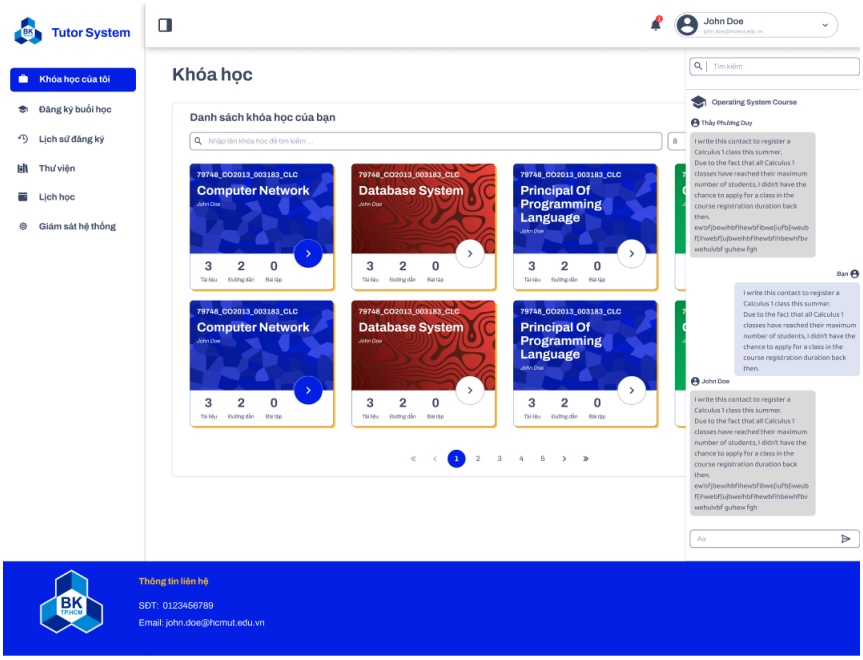


Figure 5.3: Chat Interface – Tutor–Student Messaging Detail

Purpose: The **Course Dashboard** serves as the central hub for students and tutors to manage their enrolled or assigned courses within the Tutor Support System (TSS). It consolidates course access, learning materials, notifications, and real-time communication, ensuring a cohesive environment for teaching and learning interactions.

Features:

- Role-Based Display: Automatically loads course information according to the user's role (Student or Tutor).
- Dynamic Course Grid: Presents all enrolled or assigned courses as interactive cards with course codes, titles, and tutor names.
- Search and Filtering: Supports keyword search, pagination, and sorting to navigate extensive course lists efficiently.
- Notification Drawer: Centralizes course-related updates and announcements for easy access.
- Integrated Chat Panel: Enables real-time messaging between tutors and students for feedback, clarification, and academic discussions.

User Flow:

- After login, the user is redirected to the Course Dashboard.
- The system retrieves and displays all enrolled or assigned courses as interactive cards.
- The user can search, sort, or paginate through the list to find a specific course.
- Clicking on a course card opens its detailed view with access to learning materials and the chat interface.
- The Chat Panel allows tutors and students to exchange messages, share feedback, and coordinate sessions in real time.
- Notifications update dynamically to reflect new course activities or messages.

5.2.4 Course Materials Interface

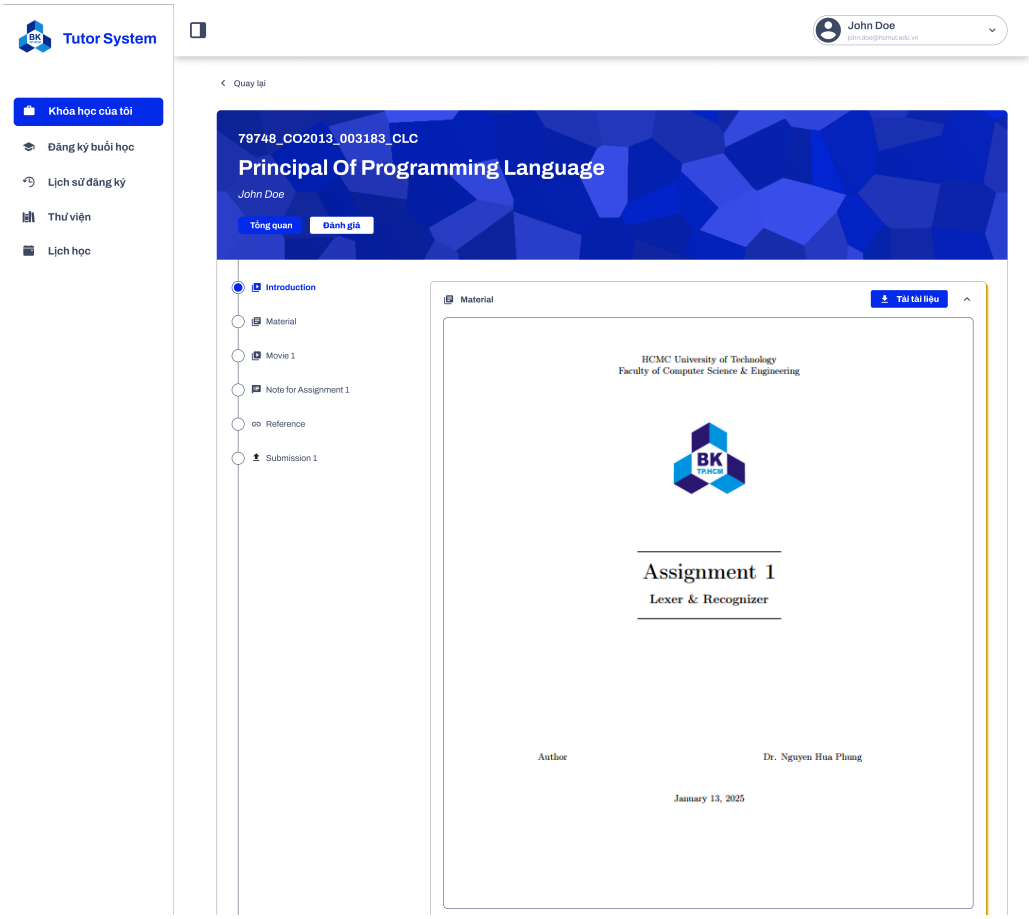


Figure 5.4: Students Course Materials Interface

Purpose: The Course Materials Interface enables tutors and students to interact with course's materials such as file, video, note, hyperlink and library's material.

Features:

- **Material Catalog:** Lists all materials in a course, organized by topic and type for quick scanning.
- **Document Viewer & Downloads:** Previews documents in-browser and allows downloading of the original files.
- **External Links:** Opens hyperlink materials in a new tab, taking users directly to the referenced page.

- **Library Resources:** Deep-links to the library catalog item or its digital copy (when available).
- **Content Management (Tutor):** Supports uploading and editing of course materials with clear, guided forms.

User flow:

- For student:
  1. **Select Course** : The student navigates from "Khóa học của tôi" and clicks on a specific course to view its materials
  2. **Navigate Content** : The student uses the left-hand table of contents to select a topic
  3. **Consume Materials** : The main panel updates, allowing the student to view document, download file, watch embedded videos, access external link and check assignment detail
- For tutor:
  1. **Access course** : The tutor navigates to the course materials page
  2. **Manage Content** : The tutor adds or edits sections in the navigation panel
  3. **Upload Materials** : For each section, the tutor uploads files, embeds videos, adds links, and writes notes for the students.
  4. **Publish** : The tutor saves the changes, making the materials instantly available to students

5.2.5 Registration and Matching Module

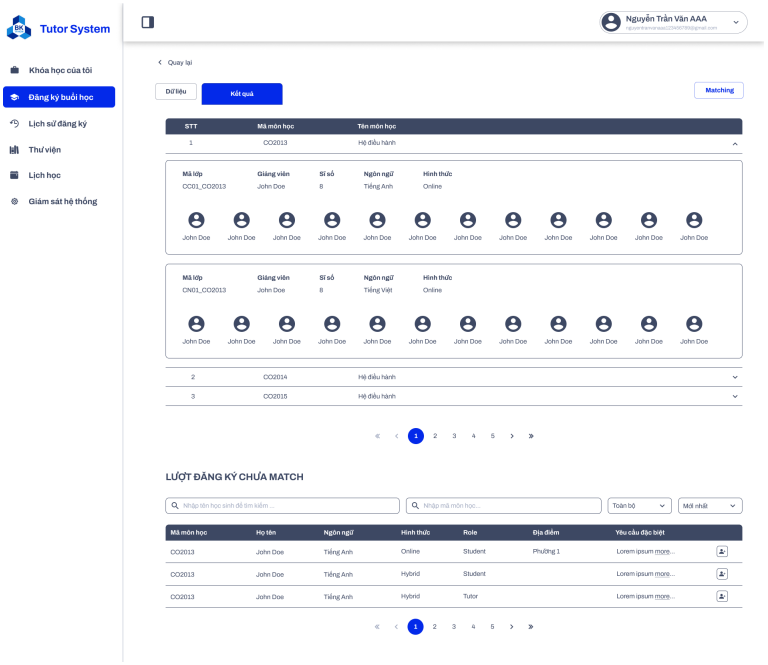


Figure 5.5: Course Matching Result

Purpose: This module manages the registration and pairing process for tutoring programs.

- Students register for available tutor-led courses and specify preferences.
- Tutors submit teaching details such as subjects, languages, and delivery modes.
- Coordinators oversee registration periods and monitor submissions, while the system's AI Matching Service automatically pairs tutors and students based on predefined criteria.

Features:

- Role-Based Forms: Separate registration interfaces for Students, Tutors, and Coordinators.
- Deadline Management: Coordinators define opening and closing dates through a scheduling modal.
- AI Matching: The system filters and pairs participants automatically according to subject, language, and schedule.

- Status Tracking: Registrations display color-coded tags (Pending, In Review, Matched).
- Multilingual Support: Drop-down menus in English or Vietnamese enhance accessibility.

User Flow:

1. Students and tutors complete their respective registration forms.
2. The system validates and saves submissions.
3. Coordinators review and manage registration timelines.
4. The AI Matching Service pairs tutors and students automatically.
5. Matched records are updated in the database, while unmatched ones remain for later review.

5.2.6 HCMUT Library Integration Interface

Figure 5.6: Library Interface

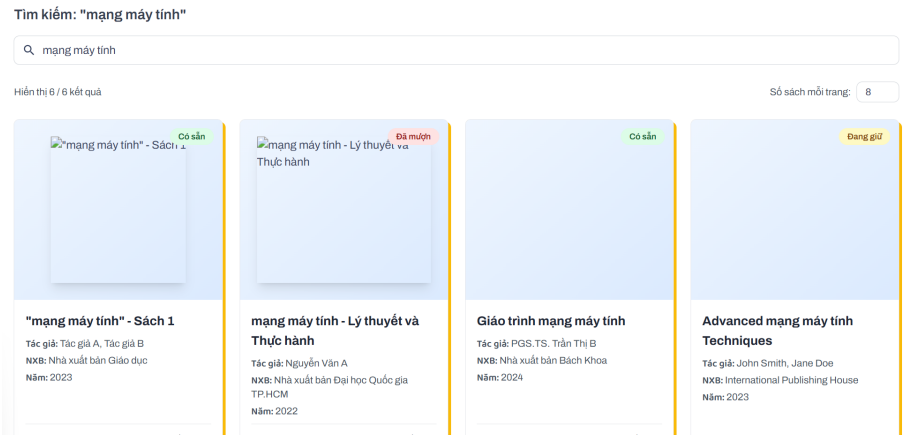


Figure 5.7: Search library

Purpose The HCMUT Library Interface links the Tutor Support System (TSS) with the university's centralized digital library, enabling tutors and students to access academic resources directly within the platform. It allows users to search, preview, and download course-related materials such as textbooks, lecture slides, and notes without leaving the system. This integration ensures a seamless transition between tutoring activities and official institutional learning resources hosted on HCMUT_LIBRARY.

Features:

- Search Functionality: Supports keyword-based queries by title or author to filter relevant materials.

User Flow:

1. The user selects "Thư viện" from the sidebar.
2. The system loads and displays available academic resources.
3. The user enters a keyword in the search bar to find specific materials.
4. The results update dynamically with pagination for navigation.
5. Selecting a resource card opens detailed information or initiates a secure download.

5.2.7 Evaluation and Report Module

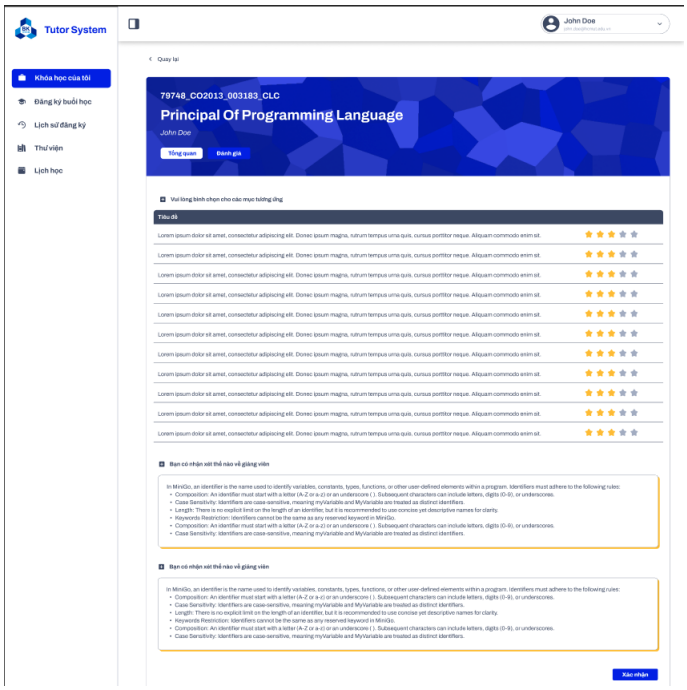


Figure 5.8: Evaluation Interface – Student Perspective with Rating Criteria

Purpose: The **Evaluation and Report Module** provides a structured mechanism for collecting, displaying, and managing feedback among students, tutors, and coordinators. It supports both qualitative comments and quantitative scoring, enabling continuous monitoring of session quality and participant performance across the tutoring program. This module ensures transparency, accountability, and systematic quality improvement in tutoring activities.

Features:

- **Star-Rating Component:** Provides 1–5 scale feedback with visual cues for clarity.
- **Role-Adaptive Interface:** UI content dynamically changes according to the user's account type.
- **Comment Section:** Enables text-based qualitative feedback from both tutors and students.
- **Pagination and Filtering:** Coordinators can search and navigate through multiple evaluations per course.

- **Consolidated Report:** Data collected from multiple participants are displayed in a structured coordinator dashboard for trend analysis.

User Flow:

1. The session concludes, prompting the user to open the evaluation form for that course.
2. Students submit ratings and written feedback about tutors.
3. Tutors reciprocate by evaluating each assigned student.
4. The coordinator accesses all related data through the report dashboard.
5. Consolidated insights are exported for program-level quality tracking and improvement planning.

5.2.8 Course Analytics Module

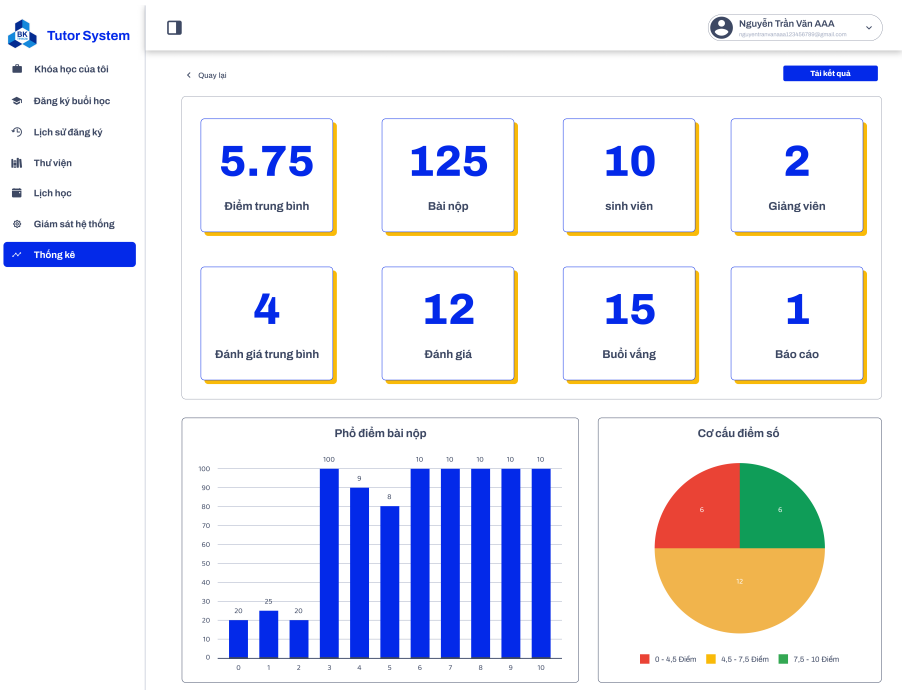


Figure 5.9: Course Analytics

Purpose: The Analytics Module provides coordinators with visual insights into course performance, student participation, and tutor activity. It enables quick evaluation of academic outcomes and helps coordinators identify areas that require support or adjustment.

The system automatically aggregates data from submissions, attendance, and assessments to ensure accuracy and reduce manual work.

Features:

- Course Dashboard Overview: Displays summarized metrics such as average score, total submissions, number of students and tutors, average rating, total evaluations, and attendance records.
- Interactive Charts: Bar and pie charts illustrate grade distribution, rating breakdown, and participation levels for each course.
- Activity Timeline: Shows historical engagement trends over time, allowing coordinators to track progress throughout the semester.
- Filtering and Comparison: Coordinators can select specific courses from their managed list to analyze results and compare performance metrics.
- Data Export: Allows downloading summarized reports in standard formats for departmental review or archival.

User Flow:

1. The coordinator accesses the Statistics section from the sidebar.
2. The system displays a list of managed courses with summary cards (course name, tutor, number of students, and progress).
3. The coordinator selects a course to view detailed analytics.
4. The dashboard loads visual reports including average score, rating distribution, attendance, and submission statistics.
5. The coordinator can export data or return to the course list to view another course.

5.2.9 System Monitoring Module

Figure 5.10: System Monitoring Interface

Purpose: The System Monitoring Module allows coordinators and administrators to track the system's real-time performance, ensuring stability and reliability. It provides live metrics on server health, enabling early detection of performance degradation or resource overload that may affect user experience.

Features:

- Resource Dashboards: Displays real-time gauges for CPU, memory, disk, and swap usage.
- Performance Charts: Line graphs visualize CPU and memory consumption trends over time.
- Dynamic Updates: Data refreshes automatically at short intervals for continuous monitoring.
- Usage Segmentation: Graphs distinguish between user-related and system-level activity (e.g., Busy User vs. Busy I/O).

- Visual Alerts: Color indicators highlight high-usage conditions, helping administrators respond promptly.

User Flow:

1. The coordinator or administrator selects the matching section from the sidebar.
2. The dashboard displays key usage indicators for CPU, memory, disk, and swap.
3. Real-time charts show historical performance trends to assess system load.
4. When abnormal resource usage occurs, visual gauges change color to alert the user.
5. Administrators use the data to optimize resource allocation or perform maintenance.

6 Sequences diagrams

6.1 Sequence Diagram – Log in Process

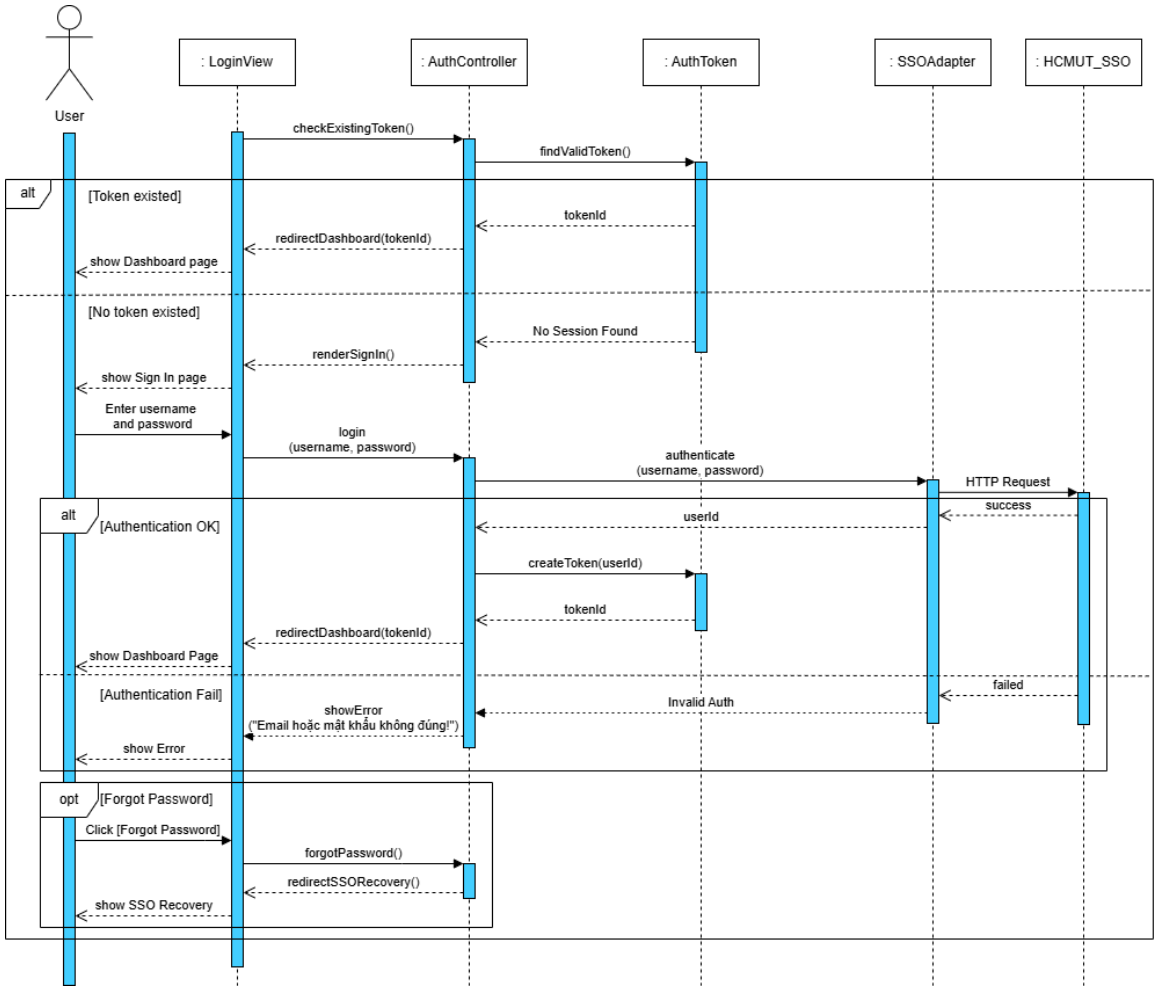


Figure 6.1: Sequence Diagram – Sign-In Process with HCMUT SSO Integration

Actors and Lifelines

- **User:** Initiates the sign-in process through the web interface.
- **Log in Page: WebAppUI** – Frontend boundary class responsible for collecting user credentials and displaying responses.
- **: AuthController** – Core control component managing the authentication flow and SSO communication.
- **:AuthToken** – Entity component that stores and validates active user sessions.

- **: SSOAdapter** – External interface responsible for communicating with HCMUT's centralized SSO service.
- **: HCMUT_SSO** – HCMUT's SSO service responsible for system validation.

Main Flow The diagram consists of three logical segments:

1. **Session Validation:** The controller first checks for an existing session via checkExistingToken(). If a valid session is found, the user is redirected directly to the dashboard.
2. **Authentication:** When no session exists, the system renders the log in page. Upon user input, the controller forwards credentials to `SSOAdapter` for validation through `authenticate(username, password)`. Successful authentication triggers `createToken(userId)` and redirects to the dashboard.
3. **Error and Recovery Handling:** The **alt** fragments describe possible exceptions:
   - **Authentication Fail:** Invalid credentials return an "Email hoặc mật khẩu không đúng!" error.
   - **Forgot Password (opt):** Users may trigger password recovery, which redirects them to the official HCMUT SSO recovery portal.

6.2 Sequence Diagram – Course Registration Process

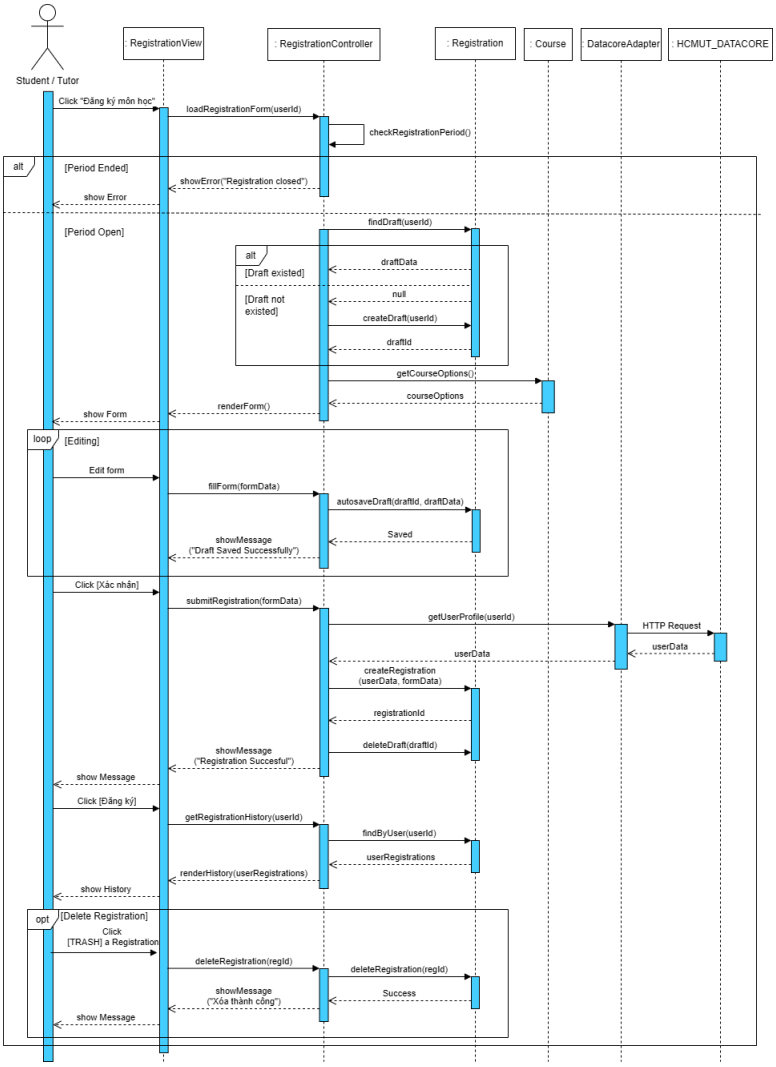


Figure 6.2: Sequence Diagram – Course Registration Workflow for Students and Tutors

Actors and Lifelines

- **Student / Tutor:** The primary actor who interacts with the system to initiate course registration, edit forms, or manage existing registrations.
- **:RegistrationView:** The presentation layer (Boundary) responsible for rendering the UI, capturing user events (clicks), and displaying system feedback or errors.
- **:RegistrationController:** The logic layer that orchestrates the workflow, validates the registration period, and coordinates data exchange between the View, Entities, and external adapters.

- **:Registration:** The entity responsible for persisting data, handling both temporary drafts and finalized registration records.
- **:Course:** A service or entity provider used to retrieve the list of available course options.
- **:DatacoreAdapter:** An integration component acting as a bridge to fetch verified user profiles from the external core system.
- **:HCMUT_DATACORE:** The external institutional system that processes HTTP requests to provide authoritative user data.

Main Flow The process consists of multiple logical stages:

1. **Form Initialization and Validation:**
   The process commences when the actor clicks "Đăng ký môn học". The View triggers `loadRegistrationForm(userId)`, then the Controller to execute `checkRegistrationPeriod()`.
   - [**Period Ended**]**:** If the registration window is closed, the Controller returns an error, and the View displays "Hết hạn đăng ký".
   - [**Period Open**]**:** If the window is active, the flow proceeds to draft retrieval.
2. **Draft Management and Rendering:**
   The Controller attempts to retrieve an existing session via `findDraft(userId)` from the `:Registration` entity.
   - If a draft exists, the system retrieves `draftData`.
   - If no draft is found (null), the system initializes one using `createDraft(userId)` and returns the new `draftId`.

   Subsequently, the Controller fetches available courses using `getCourseOptions()` from the `:Course` lifeline and instructs the View to display the interface via `renderForm()`.
3. **Iterative Editing:**
   Inside the [**Editing**] loop, as the user modifies the form, the View sends updates via `fillForm(formData)`. The Controller ensures data persistence by calling `autosaveDraft(draftId, draftData)`, triggering a "Draft Saved Successfully" confirmation message.

4. **Submission and Verification:**
   Upon clicking "Xác nhận" (Confirm), the Controller executes `submitRegistration(formData)`.
   - The Controller requests user details via `getUserProfile(userId)` from the `:DatacoreAdapter`.
   - The Adapter sends an **HTTP Request** to the external `:HCMUT_DATACORE` system to retrieve verified `userData`.
   - Once verified, the Controller finalizes the process by calling `createRegistration(userDat formData)` and simultaneously removes the temporary draft via `deleteDraft(draftId)`.
   - The View displays a "Registration Successful" message.
5. **Registration History:**
   When the user navigates to the registration tab (Click "Đăng ký"), the Controller executes `getRegistrationHistory(userId)`. The Entity retrieves records via `findByUser(userI` and the View presents the list using `renderHistory(userRegistrations)`.
6. **Optional Deletion:**
   In the **opt** [**Delete Registration**] block, if the user clicks the [TRASH] icon, the Controller invokes `deleteRegistration(regId)`. The Entity processes the deletion, and upon success, the View notifies the user with "Xóa thành công" (Deletion successful).

6.3 Sequence Diagram – Assign Classes Process

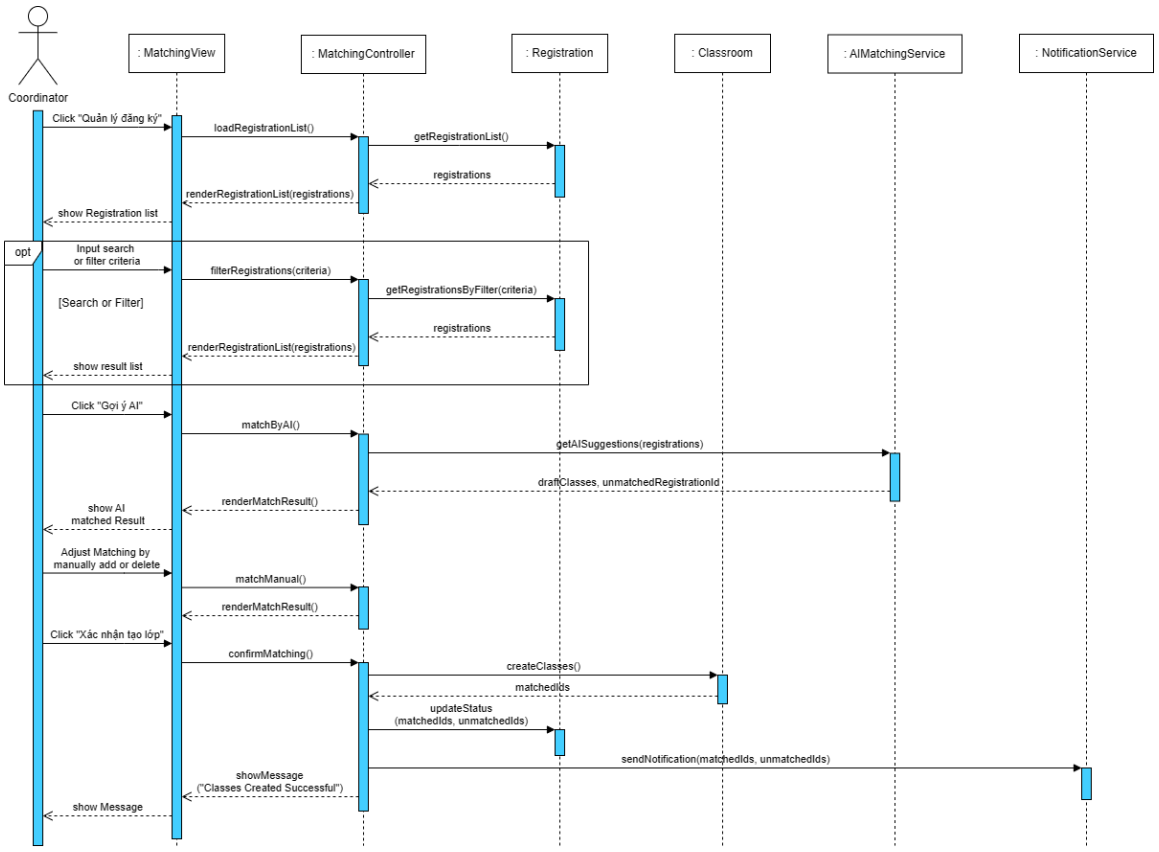


Figure 6.3: Sequence Diagram – Coordinator Matching and Classes Assignment Process

Actors and Lifelines

- **Coordinator:** The administrative user responsible for managing registration lists and confirming tutor–student assignments.
- **:MatchingView:** The boundary object representing the user interface for managing registrations, displaying AI suggestions, and showing results.
- **:MatchingController:** The control layer orchestrating the flow between user inputs, data retrieval, AI processing, and final confirmation.
- **:Registration:** The entity component responsible for retrieving registration lists and updating the status of matched/unmatched records.
- **:Classroom:** The entity component responsible for creating actual class instances upon confirmation.

- **:AIMatchingService:** The external service interface that processes registration data to generate automated class suggestions.
- **:NotificationService:** The external system responsible for delivering notifications to users regarding their class allocation status.

Main Flow The assignment process follows a structured workflow:

1. **Initialization and Loading:**
   The process begins when the Coordinator clicks "Quản lý đăng ký". The `:MatchingView` triggers `loadRegistrationList()` to the controller. The controller retrieves data via `getRegistrationList()` from `:Registration`, and the view displays the data using `renderRegistrationList(registrations)`.
2. **Search and Filtering (Optional):**
   In the **opt** block, if the Coordinator inputs search terms or criteria, the view calls `filterRegistrations(criteria)`. The controller queries the entity using `getRegistrations(criteria)`. The filtered list is then displayed via `renderRegistrationList`
3. **AI-Assisted Matching:**
   Upon clicking "Gợi ý AI", the controller executes `matchByAI()`. It delegates the logic to `:AIMatchingService` via `getAISuggestions(registrations)`. The service returns `draftClasses` and `unmatchedRegistrationId`, which are presented to the user through `renderMatchResult()`.
4. **Manual Adjustment:**
   The Coordinator may manually add or delete matches ("Adjust Matching"). The view triggers `matchManual()`, allowing the controller to process changes and update the display via `renderMatchResult()`.
5. **Confirmation and Finalization:**
   When the Coordinator clicks "Xác nhận tạo lớp", the controller executes `confirmMatching()`. This triggers a sequence of persistence and notification operations:
   - **Create Classes:** The controller calls `createClasses()` on the `:Classroom` entity, which returns the `matchedIds`.
   - **Update Status:** The controller updates the status of both matched and unmatched records in `:Registration` via `updateStatus(matchedIds, unmatchedIds)`.

- **Notification:** Finally, the controller invokes `sendNotification(matchedIds, unmatchedIds)` on the `:NotificationService`.

6. **Completion:**
   The system concludes the process by displaying the success message "Xác nhận tạo lớp thành công..." on the view.

6.4 Sequence Diagram – Learning Material Management

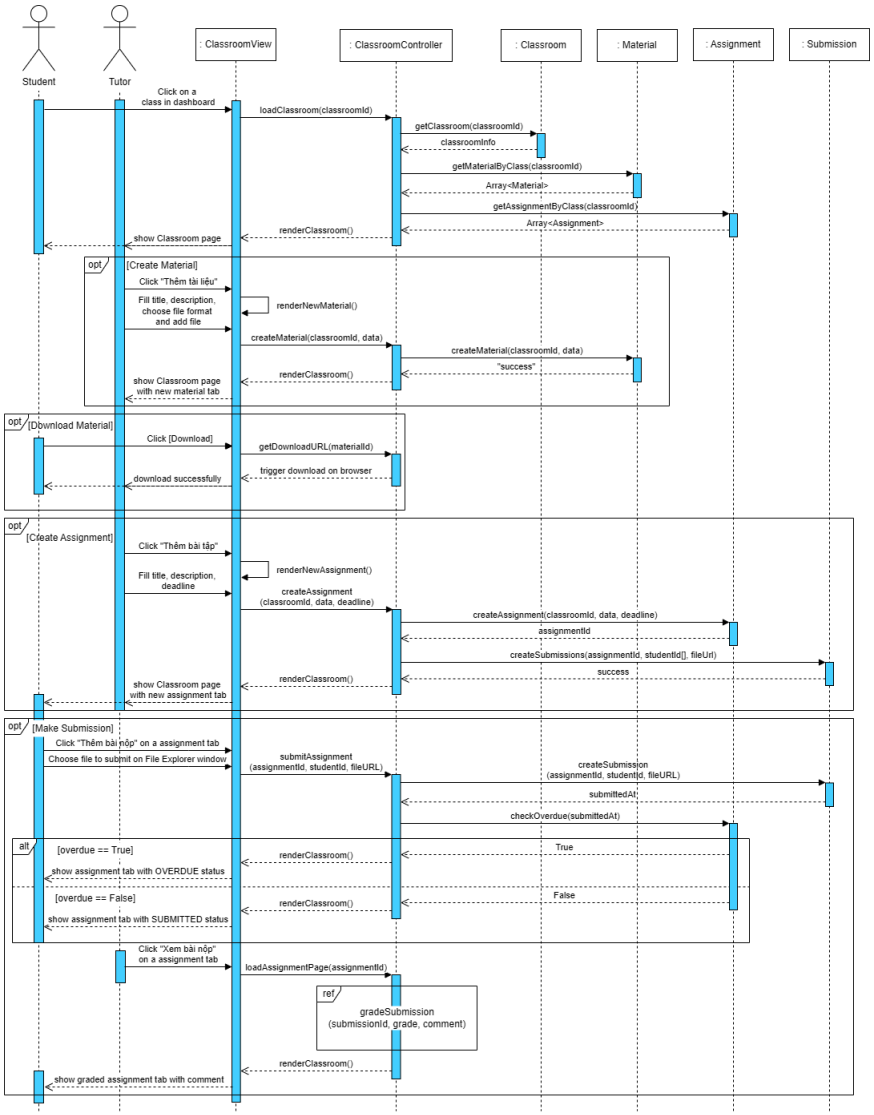


Figure 6.4: Sequence Diagram – Learning Material Management for Tutors and Students

Actors and Lifelines

- **Student / Tutor:** Users who interact with the classroom environment to view content, manage materials, or handle assignments.
- **:ClassroomView:** The user interface boundary responsible for rendering classroom details, materials, and assignment tabs.
- **:ClassroomController:** The central controller orchestrating data retrieval, material creation, and assignment processing.
- **:Classroom:** The entity managing core classroom metadata.
- **:Material:** The entity responsible for storing learning resources uploaded by tutors.
- **:Assignment:** The entity managing assignment details and deadlines.
- **:Submission:** The entity handling student submission records and status tracking.

Main Flow The classroom management process involves several key functional blocks:

1. **Classroom Initialization:**
   When a user clicks on a class in the dashboard, the view calls `loadClassroom(classroomId)`. The controller aggregates data by calling:
   - `getClassroom(classroomId)` from the `:Classroom` entity.
   - `getMaterialByClass(classroomId)` from the `:Material` entity.
   - `getAssignmentByClass(classroomId)` from the `:Assignment` entity.

   Finally, the view displays the page via `renderClassroom()`.
2. **Material Creation (Optional):**
   In the **opt** [**Create Material**] block, the Tutor clicks "Thêm tài liệu" (Add Material). After filling in the details, the controller executes `createMaterial(classroomId, data)`. This triggers the creation logic in the `:Material` entity, and the view updates via `renderClassroom()` to show the new material tab.
3. **Downloading Material (Optional):**
   In the **opt** [**Download Material**] block, when a user clicks [Download], the controller retrieves the link via `getDownloadURL(materialId)` and triggers the file download on the browser.

4. **Assignment Creation (Optional):**
   In the **opt** [**Create Assignment**] block, the Tutor clicks "Thêm bài tập" (Add Assignment). The controller executes `createAssignment(classroomId, data, deadline)`.

   - The `:Assignment` entity creates the assignment record.
   - The `:Submission` entity initializes empty submission placeholders via `createSubmissions(assi studentId, fileUrl)`.
   - The view refreshes via `renderClassroom()`.

5. **Student Submission (Optional):**
   In the **opt** [**Make Submission**] block, the Student uploads a file. The controller calls `submitAssignment(assignmentId, studentId, fileURL)`.

   - The `:Submission` entity updates the record via `createSubmission(...)` and validates the timestamp with `checkOverdue(submittedAt)`.
   - **Alt** [**Overdue Check**]**:** If `overdue == True`, the view shows the status "OVERDUE"; otherwise, it shows "SUBMITTED" via `renderClassroom()`.

6. **Grading Access:**
   Finally, if a user clicks "Xem bài nộp" (View Submission), the controller loads the assignment page, references the `gradeSubmission` process, and renders the updated classroom view.

6.5 Sequence Diagram – Grading student's submission

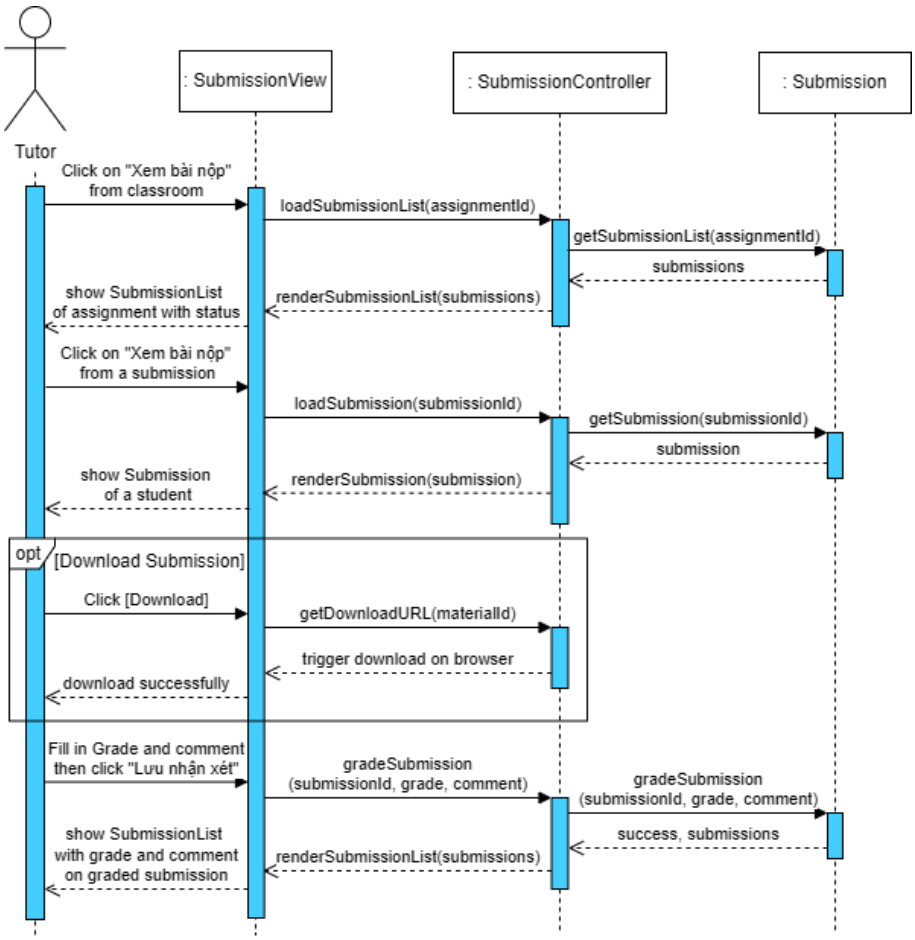


Figure 6.5: Submission: Grading student's submission

Actors and Lifelines

- **Tutor:** The academic staff responsible for reviewing student submissions, downloading attachments, and providing grades with feedback.
- **:SubmissionView:** The boundary interface that presents the list of submissions, displays individual student work, and captures grading inputs.
- **:SubmissionController:** The control class that manages the retrieval of submission data, handles file download logic, and processes grading transactions.
- **:Submission:** The entity component responsible for persisting submission records, including status, grades, and comments.

Main Flow The submission review and grading process consists of the following logical steps:

1. **Load Submission List:**
   The process begins when the Tutor clicks "Xem bài nộp" (View Submissions) from the classroom interface. The `:SubmissionView` calls `loadSubmissionList(assignmentId)`. The controller retrieves data via `getSubmissionList(assignmentId)` from the `:Submission` entity, and the view displays the results using `renderSubmissionList(submissions)`.

2. **View Specific Submission:**
   When the Tutor selects a specific student's work ("Xem bài nộp"), the view executes `loadSubmission(submissionId)`. The controller requests details via `getSubmission(submissionId` and instructs the view to display the specific submission through `renderSubmission(submission)`.

3. **Download Submission (Optional):**
   In the **opt** [**Download Submission**] block, if the Tutor clicks the [Download] button, the view requests `getDownloadURL(materialId)`. The controller processes this request and triggers the download action on the browser ("trigger download on browser"), resulting in a successful file download.

4. **Grading and Feedback:**
   To grade the work, the Tutor fills in the grade and comments, then clicks "Lưu nhận xét" (Save Feedback). The view sends this data to the controller via `gradeSubmission(submissionId grade, comment)`.

   - The controller forwards the data to the `:Submission` entity via `gradeSubmission(submissionI grade, comment)`.
   - Upon success, the entity returns the updated submissions list.
   - The controller refreshes the interface by calling `renderSubmissionList(submissions)`, showing the updated grade and comment status.

6.6 Sequence Diagram – Session management

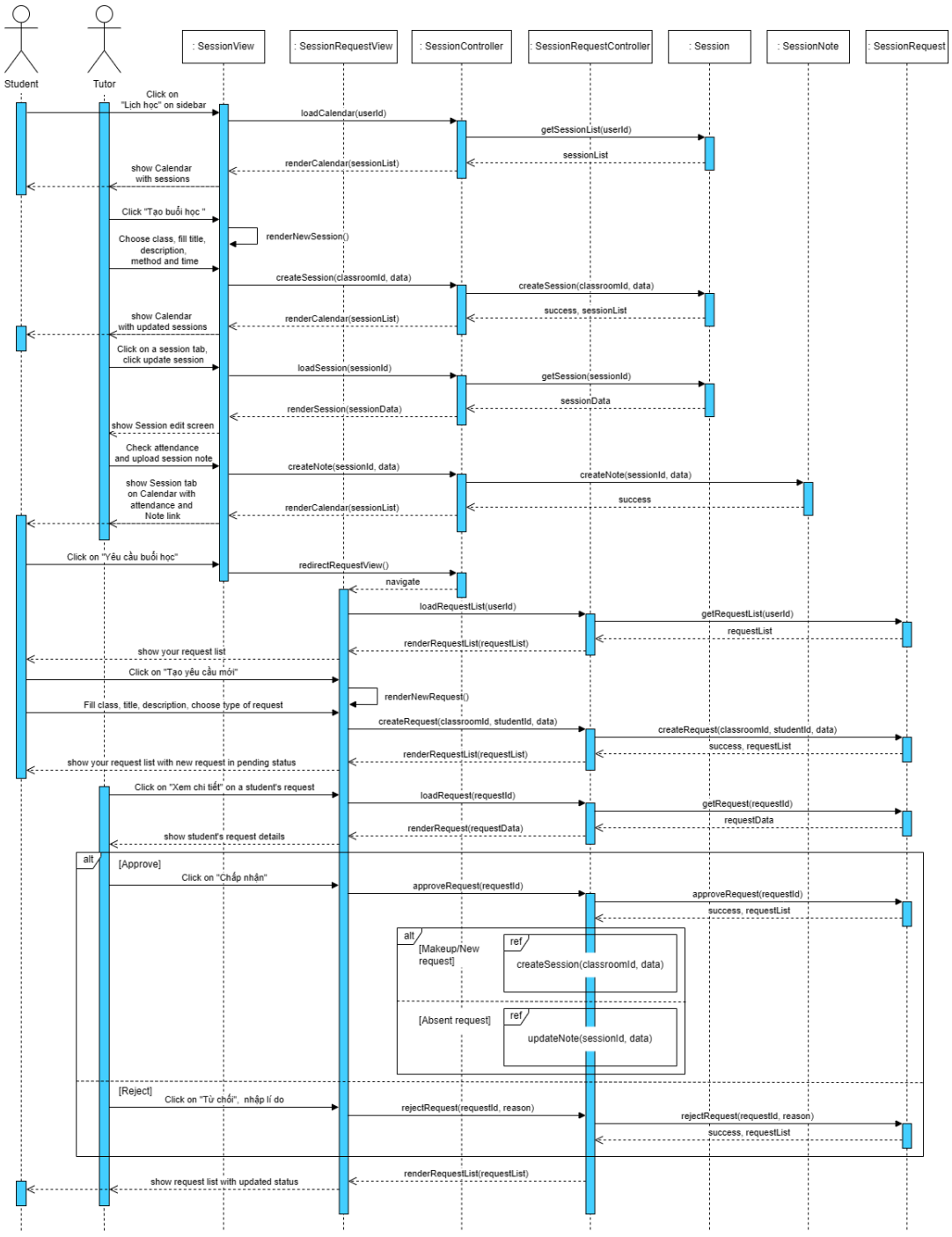


Figure 6.6: Sequence Diagram – Deadline and Assignment Management for Students and Tutors

Actors and Lifelines

- **Student / Tutor:** Users who view schedules, manage sessions, or handle session-related requests.
- **:SessionView:** The boundary interface for the calendar view, responsible for displaying scheduled sessions and capturing session creation events.
- **:SessionRequestView:** The boundary interface dedicated to managing student requests (e.g., makeup classes, absence reporting).
- **:SessionController:** The control layer handling logic for session retrieval, creation, and attendance tracking.
- **:SessionRequestController:** The control layer managing the lifecycle of student requests, including creation, approval, and rejection.
- **:Session:** The entity component representing scheduled class sessions.
- **:SessionNote:** The entity component storing attendance records and lesson notes.
- **:SessionRequest:** The entity component persisting user requests regarding session scheduling.

Main Flow The process encompasses two primary workflows: Calendar/Session Management and Request Processing.

1. **Calendar Initialization:**
   The workflow begins when a user clicks "Lịch học" (Calendar). The `:SessionView` invokes `loadCalendar(userId)`. The `:SessionController` fetches data via `getSessionList(userI` from the `:Session` entity, and the view displays the schedule via `renderCalendar(sessionList)`.
2. **Session Creation (Tutor):**
   The Tutor clicks "Tạo buổi học". After filling in the details, the controller executes `createSession(classroomId, data)`. This triggers the creation logic in the `:Session` entity. Upon success, the calendar is refreshed via `renderCalendar(sessionList)`.
3. **Attendance and Notes:**
   To manage a specific session, the Tutor clicks a session tab.
   - The system loads details via `loadSession(sessionId)` and `getSession(sessionId)`.
   - The Tutor checks attendance or uploads notes, triggering `createNote(sessionId, data)`.

- The `:SessionController` persists this via the `:SessionNote` entity, and the view updates via `renderCalendar()`.

4. **Request Management (Student):**
   The Student clicks "Yêu cầu buổi học", redirecting to the `:SessionRequestView`.
   - The view loads existing requests via `loadRequestList(userId)`.
   - To make a new request ("Tạo yêu cầu mới"), the Student fills the form, triggering `createRequest(...)`.
   - The `:SessionRequestController` saves the request in the `:SessionRequest` entity and refreshes the list.

5. **Request Processing (Tutor):**
   The Tutor views a student's request details via `loadRequest(requestId)`. The system enters an **Alt** block based on the Tutor's decision:
   - **[Approve]:** The Tutor clicks "Chấp nhận". The controller calls `approveRequest(request`
     - **Inner Alt [Makeup/New]:** The system references the `createSession` process.
     - **Inner Alt [Absent]:** The system references the `updateNote` process.
   - **[Reject]:** The Tutor clicks "Từ chối" and provides a reason. The controller calls `rejectRequest(requestId, reason)`.

   Finally, the view updates the request status via `renderRequestList(requestList)`.

6.7 Sequence Diagram – Chat System

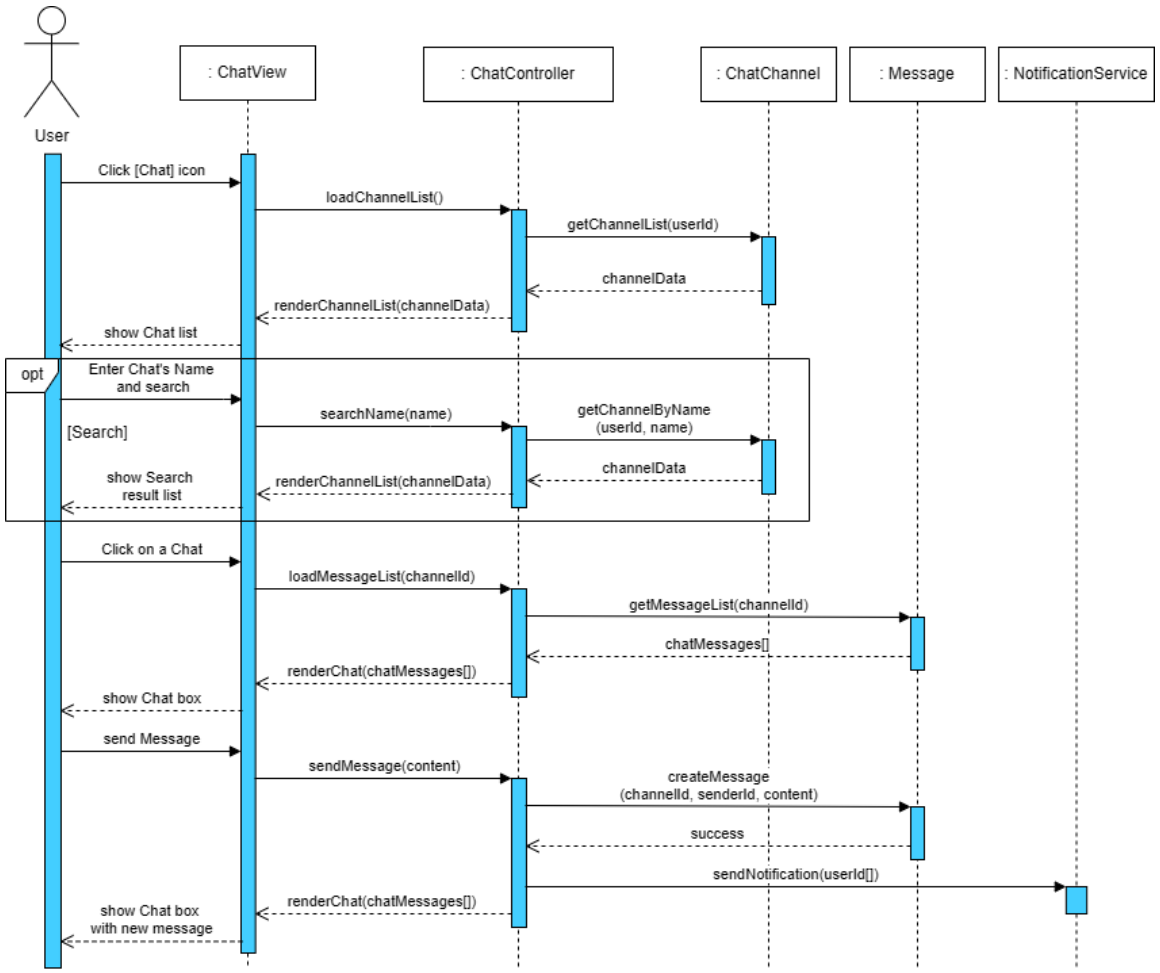


Figure 6.7: Sequence Diagram – Chat and Notification System Interaction

Actors and Lifelines

- **User:** The participant initiating conversations, searching for contacts, or sending messages.
- **:ChatView:** The boundary interface responsible for displaying the chat list, search results, and the active conversation window.
- **:ChatController:** The control component that coordinates message retrieval, search logic, and the message sending workflow.
- **:ChatChannel:** The entity component responsible for managing conversation threads (channels) and retrieving channel lists.

- **:Message:** The entity component responsible for persisting individual message records and retrieving message history.
- **:NotificationService:** The external service used to alert recipients when a new message is successfully created.

Main Flow The chat functionality follows a sequential flow from channel selection to message transmission:

1. **Channel List Initialization:**
   When the User clicks the [Chat] icon, the `:ChatView` triggers `loadChannelList()`. The controller retrieves the user's available conversations via `getChannelList(userId)` from the `:ChatChannel` entity. The view then displays the data using `renderChannelList(chan`
2. **Search Functionality (Optional):**
   In the **opt** block, if the User enters a name to search, the view executes `searchName(name)`.
   - The controller queries the `:ChatChannel` entity using `getChannelByName(userId, name)`.
   - The entity returns the matching `channelData`.
   - The view updates the list via `renderChannelList(channelData)`.
3. **Conversation Retrieval:**
   When the User selects a specific chat, the view calls `loadMessageList(channelId)`. The controller requests the conversation history from the `:Message` entity using `getMessageList(channelId)`. The view then renders the conversation window via `renderChat(chatMessages[])`.
4. **Message Transmission:**
   The User inputs content and clicks send. The view triggers `sendMessage(content)`.
   - The controller persists the new message by calling `createMessage(channelId, senderId, content)` on the `:Message` entity.
   - Upon success, the controller triggers a notification to relevant users via `sendNotification` in the `:NotificationService`.
   - Finally, the controller updates the view to reflect the new message using `renderChat(chatM`

6.8 Sequence Diagram – Rating and Feedback

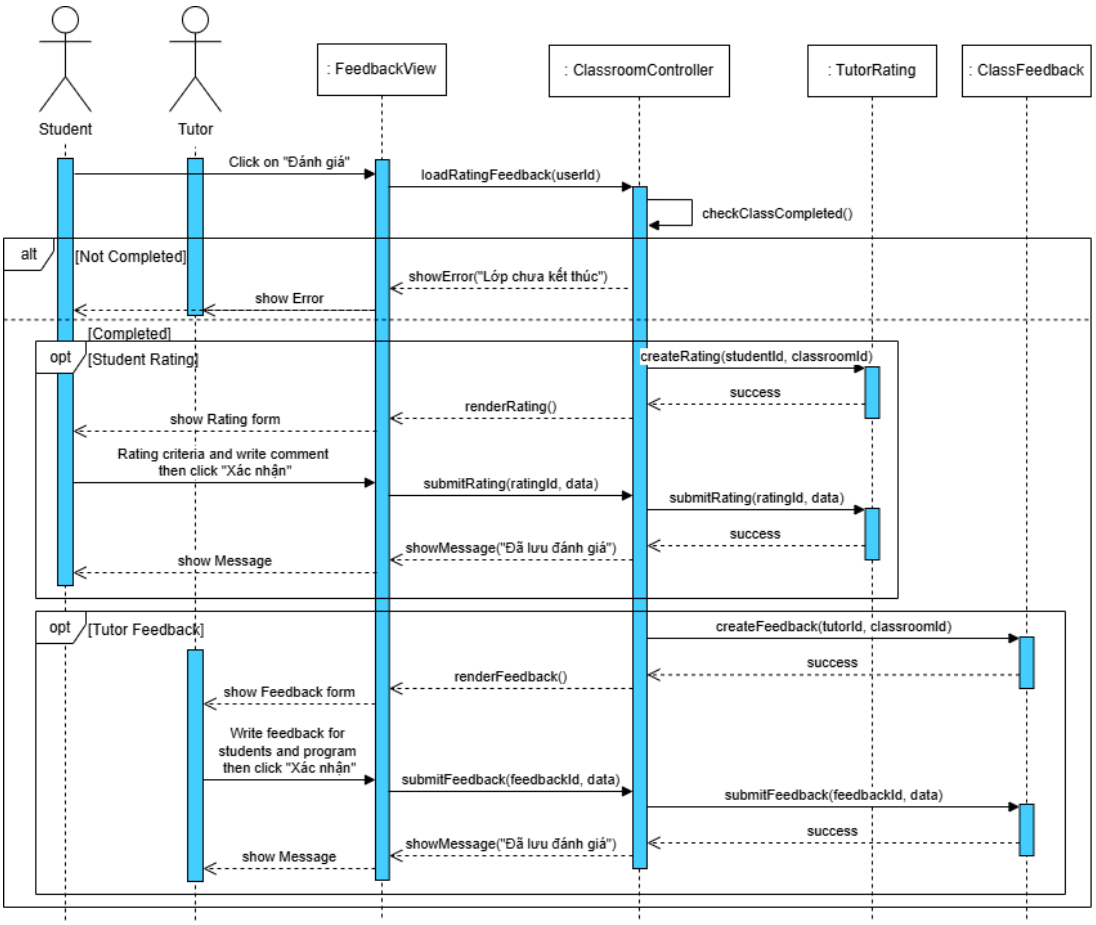


Figure 6.8: Sequence Diagram – Tutor Evaluation and Feedback Process

Actors and Lifelines

- **Student / Tutor:** The users participating in the evaluation process. The Student rates the Tutor, while the Tutor provides feedback on the class.
- **:ClassroomView:** The boundary interface responsible for displaying error messages, rating forms, and feedback forms to the users.
- **:ClassroomController:** The control component that validates course completion status and coordinates the creation and submission of ratings/feedback.
- **:TutorRating:** The entity component responsible for initializing and persisting rating records created by students.

- **:ClassFeedback:** The entity component responsible for initializing and persisting feedback records created by tutors.

Main Flow The evaluation process includes validation logic followed by distinct flows for students and tutors:

1. **Initialization and Validation:**
   When a user clicks "Đánh giá" (Rating), the `:ClassroomView` invokes `loadRatingFeedback(us`
   The `:ClassroomController` internally executes `checkClassCompleted()` to verify the course status.

2. **Completion Status Check (Alt Block):**
   The system behavior diverges based on the class status:

   - **[Not Completed]:** If the class is still active, the controller triggers `showError("Lớp chưa kết thúc")` (Class not ended), and the view displays the error to the user.
   - **[Completed]:** If the class is finished, the flow proceeds to the specific rating or feedback options below.

3. **Student Rating (Optional):**
   In the **opt** [**Student Rating**] block, if the actor is a Student:

   - The controller initializes a rating session via `createRating(studentId, classroomId)` in the `:TutorRating` entity.
   - The view displays the form via `renderRating()`.
   - After the student inputs criteria and clicks "Xác nhận" (Confirm), the view sends `submitRating(ratingId, data)`.
   - The controller persists the data in the entity, and the view displays "Đã lưu đánh giá" (Rating saved).

4. **Tutor Feedback (Optional):**
   In the **opt** [**Tutor Feedback**] block, if the actor is a Tutor:

   - The controller initializes a feedback session via `createFeedback(tutorId, classroomId)` in the `:ClassFeedback` entity.
   - The view displays the form via `renderFeedback()`.

- After the tutor inputs the program feedback and clicks "Xác nhận", the view sends `submitFeedback(feedbackId, data)`.
- The controller persists the data in the entity, and the view displays "Đã lưu đánh giá" (Rating saved).

6.9 Sequence Diagram – Searching and Importing Library References

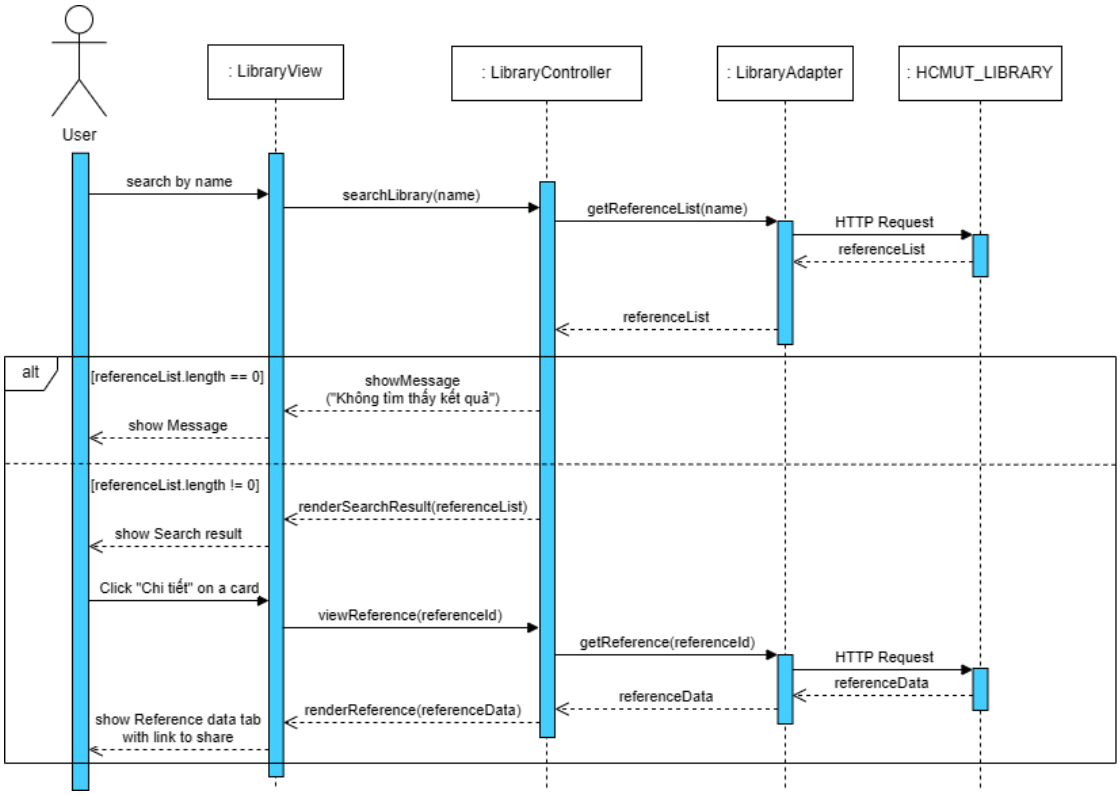


Figure 6.9: Sequence Diagram – Searching and Importing Library References

Actors and Lifelines

- **User:** The Student or Tutor initiating a search for academic resources.
- **:LibraryView:** The boundary interface responsible for capturing search inputs, displaying results, and showing detailed reference cards.
- **:LibraryController:** The control component that coordinates the search logic and processes data returned from the adapter.

- **:LibraryAdapter:** The integration layer acting as a bridge to fetch data from the external institutional library.
- **:HCMUT_LIBRARY:** The external system that handles HTTP requests and provides the actual reference metadata.

Main Flow The library search process involves querying an external system and handling the response states:

1. **Search Initialization:**
   The process begins when the User inputs a name to search. The `:LibraryView` triggers `searchLibrary(name)`. The `:LibraryController` forwards this request to the `:LibraryAdapter` via `getReferenceList(name)`.
2. **External Data Retrieval:**
   The `:LibraryAdapter` sends an **HTTP Request** to the `:HCMUT_LIBRARY`. The external system responds with a `referenceList`, which is passed back to the controller.
3. **Search Result Handling (Alt Block):**
   The controller evaluates the returned list length:
   - **[No Results]:** If `referenceList.length == 0`, the controller executes `showMessage("Kh tìm thấy kết quả")` (No results found), and the view displays the message.
   - **[Results Found]:** If `referenceList.length != 0`, the controller calls `renderSearchRes` and the view presents the list of reference cards to the user.
4. **View Reference Details:**
   When the User clicks "Chi tiết" (Details) on a specific card, the view invokes `viewReference(referenceId)`.
   - The controller requests specific data via `getReference(referenceId)` from the adapter.
   - The adapter initiates another **HTTP Request** to `:HCMUT_LIBRARY` to fetch the full `referenceData`.
   - Finally, the controller updates the interface using `renderReference(referenceData)`, showing the reference details and a shareable link.

7 Activity diagram

7.1 User - login

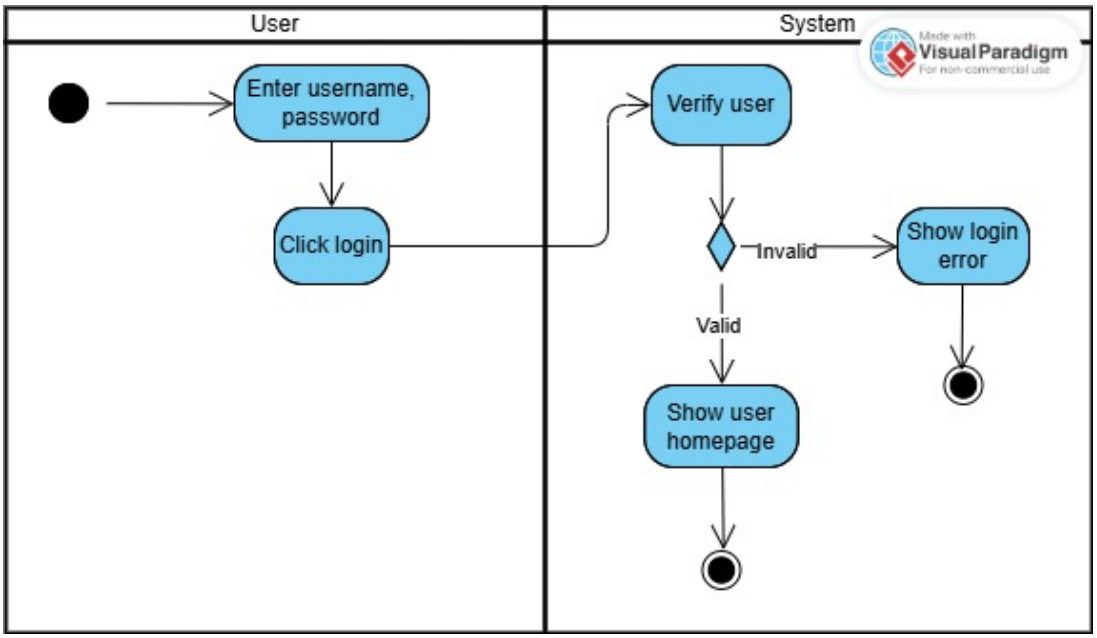


Figure 7.1: User - login

- **Login Process**

  1. **Enter Credentials**: The User initiates the session by entering their username and password.
  2. **Click Login**: The User submits the credentials by performing the click login action.
  3. **Verify User**: The System receives the input and proceeds to verify the user's information.
  4. **Validation Decision**: The System checks if the provided credentials are valid or invalid.
  5. **Show User Homepage**: If the credentials are **Valid**, the System grants access and displays the user homepage. The activity path concludes here.
  6. **Show Login Error**: If the credentials are **Invalid**, the System denies access and displays a login error message. The activity path concludes here.

7.2 User - register for program

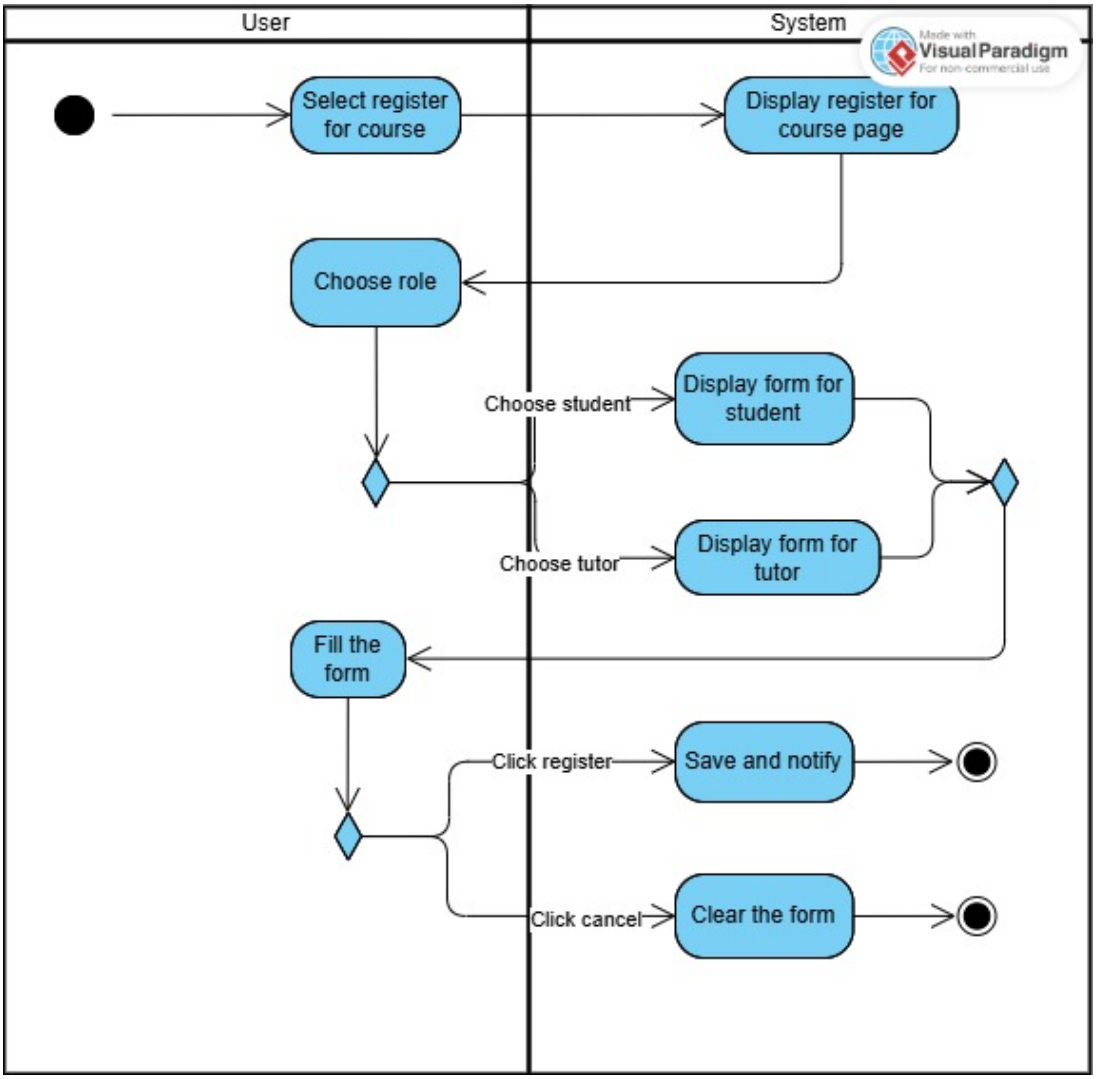


Figure 7.2: User - register for program

- **Registration**

  1. **Select Register**: The User starts the process by performing the select register for course action.
  2. **Display Page**: The System responds by displaying the register for course page.
  3. **Choose Role**: The User interacts with the page to choose their role.
  4. **Display Form**: Based on the decision made by the User:
     - If the User chooses **Student**, the System displays the form for students.

- If the User chooses **Tutor**, the System displays the form for tutors.

5. **Fill the Form**: The User proceeds to fill the form with the required information.
6. **Submission**: The process splits based on the User's final decision:
   - **Register**: If the User performs the click register action, the System executes the save and notify process. The activity path concludes at the final node.
   - **Cancel**: If the User performs the click cancel action, the System executes the clear the form process. The activity path concludes at the final node.

7.3 Tutor - Create new session

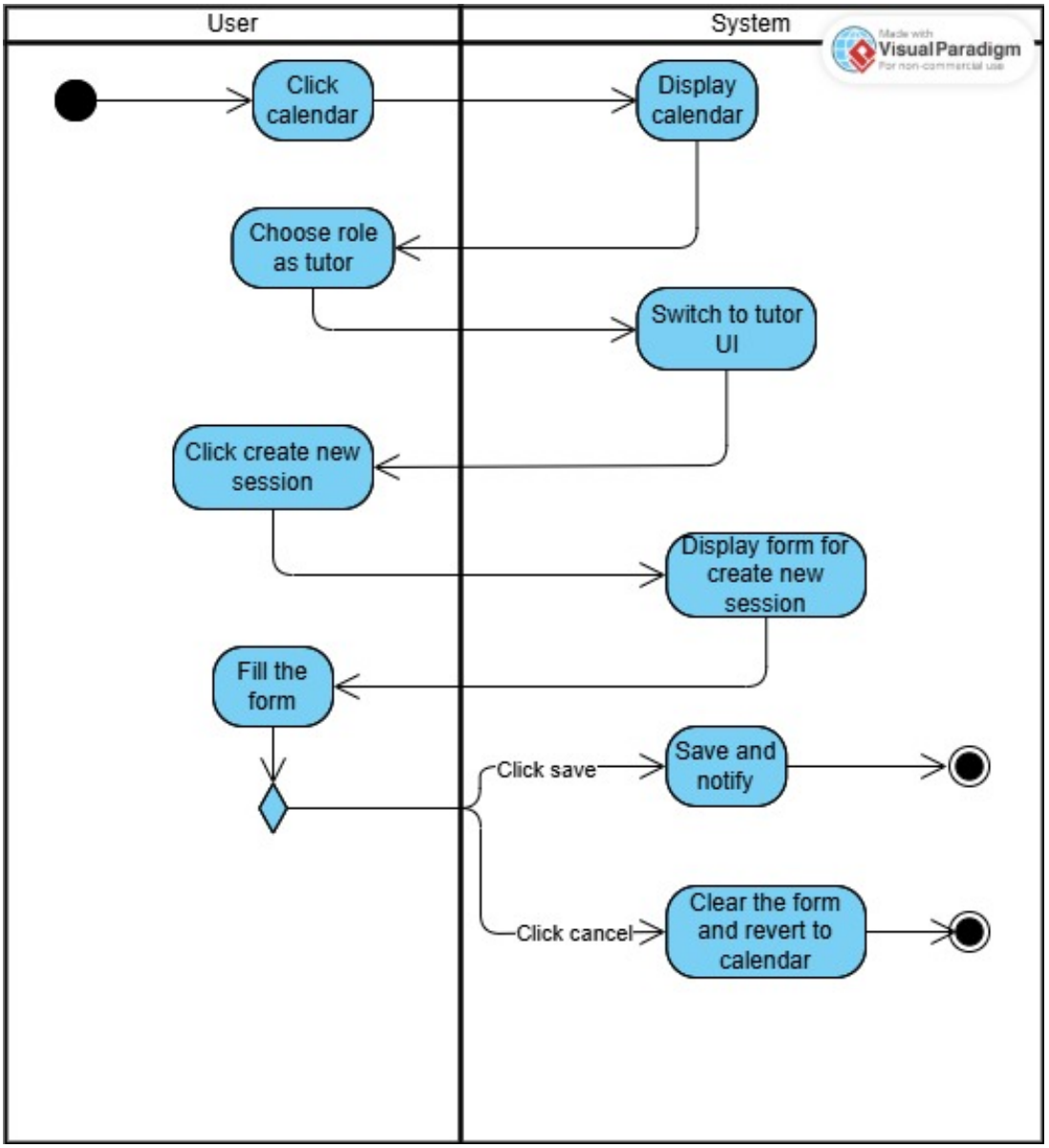


Figure 7.3: Tutor - create new class

- **Create New session**

  1. **Click Calendar**: The User initiates the process by performing the click calendar action.
  2. **Display Calendar**: The System responds by displaying the calendar interface.
  3. **Choose Role**: The User interacts with the system to choose their role as a tutor.

4. **Switch UI**: The System processes this selection and switches to the tutor user interface (UI).
5. **Click Create New session**: The User selects the specific action to click create new class.
6. **Display Form**: The System displays the form for create new session.
7. **Fill the Form**: The User proceeds to fill the form with the session details.
8. **Completion**: The process splits based on the User's final decision:
   - **Save**: If the User performs the click save action, the System executes the save and notify process. The activity path concludes at the final node.
   - **Cancel**: If the User performs the click cancel action, the System executes the clear the form and revert to calendar process. The activity path concludes at the final node.

7.4 Student - Request for new session

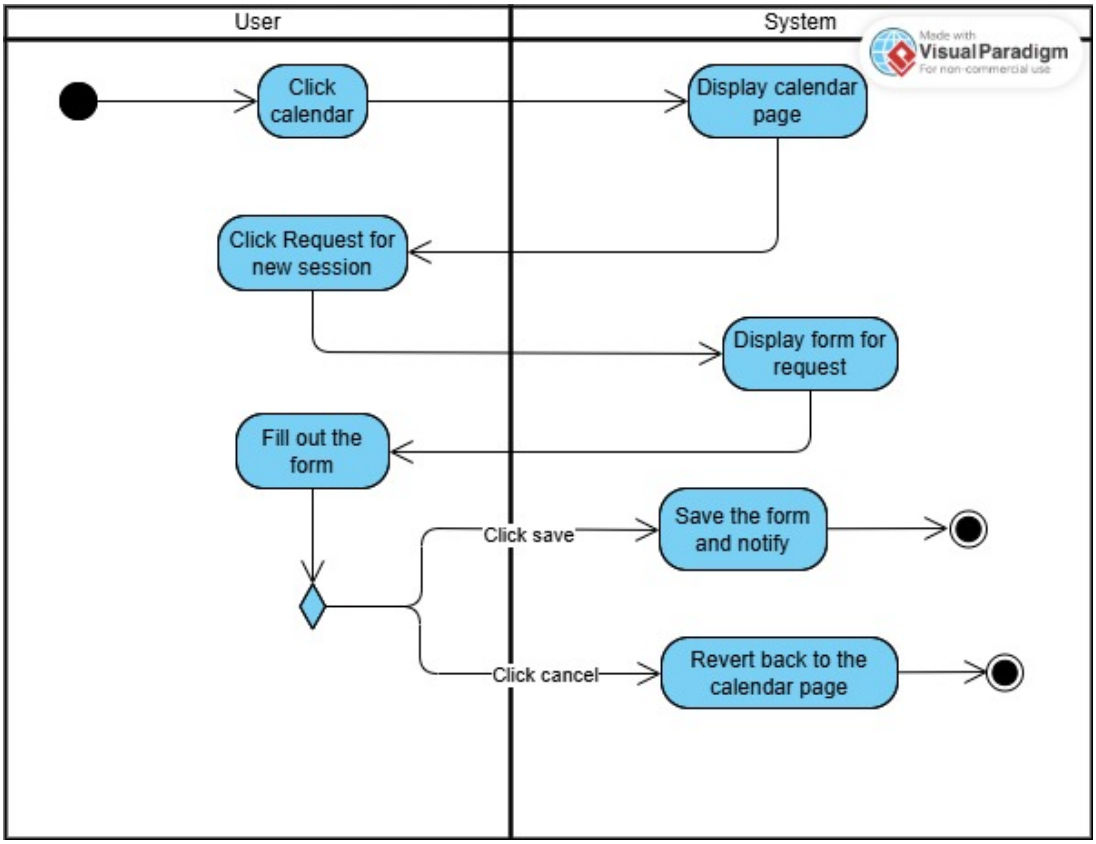


Figure 7.4: User - request new session

- **Request New session**

  1. **Click Calendar**: The User initiates the process by performing the click calendar action.
  2. **Display Calendar Page**: The System responds by displaying the calendar page.
  3. **Click Request**: The User selects the option to click request for new session.
  4. **Display Form**: The System displays the form for request.
  5. **Fill out the Form**: The User proceeds to fill out the form with the necessary details.
  6. **Completion**: The process splits based on the User's final decision:
     - **Save**: If the User performs the click save action, the System executes the save the form and notify process. The activity path concludes at the final node.
     - **Cancel**: If the User performs the click cancel action, the System executes the revert back to the calendar page process. The activity path concludes at the final node.

7.5 Tutor - Manage request

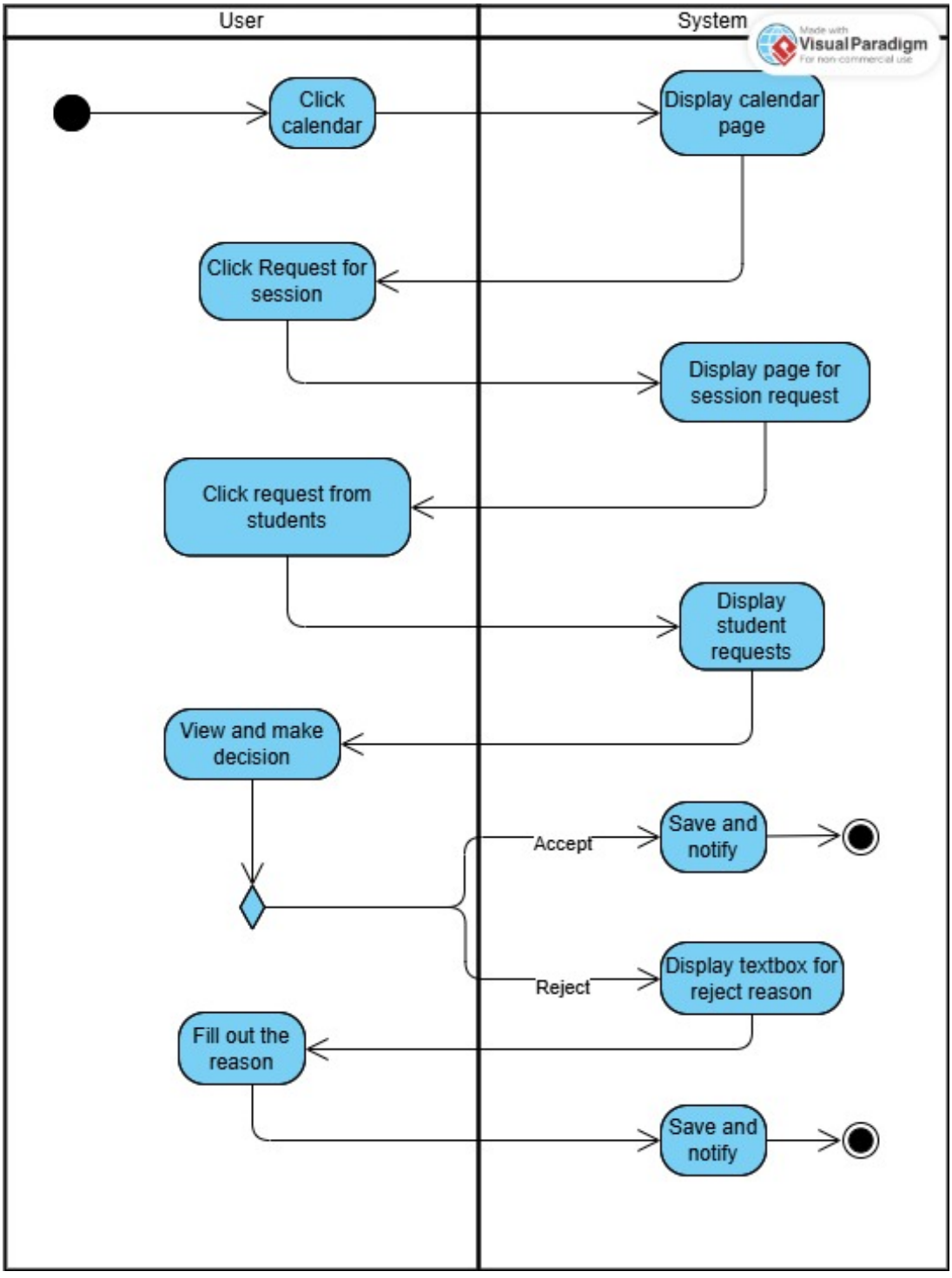


Figure 7.5: User - manage request

- **Manage Request**

1. **Click Calendar**: The User initiates the process by performing the click calendar action.
2. **Display Calendar**: The System responds by displaying the calendar page.
3. **Access Session Requests**: The User selects the option to click request for classroom.
4. **Display Request Page**: The System displays the page for session request.
5. **Select Student Requests**: The User proceeds to click request from students.
6. **Show Requests**: The System displays the pending student requests.
7. **Decision Making**: The User performs the view and make decision action regarding the request.
8. **Process Decision**: The workflow splits based on whether the User accepts or rejects the request:
   - **Accept**: If the decision is to accept, the System executes the save and notify process. The activity path concludes at the final node.
   - **Reject**: If the decision is to reject, the System displays a textbox for reject reason.
   - **Enter Reason**: The User is required to fill out the reason for the rejection.
   - **Finalize Rejection**: After the reason is provided, the System executes the save and notify process. The activity path concludes at the final node.

7.6 Tutor - Grading and rating

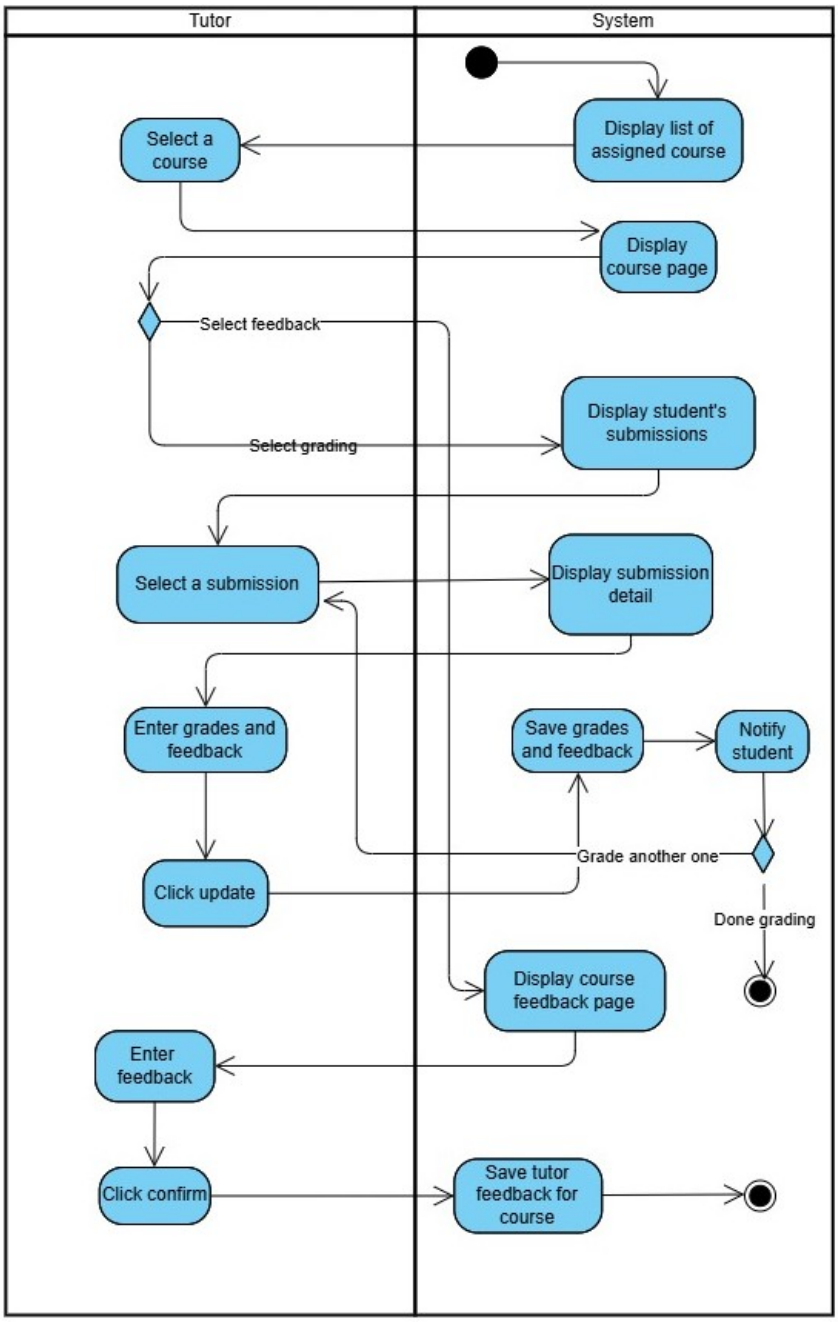


Figure 7.6: Tutor's grading and rating activity diagram

- **Grading**

  1. The system display list of assigned course

2. **Select Course**: The tutor then chooses a specific course by performing the select course action.
3. **Display Course Page**: The System will display the course page for the selected course.
4. **Select Grading**: From the course page, if the Tutor selects grading, the System proceeds to the grading page.
5. **Display Submissions**: The System displays a list of the student's submissions for that course.
6. **Select a Submission**: The Tutor then selects a submission from the list to grade.
7. **Display Submission Detail**: The System shows the details of the selected submission.
8. **Enter Grades**: The Tutor reviews the work and proceeds to enter grades and feedback.
9. **Update**: The Tutor finalizes the grading by clicking update.
10. **Save and Notify**: The System receives the update command, then saves the grades, feedback, and notifies the student.
11. If the tutor choose to grade another student, the loop comeback to select submission. If the tutor has done grading, the activity path concludes at the final node

- **Feedback**

1. If the tutor choose feedback, the System presents a course feedback page
2. **Enter Rating**: The Tutor interacts with this page to enter a rating and provide comments about the course.
3. **Confirm**: The Tutor finalizes their feedback by clicking confirm.
4. **Save Feedback**: The System then takes this input and saves the Tutor's feedback for the course.

7.7 Coordinator - Matching

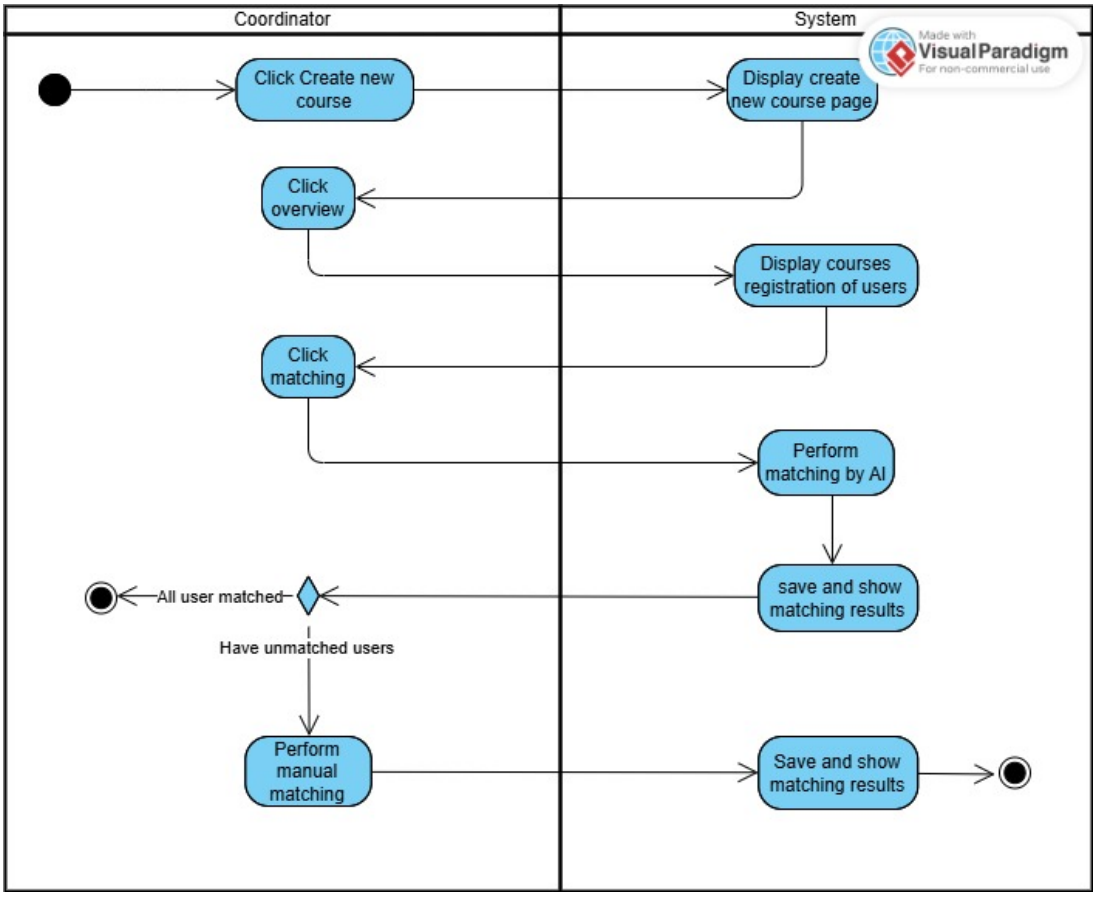


Figure 7.7: Coordinator - matching users

- **Matching Process**

  1. **Create Course**: The Coordinator initiates the workflow by performing the click Create new course action.
  2. **Display Page**: The System responds by showing the display create new course page.
  3. **View Overview**: The Coordinator proceeds to click overview.
  4. **Show Registrations**: The System displays the courses registration of users.
  5. **Initiate Matching**: The Coordinator selects the option to click matching.
  6. **AI Processing**: The System executes the perform matching by AI process.
  7. **Save Results**: The System proceeds to save and show matching results.

8. **Verification**: The process splits based on the completeness of the matching results:
   - **Complete**: If all users are matched, the activity path concludes at the final node.
   - **Incomplete**: If there are unmatched users, the Coordinator performs manual matching. The System then executes the save and show matching results process again, and the activity concludes at the final node.

7.8 Coordinator - Create new course

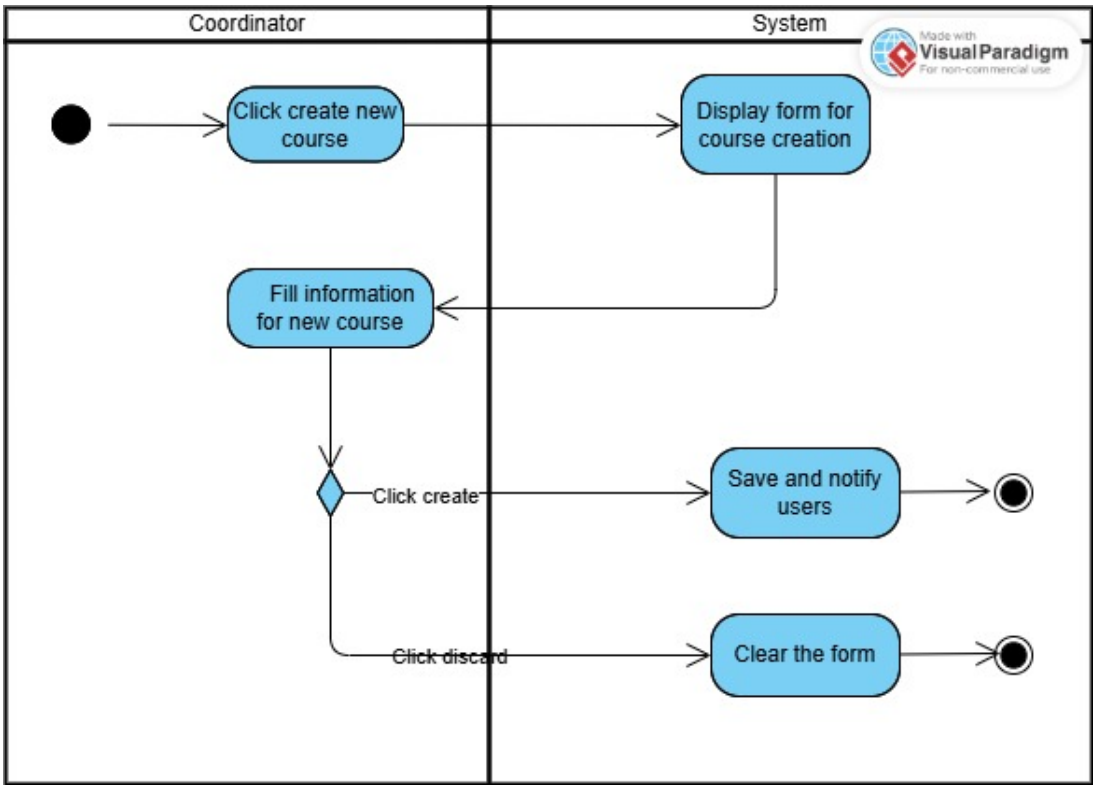


Figure 7.8: Coordinator - Create new course

- **Course Creation Process**

1. **Click Create**: The Coordinator initiates the workflow by performing the click create new course action.
2. **Display Form**: The System responds by showing the display form for course creation.
3. **Fill Information**: The Coordinator proceeds to fill information for new course.

4. **Completion**: The process splits based on the Coordinator's decision:
   - **Create**: If the Coordinator performs the click create action, the System executes the save and notify users process. The activity path concludes at the final node.
   - **Discard**: If the Coordinator performs the click discard action, the System executes the clear the form process. The activity path concludes at the final node.

7.9 Coordinator - Handle feedback and report

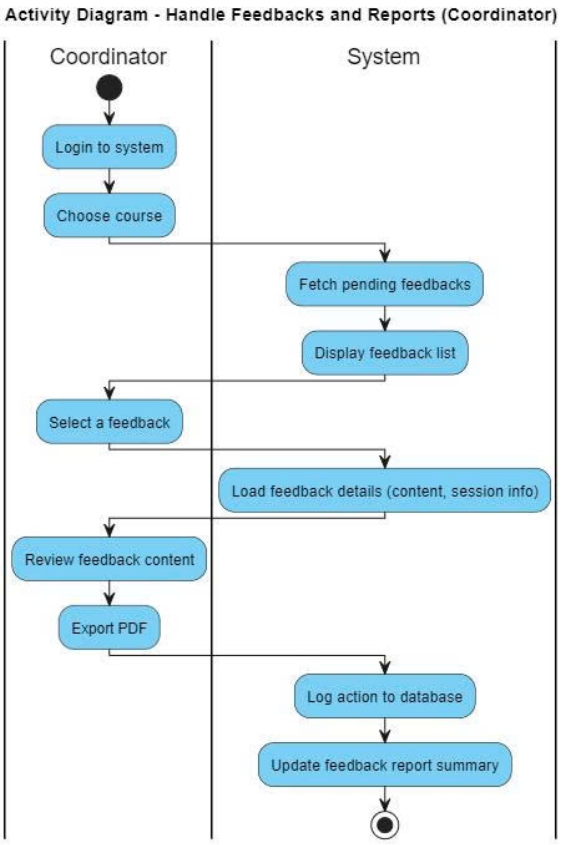


Figure 7.9: Coordinator - Handle feedback and report

- **Login to system:** The Coordinator logs into the system using their credentials.
- **Choose course:** The Coordinator selects the course for which feedback needs to be reviewed.

- **Fetch pending feedback** : The system retrieves all pending feedback related to the chosen course.
- **Display feedback list** : The system displays the list of pending feedback for the Coordinator to review.
- **Select a feedback** : The Coordinator selects a specific feedback from the displayed list.
- **Load feedback details:** The system loads the details of the selected feedback, including the content and session information.
- **Review feedback content** : The Coordinator reviews the content of the feedback to analyze the student's comments.
- **Export PDF** : After reviewing, the Coordinator can export the feedback details and reports in PDF format.
- **Log action to database** : The system saves the Coordinator's action (review, export, etc.) to the database for auditing and records.
- **Update feedback report summary** : The system updates the feedback report summary to reflect the reviewed feedback and associated actions.

8 State-chart

8.1 Statechart Diagram – Assignment Lifecycle

Purpose The **Assignment Lifecycle Statechart Diagram** models the various states an assignment passes through within the Tutor Support System (TSS). It captures both tutor and student interactions from assignment creation to review, including exceptional conditions such as late submissions, upload errors, and grading appeals.

Description The diagram defines how the system dynamically transitions between assignment states in response to user actions and system triggers. Each state represents a meaningful stage in the lifecycle of an assignment — ensuring clear traceability of submission and evaluation events across the platform.

States and Transitions

- **Created:** The tutor creates a new assignment. The system initializes assignment metadata such as title, description, deadline, and associated course.
- **Assigned:** The system assigns the created assignment to the respective students and dispatches notifications. Students can view assignment details and begin working toward submission. Two possible transitions emerge:
  - *Late:* If the deadline passes before submission, the assignment transitions to the `Overdue` state.
  - *Submit before deadline:* If the student successfully submits work, the system transitions to `Submitted`.
- **Submitted:** The student has submitted the assignment before or after the deadline. In the case of an upload error, the system permits re-uploading before marking the submission as final. Tutors then review and evaluate the submission, transitioning it to `Graded`.
- **Graded:** The tutor completes the review and assigns a score. If a student fails to submit, the system automatically assigns a grade of zero and terminates the process. A student may file an appeal, transitioning the assignment into `UnderReview`.
- **UnderReview:** The coordinator or tutor revisits the evaluation after an appeal, potentially updating the score. Once the review is finalized, the assignment lifecycle concludes.
- **Overdue:** The system marks assignments that missed the deadline. Students can still submit late, but the submission is flagged and may result in partial credit or administrative review.

Design Notes

- The diagram follows the **finite state machine (FSM)** paradigm, ensuring that every assignment has a clearly defined state at all times.
- Circular transitions such as `uploadError → Submitted` allow recovery from common user errors without data loss.
- Late submissions and appeals introduce realistic academic governance rules handled automatically by the system.

- The lifecycle closes only after all grading and review actions are completed, maintaining system consistency and audit traceability.

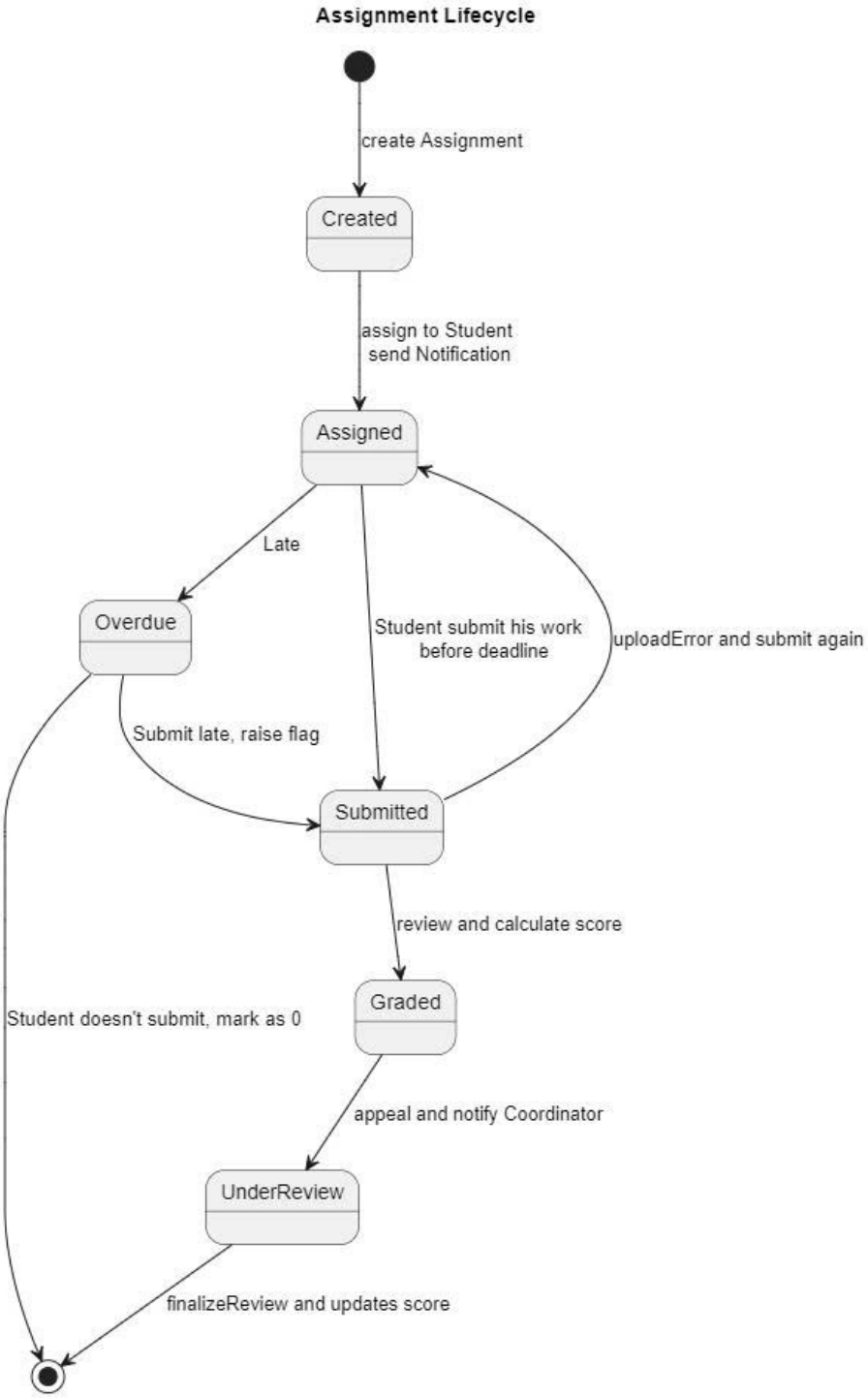


Figure 8.1: Statechart Diagram – Assignment Lifecycle in TSS

Figure

Summary This statechart encapsulates the behavioral flow of an assignment entity within the TSS ecosystem, bridging tutor evaluation workflows and student submission activities. By enforcing explicit transitions between states, the system maintains robust status tracking, late-handling mechanisms, and transparent feedback loops to ensure academic integrity and operational reliability.

8.2 Statechart Diagram – Session Lifecycle

Purpose The **Session Lifecycle Statechart Diagram** illustrates the dynamic behavior of a tutoring session within the Tutor Support System (TSS), capturing how each session transitions between states from scheduling to completion or cancellation. This model ensures traceability of all tutor–student interactions, reflecting real-world scheduling, confirmation, rescheduling, and attendance patterns.

Description Each session in TSS is instantiated after the matching process between a tutor and a student. The diagram tracks subsequent transitions based on confirmations, rescheduling requests, cancellations, and the passage of scheduled time. It provides a clear behavioral definition for managing active sessions and maintaining historical consistency in reports and analytics.

States and Transitions

- **Draft:** The initial state of a session created automatically after tutor–student matching. The session details (time, location, format) are editable by the tutor until confirmation. The tutor may cancel before confirming the session (*Cancel before confirmation*), which moves the session to `Cancelled`.
- **Confirmed:** When the tutor confirms the schedule, the session transitions to the `Confirmed` state. Both tutor and student receive notifications. At this stage:
  - Either party may cancel before the session begins (*Cancel before start*).
  - A rescheduling request triggers a transition to the `Rescheduling` state.
- **Rescheduling:** This state captures the process of updating a session's schedule upon a rescheduling request. Possible outcomes include:

  - The session returns to `Confirmed` once the schedule is successfully updated.
  - The session transitions to `Cancelled` if the session is cancelled during the rescheduling process.

- **Ongoing:** When the scheduled start time is reached, the system automatically transitions the session to the `Ongoing` state. It represents an active tutoring session currently in progress. Either party may still issue a cancellation request during the session, resulting in a transition to `Cancelled`.

- **Cancelled:** A terminal state indicating that the session has been cancelled at any point before or during its execution. The system updates records accordingly and notifies relevant users.

- **Completed:** Upon successful conclusion of a session, the system transitions it to `Completed`. Attendance records are finalized, and both tutor and student may proceed to evaluation and rating activities. Once completed, the session becomes immutable for reporting integrity.

Design Notes

- The lifecycle enforces distinct paths for **confirmation, rescheduling, execution, and cancellation**, preventing ambiguous session states.

- Transitions are governed by role-based triggers: tutors control schedule confirmation; both tutor and student may initiate cancellation or rescheduling requests.

- The addition of the `Rescheduling` state enhances realism by modeling schedule adjustments commonly occurring in tutoring workflows.

- Once a session reaches `Completed`, it becomes immutable to preserve attendance integrity for analytics and performance evaluation.

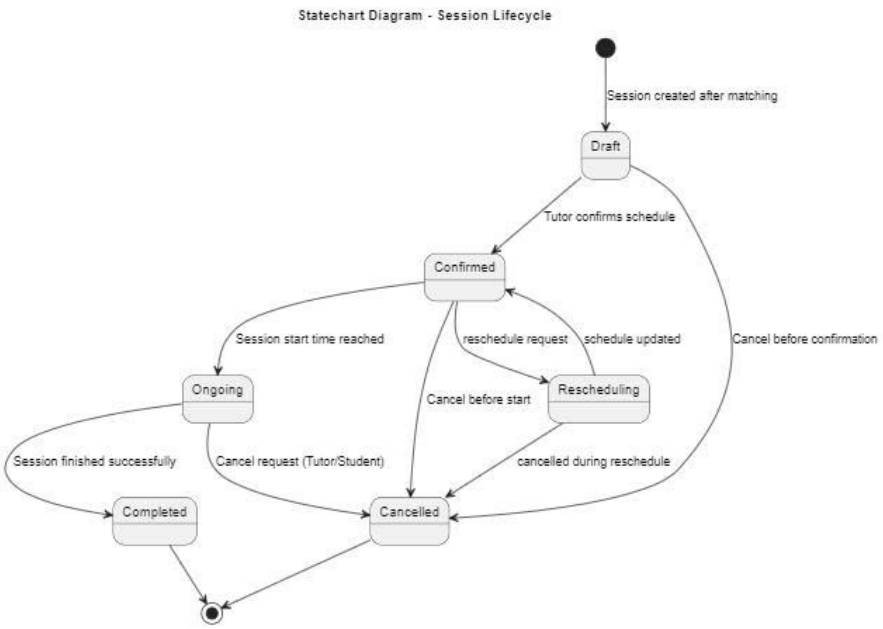


Figure 8.2: Statechart Diagram – Session Lifecycle in TSS

Figure

Summary This updated statechart defines the full behavioral lifecycle of tutoring sessions in the TSS environment. By formalizing transitions from creation to rescheduling, execution, and completion or cancellation, it ensures consistency between scheduling logic, calendar synchronization, and user notification flows. The inclusion of a dedicated `Rescheduling` state supports realistic session adjustments and enhances transparency and accountability in session management.

8.3 Statechart Diagram – Register Tutor Program Form

Purpose The **Register Tutor Program Form Statechart Diagram** models the user behavior and system reactions during the tutor registration process. It describes how a user interacts with the form—from initial access, drafting, auto-saving, validation, and finally official submission. This model captures realistic client–server interactions and ensures data integrity throughout the registration workflow.

Description When a user accesses the tutor registration form, the system retrieves any previously saved draft or initializes a new editable form. The statechart tracks transitions between editing, saving, validating, and submitting activities. It provides a clear

behavioral specification for handling draft preservation, error recovery, and submission confirmation.

States and Transitions

- **Idle:** The default resting state when the user has not yet interacted with the registration form. Transitions occur when:
  - The user opens the form, moving to `EditingDraft`.
  - A previously saved draft exists, which the system loads into `DraftSaved`.
- **EditingDraft:** The user is actively editing the form. Key transitions include:
  - The system periodically performs an *Auto-save draft*, moving to `DraftSaved`.
  - The user continues editing, remaining in the current state.
  - The user clicks *Submit*, initiating a transition to `Submitting`.
- **DraftSaved:** The form has been saved automatically or manually without being submitted. At this stage:
  - The user may resume editing, returning to `EditingDraft`.
  - A submission attempt triggers:
    - Successful validation: transition to `Submitting`.
    - Validation failure or unexpected errors: return to `EditingDraft`.
- **Submitting:** The user has requested submission, and the system is verifying and processing the form. Possible outcomes:
  - The server confirms the submission, transitioning to `Submitted`.
  - Validation errors occur, returning the user to `EditingDraft`.
- **Submitted:** The final state, reached once the server officially acknowledges form submission. The form becomes immutable, and the user is notified of successful registration.

Design Notes

- The statechart accommodates real-world form behaviors such as auto-saving, draft recovery, and validation loops.
- Error handling ensures users never lose progress, supporting iterative editing and correction.
- The separation between `DraftSaved` and `EditingDraft` models the distinction between live editing and preserved data states.
- The submission process is explicitly split into client-side initiation (`Submitting`) and server-side confirmation (`Submitted`).

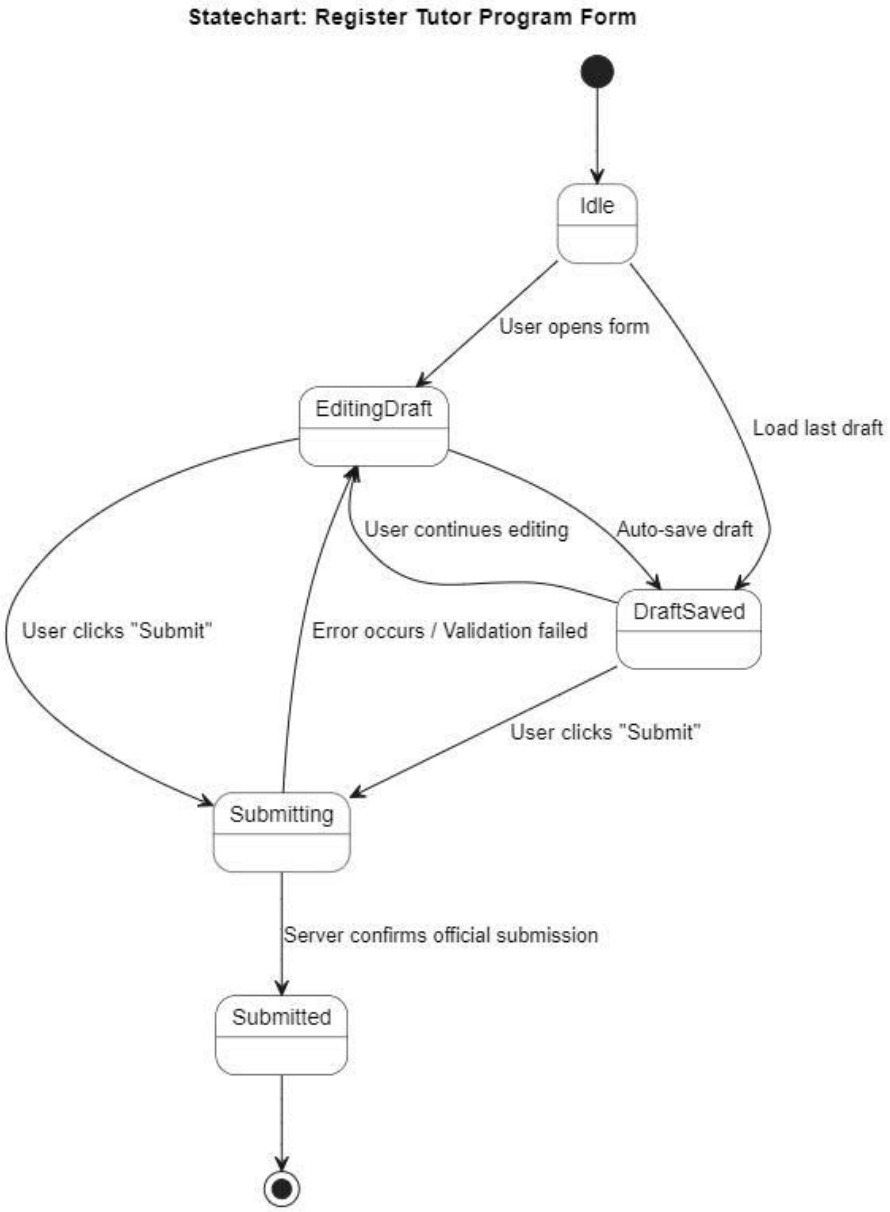


Figure 8.3: Statechart Diagram – Register Tutor Program Form

Figure

Summary This statechart provides a complete behavioral specification for the tutor registration workflow in the TSS. It formalizes transitions between editing, saving, validating, and submitting, ensuring both user convenience and system robustness. The model supports draft recovery, error resilience, and reliable submission confirmation, offering a seamless registration experience.

9 Matching Module

9.1 Current Approach: Hard Matching Algorithm

Purpose The current implementation adopts a **Hard Matching Algorithm** to assign suitable tutors to students based on strict one-to-one compatibility rules. A tutor is considered a valid match only when **all required criteria** align with the student's preferences.

Description Both students and tutors provide registration information including subjects, languages, session types (online/offline), and preferred locations. The system evaluates these attributes using deterministic constraints. If any constraint fails, the tutor is immediately excluded from the candidate list.

Matching Criteria

- **Subject Compatibility** The tutor must teach at least one subject selected by the student:

$$\exists\, t \in TutorSubjects,\ \exists\, s \in StudentSubjects : t.id = s.id$$

- **Language Compatibility** The tutor must support at least one language preferred by the student:

$$\exists\, t \in TutorLanguages,\ \exists\, s \in StudentLanguages : t.id = s.id$$

- **Session Type Compatibility** The tutor must provide the student's desired session type (online or offline):

$$\exists\, t \in TutorSessionTypes,\ \exists\, s \in StudentSessionTypes : t.id = s.id$$

- **Location Constraint (Offline Only)** If the student prefers offline learning, tutor and student must share at least one location:

$$StudentOffline \Rightarrow \exists\, t \in TutorLocations,\ \exists\, s \in StudentLocations : t.id = s.id$$

Algorithm Flow The matching process is implemented using three functions:

1. `hardMatch(student, tutor)` Evaluates all constraints and returns `true` only when all conditions pass.

2. `getMatchedTutors(student)` Filters the tutor list by applying the hard match test.
3. `matchAllStudents()` Performs matching for every registered student and aggregates results into:

$$(StudentName,\ MatchedTutors)$$

Advantages

- Simple, predictable, and easy to verify.
- Ensures high compatibility because all constraints must match.
- Suitable for early-phase system deployment.

Limitations

- Too strict: many tutors may be excluded even if partially suitable.
- No ranking or scoring mechanism.
- Does not consider tutor workload, availability, or rating.

9.2 Future Extension: AI-Assisted Matching for Special Requests

Motivation While the current system relies on strict rule-based *Hard Matching* to ensure compatibility between students and tutors (subjects, languages, session types, and locations), many students also provide a `specialRequest` field describing their personal learning needs. These requests are often diverse, unstructured, and written in natural language, e.g.:

- "Muốn luyện kỹ năng giải đề và chấm bài mẫu trước kỳ thi cuối khoá."
- "Em bị mất gốc thuật toán và muốn học lại từ đầu."
- "Cần ôn tập Big-O và các thuật toán sắp xếp."
- "Muốn được hướng dẫn làm assignment tuần này."

Such requests cannot be handled using the existing Hard Matching logic. Therefore, an AI-assisted extension is proposed to interpret, classify, and refine these requests to support more personalized tutor recommendations.

Diversity of Special Requests Student requests may belong to various categories, including but not limited to:

- **Exam preparation** (e.g., luyện đề, chấm bài mẫu, ôn thi cuối khoá)
- **Foundational learning** (e.g., "mất gốc", học lại từ đầu)
- **Topic-specific learning** (Big-O, Sorting, Recursion, OOP concepts)
- **Homework or assignment support**
- **Learning strategies or study skills** (kỹ năng phân tích đề, tối ưu code)
- **Teaching style preferences** (muốn dạy chậm, nhiều ví dụ)
- **Scheduling constraints** (chỉ học được cuối tuần)
- **Advanced training** (ICPC, design patterns)

The high variability of these requests motivates the need for an AI-driven interpretation layer.

Proposed AI Processing Pipeline To enrich the matching results without altering the existing Hard Matching rules, the system introduces an **AI-assisted Special Request Pipeline**. Hard constraints remain mandatory, while AI provides additional ranking and interpretation.

1. **Hard Matching (Rule-Based)** The system first applies strict constraints: subjects, languages, session types, and locations (for offline). If any required condition fails, the tutor is automatically excluded.

2. **AI Intent Classification** The model identifies the category of the request using NLP techniques. The intent label is chosen from:

   $$\text{Intent} \in \{\text{ExamPrep}, \text{Foundational}, \text{TopicLearning}, \text{AssignmentSupport}, \text{Preference}, \text{Scheduling}, \text{Adva}$$

3. **Topic Extraction** The AI extracts specific concepts from the request:

   $$\text{Topics} = \{\text{Big-O}, \text{Sorting}, \text{Recursion}, \text{OOP}, \dots\}$$

   This enables finer-grained matching between student needs and tutor skills.

4. **Tutor Capability Mapping** Each tutor profile is extended with skill tags, experience scores, and teaching preferences. The system evaluates how well a tutor aligns with the extracted topics.

5. **AI-Based Ranking (Soft Matching Layer)** Tutors who pass Hard Matching are scored using:

$$\text{AIScore} = f(\text{TopicFit}, \text{Experience}, \text{DifficultyFit}, \text{StyleFit})$$

   where:

   - **TopicFit** — the degree to which the tutor's expertise aligns with the topics extracted from the student's `specialRequest`.
   - **Experience** — the tutor's prior experience handling similar learning needs, based on past sessions or performance records.
   - **DifficultyFit** — the tutor's capability to address the academic difficulty level implied in the request.
   - **StyleFit** — the compatibility between the tutor's teaching style and the student's expressed learning preferences.

   The final ranking score is computed as:

$$\text{FinalScore} = 1.0 \cdot \text{HardMatch} + 0.3 \cdot \text{AIScore}$$

6. **Adaptive Feedback Loop** Student feedback, tutor ratings, and session outcomes continuously update the dataset, enabling periodic retraining of the AI model to improve accuracy.

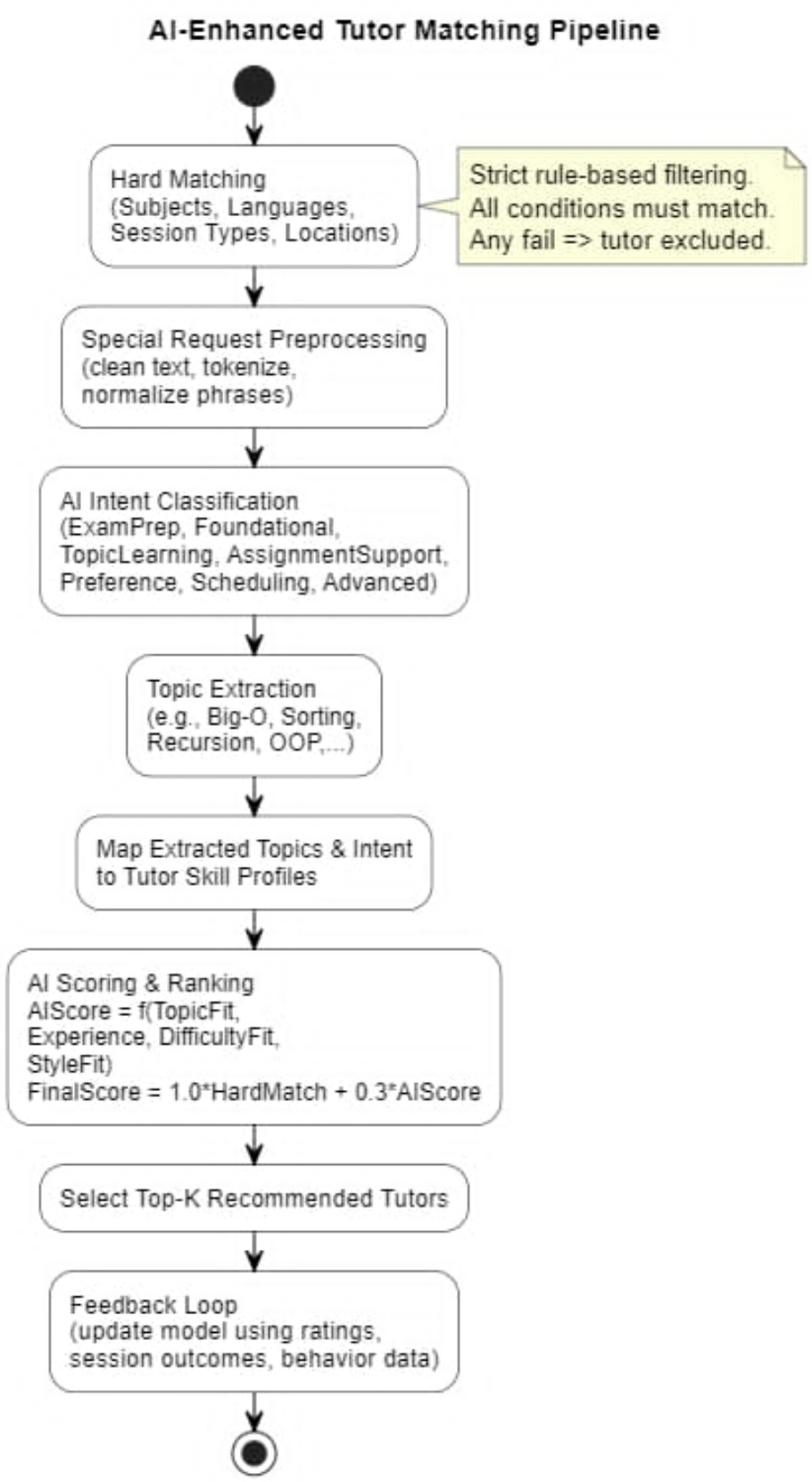


Figure 9.1: AI Matching Pipeline

Figure

Expected Benefits

- Enhances personalization without compromising strict compatibility rules.
- Helps students refine overly broad or ambiguous requests.
- Identifies tutors with the most relevant skills for specific learning needs.
- Improves matching success rate by ranking tutors based on deeper semantic understanding.
- Enables long-term evolution into a full recommendation system.

Summary The proposed AI-assisted matching extension preserves the reliability of Hard Matching while adding intelligent handling of unstructured `specialRequest` data. This hybrid approach ensures both strict rule compliance and personalized tutor recommendations, leading to a more flexible, scalable, and user-centric matching system.

10 Class diagram

10.1 View

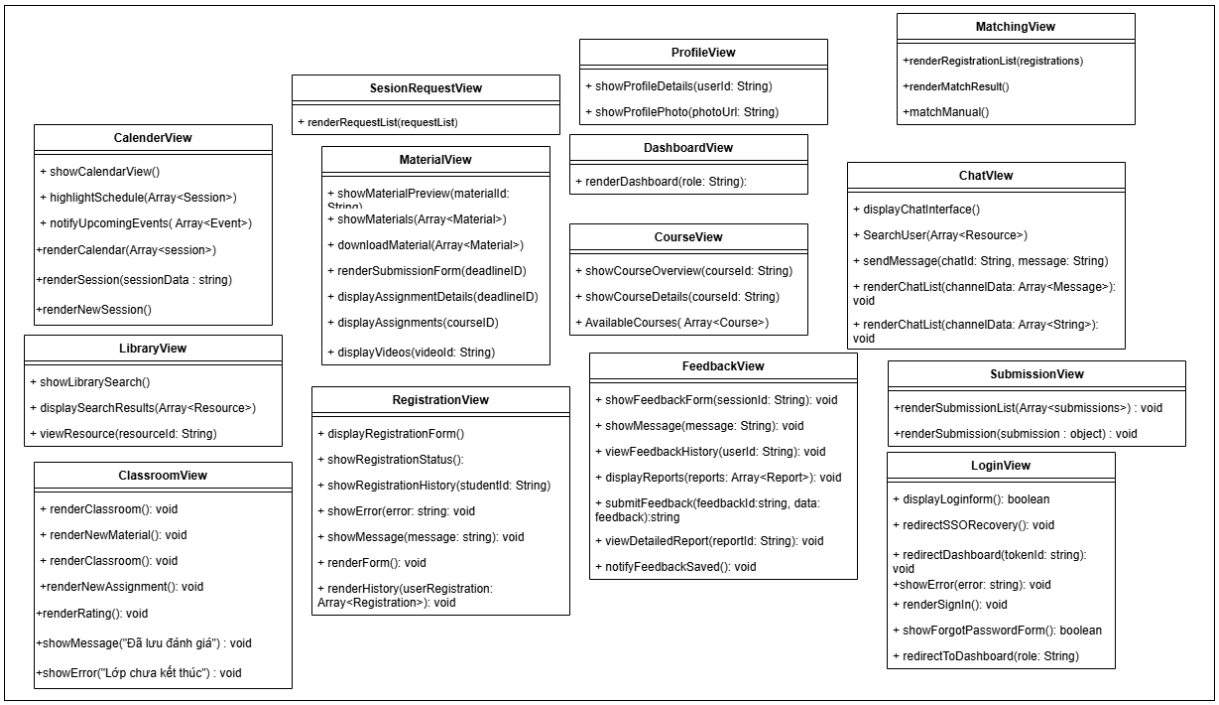


Figure 10.1: View-Group

This diagram represents the **View** layer for an online educational or tutoring platform. The View component is responsible for rendering the user interface and presenting data in a user-friendly and accessible manner. It is structured around several key domains: user interfaces for account management, course and classroom dashboards, content displays, scheduling user interactions, communication tools, and feedback forms. The View layer ensures seamless interaction between users and the application's functionalities while providing an intuitive and engaging experience.

1. **Class: LoginView**

   - **Purpose**: Responsible for displaying the login form to users, handling user interactions, and other related functionalities.
   - **Methods**:
     - `displayLoginForm()`: Displays the user login form.
     - `showForgotPasswordForm()`: Displays the "Forgot Password" form for users to recover their account.

- `redirectSSORecovery()`: Handles the redirection for Single Sign-On (SSO) recovery.
- `redirectDashboard(tokenId: string)` : Redirects the user to the dashboard using a security token.
- `showError(error: string)` : Displays an error message to the user.
- `renderSignIn()` : Renders the main sign-in interface.
- `showForgotPasswordForm()` : Displays the "Forgot Password" form and returns a status.
- `redirectToDashboard(role: String)` : Redirects the user to the appropriate dashboard based on their role.

2. **Class: CalendarView**

- **Purpose**: Responsible for displaying the Calendar view to users, handling interactions, and rendering session-related information.
- **Method**:
  - `showCalendarView()`: Displays the calendar interface to the user.
  - `highlightSchedule(Array<Session>)`: Highlights scheduled sessions in the calendar based on the provided session data.
  - `notifyUpcomingEvents(Array<Event>)`: Notifies users about upcoming events using the provided list of events.
  - `renderCalendar(Array<session>)` : Renders the calendar with a given set of sessions.
  - `renderSession(sessionData: string)` : Renders the details of a specific session.
  - `renderNewSession()` : Displays the interface for creating a new session.

3. **Class: CourseView**

- **Purpose**: Responsible for displaying course-related information and providing an interface for users to explore and interact with courses.
- **Methods**:
  - `showCourseOverview(courseId: String)`: Displays a brief overview of the selected course specified by its ‘courseId‘.

- `showCourseDetails(courseId: String)`: Displays detailed information about the course, including syllabus, instructor information, and course requirements.
- `availableCourses(courses: Array<Course>)`: Displays a list of available courses to the user based on the provided array of course objects.

4. **Class: ProfileView**

   - **Purpose**: Manages the display and interaction of user profile information, allowing users to view and update their personal details.
   - **Methods**:
     - `showProfileDetails(userId: String)`: Displays detailed information of the user's profile based on the provided 'userId', including personal information, contact details, and account settings.
     - `showProfilePhoto(photoUrl: String)`: Displays the user's profile photo using the specified 'photoUrl'.

5. **Class: FeedbackView**

   - **Purpose**: Responsible for managing the display and interaction of feedback-related interfaces, including providing forms for users to give feedback, viewing feedback history, and notifying users of feedback submissions.
   - **Methods**:
     - `showFeedbackForm(sessionId: String)`: Displays the feedback form for a specific session, identified by the provided 'sessionId'. Allows users to give feedback on the session.
     - `showMessage(message: String)` : Displays a general message to the user
     - `viewFeedbackHistory(userId: String)`: Displays the history of feedback provided by a specific user, identified by the given 'userId'.
     - `displayReports(reports: Array<Report>)` : Displays a list of reports
     - `submitFeedback(feedbackId: string, data: feedback)` : Submits user feedback and returns a confirmation
     - `viewDetailedReport(reportId: String)` : Displays detailed information for a specific report
     - `notifyFeedbackSaved()`: Shows a notification or message to the user indicating that their feedback has been successfully saved.

6. **Class: MaterialView**

   - **Purpose**: Responsible for managing the display and interaction of educational materials, including previews, material downloads, and video content.
   - **Methods**:
     - `showMaterialPreview(materialId: String)`: Displays a preview of a specific material identified by the 'materialId', allowing users to get an overview of the content before interacting further.
     - `showMaterials(materials: Array<Material>)`: Displays a list of available materials from the provided array of Material objects, enabling users to browse and select resources.
     - `downloadMaterial(materials: Array<Material>)`: Provides the functionality for users to download one or more materials from the provided array of Material objects.
     - `renderSubmissionForm(deadlineID)` : Renders the form for submitting an assignment
     - `displayAssignmentDetails(deadlineID)` : Displays detailed information about a specific assignment `displayAssignments(courseID)` : Displays a list of assignments for a specific course
     - `displayVideos(videoId: String)`: Displays video content associated with the given 'videoId', allowing users to view educational or tutoring videos directly within the interface.

7. **Class: MatchingView**

   - **Purpose**: Responsible for managing the interfaces related to matching tutors and students, including rendering registration lists and match results.
   - **Methods**:
     - `renderRegistrationList(registrations)`:Renders a list of registrations for matching
     - `renderMatchResult()`: Displays the result of a matching process.
     - `matchManual()`: Initiates the manual matching interface or process.

8. **Class: RegistrationView**

- **Purpose**: Responsible for managing the user registration interface, including displaying forms for new registrations, showing registration status, and viewing past registration history.
- **Methods**:
  - `displayRegistrationForm()`: Displays the registration form to allow users to input their details and register for the platform.
  - `showRegistrationStatus()`: Displays the current registration status, such as whether the registration is completed successfully, pending approval, or incomplete.
  - `showRegistrationHistory(studentId: String)`: Displays the registration history for a given student, identified by the 'studentId'. This includes past registrations or enrolled courses.
  - `showError(error: string)` : Displays a registration-related error message
  - `showMessage(message: string)` : Shows a notification or message to the user
  - `renderForm()` : Renders the main registration form structure
  - `renderHistory(userRegistration: Array<Registration>)` : Renders the user's registration history

9. **Class: DashboardView**

- **Purpose**: Responsible for displaying the main dashboard for users, customized based on their assigned role (e.g., student, tutor, or admin). The dashboard provides access to key features and relevant information.
- **Methods**:
  - `renderDashboard(role: String)`: Renders a personalized dashboard based on the user's role. For example, students may see their courses and progress, while tutors may see their class schedules and assigned students.

10. **Class: LibraryView**

- **Purpose**: Responsible for managing the interface to search, browse, and view resources within the library, providing users with tools to explore educational materials effectively.
- **Methods**:

- `showLibrarySearch()`: Displays the search interface for users to query the library and look for specific educational resources.
- `displaySearchResults(resources: Array<Resource>)`: Displays a list of search results based on the query, using provided data in the form of an array of 'Resource' objects.
- `viewResource(resourceId: String)`: Displays detailed information about a specific resource, identified by the 'resourceId'. This may include preview, metadata, and download options.

11. **Class: ChatView**

- **Purpose**: Responsible for managing the user interface for chat functionality, including sending messages, searching for users, and displaying chat interactions in real-time.
- **Methods**:
  - `displayChatInterface()`: Displays the main chat interface for users, allowing them to view ongoing conversations and navigate between different chats.
  - `searchUser(users: Array<Resource>)`: Enables users to search for other participants or contacts within the user list, provided as an array of 'Resource' objects.
  - `sendMessage(chatId: String, message: String)`: Sends a message to a specific chat, identified by its 'chatId'. The method takes the message content as a parameter and processes it for delivery.
  - `renderChatList(channelData: Array<Message>)` : Renders the chat list from message objects
  - `renderChatList(channelData: Array<String>)` : Renders the chat list from string data

12. **Class: SubmissionView**

- **Purpose**: Manages the user interface for viewing assignment submissions, including lists of submissions and details of a single submission.
- **Methods**:
  - `renderSubmissionList(Array<submissions>)`: Displays a list of all submissions for an assignment.

- `renderSubmission(submission: object)`: Displays the details of a single submission.

13. **Class: SessionRequestView**

- **Purpose**: Manages the display of tutoring session requests.
- **Methods**:
  - `renderRequestList(requestList)`: Renders and displays a list of session requests.

14. **Class: ClassroomView**

- **Purpose**: Manages the main classroom interface, allowing for the rendering of the classroom itself and interfaces for adding new materials or assignments.
- **Methods**:
  - `renderClassroom()`: Renders the main view of the virtual classroom.
  - `renderNewMaterial()` : Renders the form or interface for adding new material
  - `renderNewAssignment()` : Renders the form or interface for creating a new assignment
  - `renderRating()` : Renders the main view for the rating of the classroom.
  - `showMessage("Đã lưu đánh giá")` : Show the corresponding message to the user.
  - `showError("Lớp chưa kết thúc")` : Display an error with the corresponding message to the user.

10.2 Controller

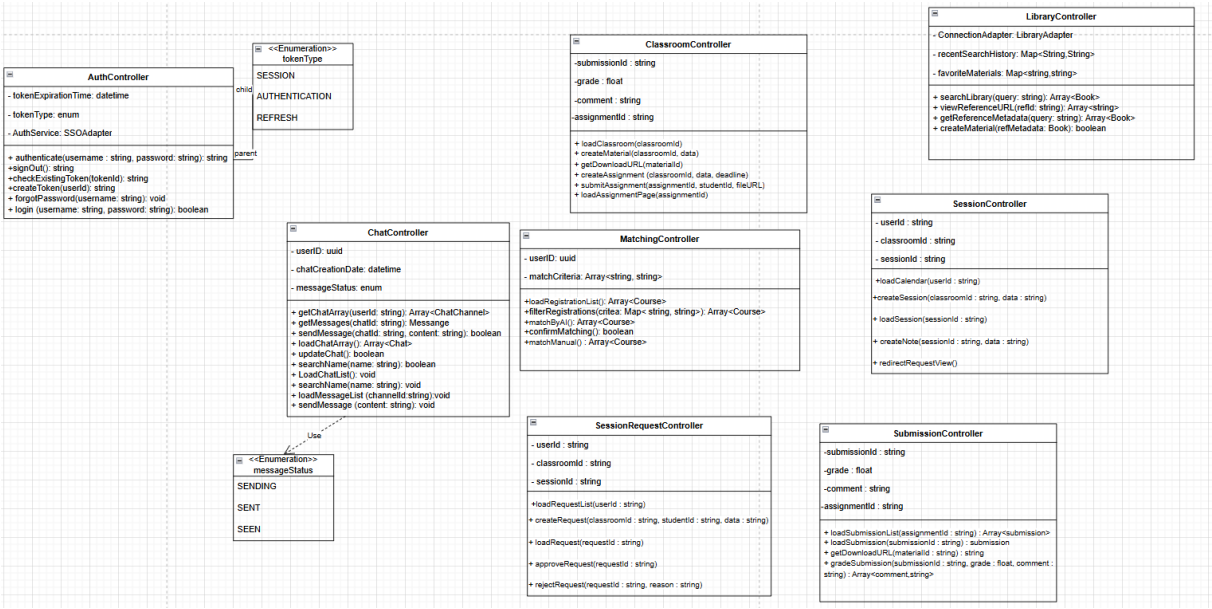


Figure 10.2: Controller-Group

This diagram illustrates the **Controller** layer of the system, focusing on managing user sessions, classroom interactions, content delivery, and administrative workflows. These controllers serve as the bridge between the user interface and the backend business logic, handling data processing, state management, and the routing of requests for authentication, communication, and educational activities.

1. **AuthController**: Manages user authentication, token generation, and account security.

   - **Purpose**: To handle the security gateway of the application, ensuring users are verified and their sessions are managed correctly via tokens.
   - **Attributes**
     - **tokenExpirationTime : datetime**: Stores the timestamp indicating when the current authentication token becomes invalid.
     - **tokenType : enum**: Specifies the category of the token (e.g., SESSION, AUTHENTICATION, REFRESH) used for validation.
     - **AuthService : SSOAdapter**: A reference to the Single Sign-On (SSO) adapter used for external authentication services.
   - **Methods**

- **authenticate(username: string, password: string): string**: Validates user credentials and returns an authentication token upon success.
- **signOut(): string**: Logs the user out and invalidates the active session.
- **checkExistingToken(tokenId: string): string**: Verifies the validity of a specific token ID to maintain session continuity.
- **createToken(userId: string): string**: Generates a new access token for a specific user.
- **forgotPassword(username: string): void**: Initiates the password recovery process for the specified user.
- **login(username: string, password: string): boolean**: Performs the login action, returning true if the credentials are correct.

2. **ChatController**: Facilitates real-time communication between users, managing chat logs and message statuses.

- **Purpose**: To handle the storage, retrieval, and transmission of messages within the platform's chat feature.
- **Attributes**
  - **userID : uuid**: The unique identifier of the user engaged in the chat.
  - **chatCreationDate : datetime**: The timestamp marking when a specific chat thread was initiated.
  - **messageStatus : enum**: Tracks the delivery state of messages (SENDING, SENT, SEEN).
- **Methods**
  - **getChatArray(userId: string): Array<ChatChannel>**: Retrieves a list of chat channels available for a specific user.
  - **getMessages(chatId: string, content: string): boolean**: Fetches the message history for a specific chat ID.
  - **sendMessage(chatId: string, content: string): boolean**: Dispatches a message to a specific chat room and returns the success status.
  - **loadChatArray(): Array<Chat>**: Loads the collection of chat objects into the view.
  - **updateChat(): boolean**: Refreshes the chat state or metadata.
  - **searchName(name: string): boolean**: Searches for a chat participant or group by name.

  - **LoadChatList(): void**: Initiates the loading sequence for the list of chats.
  - **loadMessageList(channelId: string): void**: Loads the specific messages contained within a selected channel.

3. **ClassroomController**: Manages the core educational workspace, including assignments and materials.

   - **Purpose**: To orchestrate the activities within a virtual classroom, handling material creation, assignment distribution, and classroom loading.
   - **Attributes**
     - **submissionId : string**: The identifier for student submissions within the classroom.
     - **grade : float**: The numerical score assigned to a submission.
     - **comment : string**: Feedback text provided by the instructor.
     - **assignmentId : string**: The unique identifier for a specific assignment.
   - **Methods**
     - **loadClassroom(classroomId)**: Initializes and retrieves data for a specific classroom environment.
     - **createMaterial(classroomId, data)**: Adds new educational resources to the classroom.
     - **getDownloadURL(materialId)**: Generates a link to download specific classroom materials.
     - **createAssignment(classroomId, data, deadline)**: Publishes a new assignment with associated data and a due date.
     - **submitAssignment(assignmentId, studentId, fileURL)**: Processes the submission of an assignment by a student.
     - **loadAssignmentPage(assignmentId)**: Prepares the view for a specific assignment details page.

4. **MatchingController**: Automates and manages the pairing of students with courses or tutors.

   - **Purpose**: To handle the logic regarding the registration and matching process, utilizing both manual inputs and AI algorithms.
   - **Attributes**

  - **userID : uuid**: The ID of the user triggering the matching process.
  - **matchCriteria : Array<string, string>**: A set of parameters used to filter and match courses or tutors.
- **Methods**
  - **loadRegistrationList(): Array<Course>**: Retrieves the list of current course registrations.
  - **filterRegistrations(critea: Map<string, string>): Array<Course>**: Filters available courses based on specific criteria maps.
  - **matchByAI(): Array<Course>**: Executes an AI algorithm to suggest optimal course or tutor matches.
  - **confirmMatching(): boolean**: Finalizes the selected matches in the system.
  - **matchManual(): Array<Course>**: Allows for manual selection and matching of courses.

5. **LibraryController**: Handles the search and retrieval of educational resources from the library.
   - **Purpose**: To provide an interface for users to search, view, and manage library materials and references.
   - **Attributes**
     - **ConnectionAdapter : LibraryAdapter**: The interface used to connect to the external or internal library database.
     - **recentSearchHistory : Map<string, String>**: Caches the user's recent search terms.
     - **favoriteMaterials : Map<string, string>**: Stores a list of materials marked as favorites by the user.
   - **Methods**
     - **searchLibrary(query: string): Array<Book>**: Queries the library database for books matching the input string.
     - **viewReferenceURL(refId: string): Array<string>**: Retrieves access URLs for specific reference materials.
     - **getReferenceMetadata(query: string): Array<Book>**: Fetches detailed metadata for books or materials.

  - **createMaterial(refMetadata: Book): boolean**: Adds a new material entry to the library system.

6. **SessionController**: Manages the scheduling and lifecycle of individual learning sessions.
   - **Purpose**: To handle the organization of sessions, including calendar views and note-taking functionality.
   - **Attributes**
     - **userId : string**: The user associated with the session.
     - **classroomId : string**: The classroom context for the session.
     - **sessionId : string**: The unique identifier for the session instance.
   - **Methods**
     - **loadCalendar(userId: string)**: Retrieves the schedule/calendar for a specific user.
     - **createSession(classroomId: string, data: string)**: Initializes a new session within a classroom.
     - **loadSession(sessionId: string)**: Fetches the details of an active or past session.
     - **createNote(sessionId: string, data: string)**: Saves notes associated with a specific session.
     - **redirectRequestView()**: Navigates the user to the session request interface.
7. **SessionRequestController**: Handles the workflow for requesting and approving new sessions.
   - **Purpose**: To manage the administrative side of scheduling, allowing for the creation, approval, and rejection of session requests.
   - **Attributes**
     - **userId : string**: The ID of the requester or approver.
     - **classroomId : string**: The target classroom for the request.
     - **sessionId : string**: The ID of the session being requested.
   - **Methods**

- **loadRequestList(userId: string)**: Displays a list of pending session requests for a user.
- **createRequest(classroomId: string, studentId: string, data: string)**: Submits a new request for a session.
- **loadRequest(requestId: string)**: Retrieves details of a specific request.
- **approveRequest(requestId: string)**: Grants approval for a session request.
- **rejectRequest(requestId: string, reason: string)**: Denies a session request and records the reason.

8. **SubmissionController**: Dedicated to the grading and management of student work.

   - **Purpose**: To provide specific functionality for handling assignment submissions, including retrieval and grading.
   - **Attributes**
     - **submissionId : string**: The unique ID of the submission.
     - **grade : float**: The numerical grade given.
     - **comment : string**: Feedback text.
     - **assignmentId : string**: The assignment this submission belongs to.
   - **Methods**
     - **loadSubmissionList(assignmentId: string): Array<submission>**: Fetches all submissions for a given assignment.
     - **loadSubmission(submissionId: string): submission**: Retrieves a specific submission entity.
     - **getDownloadURL(materialId: string): string**: Gets the URL to download the submitted file.
     - **gradeSubmission(submissionId: string, grade: float, comment: string): Array<comment, string>**: Updates the submission with a grade and feedback.

10.3 Model

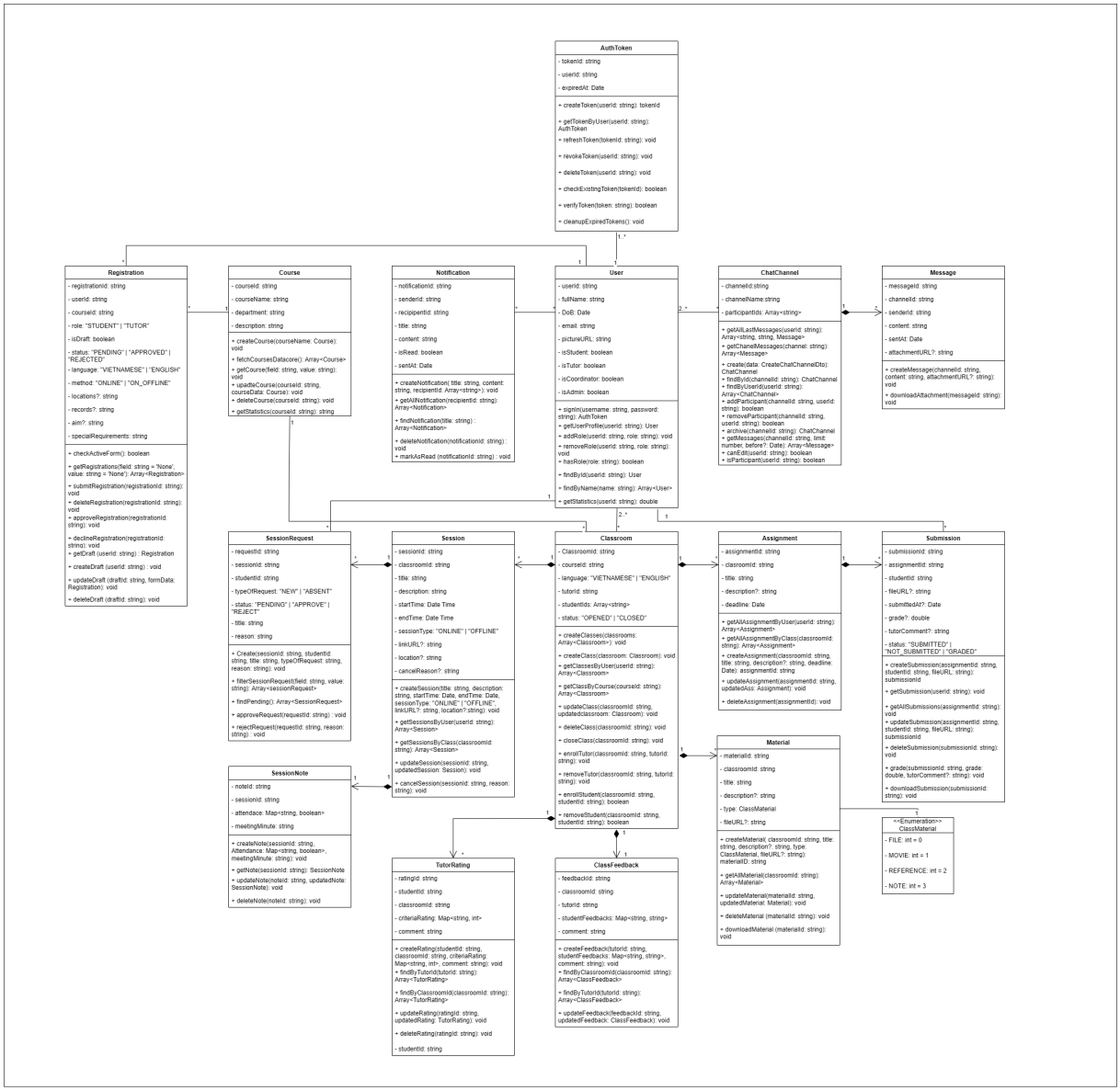


Figure 10.3: Model class diagram

This diagram service illustrates the core service layer for an online educational or tutoring platform. This layer is responsible for managing the application's data, business logic, and core functionalities. It is structured around several key domains: user management, course and classroom organization, content delivery, scheduling, communication, and feedback.

1. **User Management and Authentication**

   Class User : Primary entity representing an individual

- **Purpose** : To store profile information and define the roles of users, such as students, tutors, or coordinators.
- **Attributes** : Includes userId, fullName, email, pictureURL, and boolean flags like isStudent, isTutor, and isCoordinator to manage roles
- **Methods** : Contains methods for core user actions like signIn(...), findById(...), finding users (findBy...), managing roles (addRole, hasRole), and password verification
- **Relationships** : It's a central hub connected to nearly every other part of the service, including AuthToken, Registration, Notification, and Submission

Class AuthToken : Primary entity representing an authorize token

- **Purpose** : To manage user sessions and API security through authentication tokens
- **Attributes** : Stores the token, their expiresAt date, the associated userId, and a revoked status
- **Methods** : Provides functionality to create, verify, refresh, and revoke tokens, ensuring secure and stateful user sessions. It also includes a cleanupExpiredTokens() method for maintenance
- **Relationships** : A User can have one or more AuthTokens

2. **Course and Classroom Management**
   Class Course : Represents a high-level subject or curriculum

- **Purpose** : To define a course offering
- **Attributes** : Contains courseId, courseName, description, language, and department (linking it to an existing department)
- **Methods** : Includes standard Create, Get, Update, Delete (CRUD) operations for managing the course catalog
- **Relationships** : A Course serves as a template for one or more Classrooms

Class Classroom : Represents a specific instance of a Course

- **Purpose** : To bring together a specific tutor, a group of students, and learning materials for a Course

- **Attributes** : Includes classroomId, courseId, tutorId, a list of studentIds, and a status and other meta data of a classroom
- **Methods** : Manages the lifecycle of a class, including creating the class, enrolling/removing students, and closing the classroom
- **Relationships** : A Classroom belongs to one Course and contains multiple Sessions, Assignments, and Materials along with other ratings associate with tutor and students accordingly.

Class Registration : Represents a specific instance of a Registration

- **Purpose** : To handle a student's request to enroll in a course
- **Attributes** : Stores registrationId, userId, courseId, and a status (e.g., PENDING, APPROVED), applied role ("TUTOR" | "STUDENT"), languages, specialRequirements with other optional input as locations, records for tutor role or grade aim for students. It can also capture preferences like Location (ONLINE/OFFLINE)
- **Methods** : Manages the registration workflow, allowing users to submitRegistration, createDraft, and for coordinators to approveRegistration or declineRegistration, deleteRegistration
- **Relationships** : It acts as a link between a User and a Course during the enrollment phase.

3. **Content, Assignments, and Submissions**
   Class Material : Represents any instance of learning resource

- **Purpose** : To manage documents, videos, notes, or any file associated with a classroom
- **Attributes** : Includes materialId, title, description, fileURL, and a MaterialType (an enumeration including LECTURE, MOVIE, REFERENCE, etc.)
- **Relationships** : Many Materials can belong to one Classroom

Class Assignment : Represents a task given to students

- **Purpose** : To define homework, quizzes, or projects with details and deadlines
- **Attributes** : Contains relevant attributes such as assignmentId, classroomId, title, description, deadline, and publishDate

- **Relationships** : An Assignment is part of a Classroom and is associated with multiple student Submissions

Class Submissions : Represents a student's submitted work

- **Purpose** : To track a student's response to an Assignment
- **Attributes** : Includes attributes such as submissionId, studentId, fileURL, submitDate, a grade, tutorComment, a GradingStatus (e.g., SUBMITTED, GRADED)
- **Relationships** : Each Submission is linked to one Assignment and one User (the student)

4. **Scheduling and Sessions**

Class Session : Represents a scheduled event within a classroom

- **Purpose** : To define a specific lecture, tutorial, or meeting
- **Attributes** : Contains attributes such as sessionId, classroomId, startTime, endTime, title, location
- **Methods** : Allows for creating, canceling, and marking attendance for sessions
- **Relationships** : A Classroom can have many Sessions

Class SessionNote : Represents a note for a session

- **Purpose** : To store detailed records for a specific session
- **Attributes** : Holds noteId, sessionId, an attendanceMap (linking students to attendance status), and meetingMinutes
- **Relationships** : A Session can have associated SessionNotes

Class SessionRequest : Represents a request for a session

- **Purpose** : To allow students to request new or special sessions
- **Attributes** : Includes atributes like typeOfRequest("NEW"|"ABSENT"), requestId, studentId, reason, and a status (PENDING, APPROVED, REJECT)
- **Methods** : Manages the request workflow (create, approve, reject, cancel)
- **Relationships** : Links a User (student) to a Classroom for scheduling purposes

5. **Communication and Feedback**
   Class ChatChannel and Message : Together, these implement a messaging system

   - **ChatChannel** : Represents a conversation, with attributes like channelId, channelName and participantIDs
   - **Message** : Represents a single message sent in a channel, with attributes like messageId, senderId, content, and sentAt
   - **Relationships** : A ChatChannel contains many Messages

   Class Notification

   - **Purpose** : To send system-generated alerts to users
   - **Attributes** : Includes attributes such as notificationId, recipientId, senderId,sentTime, title, content, and an isRead flag
   - **Methods** : Supports creating, deleting notifications, marking them as read, and fetching unread counts
   - **Relationships** : A Notification is directed at a specific User

   Class TutorRating and ClassFeedback

   - **Purpose** : To collect feedback on the quality of teaching and courses
   - **TutorRating** : Allows a student to rate a tutor within a specific classroomId. It captures detailed criteriaRating and a comment
   - **ClassFeedback** : Allows a tutor to provide general feedback on a courseId, including overallPerformance and suggestion
   - **Relationships** : These classes link Users (students) to Classrooms and Courses to capture their evaluations

10.4 Adapter

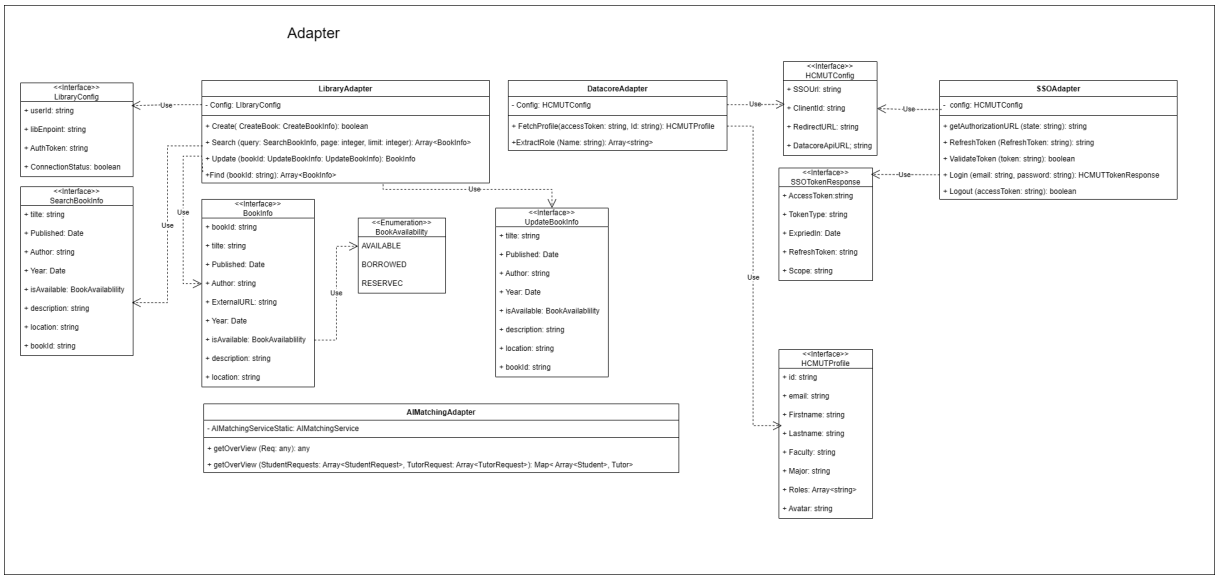


Figure 10.4: Adapter-Group

The Adapter Layer is a strategic architectural tier designed based on the **Adapter Design Pattern**. Its role is not merely "connection" but that of a key **mediator and translator**, standing between the system's core **Business Logic Layer** (Services) and all external or complex services. These services include: (1) third-party APIs (e.g., HCMUT_SSO, Library API) and (2) complex internal subsystems (e.g., the AI Inference Server).
Design Rationale: The core purpose of this tier is to enforce the **Separation of Concerns** and **Dependency Inversion** principles. The system's business logic should not, and does not need to, know about the complex and often volatile implementation details (e.g., specific protocols, data-formatting, or authentication flows) of the services it consumes.

- **Maintainability:** This is the primary benefit. When the HCMUT SSO API changes (e.g., its endpoints are updated, its response structure is altered, or its auth flow is modified), we *only* need to update the logic within the `SSOAdapter`. The core `AuthService` and the rest of the system remain entirely unaffected. The impact of such changes is effectively **contained** within the adapter.

- **Standardization:** Adapters act to **normalize and standardize external data**. Data from external APIs can be messy, inconsistent, or structured in a way that is not optimal for the application. The adapter translates this raw data into standardized, clean internal `interfaces` or `DTOs` (Data Transfer Objects), such as

`HCMUTProfile` or `BookInfo`. This ensures the core system always works with consistent, predictable data models.

- **Testability:** When performing a unit test on the `AuthService`, we do not require a live network connection to the HCMUT SSO server. Instead, we can easily "mock" the `SSOAdapter` to return a hypothetical `SSOTokenResponse`. This makes testing fast, deterministic, reliable, and entirely independent of external environments.

Adapter Component Analysis:

- **SSOAdapter:** This is a complex adapter responsible for encapsulating the entire HCMUT SSO authentication flow.
  - `getAuthorizationURL`: Constructs the precise URL to redirect the user to the HCMUT login page if a web-based flow (like OAuth2) were used for parts of the system.
  - `Login`: As per the diagram, this method directly exchanges the user's credentials (`email`, `password`) for an `AccessToken` and `RefreshToken`. It encapsulates the secure, back-channel request to the SSO's token endpoint (implementing a ROPC grant flow).
  - `ValidateToken`: Provides a simple, clean method for checking token validity, likely by calling an introspection endpoint or validating a JWT signature.
- **DatacoreAdapter:** This adapter works in close conjunction with the `SSOAdapter`.
  - `FetchProfile`: It receives the `AccessToken` (acquired by the `SSOAdapter`). Its most important task is to *translate*. It calls the HCMUT "user info" API, receives a raw JSON response, and then carefully *maps* the external fields (e.g., `"mail"`, `"surname"`, `"faculty_code"`) to the standardized internal `HCMUTProfile` interface (e.g., `email`, `lastName`, `faculty`).
- **LibraryAdapter:** This component demonstrates the pattern's reusability. It hides all the implementation details of the university's Library API (which could be a legacy REST or SOAP protocol).
  - The `Search` method does more than make an API call; it must format the internal `SearchBookInfo` object into the specific query parameters the external API expects, handle its unique pagination logic, and translate the non-standard response back into a clean array of `BookInfo` objects.

- **AIMatchingAdapter:** This is a special case illustrating that adapters are not only for external, third-party APIs. As described in **Pattern F (Complex Internal Flow)**, the AI Inference Server runs on separate GPU infrastructure and, as noted in the Deployment View, uses **gRPC**.
  - The core business logic (e.g., a `CoordinatorService`) should have zero knowledge of `gRPC`, `Protobuf` (Protocol Buffers), or the IP address of the AI server.
  - The `AIMatchingAdapter` hides this internal complexity completely. Its `getOverView` methods will: (1) Initialize a `gRPC` client, (2) "Serialize" the standard application objects (`StudentRequest`, `TutorRequest`) into `Protobuf` messages, (3) Make the `gRPC` call to the AI Inference Server, and (4) "Deserialize" the `Protobuf` response back into a standard `Map<Student, Tutor>` that the application can understand.

Conclusion: The Adapter Layer is not a simple "helper" tier; it is a deliberate **defensive** and **abstraction** layer. It ensures the system's core business logic remains clean, independent, testable, and **robust** against the inevitable volatility of both external third-party services and complex internal subsystems.

11 MVC diagrams

11.1 Login - MVC

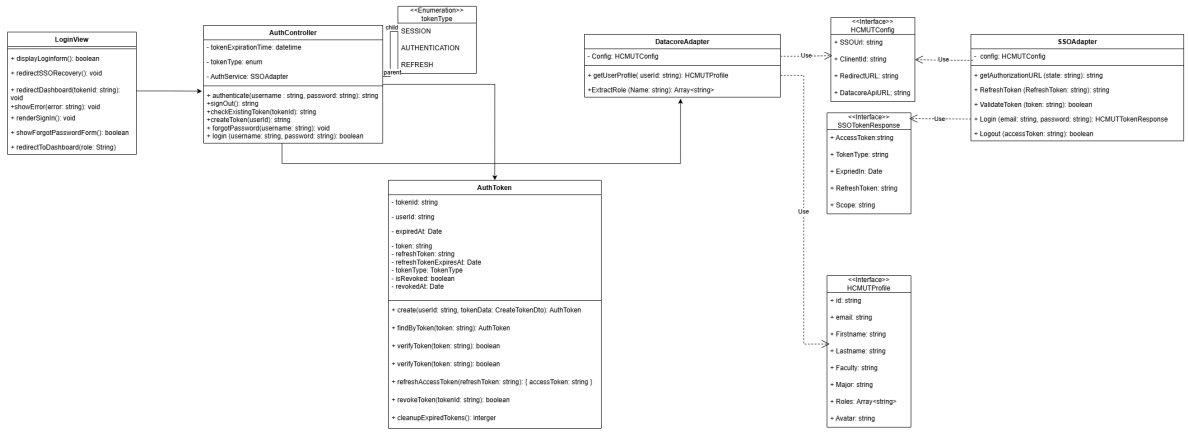


Figure 11.1: MVC diagram for login process

1. **View and controller**

- The LoginView has a one-way association with the AuthController
- The AuthController, in turn, communicates back to the LoginView to render results, such as by calling redirectToDashboard() on success or showError() on failure

2. **Model and controller**

- The AuthController acts as the central orchestrator and depends on components of the Model layer to execute tasks
- The AuthController also directly interacts with the AuthToken class to create, manage, and verify the application's internal session tokens after a successful external authentication

3. **Adapters and External Service Interfaces**

- The SSOAdapter has a dependency on the HCMUTConfig interface to obtain necessary configuration (like URLs and client IDs)
- The DatacoreAdapter depends on HCMUTConfig for its configuration and uses the HCMUTProfile interface to structure the user profile data it fetches

11.2 Chat - MVC

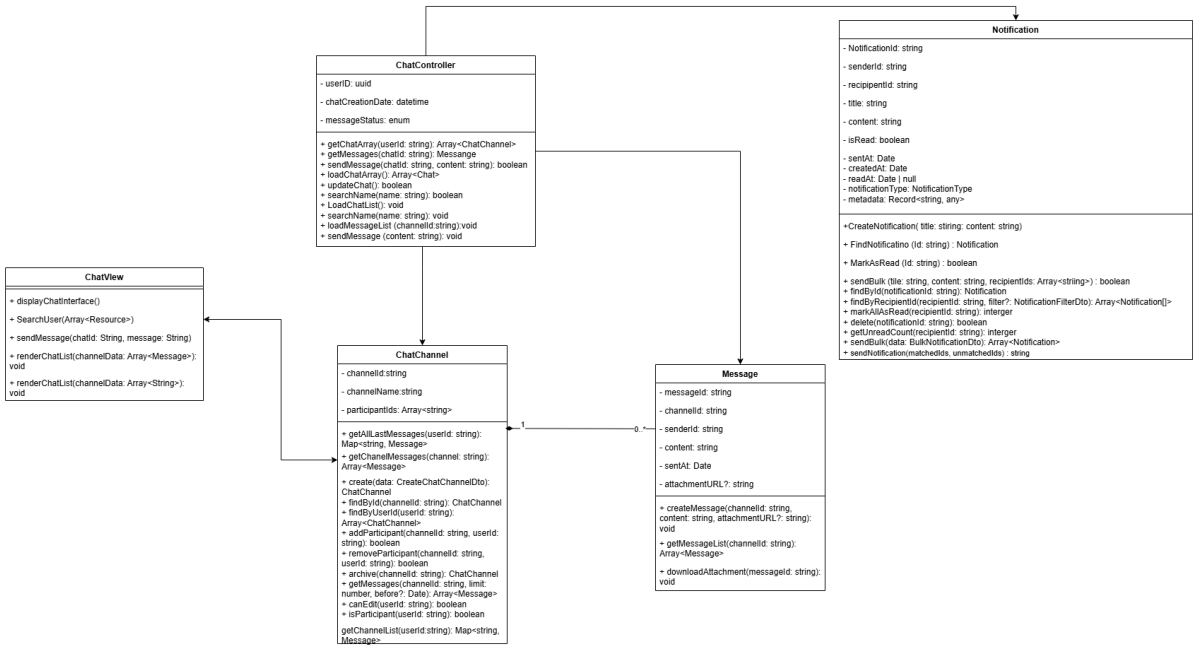


Figure 11.2: MVC diagram for chat process

1. **View and controller**
    - The ChatView depends on the ChatController. User actions, such as sending a message (sendMessage) or searching for a user (searchUser), trigger method calls on the ChatController
    - The ChatController holds a reference to the ChatView to update it with new data. After fetching or processing data, the ChatController calls methods like renderChatList() on the ChatView to display the results to the user
2. **Model and controller**
    - The ChatController acts as the central coordinator and has dependencies on all Model components
    - The controller directly interacts with the ChatChannel and Message classes to perform core chat functions
3. **Within the model**
    - A Message is intrinsically part of a ChatChannel
    - There is a one-to-many composition relationship between ChatChannel and Message, implies that one ChatChannel can contain zero or more Message objects

11.3 Session - MVC

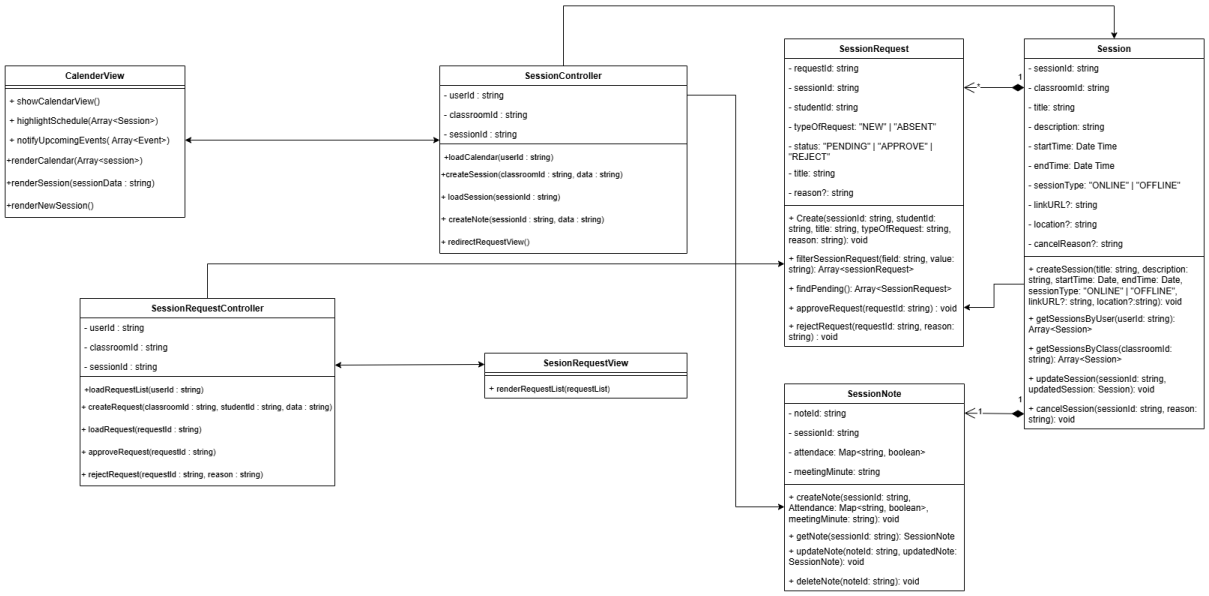


Figure 11.3: MVC diagram for sessions process

1. **View and controller**

   - There is a one-way dependency from the CalendarView to the SessionController. User interactions within the calendar to trigger methods on the SessionController. In return, the SessionController updates the CalendarView by calling its rendering methods
   - The SessionRequestController depends on the SessionRequestView to display data

2. **Model and controller**

   - The SessionController has a strong dependency on the Session and SessionNote classes. It directly uses them to create, load, and manage session data and associated notes
   - The SessionRequestController depends on the SessionRequest class to load, create, approve, and reject requests.

3. **Within the model**

   - A single Session can be associated with many SessionRequest objects

- A single Session can have many SessionNote objects associated with it. This allows for multiple notes, attendance records, or meeting minutes to be attached to one session.
- Each SessionNote is tied to exactly one Session

11.4 Submission - MVC

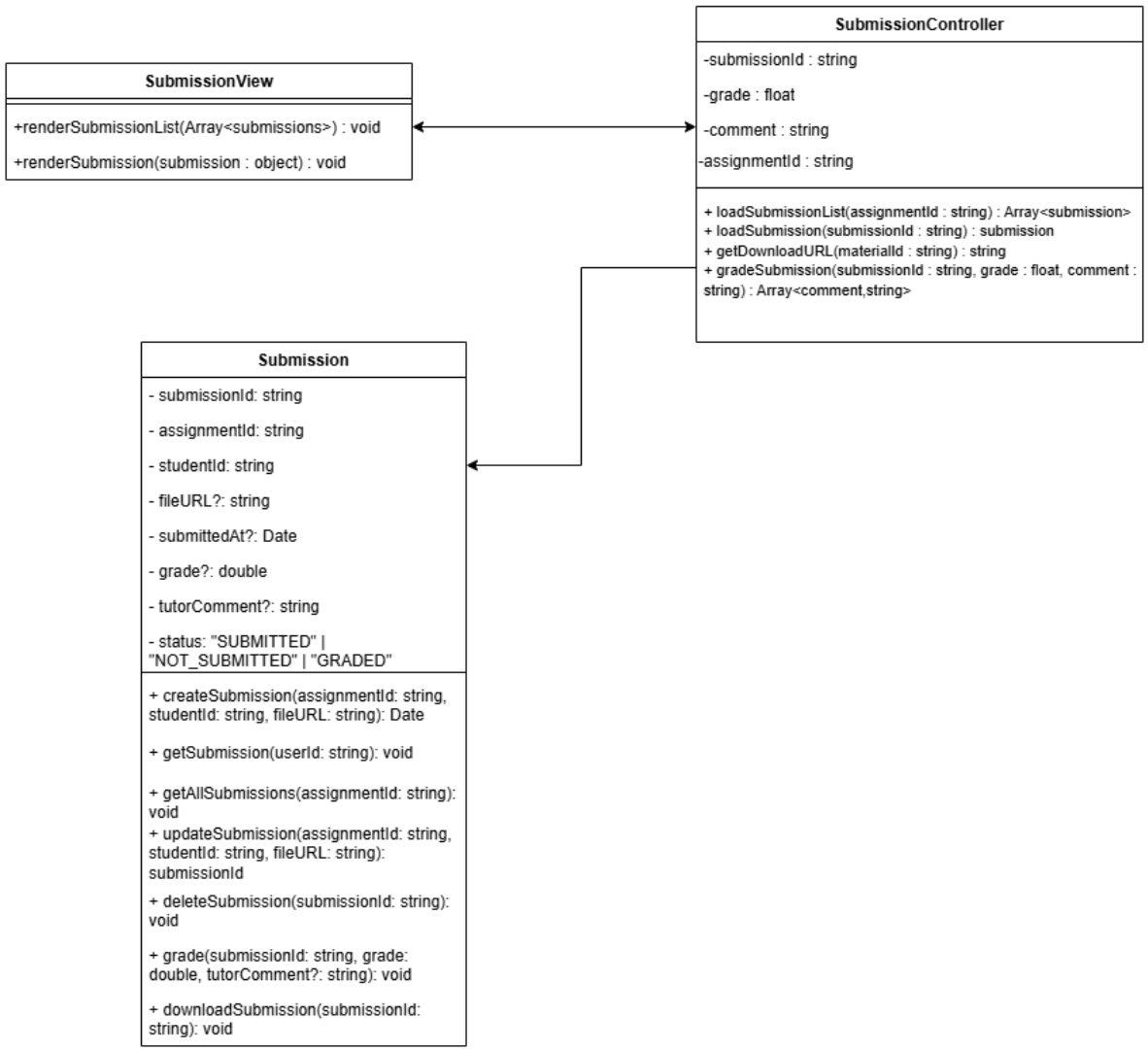


Figure 11.4: MVC diagram for submission process

1. **View and controller**

   - User actions within the SubmissionView trigger calls to methods on the SubmissionController

- After fetching or processing data, the SubmissionController calls methods on the SubmissionView to display the updated information to the user

2. **Model and controller**

- The SubmissionController has a strong dependency on the Submission model
- The Controller delegates all data-related logic to the Submission model. For example, when the loadSubmission method is called on the Controller, it in turn calls a method like getSubmission on the Submission class to actually retrieve the data

11.5 Registration - MVC

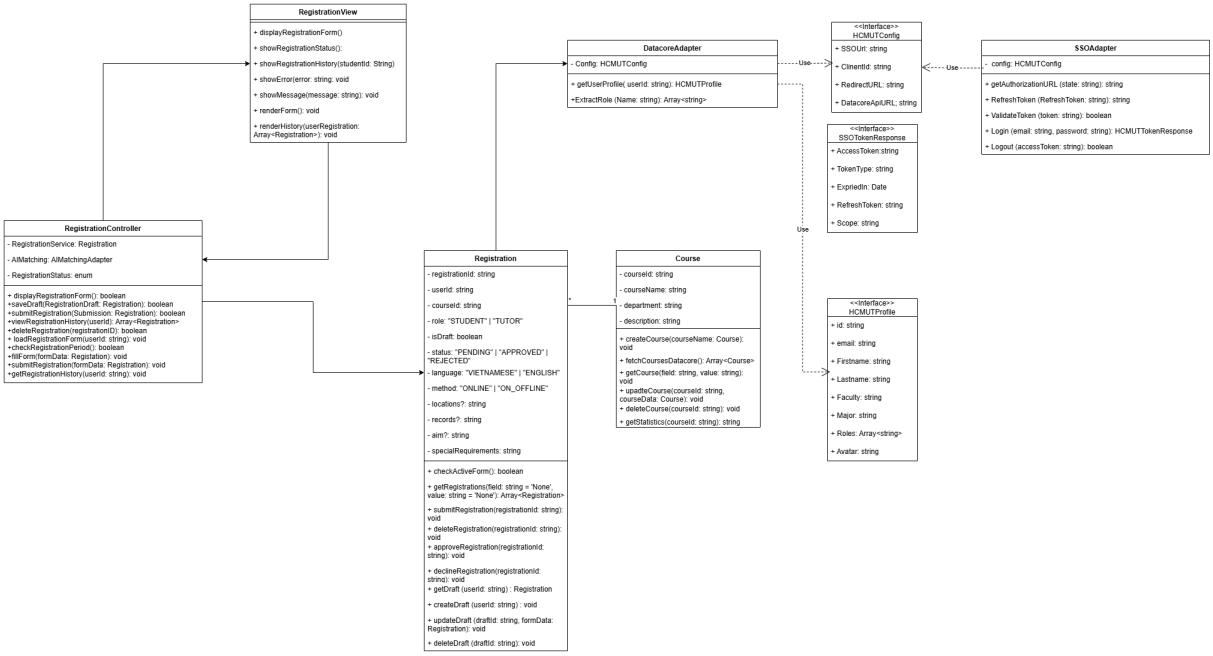


Figure 11.5: MVC diagram for registration process

1. **View and controller**

   RegistrationView depends on the RegistrationController to handle user actions like submitting a form or requesting registration history

- The RegistrationController, in turn, holds a reference to the RegistrationView to call its methods, thus updating the UI with new data or status messages

2. **Model and controller**

- The RegistrationController has a strong dependency on the Registration model
- The RegistrationController indirectly interacts with the Course model through its work with Registration objects

3. **Within the Model**

- Registration and course have a Many-to-One association, registration instances can be associated with exactly one course
- The Course model has a dependency on the DatacoreAdapter

4. **Adapters and Interfaces**

- DatacoreAdapter uses several interfaces to remain decoupled from specific implementations. It depends on HCMUTConfig for its configuration and HCMUTProfile as a data contract for the user profiles it retrieves
- The SSOAdapter uses the HCMUTConfig interface for configuration and SSOTokenResponse as a contract for authentication tokens

11.6 Matching - MVC

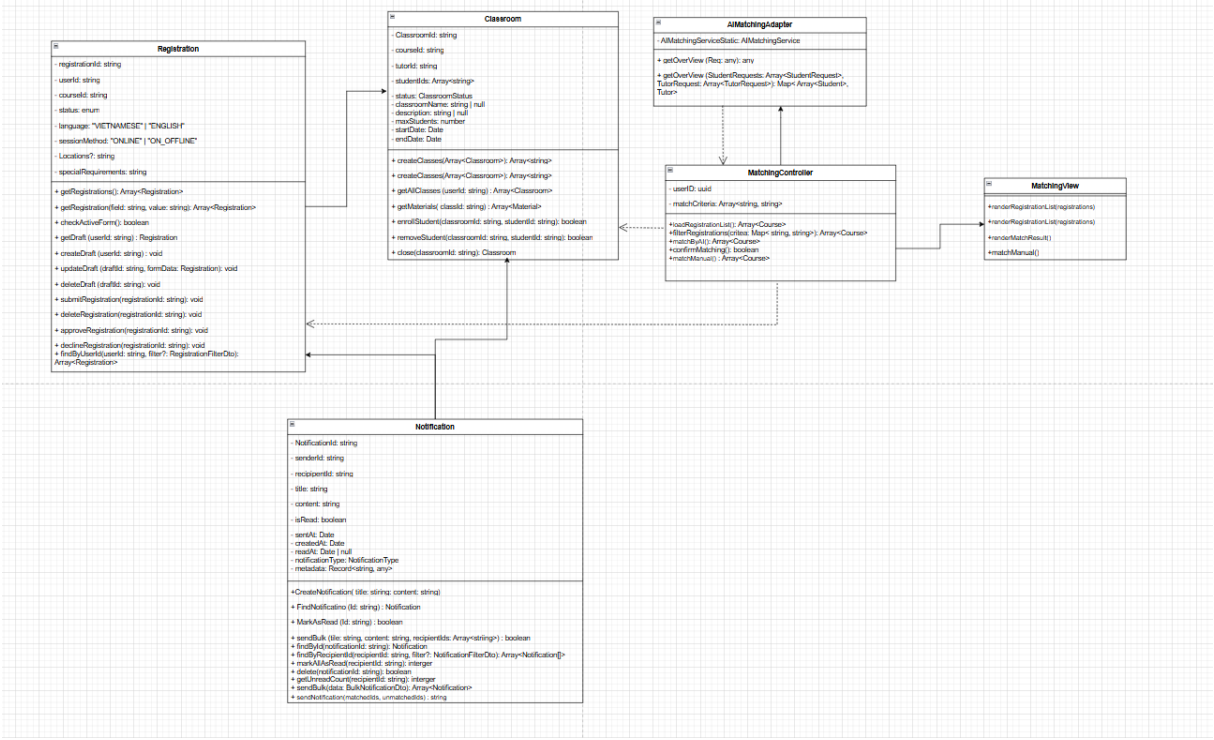


Figure 11.6: MVC diagram for the matching process

1. **View and Controller**
    - The `MatchingController` holds a reference to the `MatchingView` to update the UI by calling its methods, such as `renderRegistrationList()` and `renderMatchResult()`
    - The `MatchingView` depends on the `MatchingController` to handle user actions, specifically invoking `matchManual()` to perform a manual matching operation.
2. **Model and Controller**
    - The `MatchingController` has strong dependencies on the `Registration` and `Classroom` models.
    - It interacts with the `Registration` model to fetch data for display and filtering (e.g., via `loadRegistrationList()` and `filterRegistrations()`).
    - After a match is confirmed via `confirmMatching()`, the controller likely interacts with the `Classroom` model to create new class instances.
    - The controller also has a dependency on the `Notification` model to send notifications upon match confirmation.
3. **Within the Model**
    - There is a logical association between `Registration` and `Classroom`, where a successful and approved registration leads to the creation of a `Classroom` instance.
    - The `Classroom` model appears to have a dependency on the `Notification` model, suggesting that actions like creating a class or enrolling a student trigger notifications.
4. **Adapters and Interfaces**
    - The `MatchingController` utilizes an `AIMatchingAdapter` to decouple the application from the specific implementation of the AI matching service.
    - This adapter abstracts the external `AIMatchingService`, allowing the controller to request AI-powered matches (via `matchByAI()`) without needing to know the underlying details of the service.

11.7 Classroom Management - MVC

Figure 11.7: MVC diagram for the classroom management process

1. **View and Controller**

- The `ClassroomController` holds a direct reference to the `ClassroomView` and is responsible for calling its methods to update the user interface, such as `renderClassroom()`, `renderNewMaterial()`, and `renderNewAssignment()`.
- The `ClassroomView` in turn relies on the `ClassroomController` to handle user-initiated actions, such as creating an assignment or submitting a student's work.

2. **Model and Controller**

- The `ClassroomController` orchestrates interactions between various model components. It has strong dependencies on the `Classroom`, `Material`, `Assignment`, and `Submission` models.
- Controller methods directly trigger model operations, for example, `createAssignment()` interacts with the `Assignment` model, and `submitAssignment()` interacts with the `Submission` model.

3. **Within the Model**

- The model is structured around the central `Classroom` entity. The black diamond connectors indicate a strong composition relationship.
- `Assignment` and `Material` are composed within a `Classroom`, meaning they cannot exist without a parent classroom.
- Similarly, `Submission` is composed within an `Assignment`, establishing a clear hierarchical relationship: a `Classroom` contains `Assignments`, which in turn contain `Submissions`.
- The models show dependencies on each other, as indicated by the dashed arrows, where actions in one model (e.g., grading a `Submission`) may affect another (e.g., updating an `Assignment` status).

4. **Adapters and Interfaces**

- This design does not feature explicit adapters or interfaces to decouple the controller from the models.
- The `ClassroomController` interacts directly with the concrete model classes (`Classroom`, `Assignment`, etc.). The public methods of these model classes serve as the contract for data manipulation.

11.8 Library Search - MVC

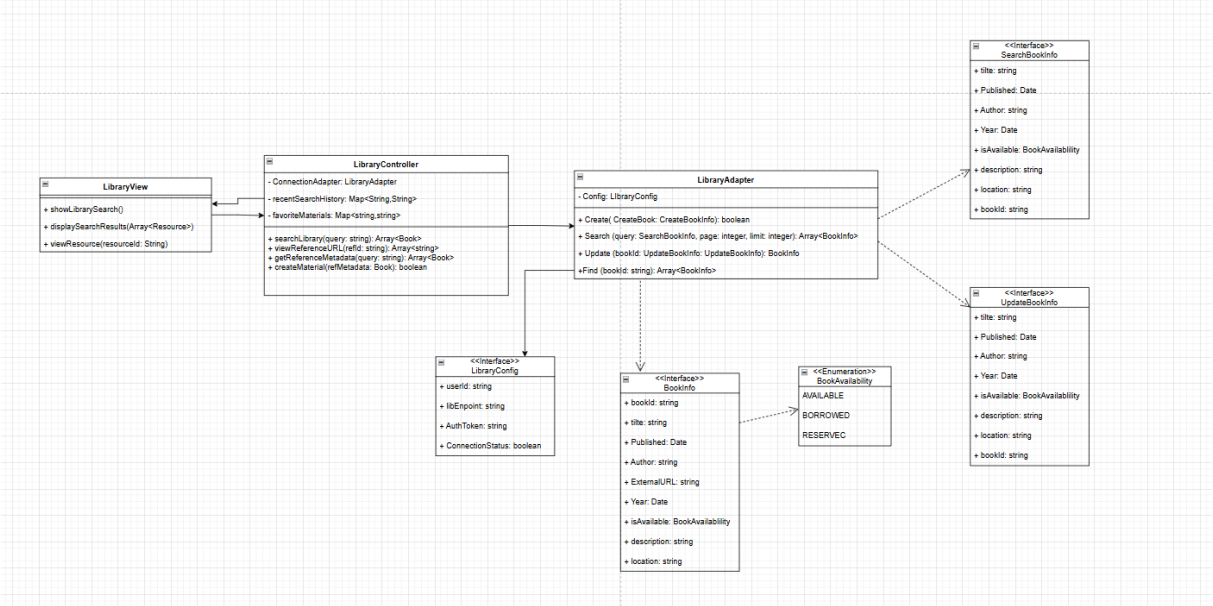


Figure 11.8: MVC diagram for the library search process

1. **View and Controller**

   - The `LibraryView` depends on the `LibraryController` to handle user actions, such as initiating a search by calling the `searchLibrary()` method.
   - The `LibraryController` in turn holds a reference to the `LibraryView` to update the UI, for example, by calling `displaySearchResults()` to render the list of books found.

2. **Model and Controller**

   - The `LibraryController` has a strong dependency on the `LibraryAdapter`, which serves as the gateway to the data layer.
   - All data-related operations, such as searching for books (`searchLibrary()`) or creating new material (`createMaterial()`), are delegated from the controller to the adapter.

3. **Within the Model**

   - The data model is defined by a set of interfaces, primarily `BookInfo`, which represents a complete book record.

- Specialized interfaces like `SearchBookInfo` and `UpdateBookInfo` are used as data contracts (Data Transfer Objects) for specific operations like searching and updating.
- The status of a book is managed by the `BookAvailability` enumeration, which includes states like `AVAILABLE`, `BORROWED`, and `RESERVED`.

4. **Adapters and Interfaces**

- The `LibraryAdapter` is key to decoupling the application logic from the external data source. It depends on the `LibraryConfig` interface for its configuration (e.g., API endpoint, authentication token).
- The adapter's methods (`Search`, `Create`, `Update`) use the specific data contract interfaces (`SearchBookInfo`, `UpdateBookInfo`) to ensure clear and consistent communication with the data source.

11.9 Feedback and Rating - MVC

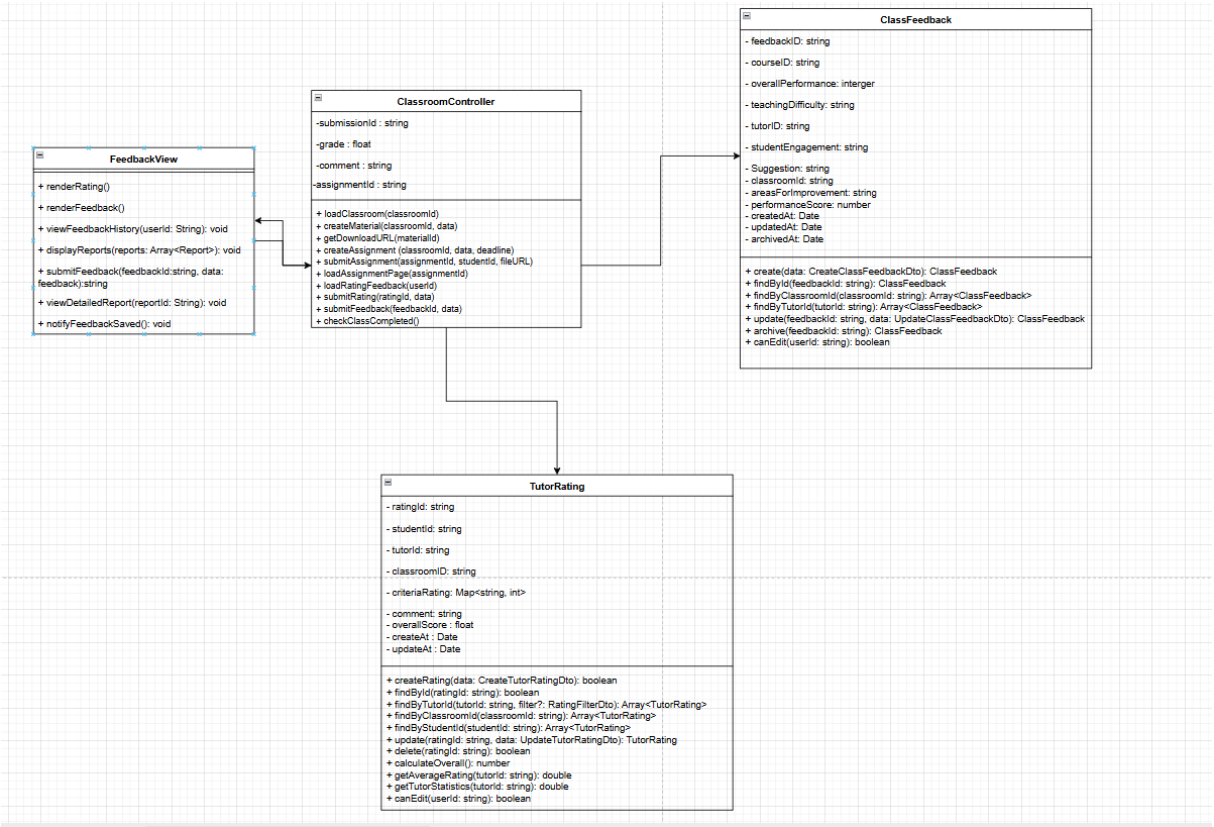


Figure 11.9: MVC diagram for the feedback and rating process

1. **View and Controller**
    - The `FeedbackView` depends on the `ClassroomController` to handle user actions, such as submitting feedback via `submitFeedback()` or requesting historical data.
    - The `ClassroomController` holds a reference to the `FeedbackView` to call its methods to update the UI, such as `displayReports()` with fetched data or `notifyFeedbackSaved()` to confirm an action.
2. **Model and Controller**
    - The `ClassroomController` acts as an intermediary with strong dependencies on both the `ClassFeedback` and `TutorRating` models.
    - When a user submits feedback, the controller's `submitFeedback()` method interacts with the `ClassFeedback` model to create or update a feedback record.
    - Similarly, when a rating is submitted, the controller's `submitRating()` method interacts with the `TutorRating` model to manage tutor ratings.
3. **Within the Model**
    - The design features two distinct but related models: `ClassFeedback` and `TutorRating`.
    - `ClassFeedback` captures comprehensive feedback about a class, including overall performance, suggestions, and areas for improvement.
    - `TutorRating` captures specific scores and comments from a student regarding a tutor, including an overall score and criteria-based ratings. Both models are linked to a classroom and tutor.
4. **Adapters and Interfaces**
    - While no explicit adapter components are shown, the system uses Data Transfer Objects (DTOs) as data contracts, serving a similar purpose to interfaces for data structure.
    - For example, the `ClassFeedback` model's `create()` and `update()` methods expect a `CreateClassFeedbackDto` and `UpdateClassFeedbackDto` respectively.
    - These DTOs define the shape of the data required for model operations, ensuring a clear and decoupled contract between the controller and the model layer.

12 Design Pattern

12.1 Model-View-Controller Pattern

- **Where it is used:** This pattern is the foundational architecture for these modules: Login, Chat, Session, Submission, Registration, Matching, Classroom Management, Library Search, and Feedback.
  - **Model:** Represents the data and business logic.
  - **View:** Represents the UI and presentation layer.
  - **Controller:** Acts as an intermediary, handling user input and orchestrating communication between the Model and the View.
- **Why it is used:** To enforce separation of concerns.
  - **Maintainability:** By decoupling the business logic (Model) from the user interface (View), changes to one layer have a minimal impact on the others.
  - **Parallel Development:** Different developers can work on the Model, View, and Controller components simultaneously.
  - **Reusability:** The same Model can be reused with different Views.

12.2 Adapter Pattern

- **Where it is used:** This pattern is utilized to integrate with external or third-party services.
  1. `SSOAdapter` **and** `DatacoreAdapter` (in Login, Registration modules): These classes wrap the specific APIs of the external HCMUT Single Sign-On (SSO) and Datacore services.
  2. `AIMatchingAdapter` (in the Matching module): This class encapsulates the communication logic for an external AI matching service.
  3. `LibraryAdapter` (in the Library Search module): This adapter wraps the API of an external library system.
- **Why it is used** : To decouple the main application from external dependencies.
  - **Flexibility** : If an external API changes, only the corresponding adapter needs to be updated.

- **Abstraction:** The adapters hide the complexity of external API calls and provide a clean interface for the controllers to use.
- **Testability:** During testing, these adapters can be replaced with mock implementations to simulate external service behavior.

12.3 Observer Pattern

- **Where it is used:** This pattern governs the communication from the Controller to the View in all modules.
  - **Class Cluster:** The Controller acts as the subject, and the View acts as the observer.
  - **Examples:** The Controller, after processing a request, calls rendering methods on the View to update the UI.
- **Why it is used:** To create a decoupled communication channel for UI updates. The Controller does not need to know the specific details of how the View is implemented. This allows the View to be modified independently of the Controller.

12.4 Composition Pattern

- **Where it is used:** Within the Model layer to create clear hierarchical object structures
  - `ChatChannel` **and** `Message`**:** A `Message` is part of a `ChatChannel`.
  - `Classroom`**,** `Assignment`**, and** `Material`**:** `Assignments` and `Materials` are components of a `Classroom`.
  - `Assignment` **and** `Submission`**:** A `Submission` is a direct response to an `Assignment` and is composed within it.
- **Why it is used:** To model strong ownership and manage object lifecycles. When a "whole" object is deleted, all its "part" objects are also deleted. This ensures data integrity.

12.5 Strategy Pattern

- **Where it is used:** In the **Matching module**, through the interaction between the `MatchingController` and its matching methods.

  - The `MatchingController` can execute a match using different strategies:
    1. A **manual strategy**, invoked by `matchManual()`.
    2. An **AI-powered strategy**, invoked by `matchByAI()`, which is encapsulated and delegated to the `AIMatchingAdapter`.

- **Why it is used:** To provide flexibility in algorithms. The controller is not hard-wired to a single way of matching. New matching algorithms could be added in the future as new strategies without modifying the `MatchingController` itself.

13 Deployment view

This section will illustrates the physical and virtual hardware infrastructure, network topology, and software artifact distribution for the Tutor Support System. The architecture is designed as a high-security, 3-zone model to protect backend services, built on VNG Cloud's (VServer) virtualized infrastructure.

The system is organized into three distinct security zones: **Public Zone**, the **Demilitarized Zone (DMZ)**, and **Secure Private Network**.

13.1 Public Zone

This zone represents untrusted external clients on the public internet.

- **User Device:** The client machine (PC, mobile) running a Web Browser. All system access originates here.
- **CDN (vCDN):** A Content Delivery Network provided by vCDN is used to improve global performance. It caches and serves the React application's static files (from the Web Server) and user-generated content (from Object Storage) to reduce latency and server load.
- **HCMUT Server:** Represent the university external services

Traffic is split: API calls are sent to the External Firewall, while the CDN serves requests for static content. Notice that the CDN is connected directly to the Object Storage in the Private Network Zone via VNG private network.

13.2 Demilitarized Zone (DMZ)

The DMZ is a "buffer" network that separates the public internet from the secure backend.

- **External Firewall (vCF)**: This is the single entry point to the system. It filters all incoming traffic, only allowing HTTPS (port 443) requests from the user Device to the Web Server.
- **Web Server (Nginx)**: A vServer running Ubuntu 22.04 LTS and Nginx. This server has two roles:
  - **Static Content**: Stores and serves the core application files (e.g., index.html, bundle.js) from its local disk.
  - **Reverse Proxy**: Forwards all dynamic API requests (e.g., /api/...) to the Internal Firewall, hiding the backend's structure.

13.3 Secure Private Network

This is the most protected zone, containing all core business logic and sensitive data. It is inaccessible from the public internet. This zone consists of:

- **Internal Firewall (vCF)**: This second firewall provides a critical layer of defense. It is configured only to accept traffic from the Web Server's reverse proxy and blocks all other requests.
- **Application Server (Node.js)**: The "brain" of the application, running Node.js on an Ubuntu 22.04 LTS VM. It stores and executes the business logic from its local disk. It orchestrates the various backend services to fulfill user requests.
- **Database Server (PostgreSQL)**: A managed vDB (PostgreSQL) service. This is the primary database, storing all relational data, including tables and indexing. It is accessed by the Application Server via the TCP/IP PostgreSQL protocol.
- **AI Inference Server (GPU Cloud)**: A specialized GPU (GH200) virtual server running Ubuntu 22.04 LTS and TensorFlow Serving. It exposes a gRPC API for the Application Server to call for complex tasks, such as AI-powered tutor matching.
- **Object Storage (vStorage)**: A high-availability storage service used to store all user-generated binary files, such as Material uploads and Assignment submissions.

13.4 Infrastructure Services

These services support the entire deployment process and ensure the system's high availability.

- **Monitoring Server**: A dedicated vServer running Prometheus and Grafana. Prometheus is configured to "scrape" metrics from the node_exporter artifact installed on every server in the DMZ and Secure Private Network. Grafana visualizes this data, providing a comprehensive dashboard for system health and performance.
- **Backup Center**: A managed vBackup service responsible for disaster recovery. It is configured to take regular, automated snapshots of critical components, including the Database Server, the Object Storage buckets.

13.5 Connectiviy

The entire system architecture is connected and demonstrated by the given diagram. For more clearer diagram, take a look at here

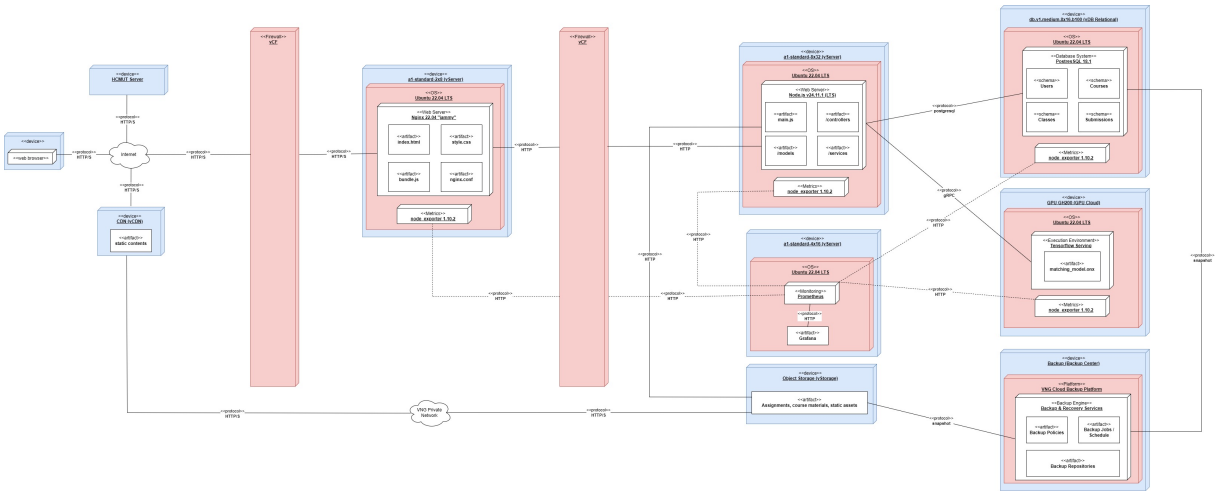


Figure 13.1: Deployment view diagram

14 Development/Implementation view

14.1 Technology Stack

With a clear architecture, we implement each layer of the system with the corresponding framework and language.

14.1.1 Client Tier (Frontend)

- **Core Framework:** React with Vite bundle (as a Single Page Application).
- **Routing:** TanStack Router, providing file-based routing, lazy loading, and route-level auth protection.
- **State Management:**
  - **Client State:** Zustand (managing stores, persist important state to the user's storage).
  - **Server State:** TanStack Query (managing caching, automatic refetching, and optimistic updates).
- **HTTP Client**: Axios, configured with a custom instance that includes:
  - Request interceptors for JWT token injection.
  - Response interceptors for global error handling.
  - withCredentials set for cookie-based sessions if needed.

14.1.2 Application Tier (Backend)

- **Core Framework**: NestJS
- **Security & Validation:**
  - **Authentication**: JWT authentication guard using Passport.js.
  - **Authorization**: Custom role guard for Role-Based Access Control (RBAC).
  - **Global Security**: Helmet (for secure HTTP headers).
  - **Validation**: class-validator (used by the global ValidationPipe).
- **Database ORM**: MikroORM provide the abstraction layer between services and the database.

14.1.3 Database Tier

- **Persistence Database**: PostgreSQL
- **In-memory Database**: Redis

14.1.4 Supporting Services

- **Caching/Queue**: Redis Cache (for sessions or queues).
- **Object Storage**: vStorage (for uploads and static files).
- **Email Service**: SendGrid (for notifications and verification).
- **External Authentication**: HCMUT_SSO .

14.2 Architecture

The application is architecturally divided into three distinct tiers, which are themselves composed of internal layers.

14.2.1 Client Layer (React SPA)

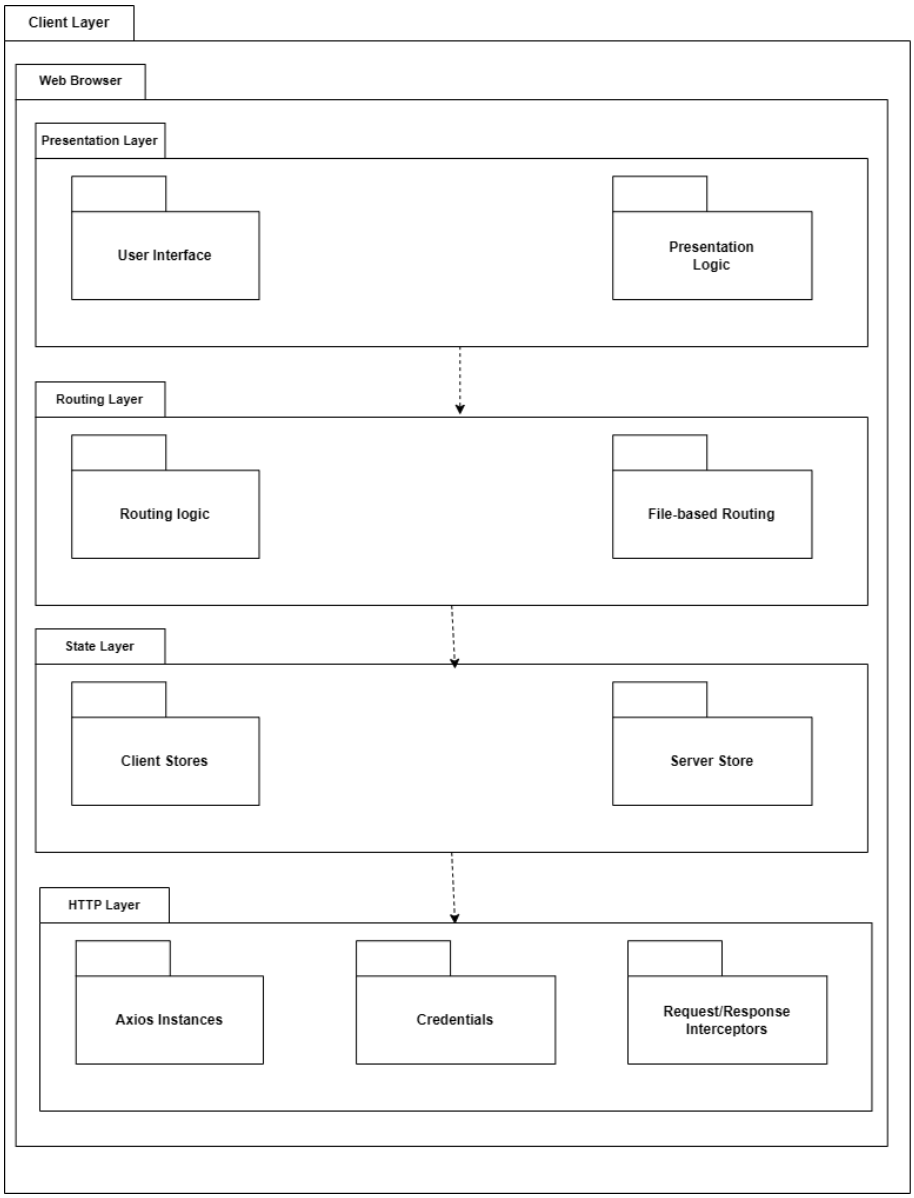


Figure 14.1: Implementation view: Client Layer

This tier runs entirely in the user's browser and is responsible for all user interaction.

- **Presentation Layer**: Contains the React components that form the UI (e.g., LoginPage, DashboardPage).
- **Routing Layer**: Handled by TanStack Router, which maps browser URLs to the correct presentation components (e,g., /login, /dashboard).

- **State Management Layer**: Manages all application data, clearly separated into:
  - **Zustand**: For managing synchronous, global client state (e.g., "is the user logged in?").
  - **TanStack Query**: For managing asynchronous server state (e.g., "what is the list of courses?").
- **HTTP Client Layer**: A dedicated Axios instance that handles all API communication, abstracting away token management and error handling from the UI components.

14.2.2 Application Layer (NestJS Backend)

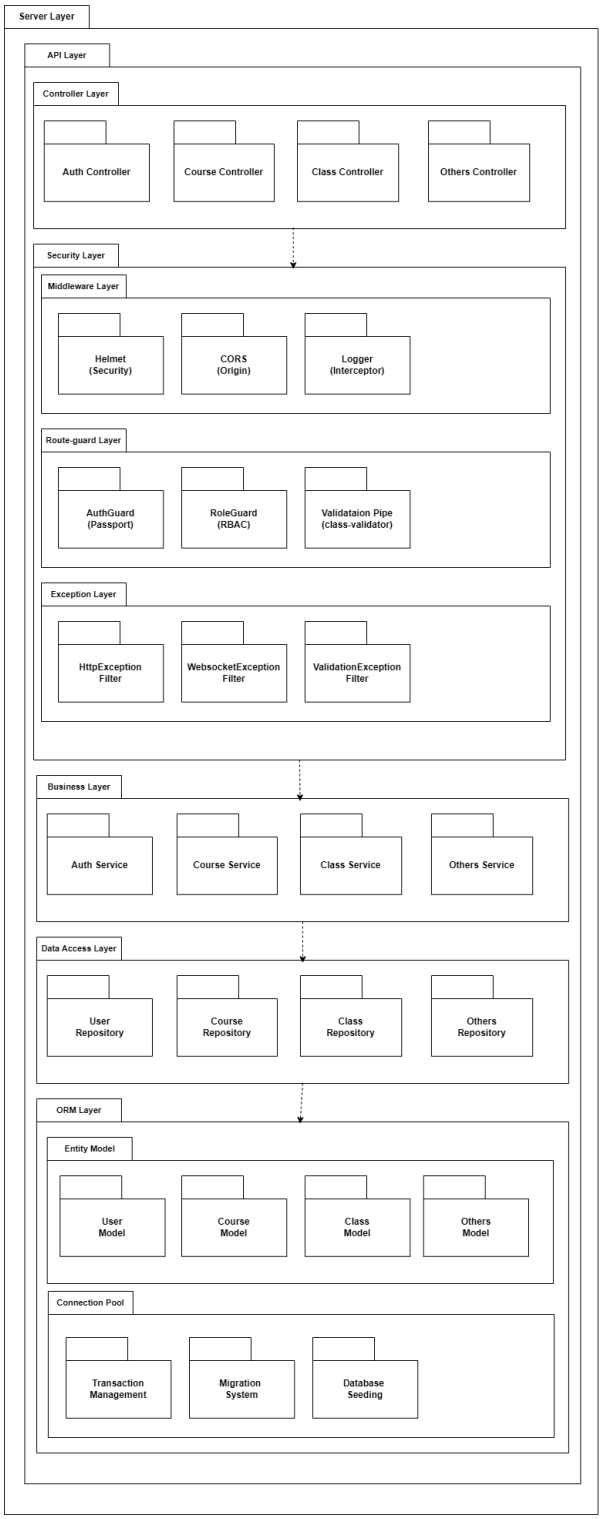


Figure 14.2: Implementation View: Server Layer

This tier acts as the system's brain, handling all business logic, security, and data coordination.

- **API Gateway Layer**: Exposes REST API Controllers (e.g., AuthController, CourseController) as the entry points for the client.
- **Middleware & Security Layer**: A robust pipeline that processes every incoming request:
  - **Global Middleware**: Helmet, CORS, and Logger run on every request.
  - **Guards & Pipes**: Route-specific modules like JWTAuthGuard (validates token), RoleGuard (checks permissions), and ValidationPipe (validates request bodies).
  - **Exception Filters**: Global filters (HttpExceptionFilter, DatabaseExceptionFilter) that catch errors and format them into consistent JSON responses.
- **Business Logic Layer**: The core logic, encapsulated in injectable NestJS Services (e.g., AuthService, CourseService). This layer is the only one that coordinates with the data layer.
- **Data Access Layer**: Composed of TypeORM Repositories (e.g., UserRepository, CourseRepository) that abstract the database queries.
- **ORM Layer**: Uses MikroORM to define Entity Models (e.g., User, Course) and manage the Connection Pool, Migrations, and Transactions.

14.2.3 Database Layer (PostgreSQL)

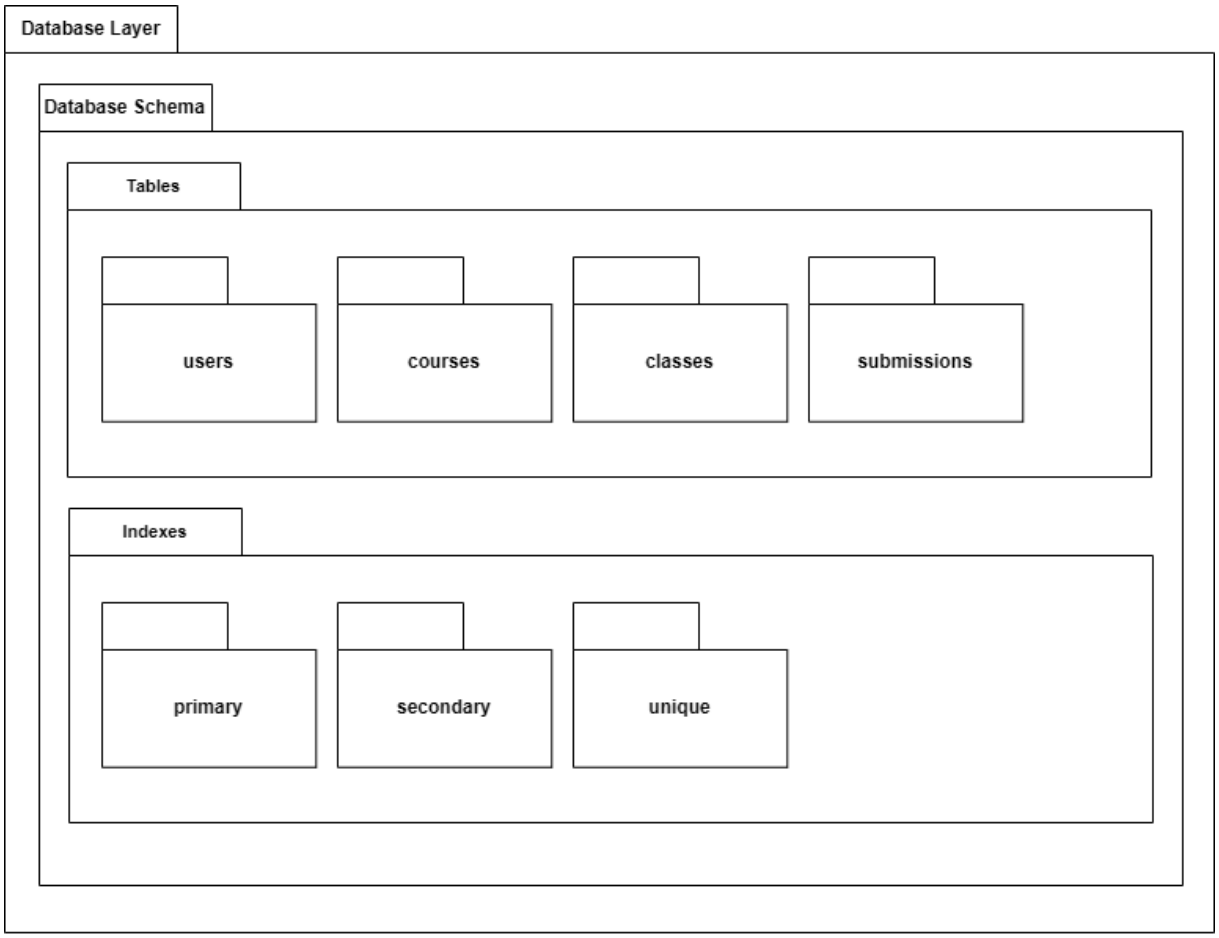


Figure 14.3: Deployment view diagram

This tier is responsible for the persistent storage and integrity of all application data.

- **Database Server**: A PostgreSQL instance listening for SQL queries.
- **Database Schema**: A collection of Tables (e.g., users, courses, orders, submissions) with defined columns, Foreign Key Relationships, and Constraints (NOT NULL, CHECK).
- **Indexes**: Performance optimizations, including Primary Keys, Unique Indexes (like on users.email), and other performance indexes.

14.2.4 Supporting Services Layer

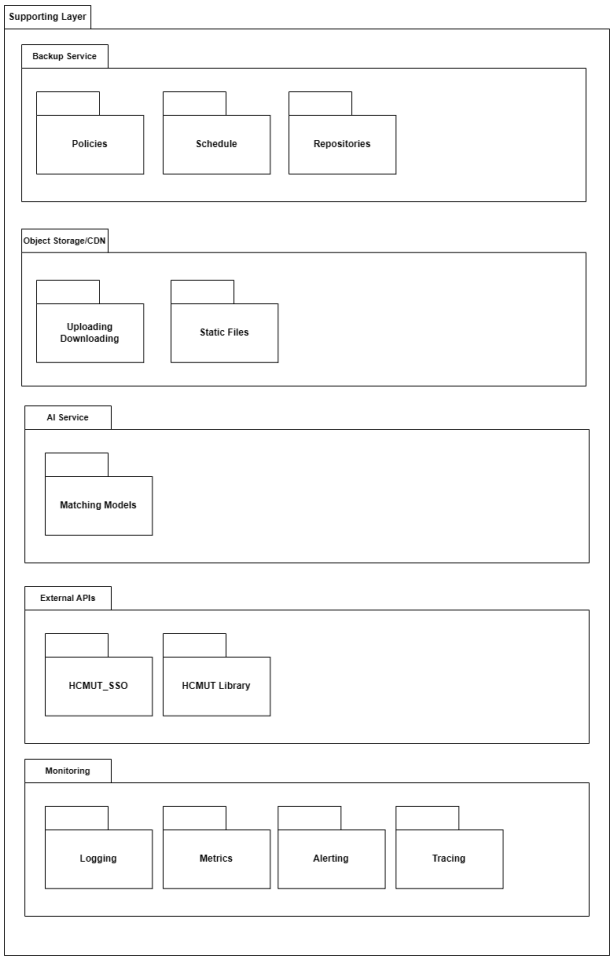


Figure 14.4: Deployment view diagram

The Supporting Services Layer consists of a set of external, decoupled services that are essential for the application's operation, maintenance, and enhancement. This layer provides auxiliary functionalities that support the core 3-tier architecture.

- **Backup Service**: Responsible for data durability and disaster recovery. This service operates based on defined Policies, runs on a fixed Schedule, and manages backup Repositories to ensure data can be restored.

  Object Storage / CDN: Manages the storage and delivery of binary data. This includes handling Uploading/Downloading of user-generated content (like submis-

sions or profile pictures) and serving Static Files (like images, CSS, and JavaScript bundles) efficiently, often via a Content Delivery Network (CDN).

- **AI Service**: Provides specialized intelligence to the application. This includes hosting and executing specific Matching Models, likely used for features such as connecting tutors with students based on specific criteria.
- **External APIs**: Integrates third-party services to extend functionality. The system connects to:
  - **HCMUT_SSO**: For handling single sign-on authentication via the university's identity provider.
  - **HCMUT Library**: For accessing or verifying information from the university library system.
- **Monitoring Service**: A comprehensive solution for observing the application's health and performance. This service is comprised of:
  - **Logging**: Aggregating and storing logs from all services for debugging.
  - **Metrics**: Collecting and visualizing time-series data (e.g., CPU usage, error rates).
  - **Tracing**: Providing end-to-end visibility for a single request as it flows through the system.
  - **Alerting**: Notifying the development team of failures or performance degradation.

14.3 Data flow

This section maps the functional requirements to the physical components of the deployment architecture. Each flow illustrates the step-by-step path of data from the user to the backend services and back.

14.3.1 Pattern A: Simple Read (View Data)

This is the most common data flow, used when a user needs to view information.

- **Request**: A user (Student, Tutor, etc.) on their User Device requests to see data (e.g., clicking on a course). The Web Browser sends an HTTPS API request (e.g., GET /api/courses/123) to the External Firewall.

- **Proxy**: The request is passed through the External Firewall to Web Server for SSL termination and proxy to the Internal Firewall.
- **Logic**: The Application Server receives the request.
- **Database**: The Application Server calls the Database Server to query for the requested data.
- **Response**: The Database Server returns the data. The Application Server formats it as JSON and sends it back to the Web Browser through the same proxy path.

This pattern is often seen in scenarios such as when users access classrooms or view course materials.

14.3.2 Pattern B: Simple Write (Submit Form)

This flow is used for any form submission that writes or updates data in the database.

- **Request**: A user on their User Device submits a form (e.g., a Tutor creating a new session). The Web Browser sends an HTTPS API request (e.g., POST /api/sessions) to the External Firewall.
- **Proxy**: The request is passed through the External Firewall to Web Server for SSL termination and proxy to the Internal Firewall.
- **Logic & Database**: The Application Server receives the JSON data. It validates the data and makes an INSERT or UPDATE query via PostgreSQL to the Database Server.
- **Response**: The Database Server confirms the write. The Application Server sends an HTTPS (200 OK) response back to the Web Browser.

This pattern is often seen in scenarios such as when users register Tutor Program or when students write class feedback

14.3.3 Pattern C: Secure File Upload

This flow is used to securely upload files (Material, Submission) without overloading the server.

- **Request Upload URL**: A Student or Tutor on their User Device selects a file to upload. The Web Browser makes an HTTPS API request (e.g., POST /api/submissions/upload-url) to the External Firewall.

- **Proxy**: The request is passed through the External Firewall to Web Server for SSL termination and proxy to the Internal Firewall.
- **Authorization**: The Application Server checks the database to confirm the user is authorized to upload this file.
- **Generate URL**: The Application Server makes an internal HTTPS API call to Object Storage, requesting a "pre-signed upload URL."
- **Response (URL)**: Object Storage returns a unique, temporary URL. The Application Server sends this URL back to the Web Browser.
- **File Upload**: The Web Browser uploads the file via PUT HTTP request directly to the Object Storage URL. This traffic does not go through your servers.
- **Confirmation**: Once the upload is complete, the Web Browser makes a final HTTPS API call (e.g., POST /api/submissions/confirm) to the Application Server.
- **Database Write**: The Application Server calls the Database Server to create the Submission or Material record and save the file's path.

This pattern is used when the tutor uploads the learning materials (documents), and students make a file submission

14.3.4 Pattern D: Secure File Download (via CDN)

This flow is used to securely download files, using the CDN for speed.

- **Request Download URL**: A user on their User Device clicks "Download." The Web Browser makes an HTTPS API request (e.g., GET /api/materials/456/download-url) to the External Firewall.
- **Proxy**: The request is passed through the External Firewall to Web Server for SSL termination and proxy to the Internal Firewall.
- **Authorization**: The Application Server checks the database to verify the user has permission to view this file.
- **Generate URL**: The Application Server generates a secure, pre-signed URL for the CDN.

- **Response (URL)**: The Application Server sends this secure CDN URL back to the Web Browser.
- **File Request**: The Web Browser makes a new, direct HTTPS request to the CDN using this secure URL.
- **Origin Pull**: The CDN validates the URL's signature. It does not have the file cached, so it makes a high-speed, private HTTPS request to Object Storage.
- **File Served**: Object Storage serves the file to the CDN, which caches it and serves it to the user's Web Browser.

This pattern is used when users want to download files from the website, such as when students want to download the course documents or a tutor wants to download a student's submission file

14.3.5 Pattern E: External API Call

This flow is used when the system must communicate with an external, third-party service provider.

- **Request**: A user makes an HTTPS API request (e.g., POST /api/auth) to the External Firewall.
- **Proxy**: The request is passed through the External Firewall to Web Server for SSL termination and proxy to the Internal Firewall.
- **Authorization**: The Application Server checks the database to verify if the user is valid to continue
- **External Call**: The Application Server acts as a client and makes a new, outgoing HTTPS API call to the external service (e.g., HCMUT Server for auth, or HCMUT Library Server for search).
- **External Response**: The external server processes the request and returns a response (e.g., "User is valid" or "Here are the search results").
- **Response**: The Application Server uses this external data to complete its task and sends a final response back to the Web Browser.

The system will call to HCMUT service provider when user want to login to the website via HCMUT_SSO or when user want to search for a textbook in the HCMUT library database

14.3.6 Pattern F: Complex Internal Flow (AI-Assisted)

This flow describes a use case that orchestrates multiple internal services.

- **Request**: A Coordinator on their User Device requests AI-assisted course assignments by calling HTTPS request (e.g., POST /api/matching/suggest).
- **Proxy**: The request is passed through the External Firewall to Web Server for SSL termination and proxy to the Internal Firewall.
- **Authorization**: The Application Server checks the database to verify if the user is valid to continue
- **Data Collection**: The Application Server first calls the Database Server to get all relevant data (e.g., student registrations, tutor profiles).
- **AI Call**: The Application Server formats this data and makes a gRPC call to the AI Inference Server.
- **AI Processing**: The AI Inference Server's TensorFlow Serving environment uses the matching model to generate a list of optimal pairings.
- **Response**: The AI Server returns the suggestions to the Application Server, which then sends them back to the Coordinator's Web Browser.

The AI model is used when the coordinator want to match students with an ideal tutor

14.3.7 Pattern G: Non-Interactive / Scheduled Flows

This flow describes automated, background tasks that are not triggered by a direct user action. The list of non-interacting actions include:

- **Backup**: The Backup Center makes API calls on a specific time of the day (usually 2 a.m) to take snapshots of the Database Server and Object Storage.
- **Monitoring**: The Monitoring Server (Prometheus) scrapes metrics from the node_exporter artifacts on all servers to monitor the health of each server. If any service is down, an alert is sent to developers for checking and fixing.

14.4 Deployment

- **Version Control**: Managing version of applications using github
- **CI/CD**: Build and push dist files to remote server via ssh
- **Environment Configuration**: Managed via .env files (with examples provided)
- **API Versioning**: (Implied) Would be handled through URL paths or headers

14.5 Development

- Backend API development can proceed independently from frontend, documentation by Swagger
- Frontend can use mocked API responses during development
- Environment configuration separates development from production settings

14.6 Diagram

Due to the scale of the architecture, the complete development diagram is provided in here

15 Test Cases

15.1 Successful User Login

black	
Test description	Verify user can successfully authenticate and access the system.
Related screens	Login page, Dashboard
Pre-conditions	1. User is not authenticated 2. Login page is displayed

black	
Actions	1. Enter valid email address 2. Enter corresponding password 3. Click login button
Inputs	Valid user credentials
Expected Outputs	• Success message is displayed • User is redirected to dashboard • User session is active
Testing environment	Chrome/Firefox browser on Windows 10/11 or macOS

Practical results

Input credentials

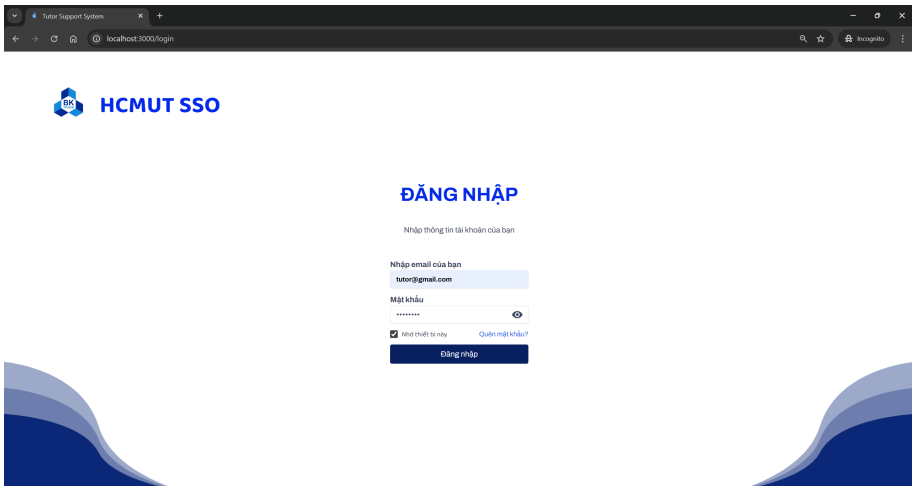


Figure 15.1: User input credentials in login page

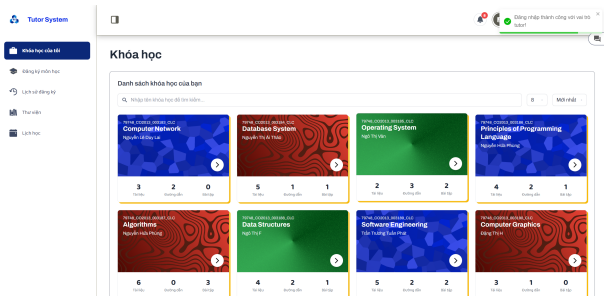


Figure 15.2: Dashboard with role

15.2 Invalid Login Credentials

black	
Test description	Verify system rejects invalid credentials and displays appropriate error.
Related screens	Login page
Pre-conditions	1. User is on login page
Actions	1. Enter invalid email or password 2. Click login button
Inputs	Invalid credentials
Expected Outputs	• Error message is displayed • Input fields are highlighted • User remains on login page
Testing environment	Chrome/Firefox browser

15.2.1 Practical Results

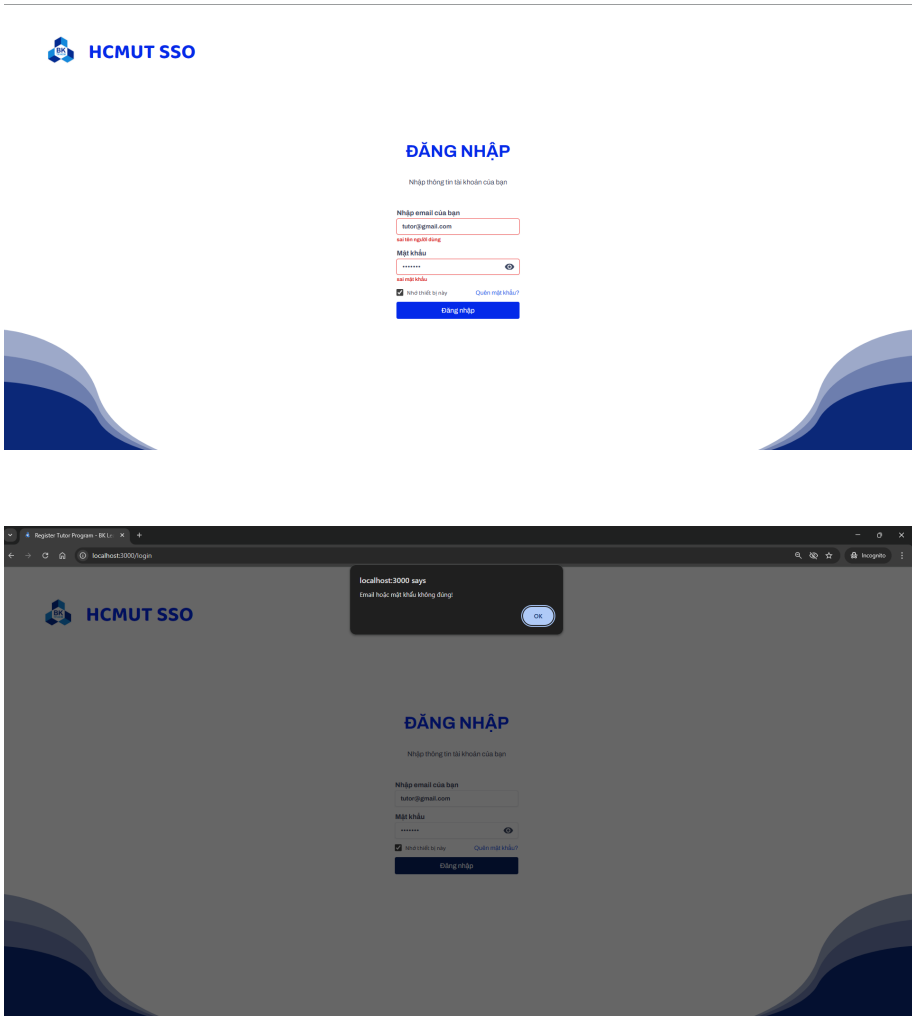


Figure 15.3: Wrong credentials

15.3 Dashboard Display

black	
Test description	Verify dashboard displays correctly based on user role.
Related screens	Dashboard
Pre-conditions	1. User is authenticated

black	
Actions	1. Navigate to dashboard 2. Verify displayed components
Inputs	User role (Student/Tutor/Coordinator)
Expected Outputs	• Role-appropriate menu items are visible • User information is displayed • Statistics and widgets load correctly
Testing environment	Chrome/Firefox browser

Practical results

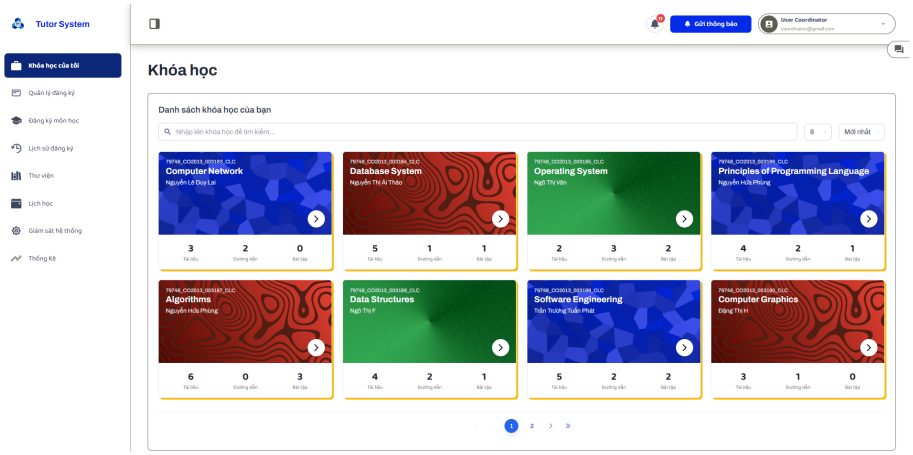


Figure 15.4: Dashboard

15.4 Course List Display

black	
Test description	Verify all courses are displayed with correct information.
Related screens	Course listing page
Pre-conditions	1. User is authenticated 2. Courses exist in system

black	
Actions	1. Navigate to course page 2. View course list
Inputs	None
Expected Outputs	• All courses are displayed • Each course shows title, description, and tutor • Courses are properly formatted
Testing environment	Chrome/Firefox browser

Practical results

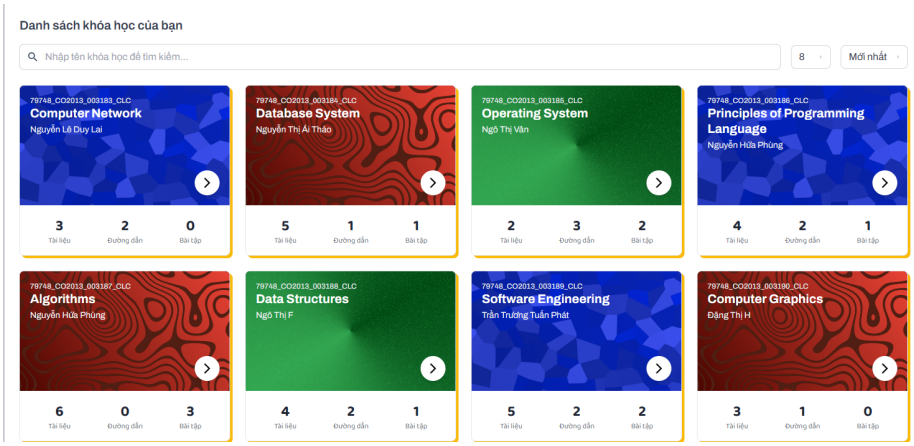


Figure 15.5: Course list display

15.5 Course Search Functionality

black	
Test description	Verify search filters courses correctly.
Related screens	Course listing page
Pre-conditions	1. User is on course page

black	
Actions	1. Enter search term in search box 2. Press enter or click search
Inputs	Course name or keyword
Expected Outputs	• Only matching courses are displayed • Results update in real-time • Empty state shows if no results found
Testing environment	Chrome/Firefox browser

Practical results

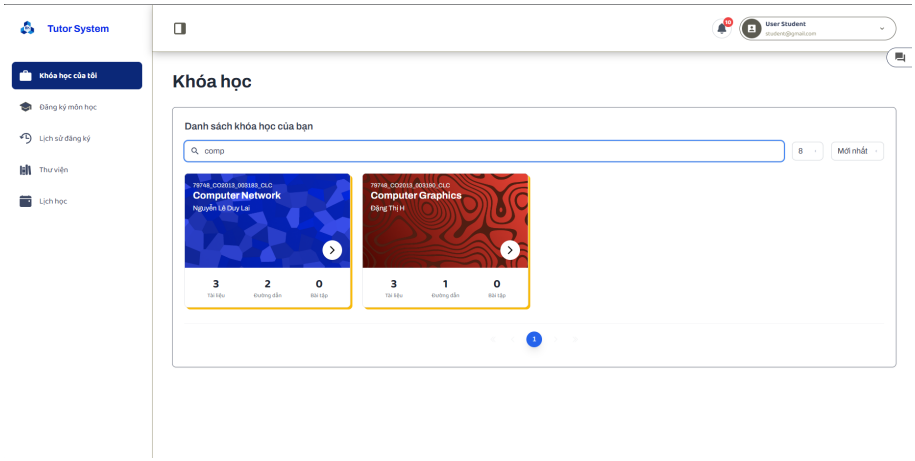


Figure 15.6: Search courses with name

15.6 Course Detail Display

black	
Test description	Verify course details are displayed correctly.
Related screens	Course detail page
Pre-conditions	1. User selects a course from list

black	
Actions	1. Click on course 2. View course information
Inputs	Course selection
Expected Outputs	• Complete course information is displayed • Enrollment status is shown • Action buttons are visible based on user role
Testing environment	Chrome/Firefox browser

Practical results

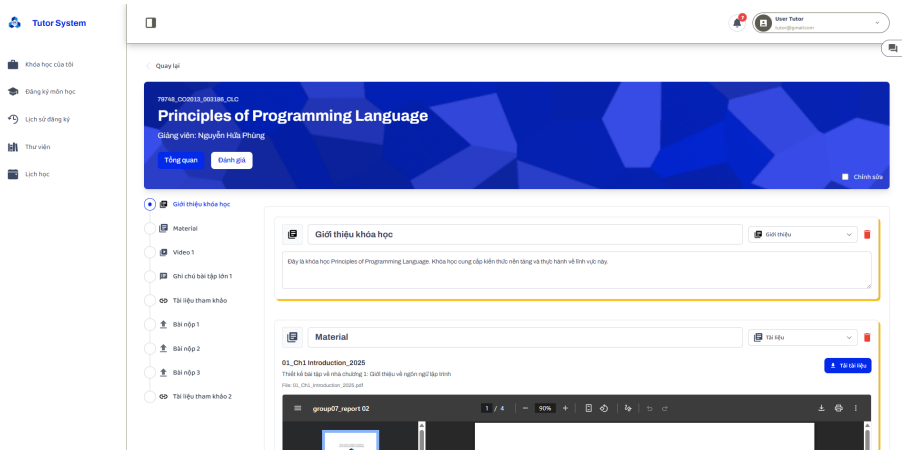


Figure 15.7: Course display-Tutor

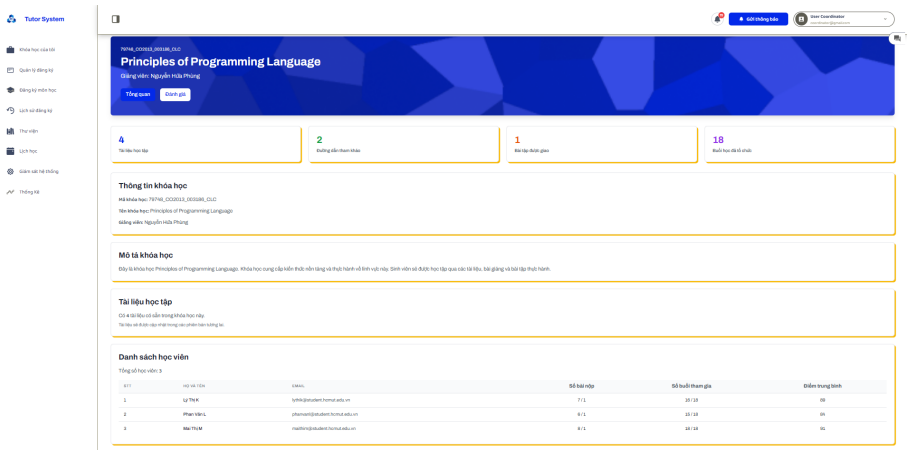


Figure 15.8: Course display for coordinator

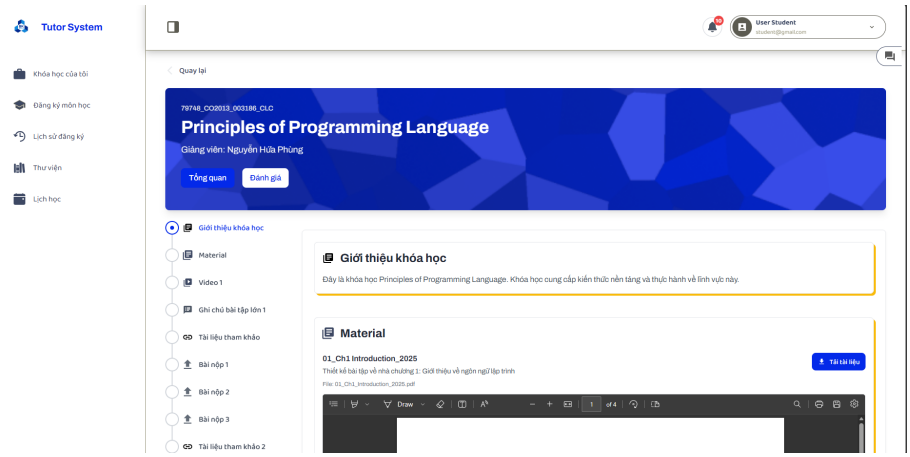


Figure 15.9: Course display for student

15.7 User Profile Management

black	
Test description	Verify user can view and update profile information.
Related screens	Profile page
Pre-conditions	1. User is authenticated

black	
Actions	1. Navigate to profile page 2. Click edit button 3. Modify profile fields 4. Save changes
Inputs	Updated profile information
Expected Outputs	• Profile information is displayed correctly • Changes are saved successfully • Success notification is shown
15.7.1 Practical results **Testing environment**	Chrome/Firefox browser

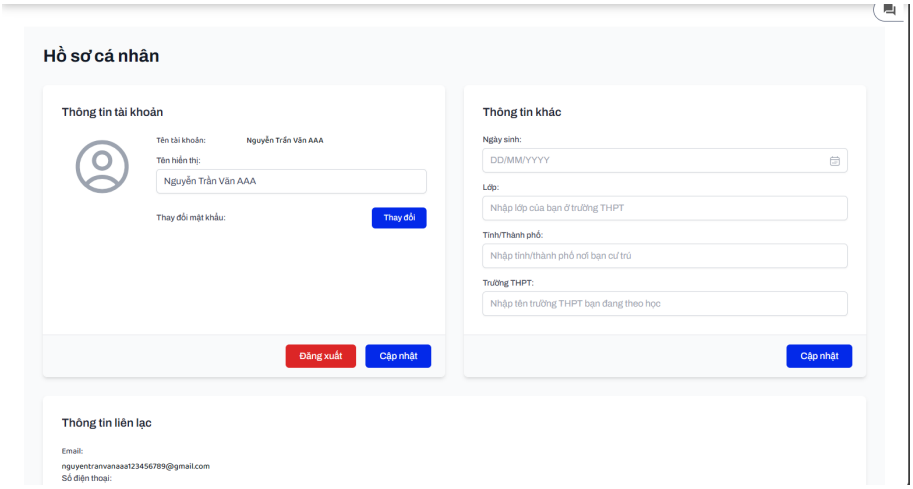
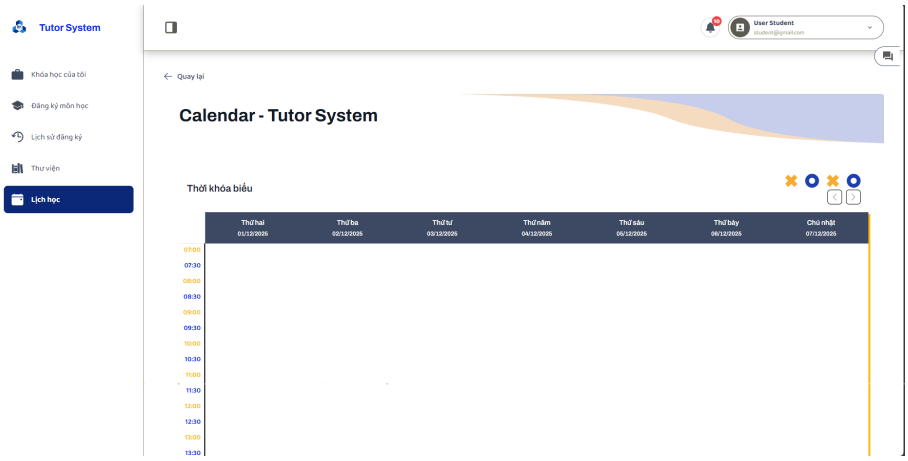


Figure 15.10: Profile

15.8 Schedule Calendar View

black	
Test description	Verify schedule is displayed in calendar format.
Related screens	Schedule page
Pre-conditions	1. User has enrolled courses 2. Sessions are scheduled
Actions	1. Navigate to schedule page 2. View calendar 3. Click on session for details
Inputs	None
Expected Outputs	• Sessions appear on correct dates • Session details are accessible • Calendar navigation works correctly
Testing environment	Chrome/Firefox browser

15.8.1 Practical results



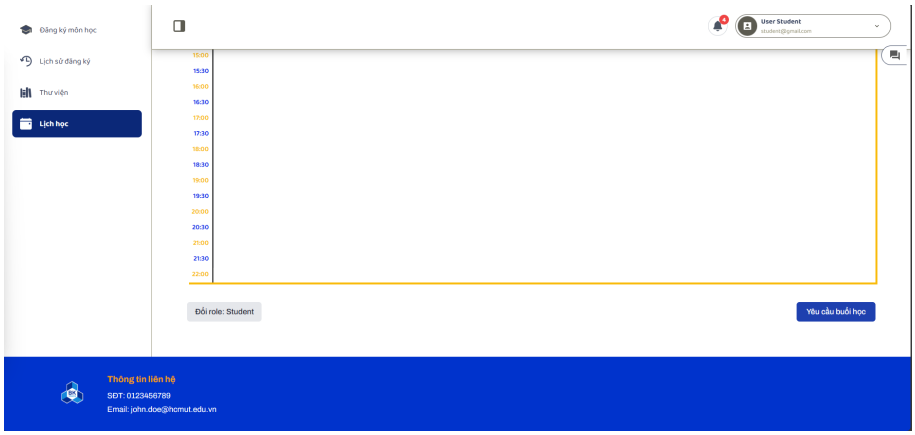


Figure 15.11: Schedule view

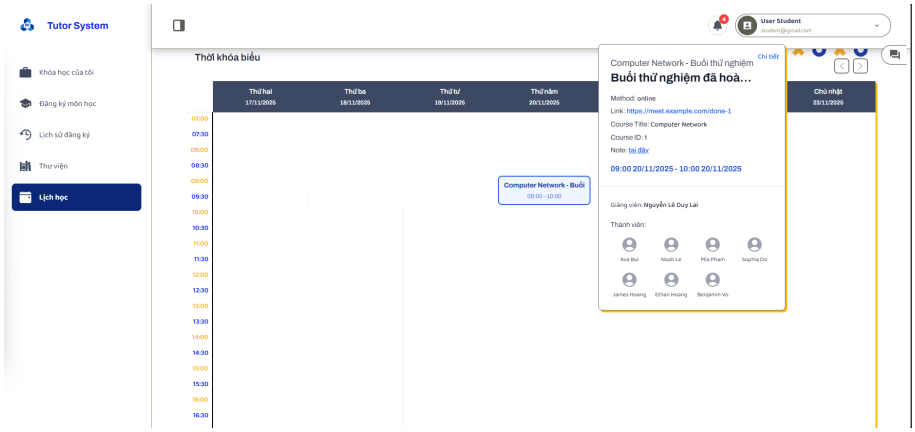


Figure 15.12: Schedule with details

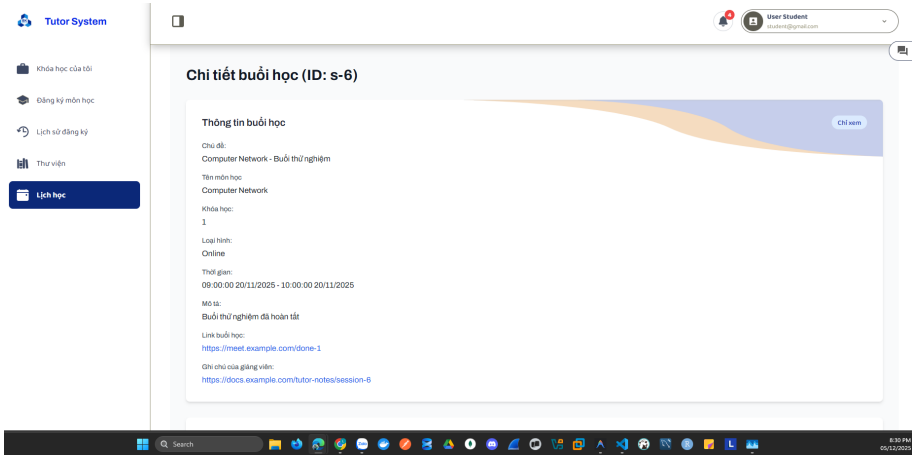


Figure 15.13: Schedule details for students

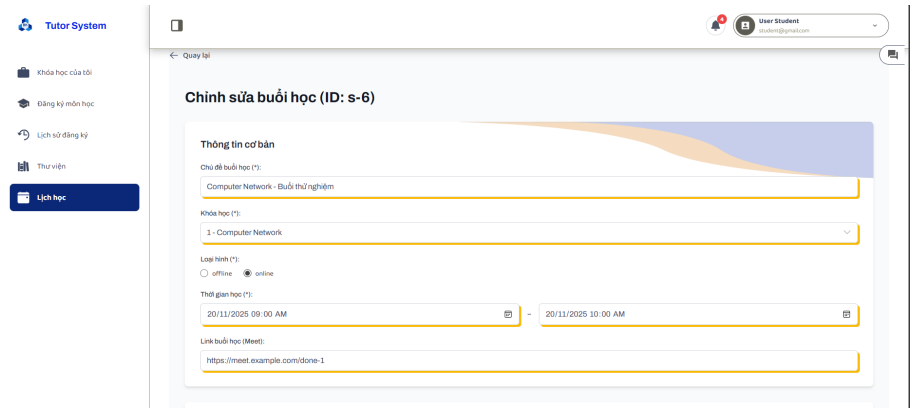


Figure 15.14: Schedule for tutors-1

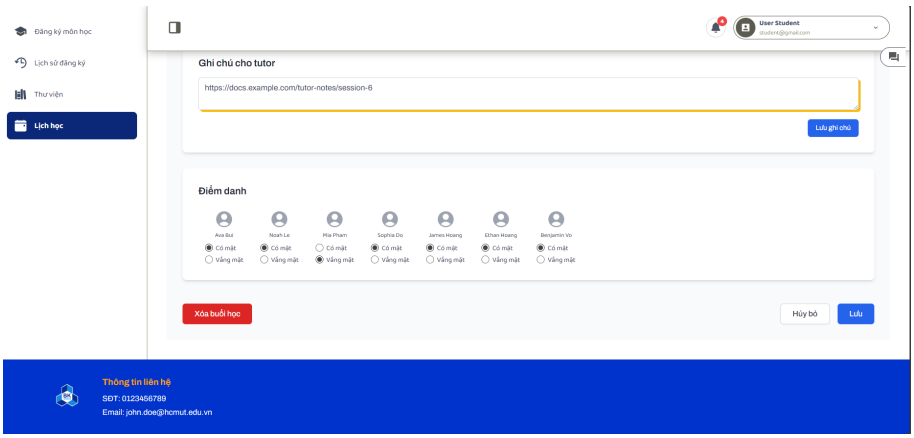


Figure 15.15: Schedule for tutors-2

15.9 Registration History Display

black	
Test description	Verify registration records are displayed with status tracking.
Related screens	Registration history page
Pre-conditions	1. User has registration history
Actions	1. Navigate to registration history 2. View list of registrations 3. Filter by status

black	
Inputs	Filter criteria
Expected Outputs	• All registrations are listed • Status badges are color-coded • Details are accessible for each record
Testing environment	Chrome/Firefox browser

15.9.1 Practical results

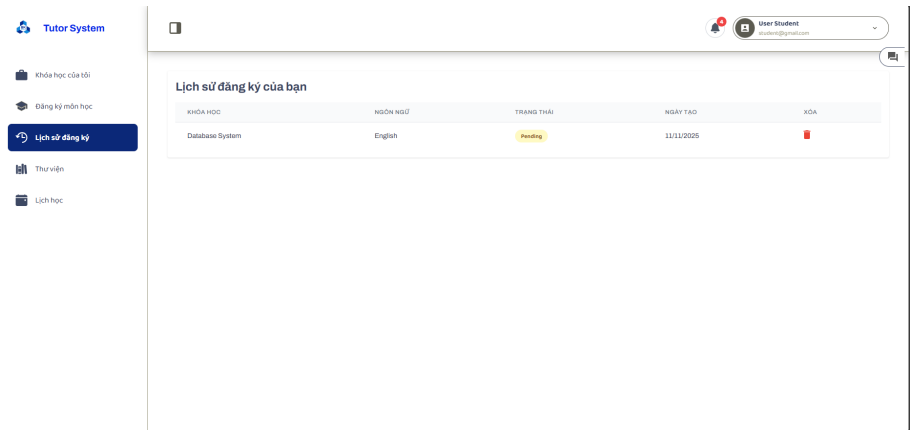


Figure 15.16: Registration history

15.10 New Course Registration

black	
Test description	Verify student can submit course registration request.
Related screens	Registration form page
Pre-conditions	1. User is logged in as student
Actions	1. Fill registration form 2. Select session type and preferences 3. Submit registration
Inputs	Course selection, session type, schedule preferences

black	
Expected Outputs	• Form validation works correctly • Registration is submitted successfully • Confirmation message is displayed
Testing environment	Chrome/Firefox browser

15.10.1 Practical results

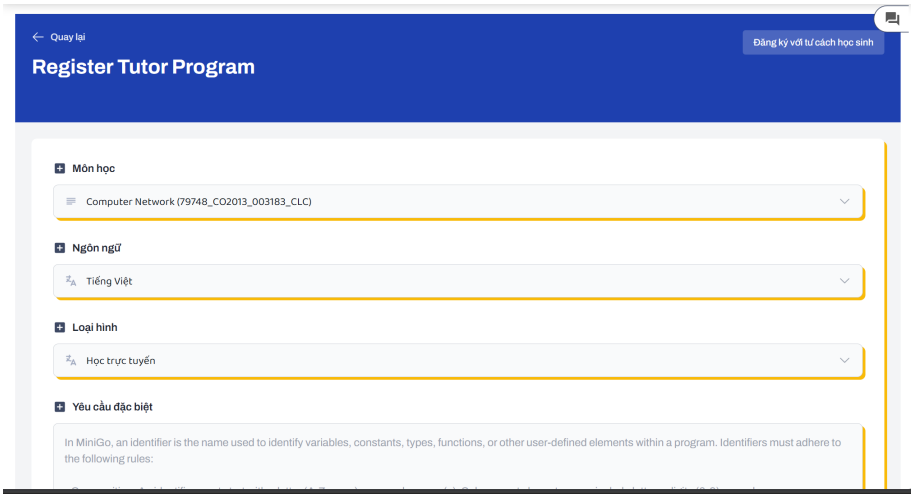


Figure 15.17: New registration for student first access

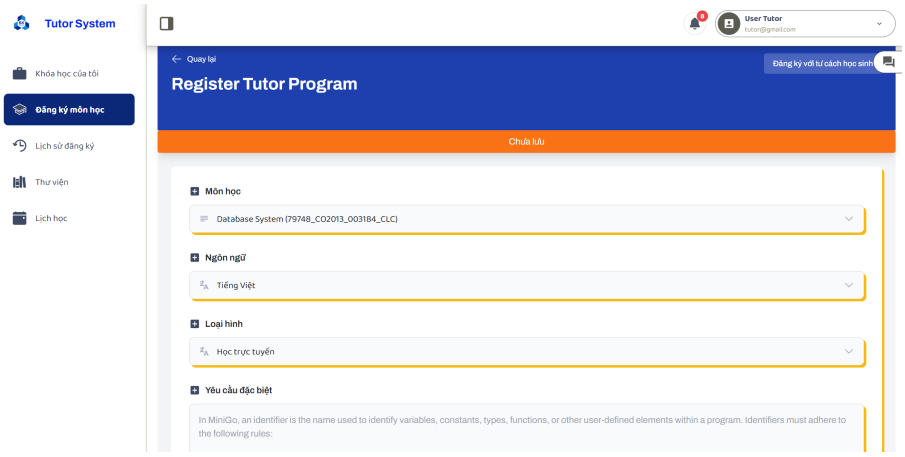


Figure 15.18: New registration student not saved

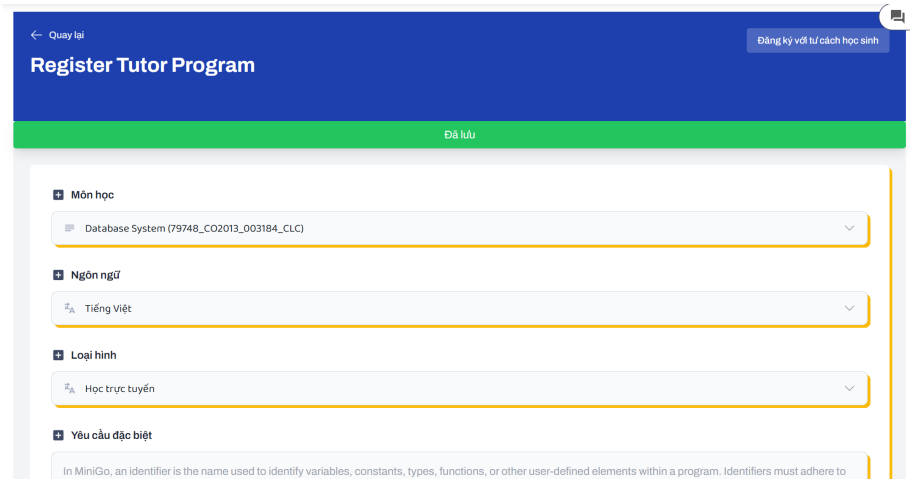


Figure 15.19: New registration from student saved

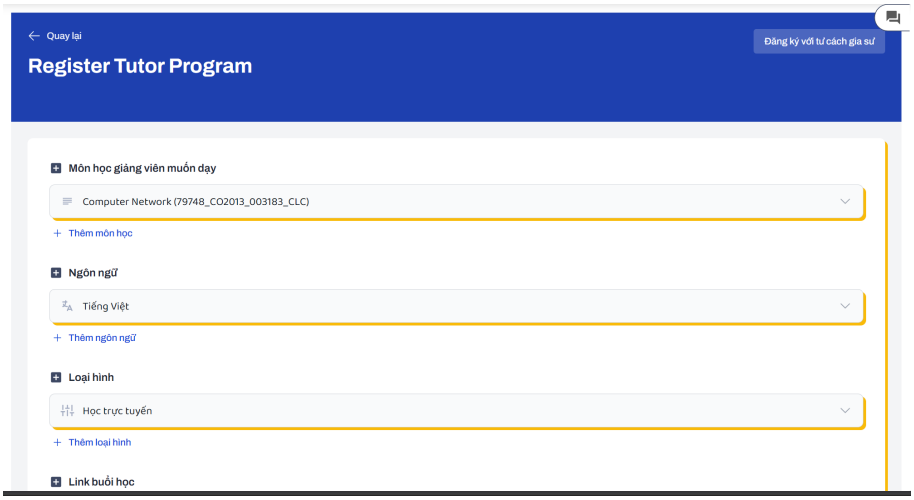


Figure 15.20: New registration tutor first access

15.11 Library Book Search

black	
Test description	Verify library search returns relevant results.
Related screens	Library search page
Pre-conditions	1. User is on library page
Actions	1. Enter search query 2. Click search 3. View results
Inputs	Book title or keyword
Expected Outputs	• Search results are displayed • Book details include availability status • Filters work correctly
Testing environment	Chrome/Firefox browser

15.11.1 Practical results

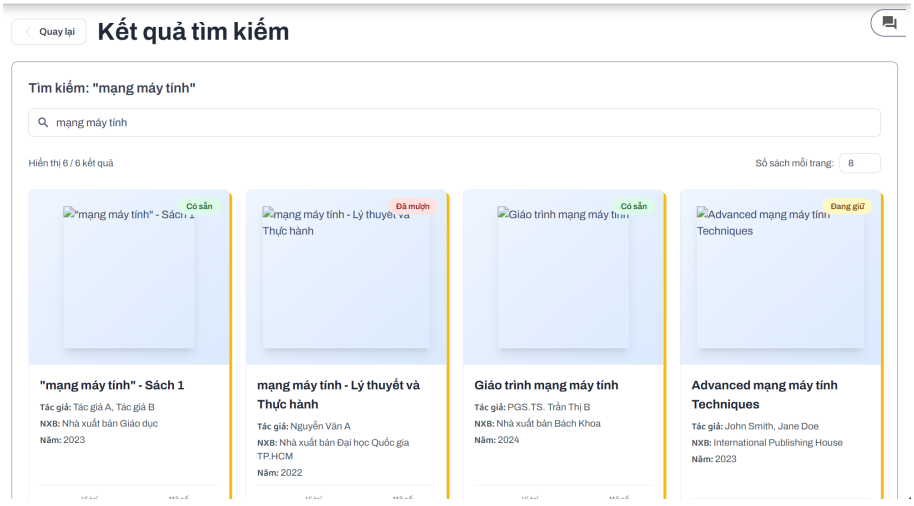


Figure 15.21: Search library

15.12 Statistical Dashboard Access

black	
Test description	Verify authorized users can access statistical reports.
Related screens	Statistical dashboard
Pre-conditions	1. User has coordinator or chairman role
Actions	1. Navigate to statistical page 2. View analytics and charts 3. Apply filters
Inputs	Filter parameters
Expected Outputs	• Dashboard loads with statistics • Charts display correctly • Unauthorized users are redirected
Testing environment	Chrome/Firefox browser

15.12.1 Practical results

Figure 15.22: Statistical list

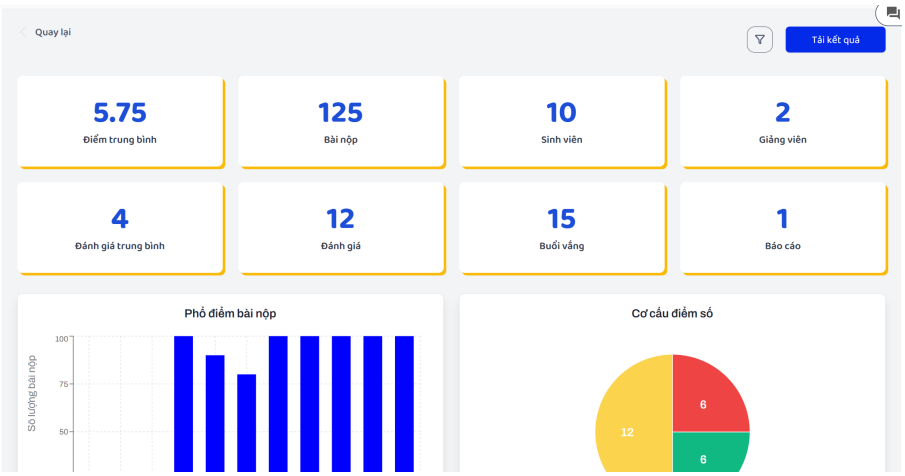


Figure 15.23: Statistacal details

15.13 System Monitoring Dashboard

black	
Test description	Verify system metrics are displayed for authorized users.
Related screens	System monitoring page
Pre-conditions	1. User has administrative privileges
Actions	1. Navigate to system monitoring 2. View performance metrics 3. Check system health indicators
Inputs	None
Expected Outputs	• Metrics are displayed with correct values • Health indicators show current status • Charts update in real-time
Testing environment	Chrome/Firefox browser

15.13.1 Practical results

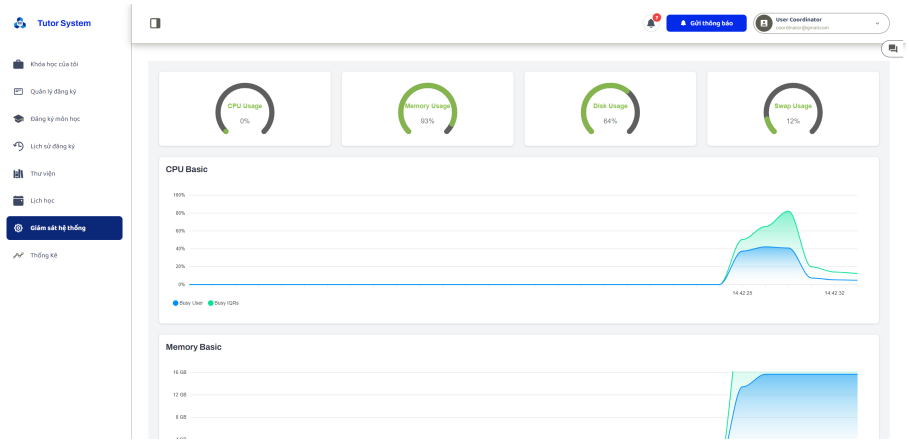


Figure 15.24: System monitoring

15.14 Protected Route Access

black	
Test description	Verify unauthorized access to protected routes is prevented.
Related screens	All protected pages
Pre-conditions	1. User is not authenticated
Actions	1. Attempt to access protected route 2. Verify redirect behavior
Inputs	Protected route URL
Expected Outputs	• User is redirected to login page • After login, user is redirected to intended page • Role-based access is enforced
Testing environment	Chrome/Firefox browser

15.15 Class Assignment Management

black	
Test description	Verify Coordinator can view registrations and manually assign students to a tutor's class.
Related screens	Registration Management (Quản lý đăng ký)
Pre-conditions	1. User is logged in as Coordinator 2. Unassigned student and tutor registrations exist in the system
Actions	1. Navigate to "Quản lý đăng ký" from sidebar 2. Select a specific subject/course to filter lists 3. Select one or multiple students from the "Pending" list 4. Select a target Tutor/Class 5. Click "Ghép đôi" button for manual way or "Gợi ý AI" for AI matching system
Inputs	Student selection, Tutor selection
Expected Outputs	• Lists of Tutors and Students are displayed correctly upon loading • Selected students are moved to the assigned class list in the UI • Success notification is displayed • Notifications are sent to relevant Tutors and Students
Testing environment	Chrome/Firefox browser

15.15.1 Practical results

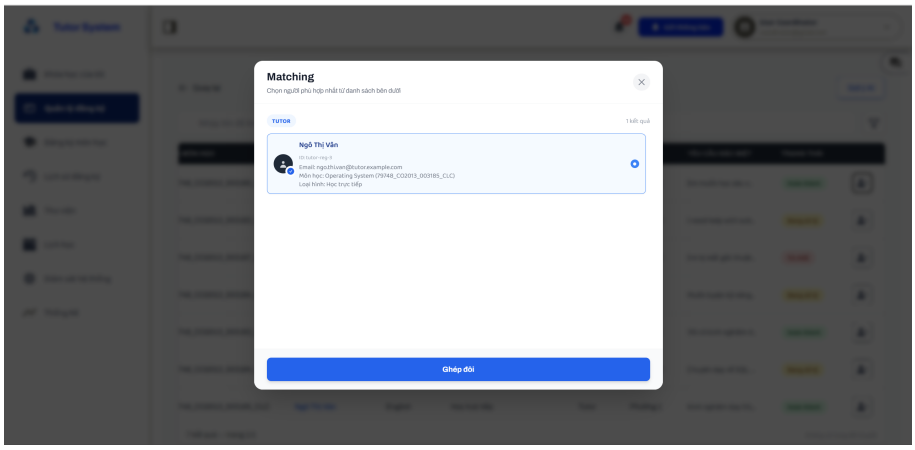


Figure 15.25: Manual Matching

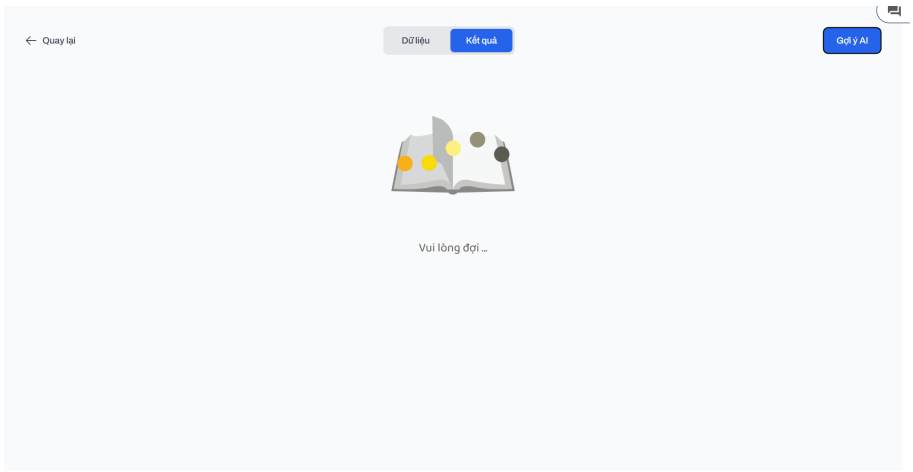


Figure 15.26: AI matching

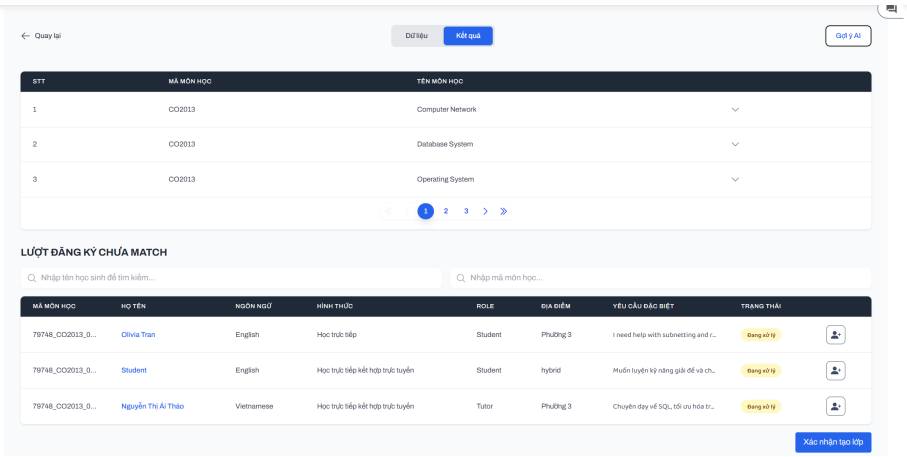


Figure 15.27: Result of matching process

16 Conclusion

The completion of Phase 4 marks a significant milestone in the development of the Tutor Support System at HCMUT. Throughout this stage, the project has successfully fulfill the primary objectives of implementation of an academic support for the university community.

The implementation of the system architecture—structured upon the MVC pattern and reinforced by a dedicated Adapter layer—has proven effective in ensuring strict separation of concerns. This technical foundation allows for seamless integration with institutional services such as HCMUT_SSO and HCMUT_DATACORE while maintaining high maintainability. Furthermore, the deployment strategy, utilizing a three-zone security model on VNG Cloud infrastructure, ensures that the system is not only functional but also secure and capable of handling concurrent user traffic during peak registration periods.

The functional integrity of the platform has been validated through rigorous testing. Over fifteen comprehensive test cases covering authentication, classroom management, course registration, and matching logic demonstrate reliability under various scenarios. Notably, the integration of AI-assisted matching logic offers a scalable solution for personalized tutor-student pairing, setting the stage for future data-driven refinements.

In summary, the Tutor Support System is now poised to serve as a unified platform that reduces administrative overhead and fosters an optimized learning environment. Future work will focus on gathering real-world user feedback and further optimizing the AI matching algorithm to support increasingly diverse academic requirements at HCMUT.

17 Materials

- UI mockup link: Figma Design Mockup
- Activity diagram link: Activity Diagram folder
- State chart diagram link: State chart diagram folder
- Sequence diagram link: Sequence diagram folder
- AI matching process: AI Matching description
- Deployment view link: Deployment view folder
- Implementation view link: Implementation view folder
- Test case practical results link: Test cases practical results folder
- Usecase link: Use cases folder
- Public github source code: Software engineer

18 Declaration of Generative AI Usage

This project utilized Generative AI as a productivity tool, contributing approximately 50% to the drafting and execution phases, under strict human oversight and verification. The application of AI was divided into two primary categories:

- **Report and Documentation:** The core concepts, structural planning, and argumentative logic were exclusively defined by the human team. Generative AI was employed to expand these human-generated ideas into formal academic prose, refine sentence structure, and ensure grammatical accuracy.
- **System Implementation:** In the development phase, AI tools were utilized to establish the foundational file structure and generate boilerplate code for simple implementations. These AI-generated components served as a baseline, upon which the team applied complex logic, debugging, and system integration.

All AI-generated output was rigorously reviewed, tested, and modified by the project team to ensure accuracy and alignment with the project objectives.

References

[1] Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson.

[2] Pressman, R. S., & Maxim, B. R. (2019). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.

[3] Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2nd ed.). Addison-Wesley Professional.

[4] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[5] Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

[6] Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.