Web Applications (082025-DMV)

GROUP ASSIGNMENT

Name **TP Number**	**OOI DUN TZI** **TP071308**
Name **TP Number**	**YAP BOON SIONG** **TP070171**
Name **TP Number**	**CHEAH JUN HENG** **TP071767**
Name **TP Number**	**CHEN XIN ZE** **TP081210**
Intake Code	**APU2F2502CS(DA)**
Lecturer Name	**Mr. Daniel Mago Vistro**

Contents

1.0 Introduction **OOI DUN TZI** 4
1.1 Project Overview 4
1.2 Project Objectives & Scope 5
1.4 Project Schedule (Gantt Chart) 5
2.0 Requirement Specification **CHEN XIN ZE** 6
2.1 Target Audience and Audience Modelling 6
2.2 Use Case Diagrams and Descriptions 7
2.3 System Flowchart 9
2.4 Major Functions of the Web Application 26
3.0 Design and Modelling **YAP BOON SIONG** 30
3.1 Entity Relationship Diagram (ERD) 30
3.2 Wireframes / Mock-ups of Major Web Pages 32
3.2.1 Public 32
3.2.2 Educator 40
3.2.3 Student 45
3.2.4 Admin 55
3.3 Website Navigational Structure 60
4.0 Implementation 62
4.1 CSS for SeaLearner Page Styling **OOI DUN TZI** 62
4.2 Database Connectivity (SQL Queries) 66
4.2.1 Insert **OOI DUN TZI** 66
4.2.2 Select **CHEN XIN ZE** 68
4.2.3 Update **YAP BOON SIONG** 71
4.2.4 Delete **CHEAH JUN HENG** 74
4.3 Explanation Form Validation and Key Source Code Features 76
4.3.1 Public/Core **OOI DUN TZI** 76
4.3.2 Student **YAP BOON SIONG** 99
4.3.3 Educator **CHEN XIN ZE** 133
4.3.4 Admin **CHEAH JUN HENG** 149
5.0 User Guidance **CHEAH JUN HENG** 163
6.0 Conclusion **CHEAH JUN HENG** 188

8.0 Appendix 189
8.1 Proposal Report 189
9.0 Workload Matrix 189

1.0 Introduction **OOI DUN TZI**

1.1 Project Overview

Figure 1 ： Logo of SeaLearner

SeaLearner is a web based learning platform that is design for able to function as a complete educational ecosystem, and it connects three primary user which is administrators, educators, and students.

To know more about administrator role is that it responsible for system governance, quality assurance, and user management. Moreover, the educator role will be act as the content creator, that responsible for designing and managing their own courses. Then the student role will be the primary consumer of educational content, that engaging with the courses and participating in the platform community.

The way that shows unique for SeaLearner platform is that the integration of gamification mechanics, this enables to provide the user with engagement and motivation. Therefore, to achieve that the system will incorporate features like the badge, pints, and competitive leaderboards. As a result, this approach aims to create a dynamic and rewarding learning experience.

1.2 Project Objectives & Scope

SeaLearner Project Objectives

- To empower educators: To enabling educators to efficiently design, develop, and manage high quality multimedia rich courses.
- To improve Student Motivation: To promote student involvement and determination via a systematic gamification platform.
- To Cultivate an Engaging Community: To create a social forum that is integrated and students and educators can pose queries and exchange knowledge thus creating a favourable learning community.
- To Secure Platform Integrity: To achieve high level of quality, safety, and accuracy by way of active administrative control and closed ended administrating feedback mechanism.

SeaLearner Project Scope

- **Educator Functionality:** Educator able to creation, management, and update of the multimedia rich courses through an SeaLearner creator studio.
- **Student Functionality:** Student able to access to courses, participate in a gamified reward system that allows students to collect the coins, badges, and leaderboard. Also, ability to provide content feedback.
- **Community Features:** In community will be able to have interaction, discussion, and knowledge sharing between the student and educator.
- Administrative able to use those tools like user management, content moderation, course quality assurance, feedback tracking, and platform integrity management.

1.4 Project Schedule (Gantt Chart)

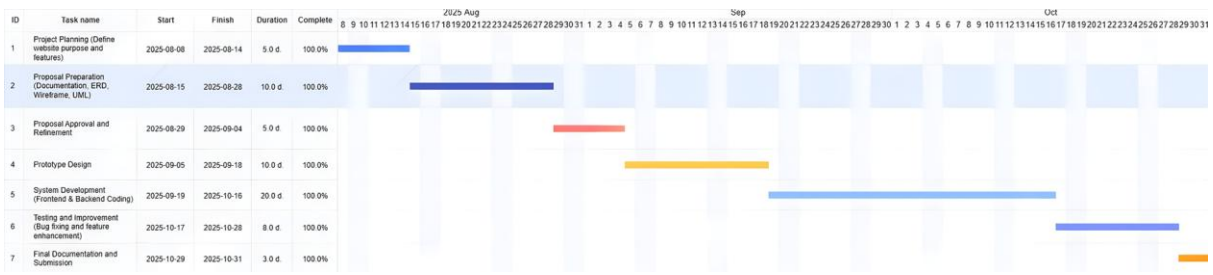

Figure 2: SeaLearner Gantt Chart

2.0 Requirement Specification **CHEN XIN ZE**

2.1 Target Audience and Audience Modelling

Target Audience:

1. **Students:** Those who access the site, and enrols in the courses, study the lessons, pass the quizzes through gamified learning process.
2. **Educators:** Professionals or individuals familiar with the topic, and perform course creation, maintenance and publication.
3. **Admins:** People capable of managing the platform, comments by users, and system content.

Audience Modelling:

Students:

- **Goals:** Access public and private courses, progress through modules, complete quizzes, earn badges and coins, and participate in discussions.
- **Key Interactions:**
    - Browse course catalogue
    - Join public courses and unlock private courses using earned coins
    - View modules, complete quizzes, receive coin, receive badges
    - Access leaderboard to compare achievements
    - Participate in community discussions
    - Manage personal profile
    - Submit feedback or complaints to administrators

Educators:

- **Goals:** Create, edit course content, and engage with learners.
- **Key Interactions:**
    - Create and edit course
    - View student progression (complete and incomplete)
    - Engage in the community discussion board
    - Manage personal profile

Admins:

- **Goals:** Ensure platform quality, maintain user and content integrity, and manage public-facing elements.
- **Key Interactions:**
  - Manage courses (view and delete courses)
  - Manage community posts (delete comment and reply)
  - Manage user accounts (active or inactive)
  - Review student feedback and complaints
  - Manage advertisements (create and delete)

2.2 Use Case Diagrams and Descriptions

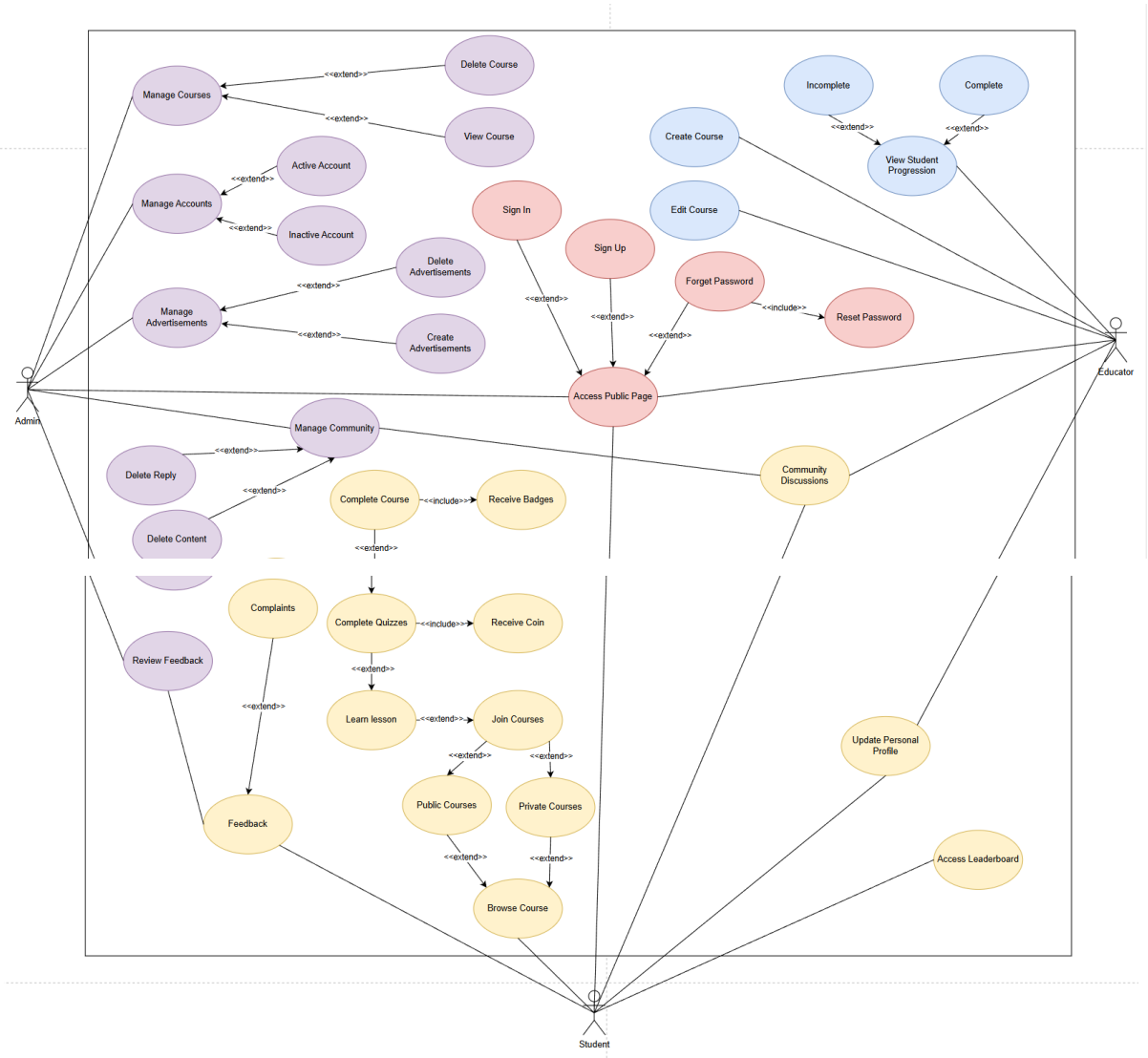

Figure 3: Use Case Diagram

Students:

Students aim to join public and private courses, go through learning modules, pass the quizzes, and receive badges and coins as rewarding their accomplishments. They can browse the course catalogue; enrol in free courses and access private courses with the coins they receive. The students can complete quizzes to get coins and badges as they progress through the lessons and can see their position on the leaderboard to compare performance with others. Moreover, students will be able to use community to get discussion, manage personal profile, and they will be able to provide feedback or complaints to administrators to assist in the improvement of the learning process.

Educators:

The educator has the responsibility of making and managing course material and interacting with learners during the learning process. They can create and edit courses to make sure that the materials are up to date and relevant. Educator can also see the progress of students, both completed and incomplete, to track learning results. Moreover, they will also be involved in the community discussion board to engage students and guide them. Educators are also able to update their own profiles to keep valid and professional information.

Admins:

Admins monitor the whole platform to assure quality, integrity of content, and control user interactions. They have the duties of controlling courses through viewing and deleting course materials (if necessary) and managing community posts to eliminate unsuitable posts or responses. Admins also do user account management by deactivating and activating accounts depending on the platform policies. Moreover, they analyse student feedback and complaints to resolve issues and increase system reliability. Admins also control advertisements on the platform by adding or removing adverts content as they find necessary to keep the environment balanced and engaging environment.

Access Public Page:

The Access Public Page gives the guest and other users a chance to view the main interface of the platform and can use the basic account features. On this page, the user will be able to view some general information, log in or register a new account. In case of passwords being forgotten, users are given an option of a password recovery process and can renew their

passwords safely. This feature acts as the gateway to the system, and easy access is granted to learning material after authentication.

2.3 System Flowchart

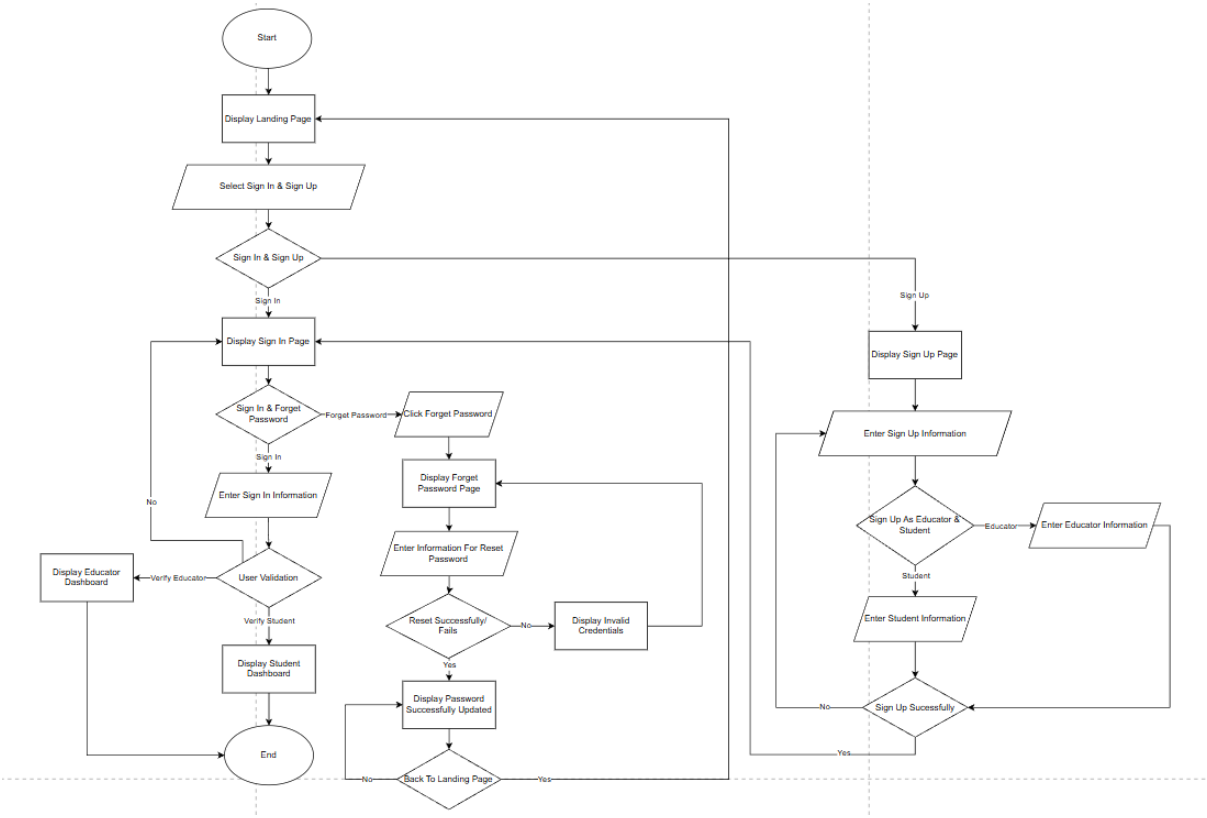

Figure 4: Public landing page (sign in and sign up）

This is “SeaLeaner” sign-in and sign-up page, sign up page will require user enter personal information, if user is registered as educator or student, then when they sign-in, it will bring them to the corresponding dashboard page. If user forget password, they can reset password.

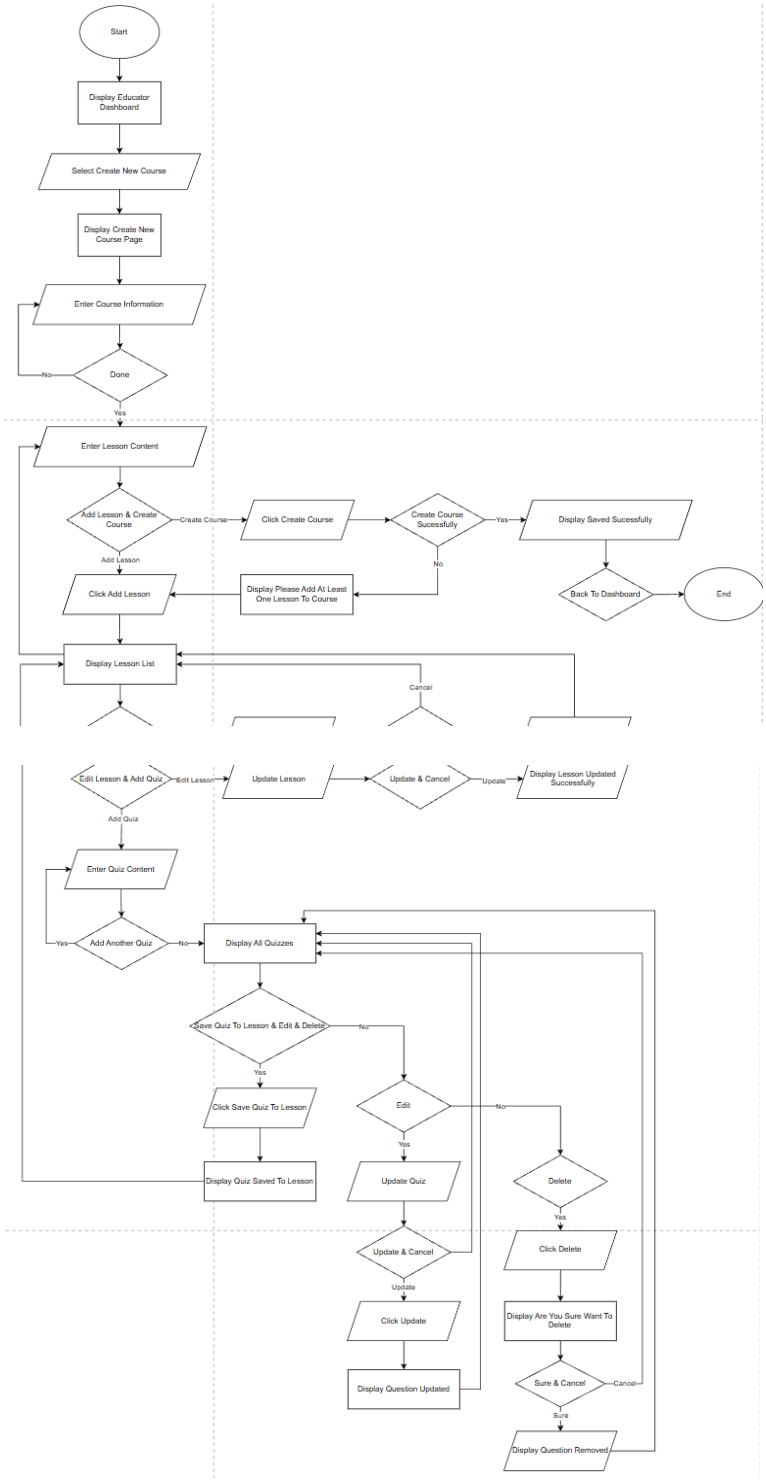

Figure 5: Educator Create Course

This is creating course page, educator need to enter course information and lesson information, and it able to adds multiple quizzes within a course. Educator also able to double check the information they enter whether is correct or not by edit or delete the lesson or quiz, before they create a course.

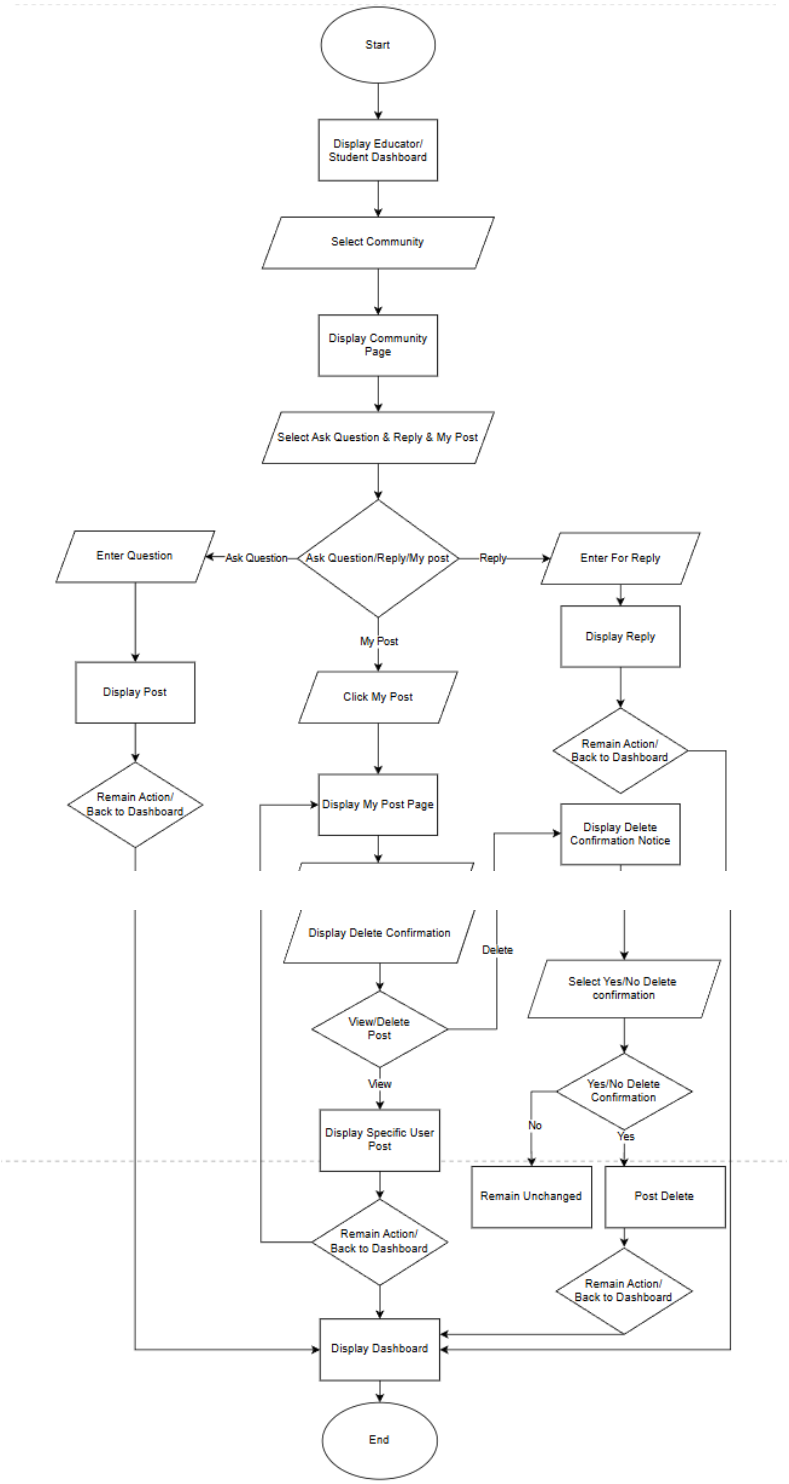

Figure 6: Educator and student community

This is community page is for educator and student to ask question, reply question, and check the previous asked questions. This page able to let them delete the posts they made before.

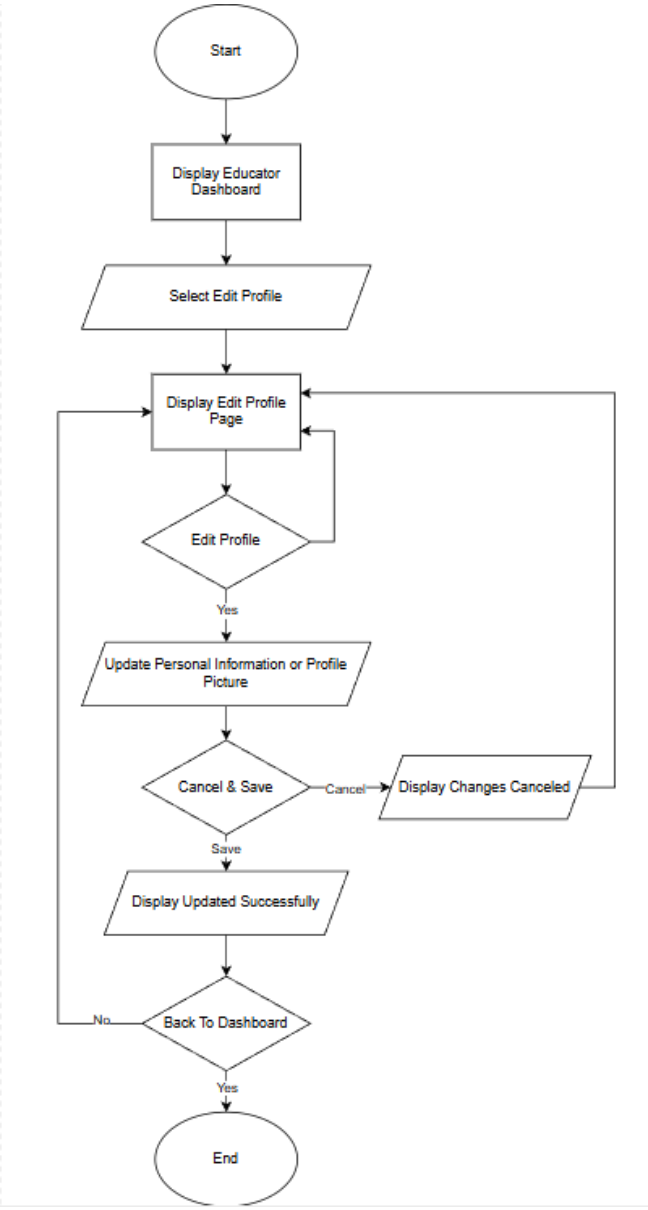

Figure 7: Educator Edit Profile

This is editing personal information profile, educator able to update his information accordingly such as “Full Name”, “Age”, “Gender”, “Education Qualification”, “Graduated University”, and “Profile Picture”.

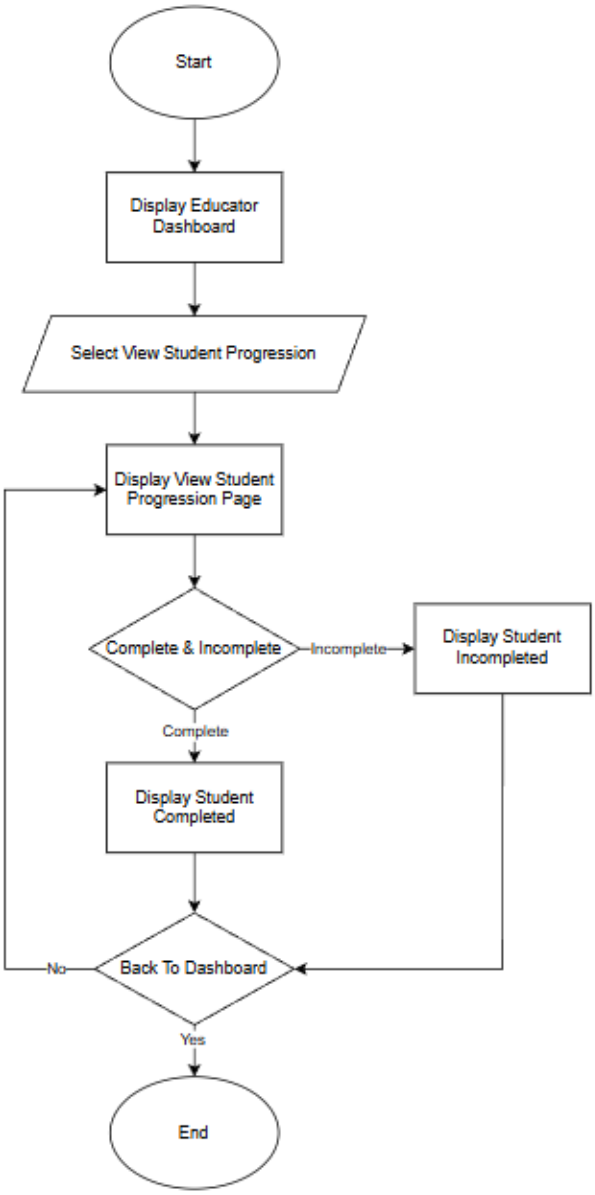

Figure 8: Educator View Progression

This is page of view student course progression whether they are “Complete” or “Incomplete”.

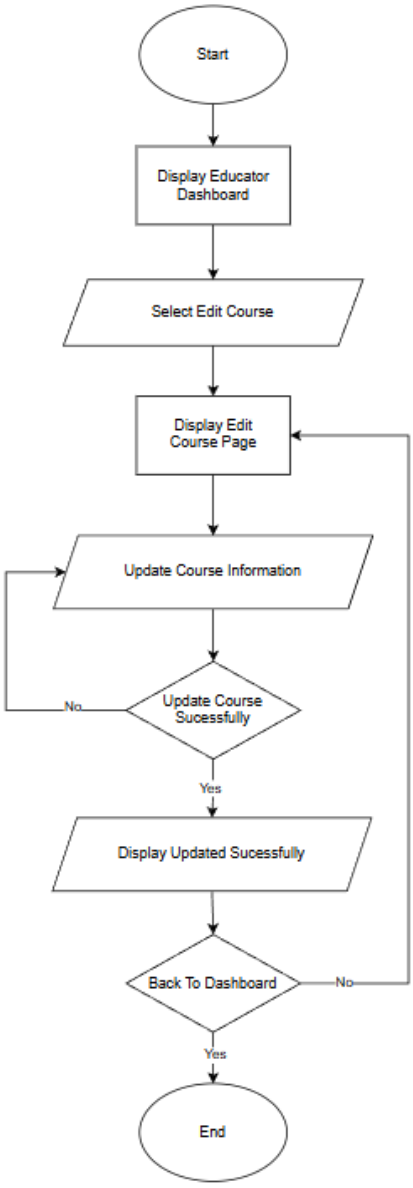

Figure 9: Educator Edit Course

This educator edit course page is same as the create course page, but in this page, they can edit or update the course they made before. The “Course Title” and “Course Type” in this page is unable to changed.

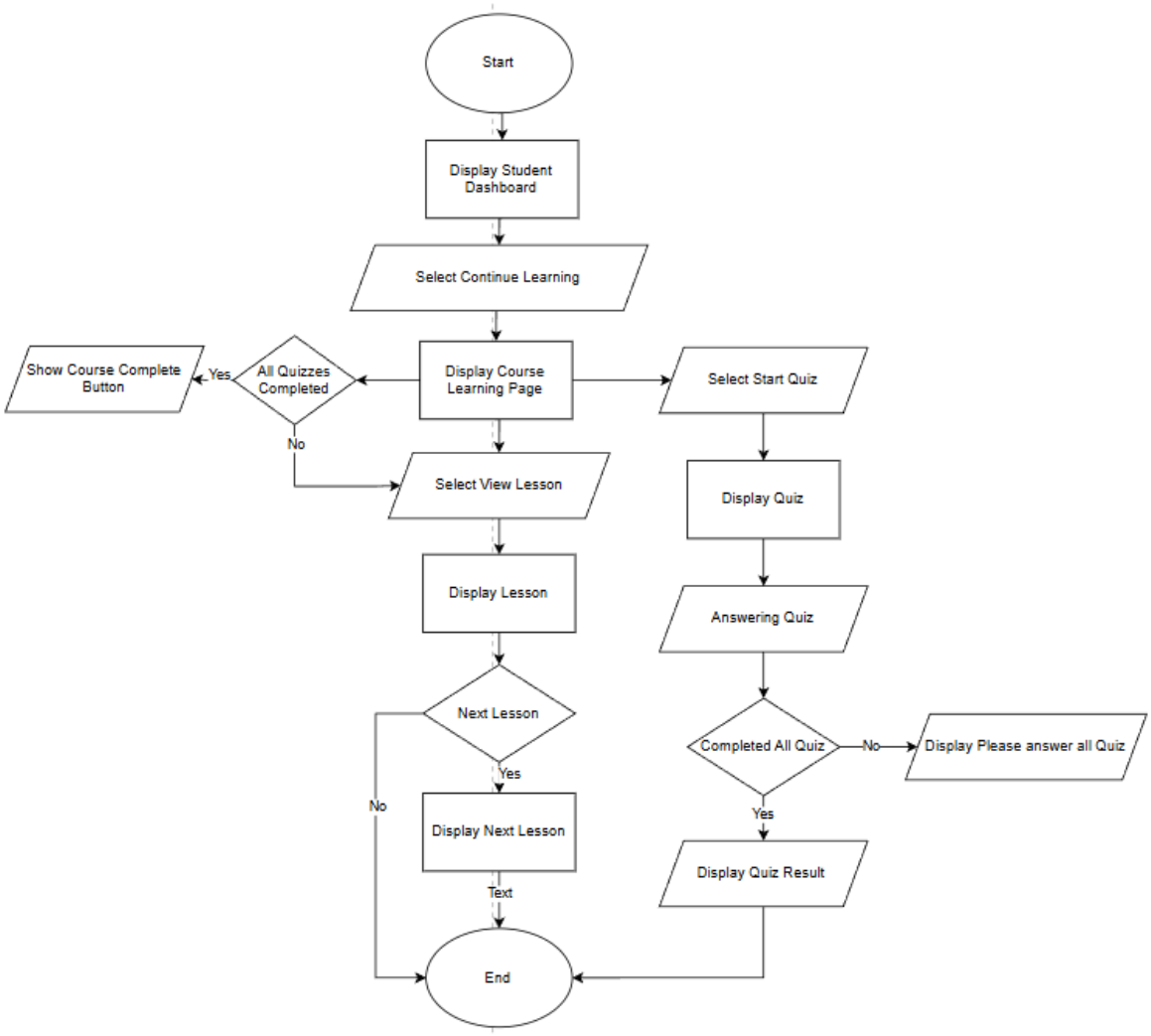

Figure 10: Student Continue Learning and Attempt Quiz

After student they joined the course, they can click the button “Continue Learning” to start learning by view the lesson and start attempt the quiz, the result will display once student submitted the quiz.

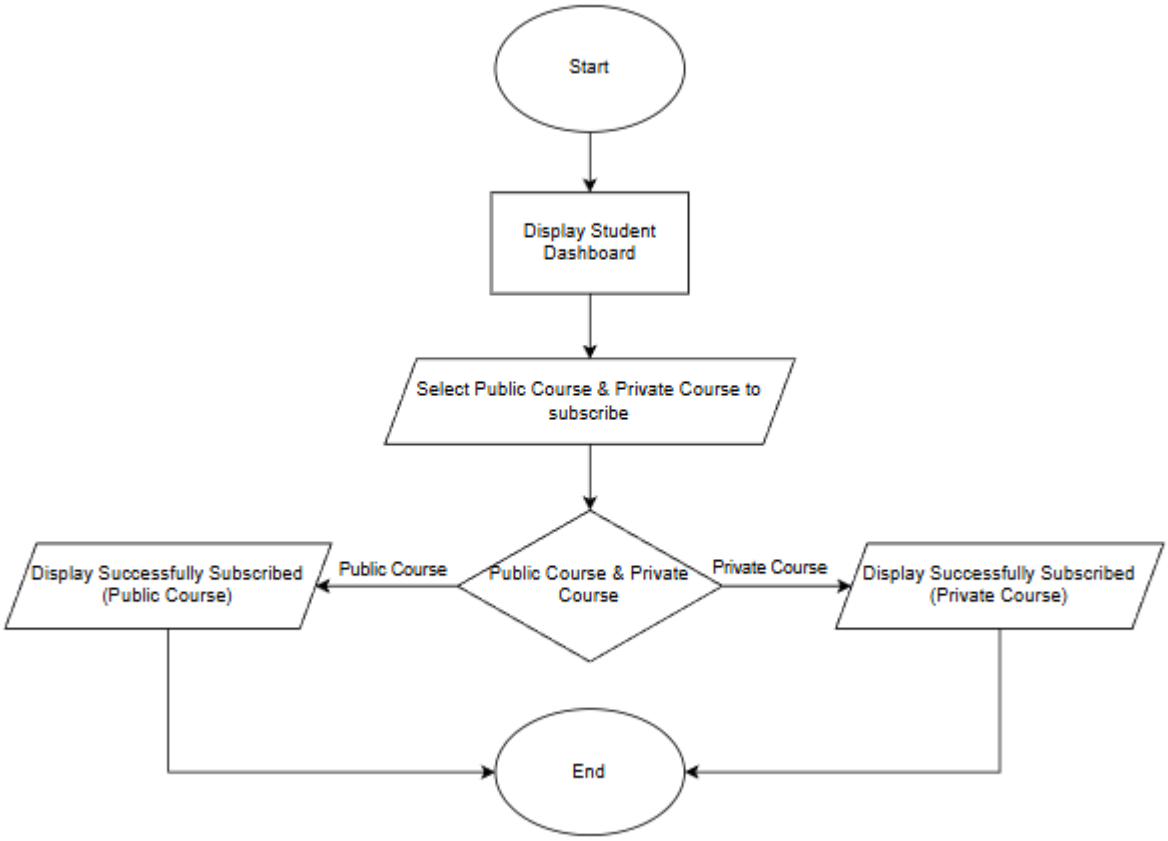

Figure 11：Student Public and Private Course Subscribe

Both pages are for student to filter the courses type by “Public” or “Private”. After they joined the course, a successfully subscribed message will be displayed.

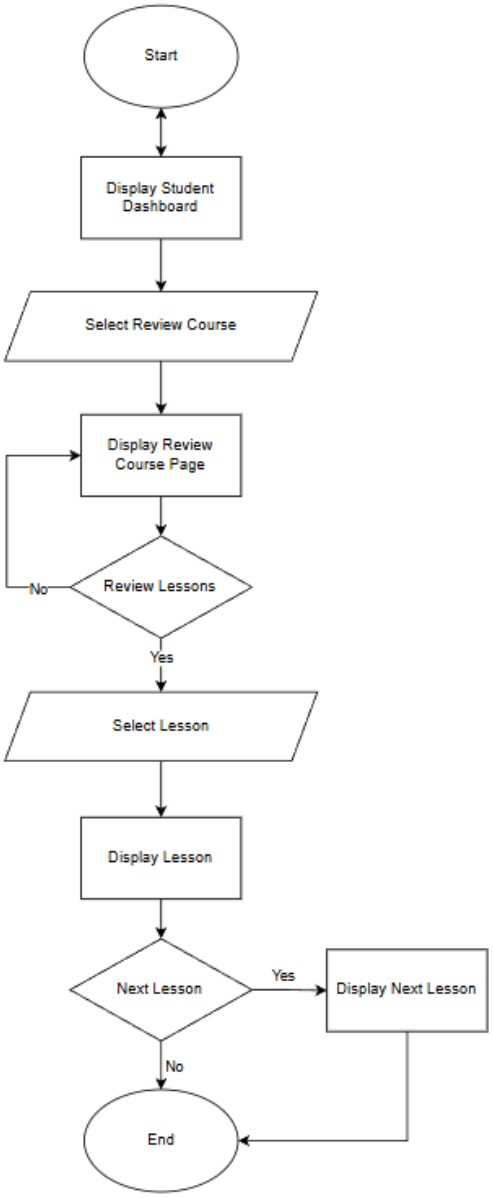

Figure 12：Student Review Course

After student completed a course, they able to review back the lesson but not attempt again to the quiz.

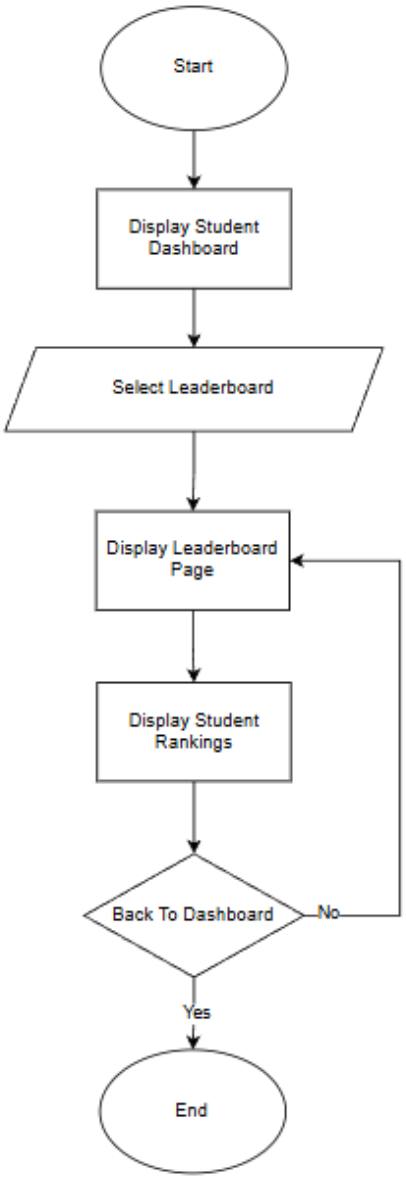

Figure 13 ：Student Leaderboard

This is leaderboard page that student can see the current ranked they are in based on badges earned and courses completed.

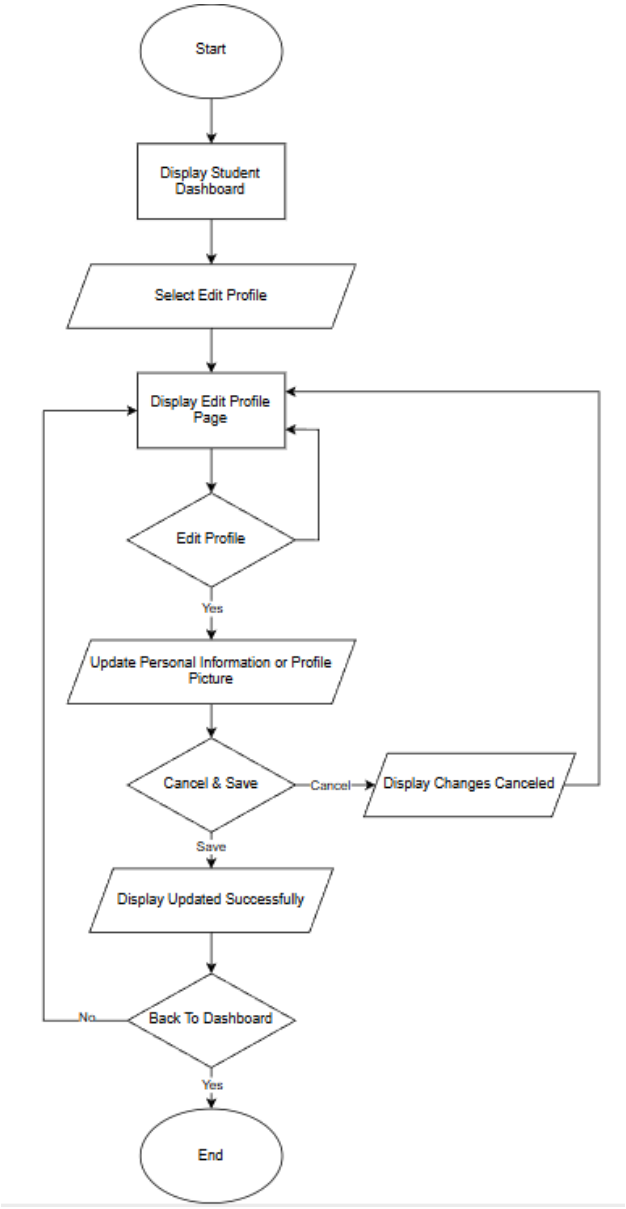

Figure 14: Student Edit Profile

This is editing personal information profile, student able to update his information accordingly such as “Full Name”, “Age”, “Gender”, “School”, “Interest Subject”, and “Profile Picture”.

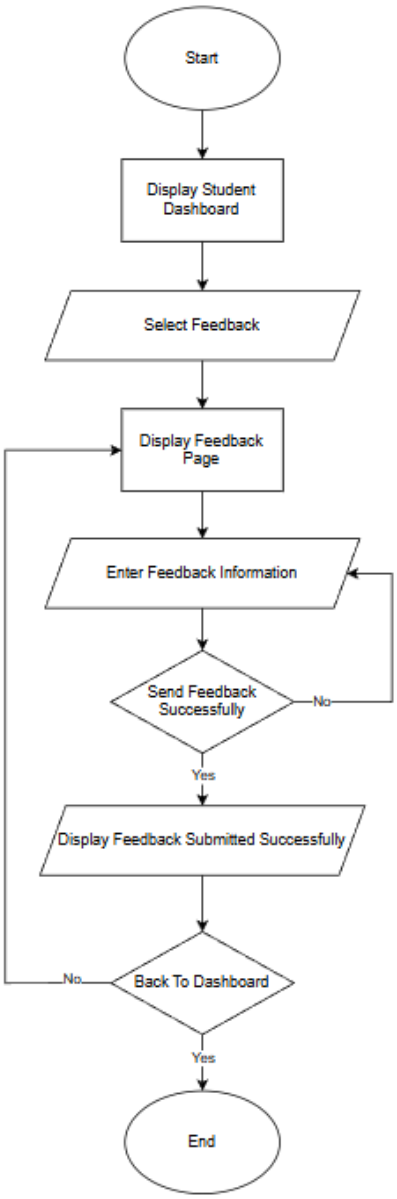

Figure 15: Student Feedback

This is the student feedback page that student able to give some feedback to improve their learning experience according to selecting the “Category”, “Priority”, “Subject”, and “Description”.

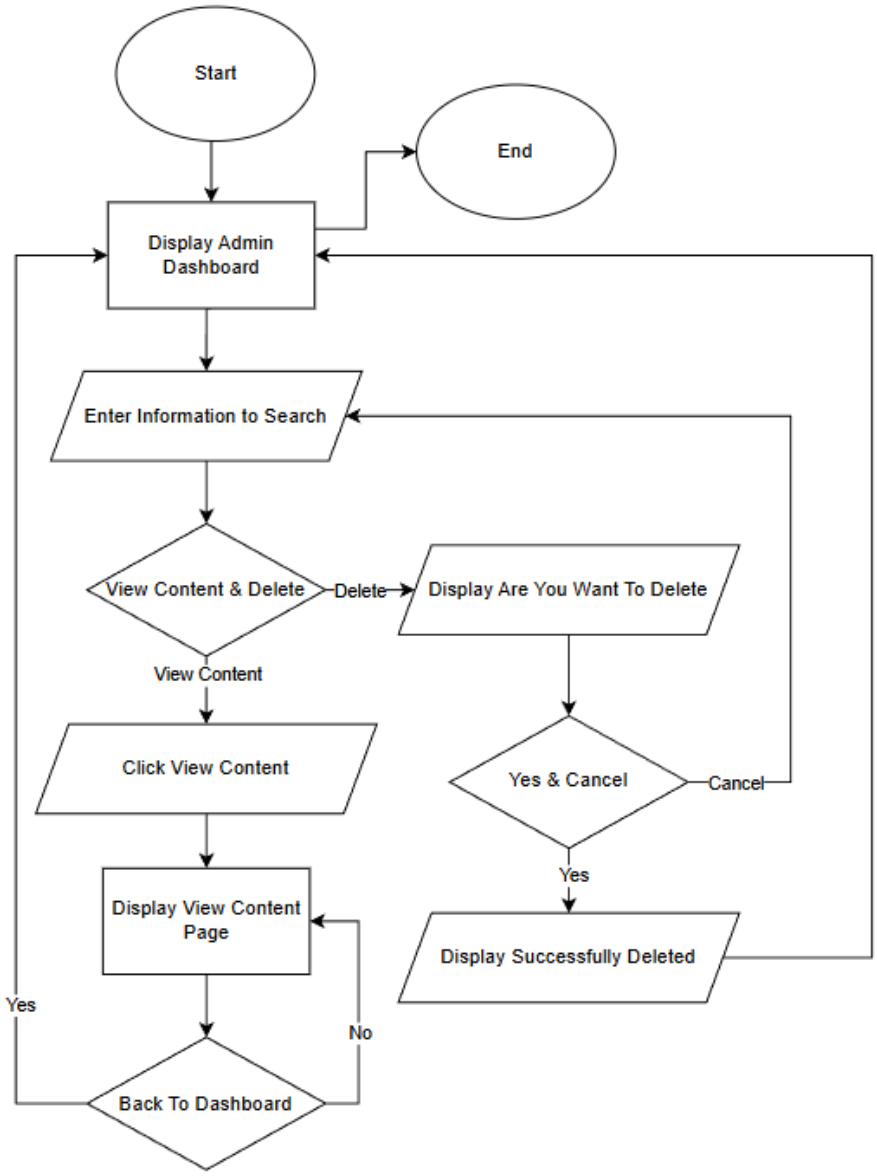

Figure 16: Admin View Content

This page is course management page that able admin to search or filter the course, and its able admin to view the course that created by an educator including all lessons, quizzes, and correct answers, admin has right to delete the course.

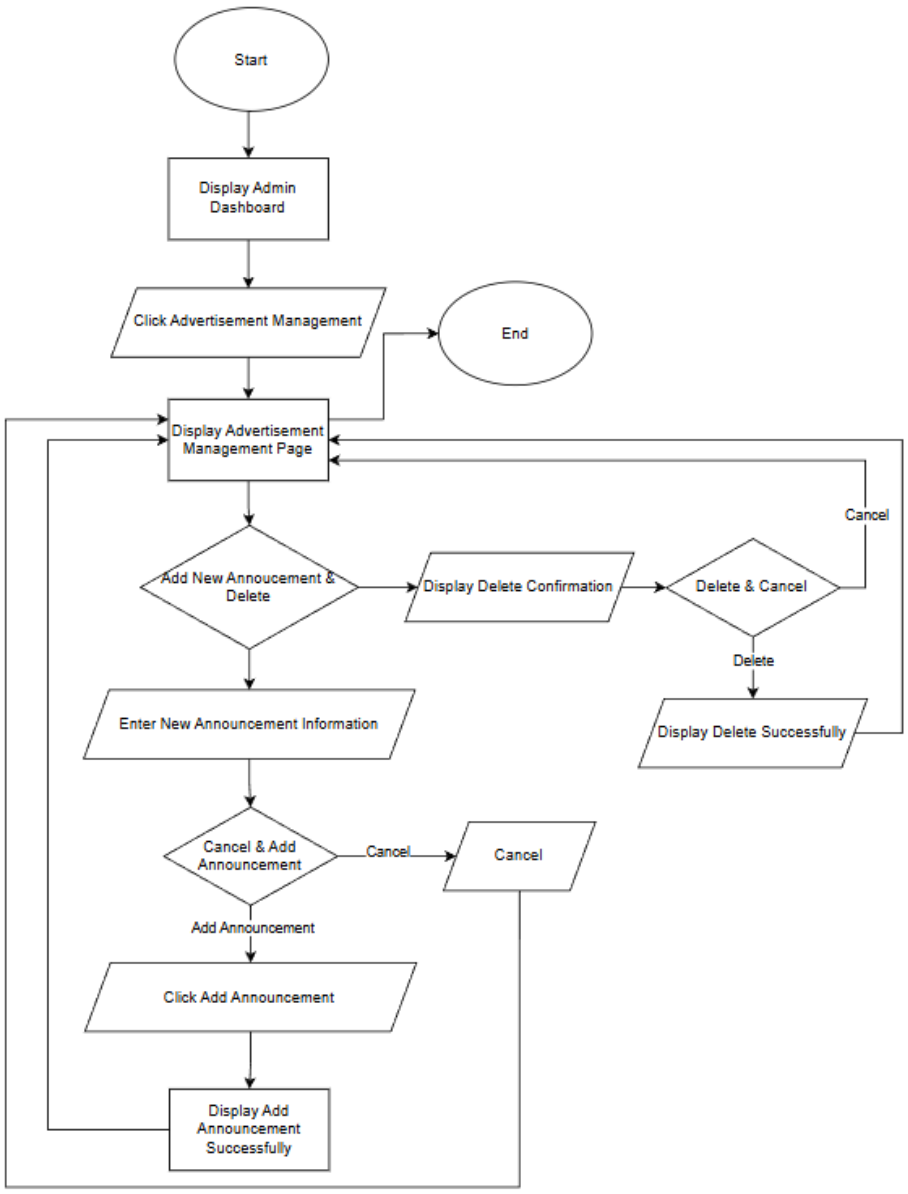

Figure 17: Admin Advertisement Management

This is advertisement management page, admin can add new advertisement by “Title”, “Content”, “Type”, and “Announcement Image” or delete an advertisement.

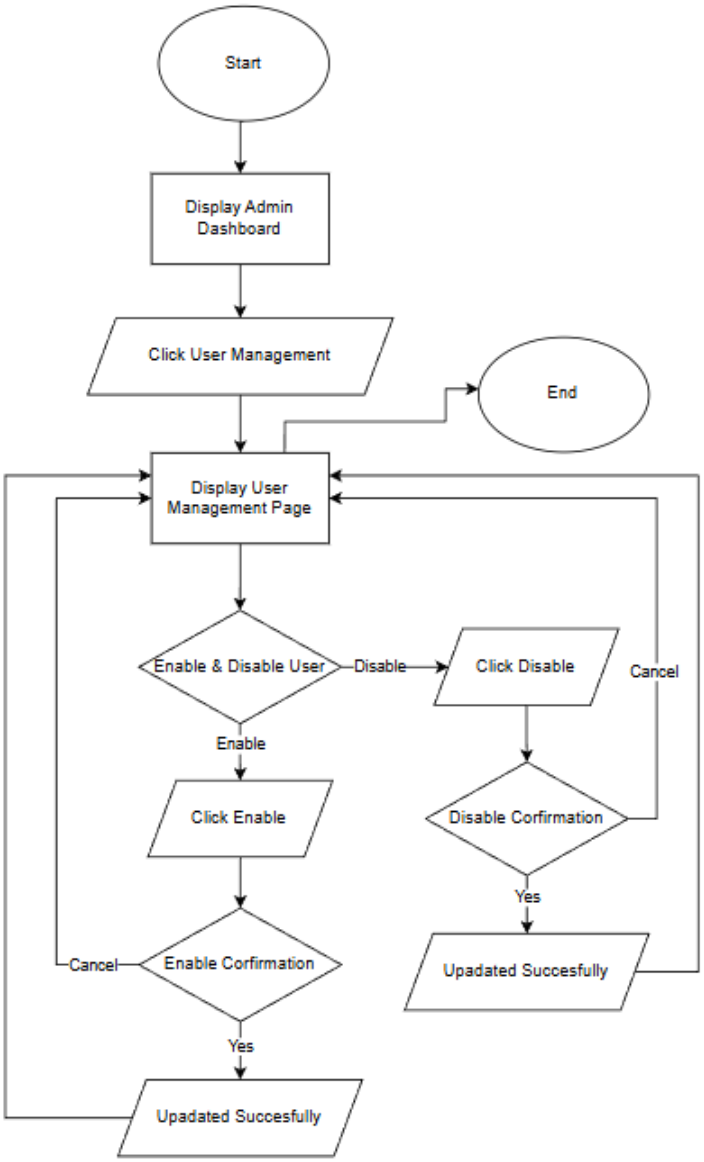

Figure 18: Admin User Management

This is user management page, admin able to search a user by name or email to “Inactive” or “Active” a user including student and educator.

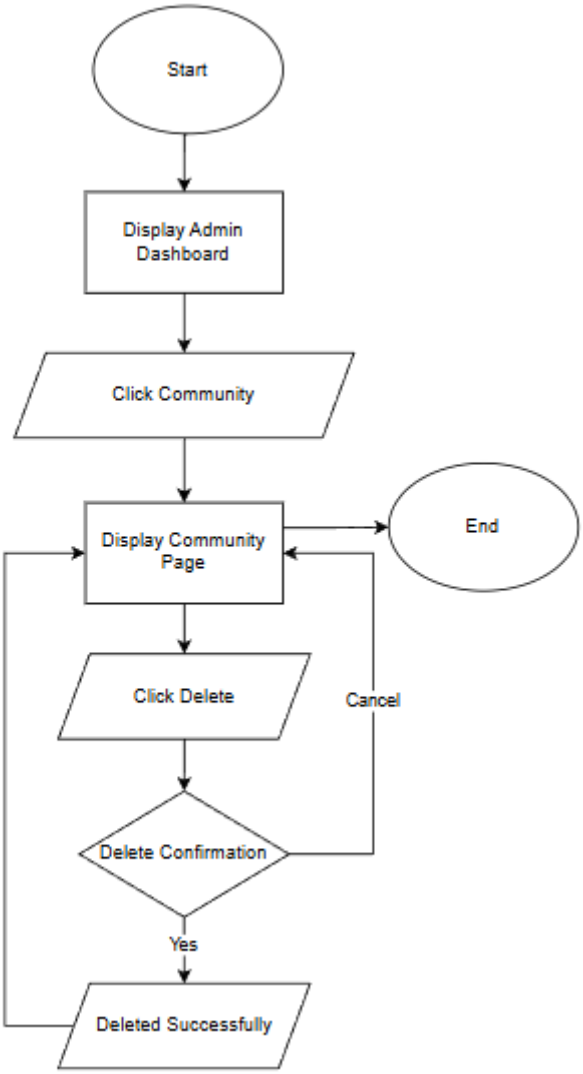

Figure 19: Admin Community Forum Management

This is community forum management page for admin to manage the community forum, inappropriate post will be deleted by admin.

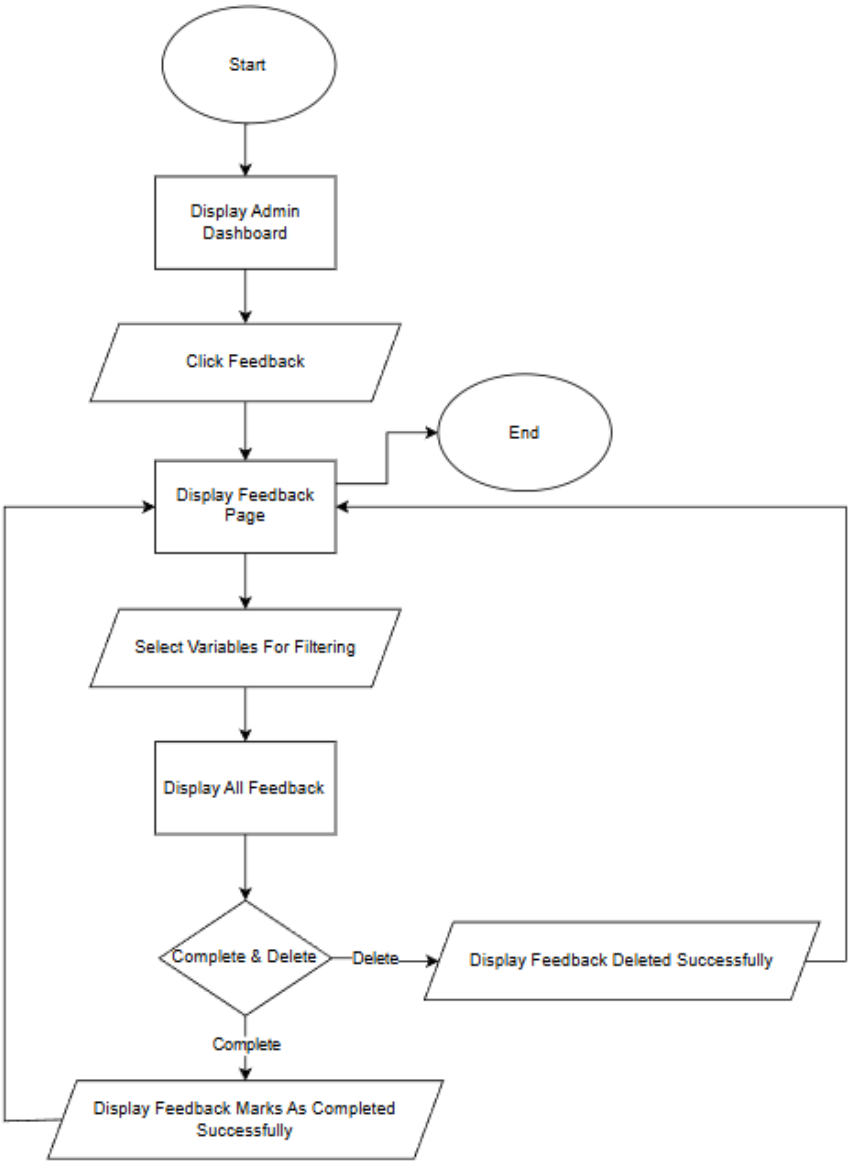

Figure 20: Admin Review Feedback

This is reviewing feedback page, admin able to review all feedback from user, then mark as "Complete" or delete the feedback.

2.4 Major Functions of the Web Application

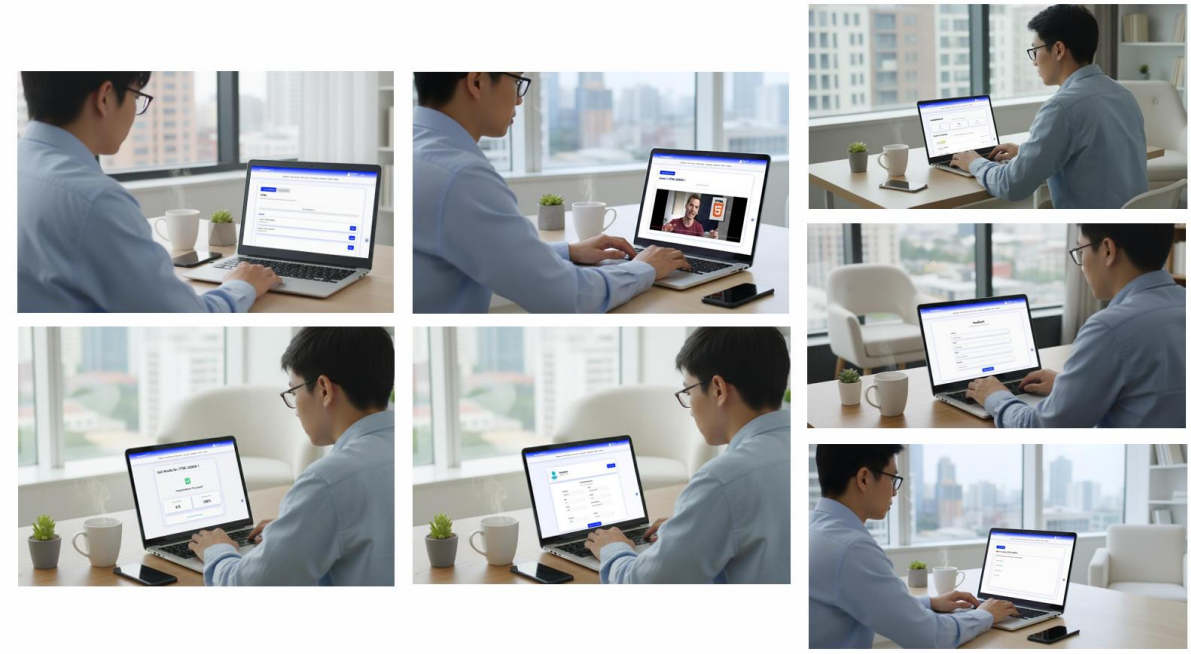

Figure 21: Student Functions

The student part is all about the learning and interaction process. Its only aim is to enable a student to consume educational information and monitor their progress. This contains all the pages to browse and enroll in public and private courses, see particular lessons and quiz. This part also handles the personal identity and student progress of the student in form of their profile, balance of coins and leaderboard as well as social and support outlets via the community and feedback pages.

Figure 22: Educator Functions

The fundamental role of the educator section is creation and management of content. This is a collection of files which allows educators to create, publish and track their own courses. This consists of their dashboard to list their created content, a profile to manage their professional identity, and their critical createcourse and editcourse pages, which contain the complicated logic of assembling lessons, uploading materials, and building quizzes. They can also see the list of students taking their respective courses in this section.

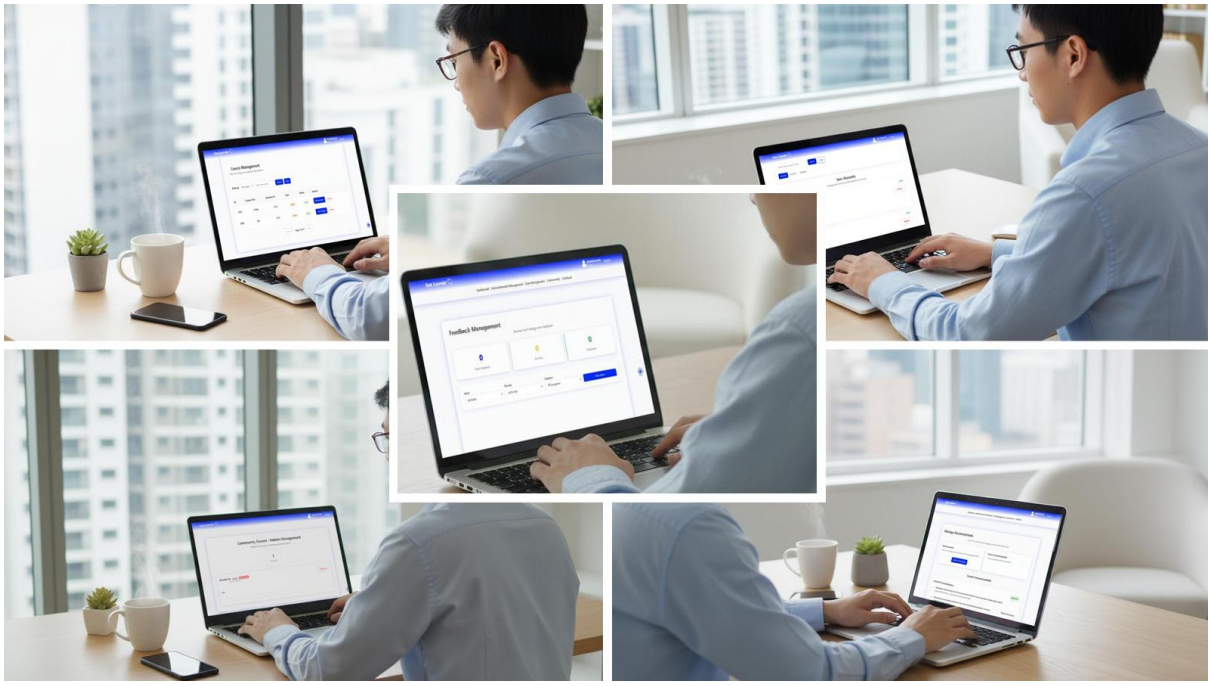

Figure 23: Admin Functions

The single purpose of the admin side is platform-wide management and moderation. The pages are not to be learned or created, but to maintain the health and integrity of the whole application. It serves this purpose by providing tools that enable an administrator to control the entire user base by deactivating accounts, moderate the community by removing inappropriate posts, and manage site-wide content such as advertisements and student feedback. This part also offers the “god-view” to view or permanently delete any course in the database.

Figure 24: Core Functions

The access control and structural layout is the main role of the “Core” section. These files act as the “front door” as well as “blueprint to the whole application. This covers user authentication including the logging in (sign_in), creating new accounts (sign_up) and the recovery of lost passwords (forgot_password). The “SiteMaster” is also included in this section, and it defines the uniform visual look and navigation of the site and dynamically adds the appropriate menus to the students, educators or admins when they are logged in. In addition, it also provides the log out and community forum, community forum capable educator and student to ask question or reply question.

3.0 Design and Modelling **YAP BOON SIONG**

3.1 Entity Relationship Diagram (ERD)

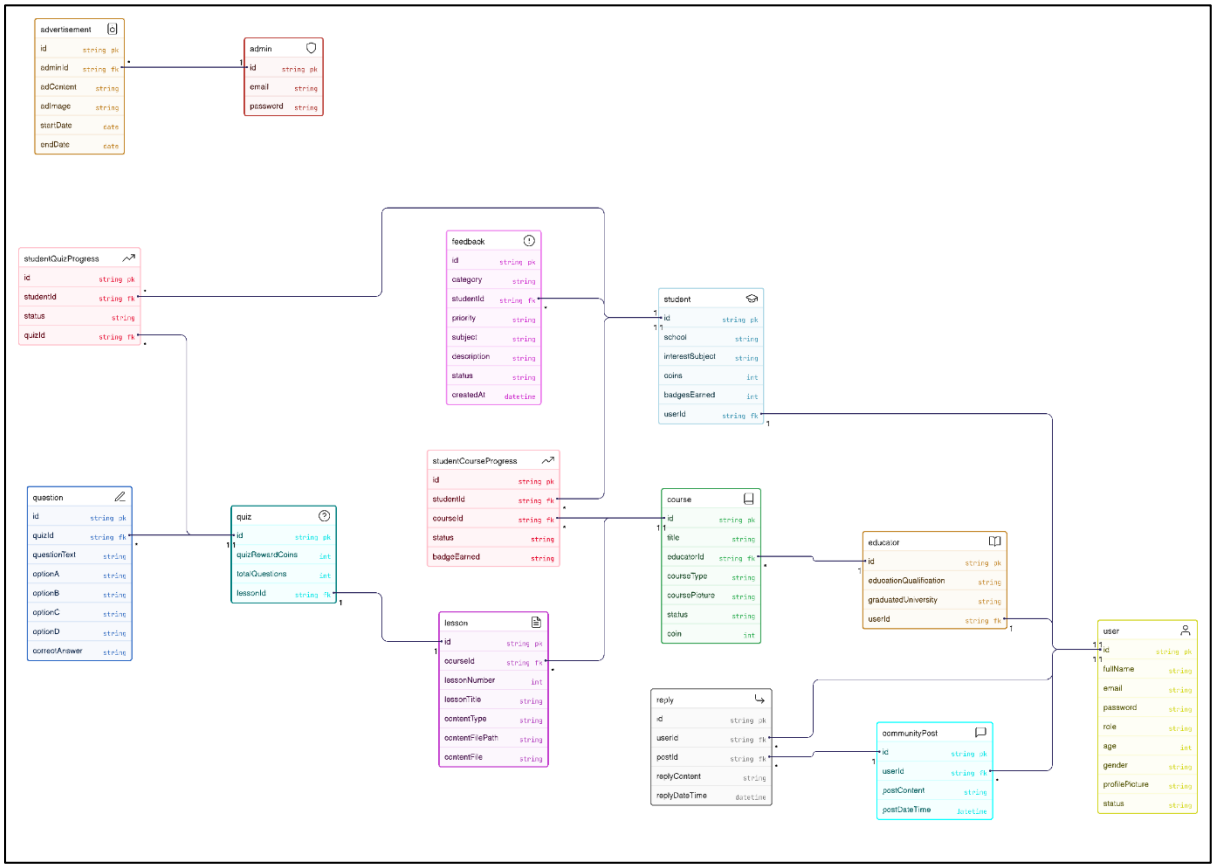

Figure 25: Entity Relationship Diagram (ERD)

Entity Relationship Diagram (ERD) of Sea Learner that is a visual representation of the interaction of data entities that will support the online learning ecosystem of the platform. It outlines the hierarchy and logical relationship between users, courses, progress tracking and the community. The User entity lies at the centre of the system, and it comprises of general details like name, email, password, role and profile details. On this foundation, the users are subdivided into distinct roles: Student, Educator and Admin, adding their own data to the basic attributes of the user.

- Students are associated with the liking of their learning styles, received badges, and acquired coins.
- Teachers are identified with their qualifications and universities in which they graduated.
- Admins deal with adverts and general operations of the platform.

The Course entity relates to educators and is the basis of the learning module. Each course has a variety of Lessons and Quizzes as well, and they are designed in such a way that they take the learners through the step-by-step progress. Educational matters are recorded in lessons, whereas comprehension is evaluated in quizzes. Each quiz is associated with the Question entity, which has a text of the question, four possible answers, and the correct answer. The progress is tracked by StudentCourseProgress and StudentQuizProgress that store the course and quiz completion status of every student and the badges that have been completed to distinguish themselves. Such organizations make sure that the progress and the achievements of each student are properly tracked. Interaction with Community is facilitated by CommunityPost and Reply institutions. Users can ask questions or engage in sharing knowledge and others can respond to the post leading to teamwork and knowledge sharing. Moreover, the Feedback entity enables passengers to provide feedback that is classified in terms of its priority and type, which will contribute to a consistent enhancement of the platform. Adverts are advertisements that are handled by the admins and are stored in the Advertisement entity giving space to either promotional or informational content.

In sum, this ERD clearly outlines the connection and ties between various elements of Sea learner, and thus, community engagement, assessment, feedback, and user management will have a seamless and effective learning management system.

3.2 Wireframes / Mock-ups of Major Web Pages

3.2.1 Public

Main Page

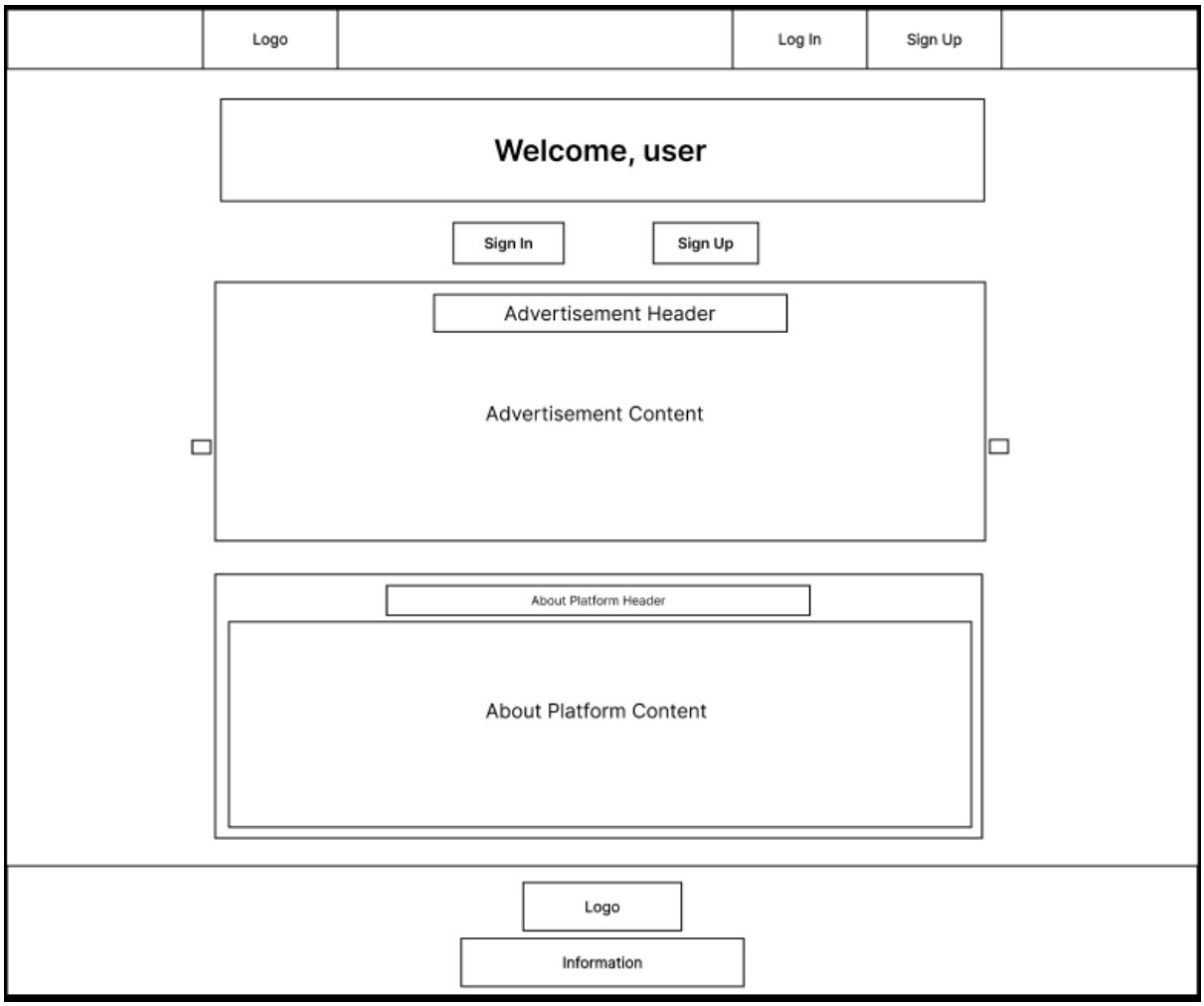

Figure 26: Main Page

The following wireframe is a representation of the homepage of the public main page of the Sealearner learning platform which is designed to attract both new and returning visitors immediately. There is a header at the top that contains the Logo, the Links of Log In and Sign Up that are necessary. The key point of interest is a large welcome section that encourages users to Sign In or Sign Up to be able to access content. Beneath it are two important informational blocks a big Advertisement home to promote marketing courses or features and an About Platform home where the value proposition of Sealearner is described. Lastly, the footer will include a secondary Logo and general Information links, which will make the page a clean and introductory portal to a learning platform that is interested in new users.

Sign In Page

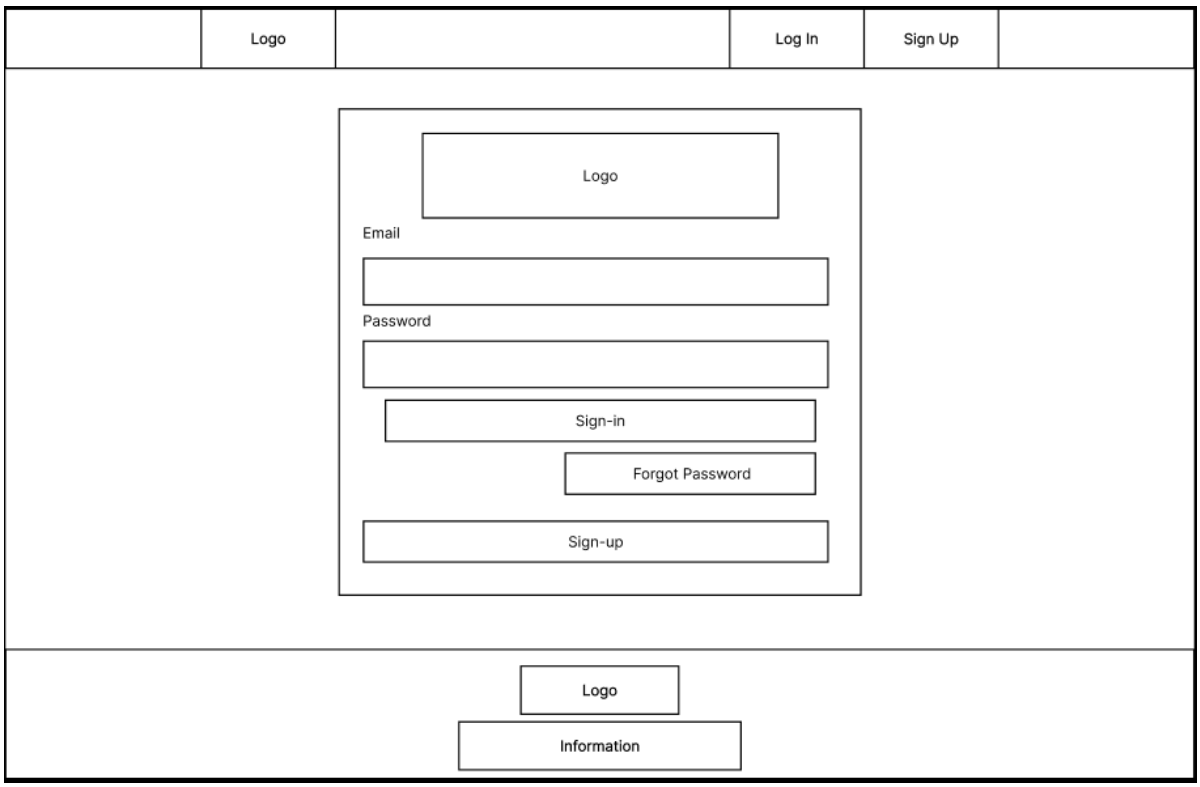

Figure 27: Sign In Page

The key part will be a large platform Logo in front of the main sign-in fields: Email and Password input boxes. Under these fields, there are the main action buttons, Sign-in to enter the credentials and Forgot Password, which helps to recover the credentials. Lastly, there is a distinct Sign-up button that one may follow once they have landed on the sign-in page but have not yet created a sign-in account and thus offering another avenue to the registration process.

Sign Up Page

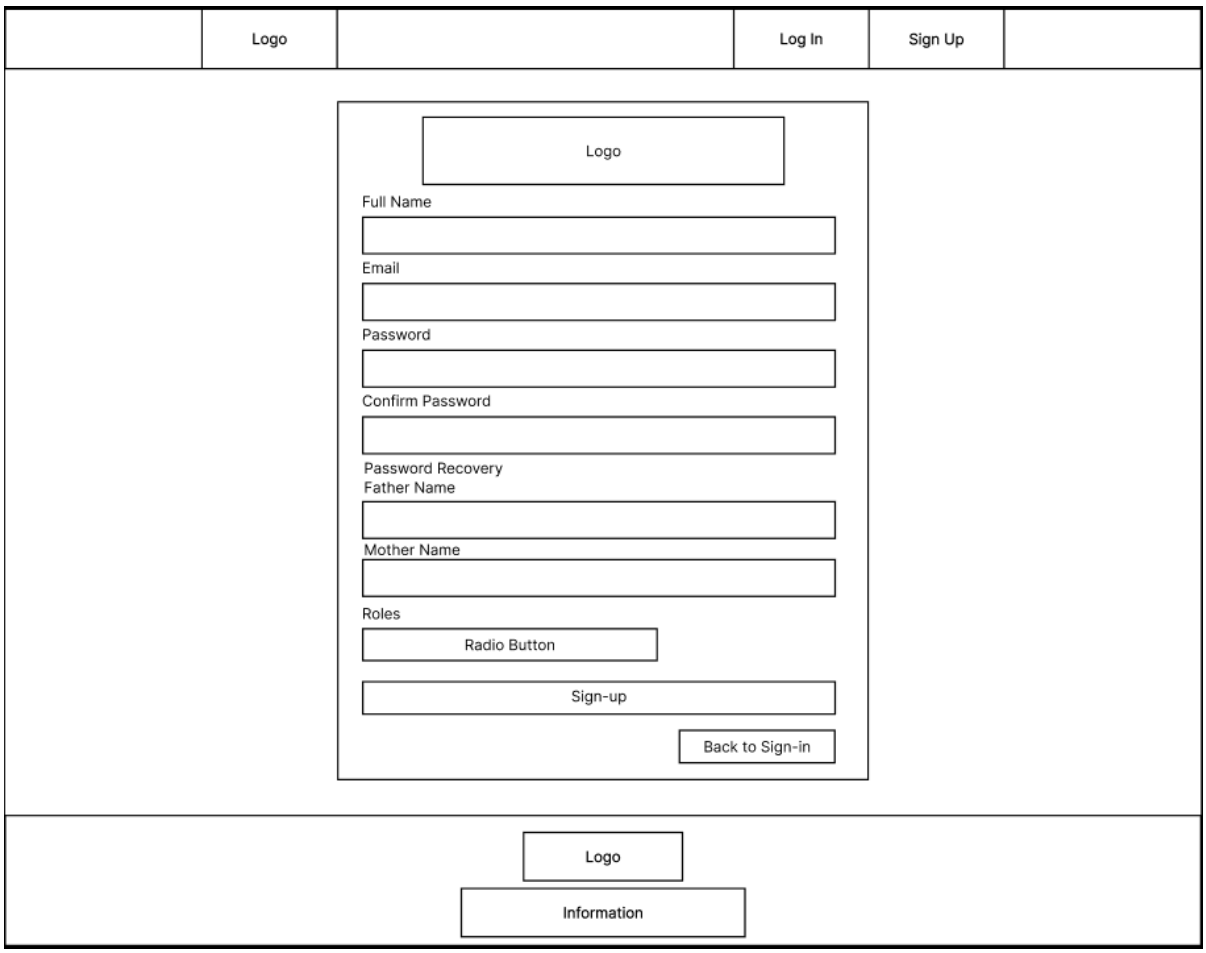

Figure 28: Sign Up Page

In the middle of the page, there is the platform Logo and a complex registration form. The form will need some necessary fields, among which are Full Name, Email, Password and Confirm Password. It also includes specific fields on Password Recovery, namely: asking Father Name and Mother Name. This is followed by a Roles section which has a Radio Button that enables the user to choose his personality type (student and educator). The Sign-up button deals with the form submission and there should be a link of Back to Sign-in as the alternative effectively as a navigation option.

Sign-Up Page Extend (Student)

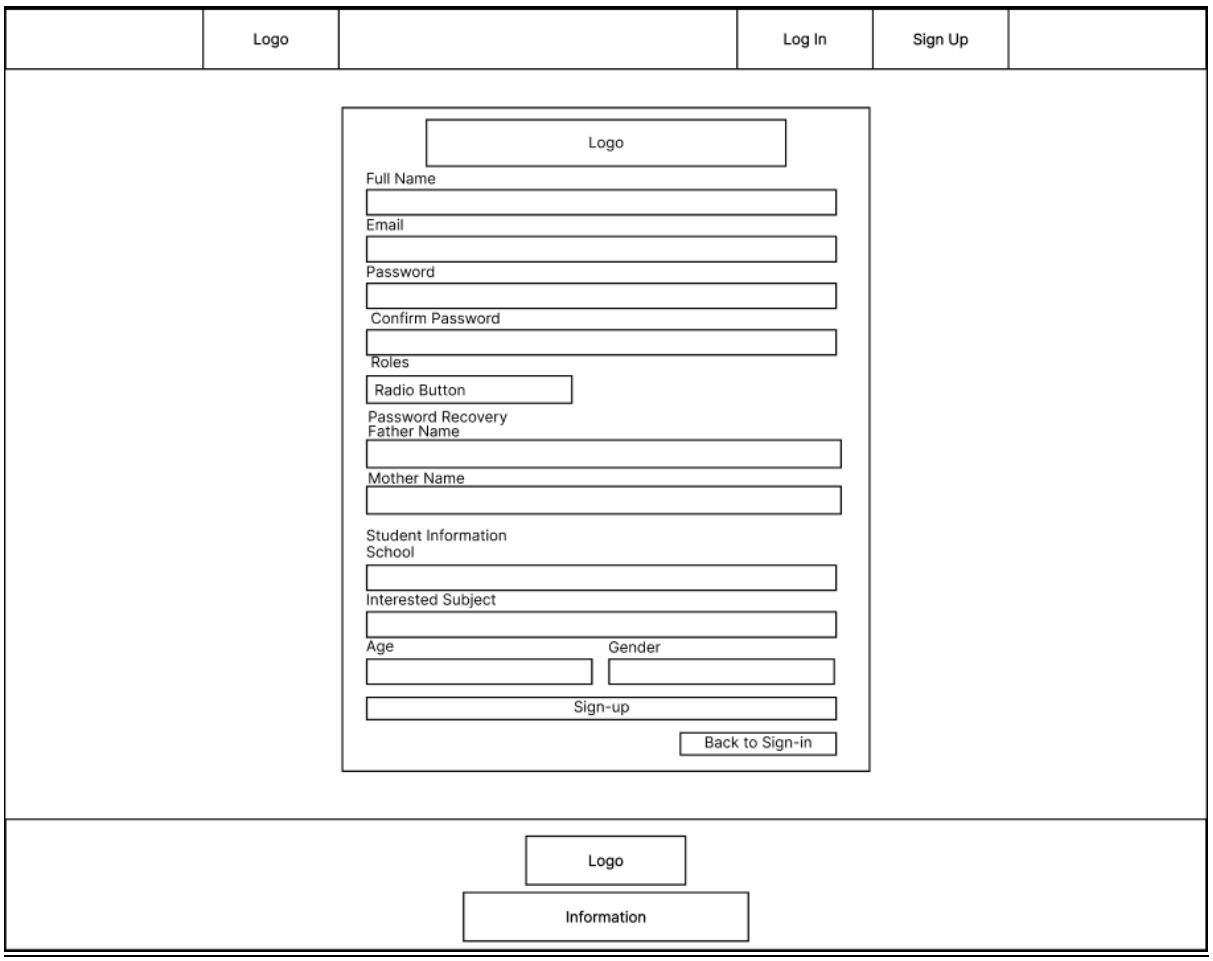

Figure 29: Sign Up Page Extend (Student)

This page consists of the platform Logo and usual initial registration fields Full Name, Email, Password, Confirm Password, and the Roles radio button list. It has two password recover (Father Name and Mother Name) fields. More importantly, the area where student specific fields used to be has been changed or not mentioned at all, with the information needed by non-students, namely the underlying and security information, being present. The Sign-up button and Back to Sign-in link also mark the end of the form.

Sign Up Page Extend (Educator)

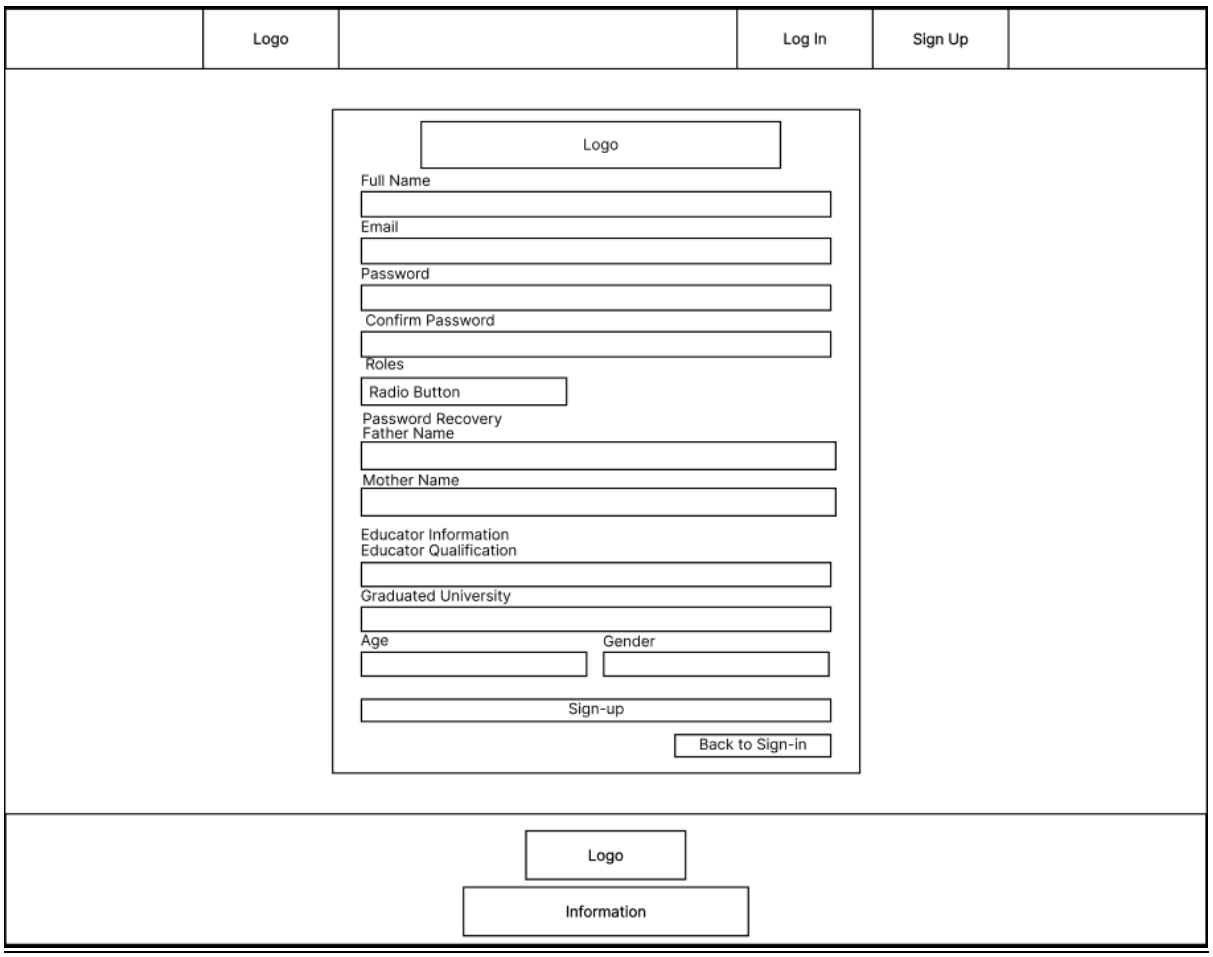

Figure 30: Sign-Up Page Extend (Educator)

The page includes the Logo and the regular first registration forms, such as Full Name, Email, Password, Confirm Password, and Roles radio button selection. It contains security areas of Password Recovery (Father Name and Mother Name). The addition is the Educator Information one that demands profession details: Educator Qualification and the Graduated University. Simple demographic information, Age and Gender are also solicited. The form is filled in with the Sign-up submit button and the other Back to Sign-in link of navigation.

Confirmation Page

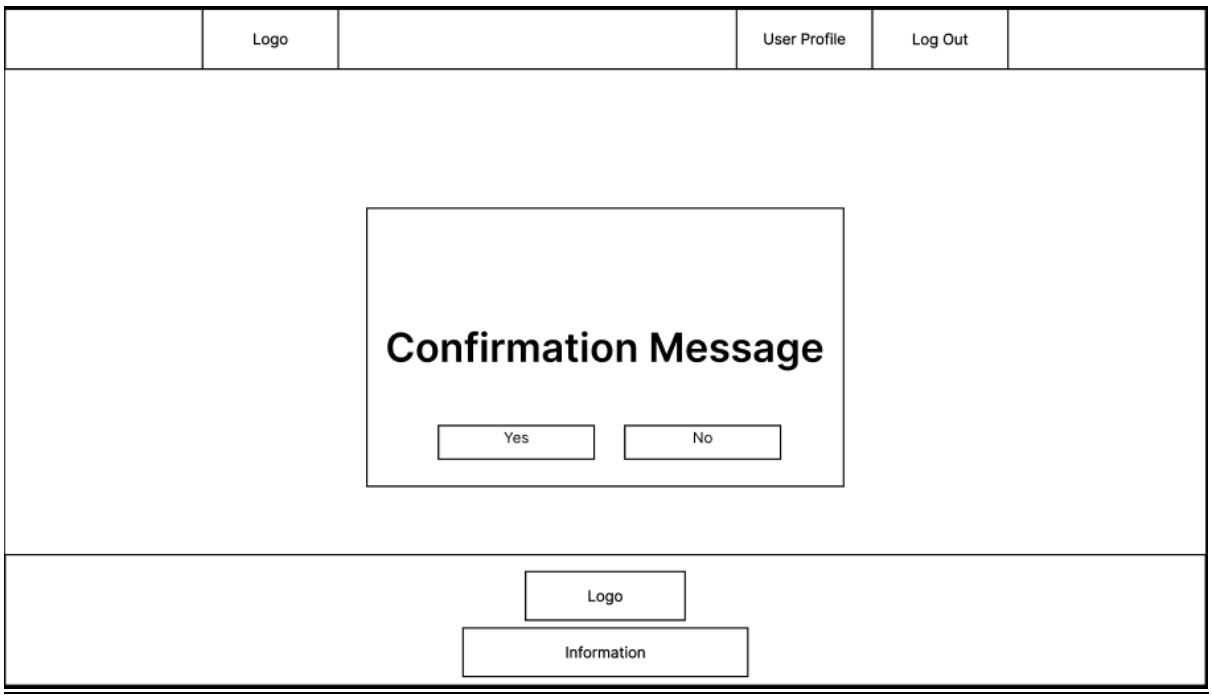

Figure 31: Confirmation Page

The main part is a big box with the words that are written in it confirmation message, this is the very question or prompt the user must answer example, “Are you sure you want to …”. Under the message there are two distinct action buttons Yes and No that can enable the user to either go ahead with the action or cancel it respectively.

Community Page (Student/Educator)

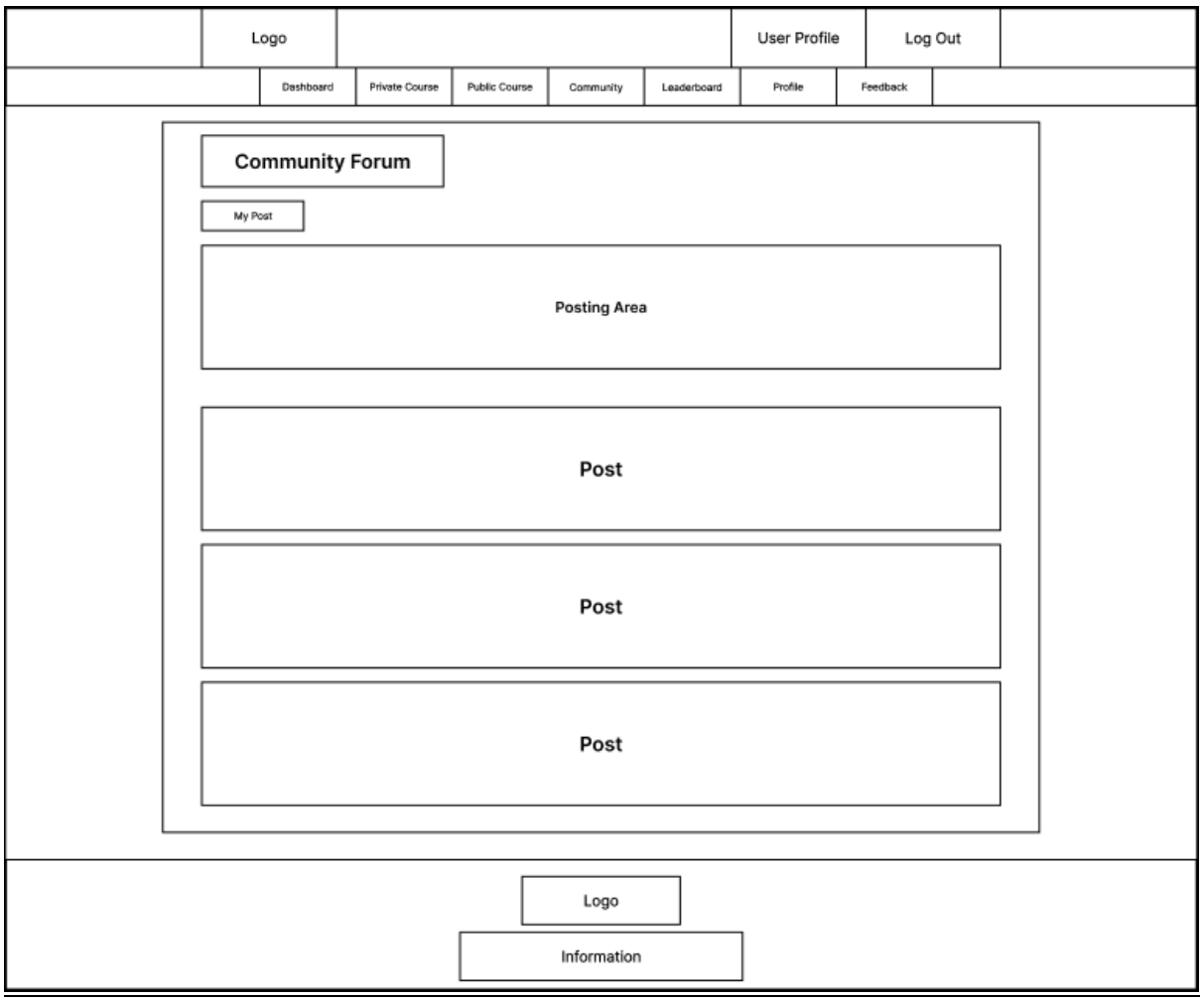

Figure 32: Community Page

This page show the layout of the Community forum. The main section of the forum is a Posting area where users can make new discussion or a post and frequently with a My Post button or filter on it. The rest of the space below this area is occupied by listings of existing content, which are multiple and repetitive Post blocks.

My Post Page (Student/Educator)

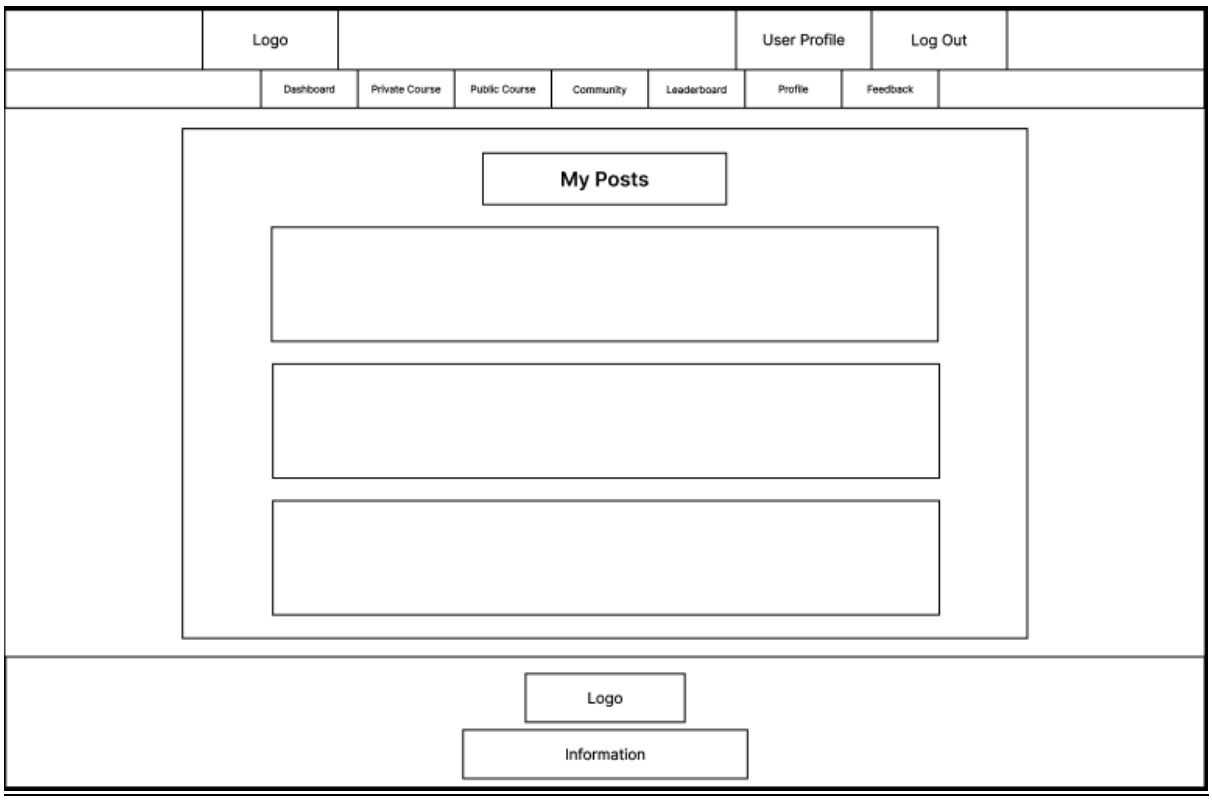

Figure 33: My Post Page (Student/Educator)

This area is “My Posts” which is post that created by the user (Student or Educator) in Community Forum. The posts are shown as several, repeating blocks which are empty, which is a sign of a feed or a list format, where one can view and possibly control their previous contributions to the community of this platform such as Delete own post or Edit existing post.

3.2.2 Educator

Educator Dashboard Page

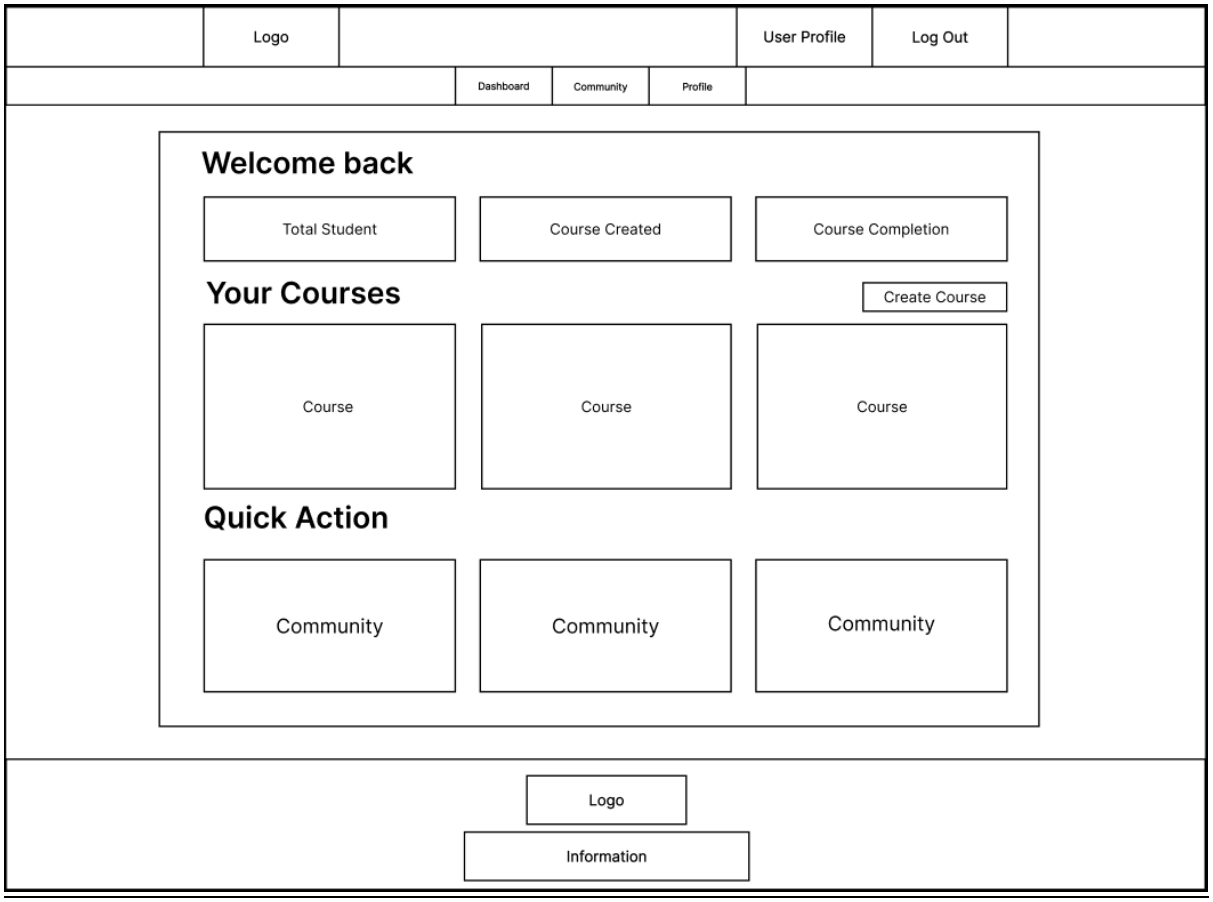

Figure 34: Educator Dashboard Page

This wireframe illustrates Educator Dashboard Page, which is a central location of instructors. The page welcome the user and it shows the key performance indicators such as Total Student, Course Created and Course Completion in prominent places. Under this overview, there is the Your Courses lists the offerings of the educator with a Create Course action button, and then a Quick Action section consisting of Community blocks that enable quick access to the relevant activities, simplified course management and oversight to the educator.

Create Course Page

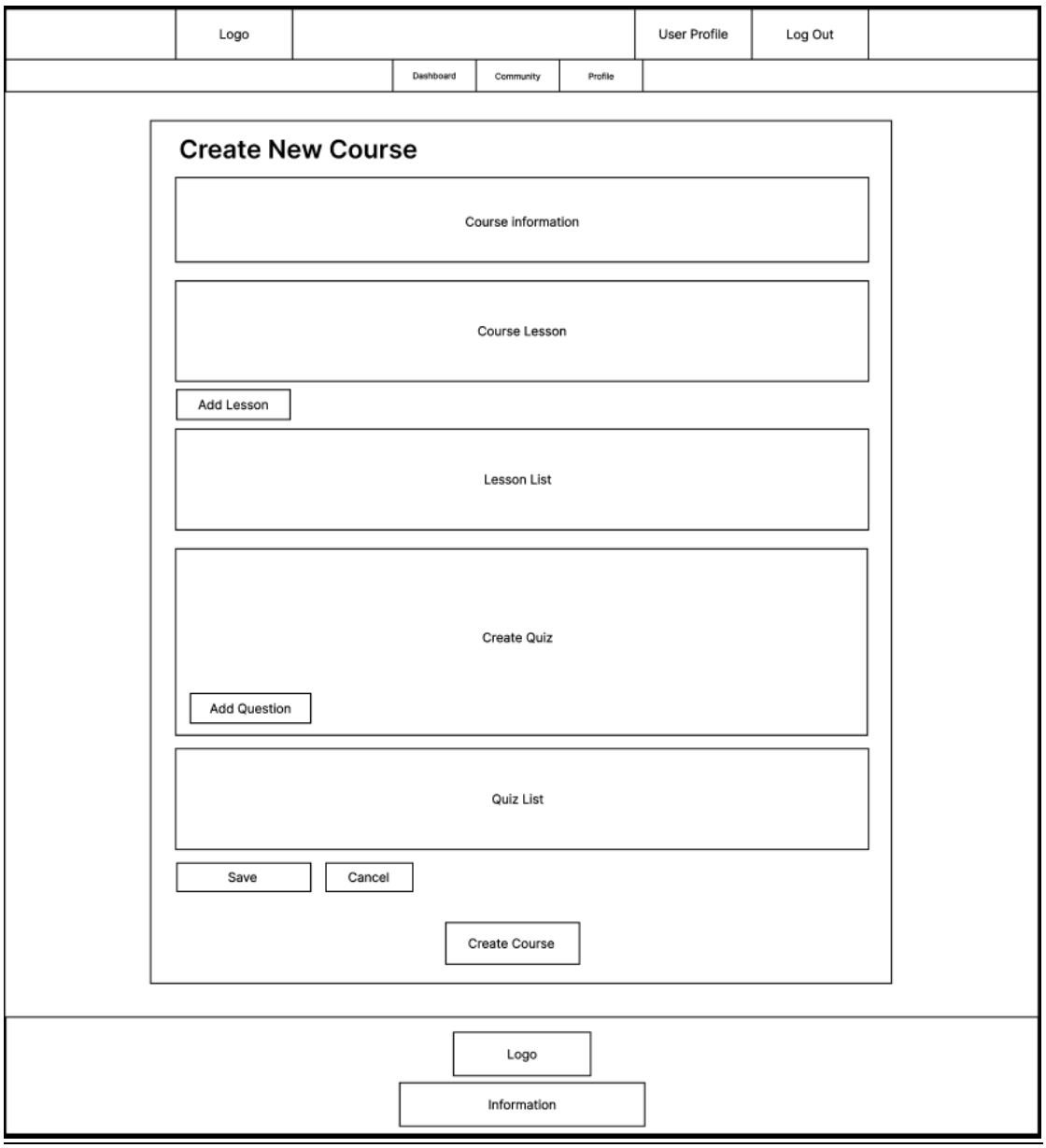

Figure 35: Create Course Page

The page is divided into consecutive stages of course creation. It starts with the section of the Course Information with basic information such as name and description. Then there is the Course Lesson area which is probably the place where one will add content and finally an Add Lesson button is provided to create the course modules. The way the course is organized is presented in the Lesson List section. With defined lessons, the educator proceeds to the Create Quiz section, which has an Add Question button and the defined assessments appear in the Quiz List. Primary action buttons Save for drafts, Cancel, and the last Create Course button to publish the course end the process.

Edit Course Page

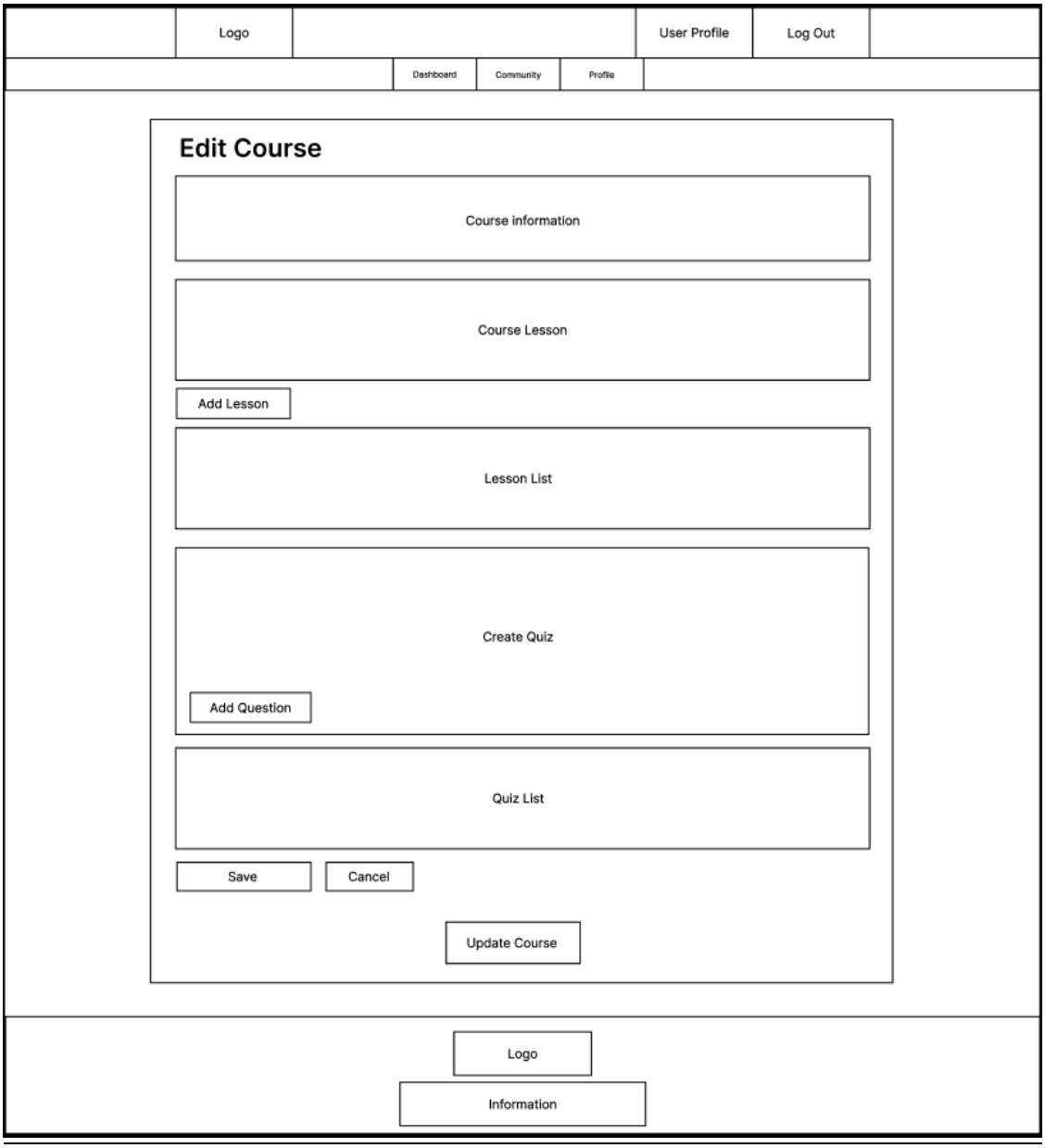

Figure 36: Edit Course Page

The page is designed in the same way the course creation form does and as a result, the educator can revisit and make changes to any aspect of a course. It is structured into parts that begin with Course Information where basic details are edited then Course Lesson content which can be expanded with the help of Add Lesson button, and the modules are arranged in the Lesson List. Under this, the teacher has the option of adjusting the assessments under the Create Quiz section and the Add Question and the organized assessments under Quiz List. The buttons at the bottom enable the user to save changes (in draft form), cancel changes and finally update course to reflect the changes onto the live form.

View Course Statistic Page

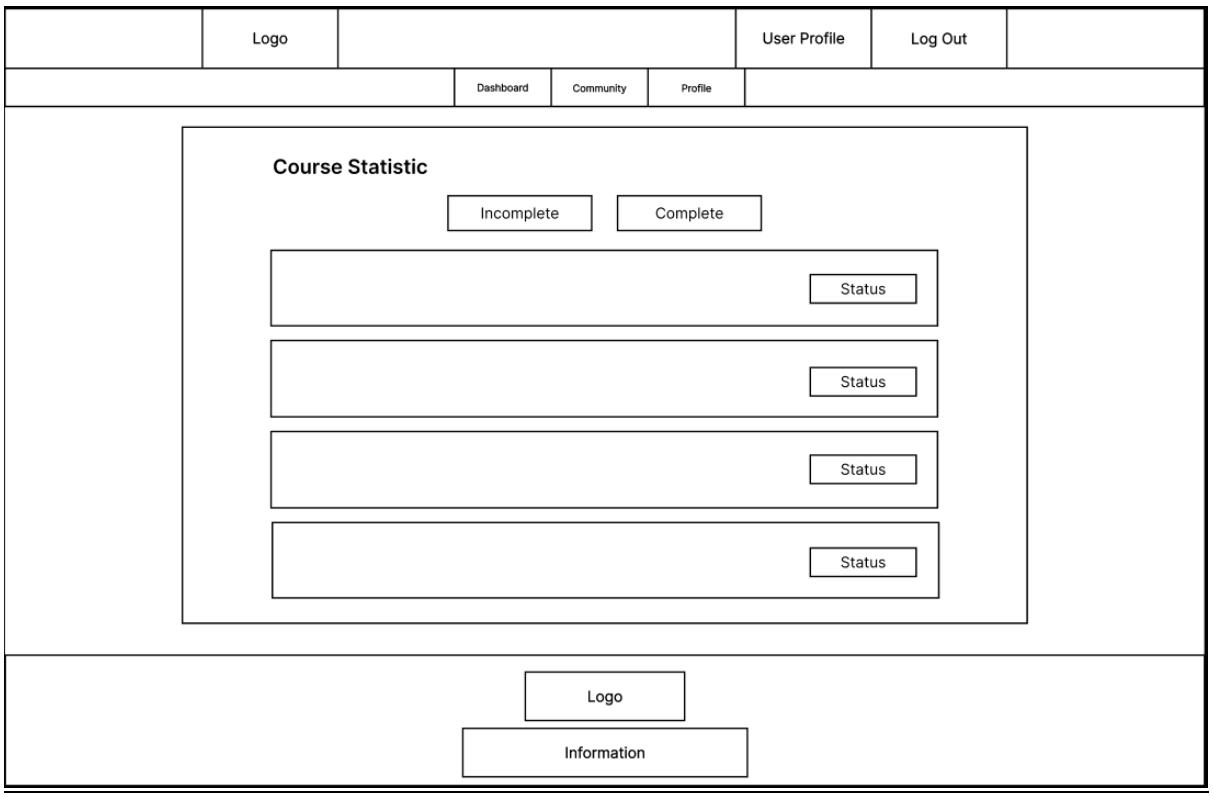

Figure 37: View Course Statistic Page

This shows wireframe for View Course Statistic. This section is meant to filter and show course progress information, and it uses two distinct clear toggle buttons, namely Incomplete and Complete. Under these filters, the groupings of statistics are presented in the form of list constituted by repeated horizontal blocks. Every block is a container of information about a particular course or perhaps the progress of a single student and has a special Status button or indicator, which lets the user examine or engage with the present status of the course quickly in comparison with the filtered display.

Educator Profile Page

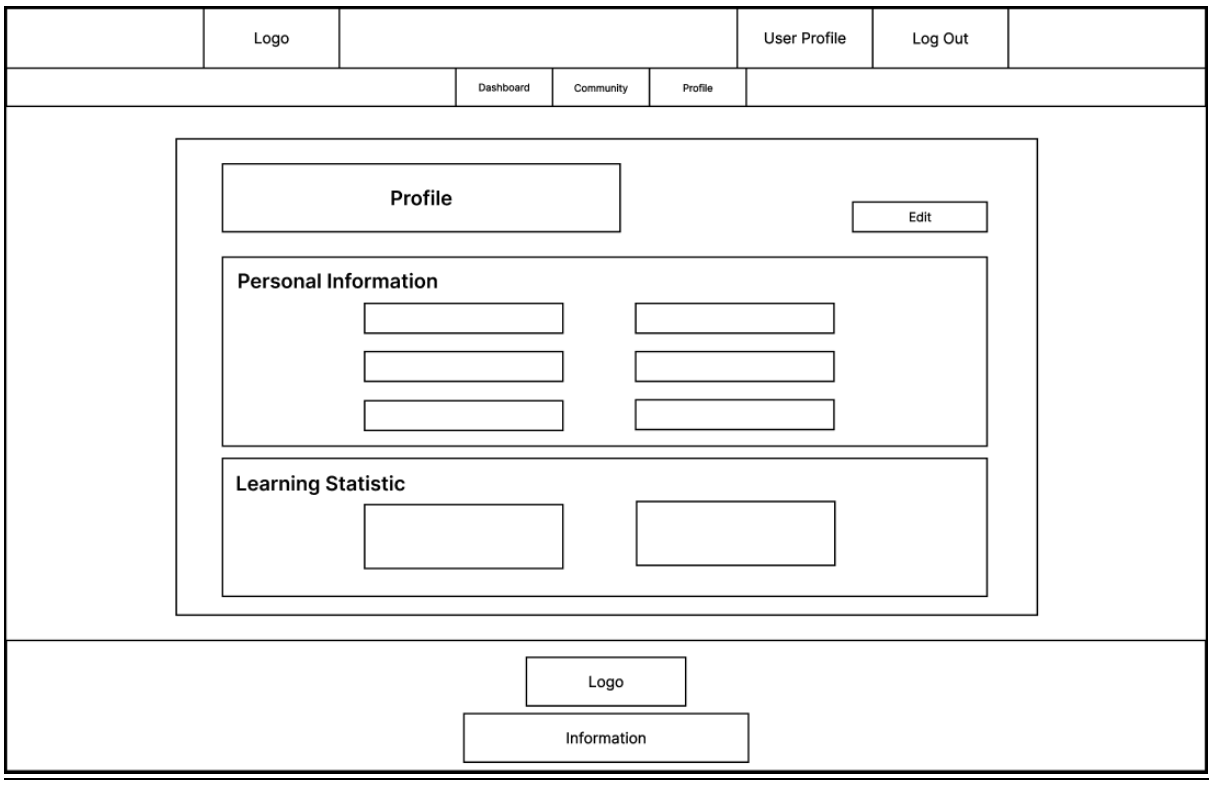

Figure 38: Educator Profile Page

This wireframe is Educator Profile Page, which shows that the information presented can be changed. It is separated into two major parts. The Personal Information section provides different information on the educator organized into multiple columns and rows with multiple input or presentation fields. This section consists of the biographical and possibly contact or professional information. The second section is Learning Statistic, which is the key performance indicator on the activities of the educator on the platform, and presumably presents such indicators as courses available, total enrolment, or student performance, which are presented in two block statistic form.

3.2.3 Student

Student Dashboard Page

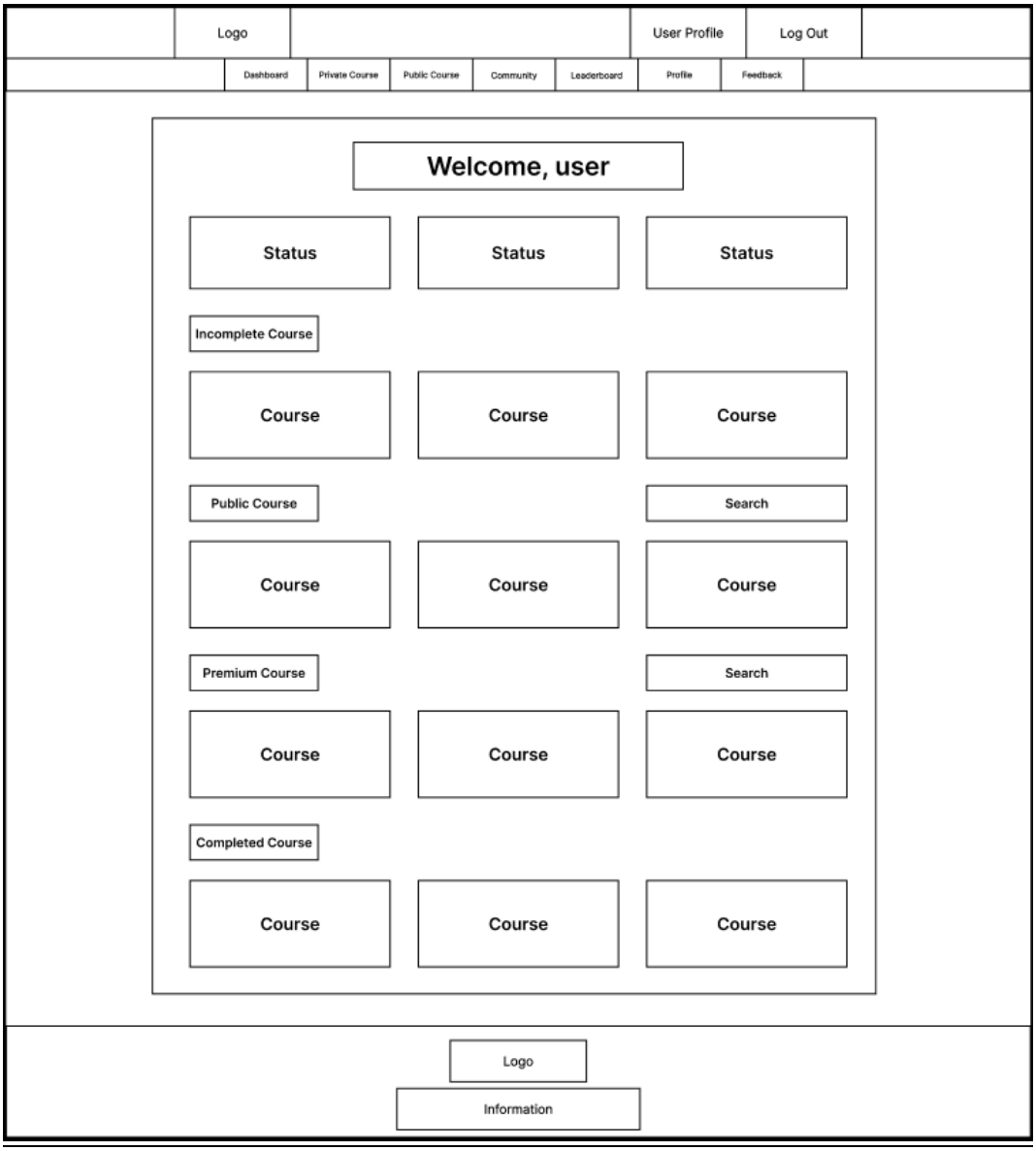

Figure 39: Student Dashboard Page

This wireframe is the Student Dashboard Page which is the main central node of a logged-in student. It starts with a welcome greeting and three major Status blocks that provide a summary of the key performance metrics of the student. The remaining page sorts the learning portfolio of the student in separate, scrollable segments, which are Incomplete Course, where active courses are studied and Public Course and Premium Course, with a Search option to discover new things and completed courses. Every part is represented by several Course blocks, through which the student can see all the content in one and full overview of their progress and options of learning on the Sealearner platform.

Public Course Page

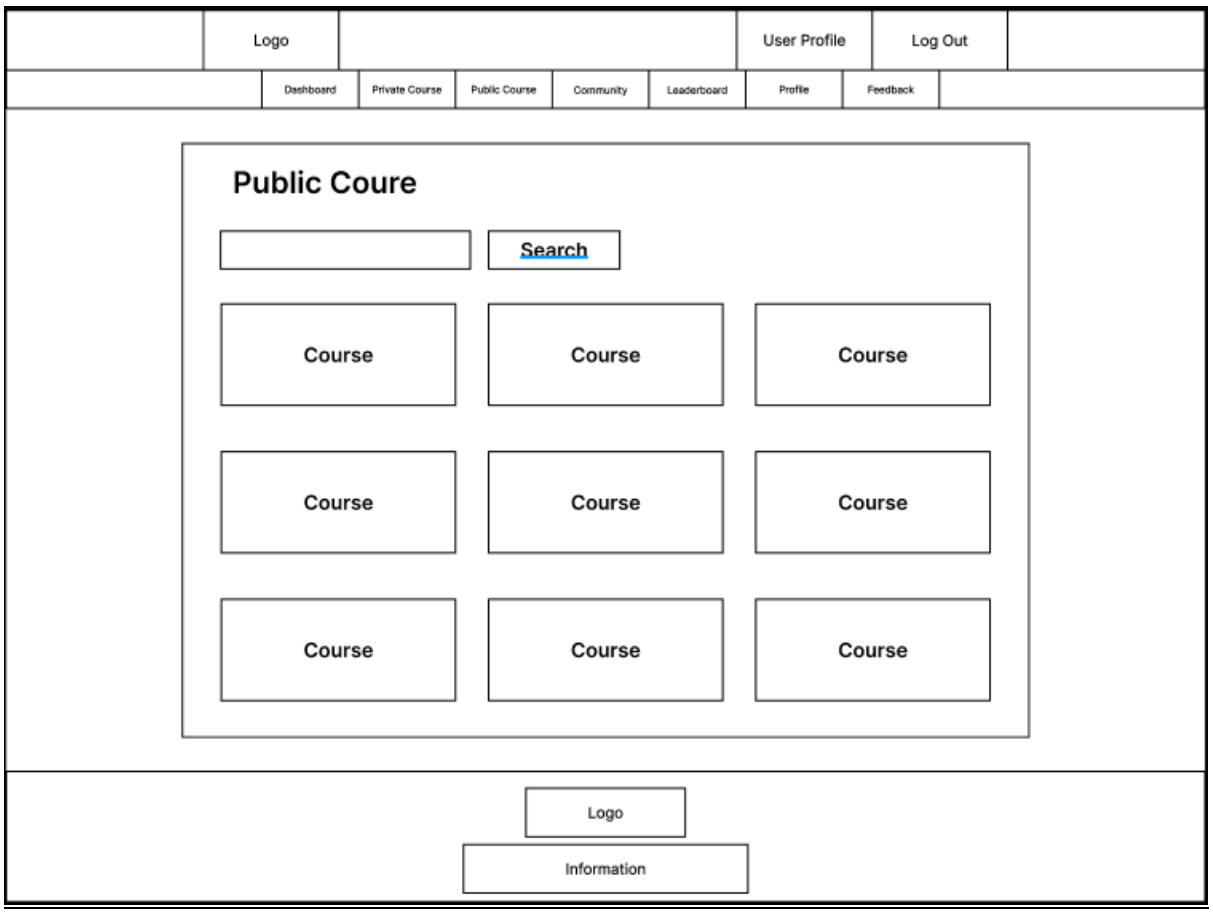

Figure 40: Public Course Page

The title of the page is very clear and is dedicated to course discovery, which is called Public Course. There is a large Search box and button at the top of the content area that would enable the user to be active in searching specific courses. The principal section of the page represents a grid or a list of displaying several Course blocks. Individual public courses can be selected by the user and each of them is represented by a block that will display the course details or allow the user to enrol into it.

Private Course Page

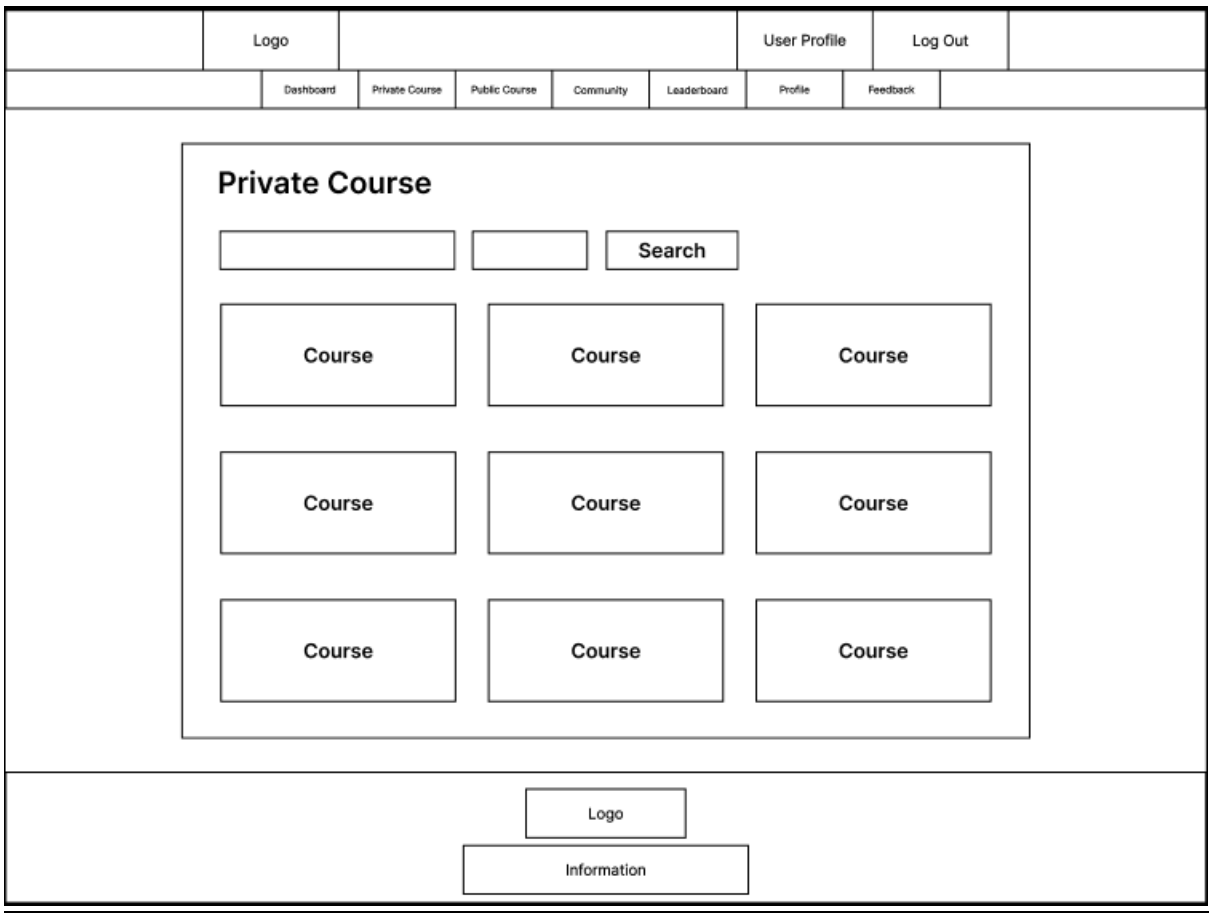

Figure 41: Private Course Page

The title of the page is called Private Course, and it implies that the content is not publicly available. It is usable as the user can find particular courses using a Search box and an extra input box that may be a filtering feature, sorting of needed coins, and then a Search button. The central content block is a grid or list which is filled with various Course blocks. The individual blocks are the individual courses that the user is currently enrolled in or given special access to, giving the block a high priority to easy navigation towards their own and individual learning materials.

Course Content Page

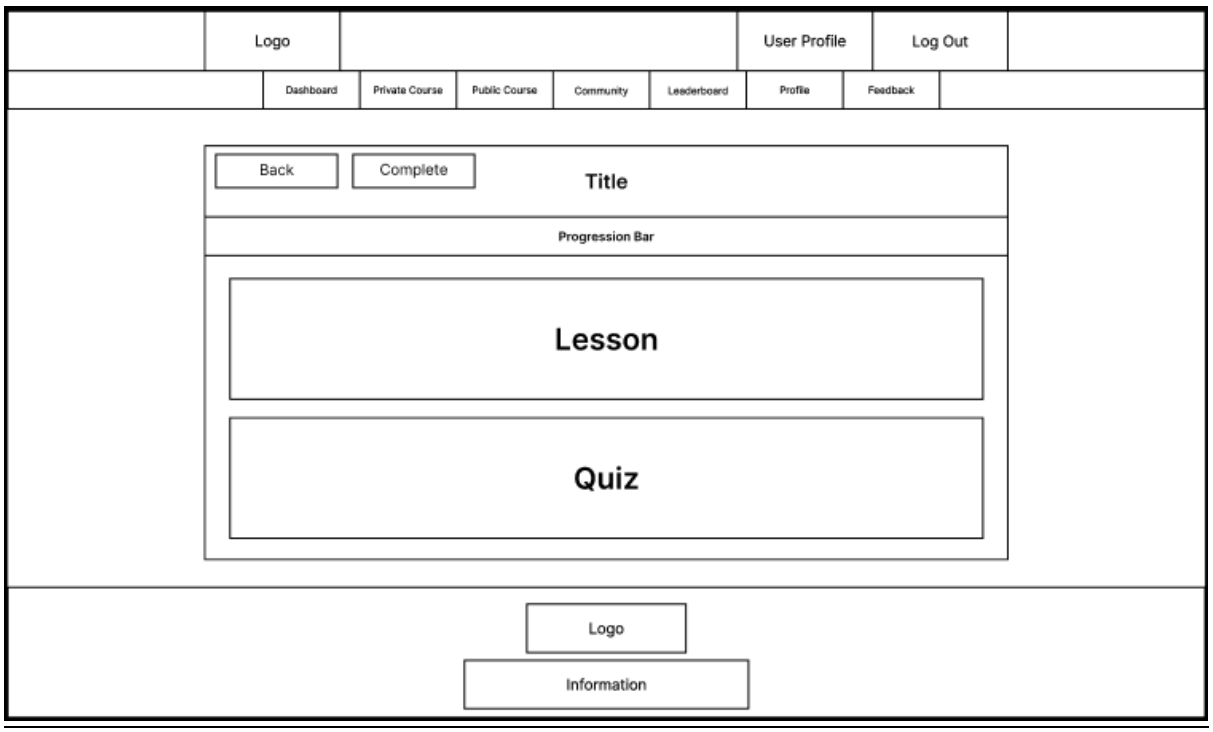

Figure 42:Course Content Page

The page is designed towards dedicated studying which is Course Content Page where it shows the Title of the lesson or module under consideration. Control functions are offered through a Back button to go earlier content and a Complete button to indicate the current content as completed. Visual Progression Bar shows the progression of the student through the course. The central content area has two unique blocks the Lesson block that includes the educational contents and the Quiz block where the student does an assessment of the lesson.

Lesson Content Page

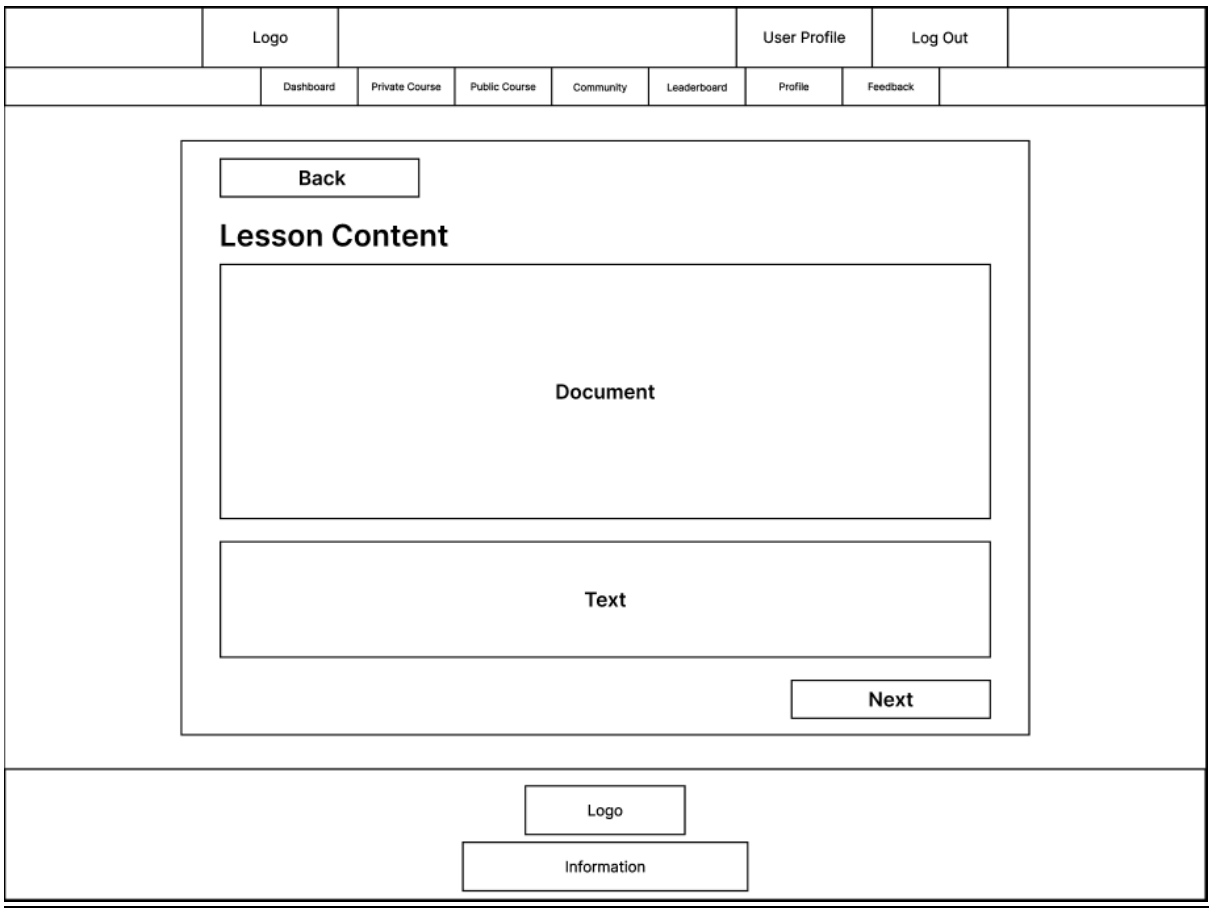

Figure 43: Lesson Content Page

The lesson content is explicitly devoted to the lesson material as it is denoted by the title of the page, which is Lesson Content. There is a Back button that enables the student to get back to the previous view. The central content element will consist of two segments, a big Document block, probably containing the core educational content ,a video or a PDF, and a Text block underneath it, which may include additional information, notes, or instructions. A Next button is placed on the bottom right, which controls the lesson progression by taking the student to the next lesson of the course.

Quiz Content Page

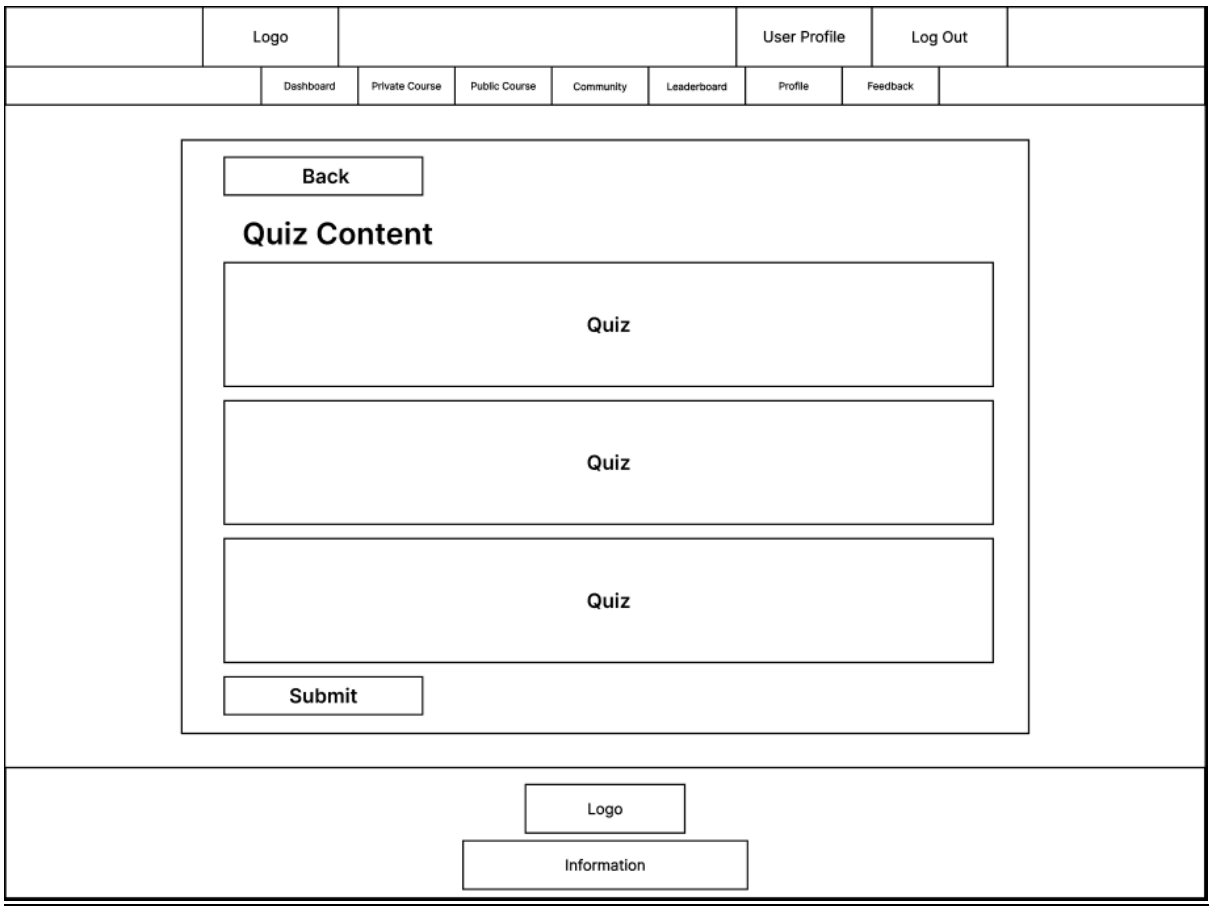

Figure 44: Quiz Content Page

It has a clear title of the page that is Quiz Content, and a Back button is placed to help navigate the page. The focus is on the assessment that is introduced as a chain of Quiz blocks where each block is a question or a group of questions. The repetitive form of the blocks suggests a standard form in multi-question format. The interaction is also closed off with a large Submit button on the bottom of the screen when the student is done with the questions and is intended to capture the answers and end the quiz attempt.

Quiz Result Page

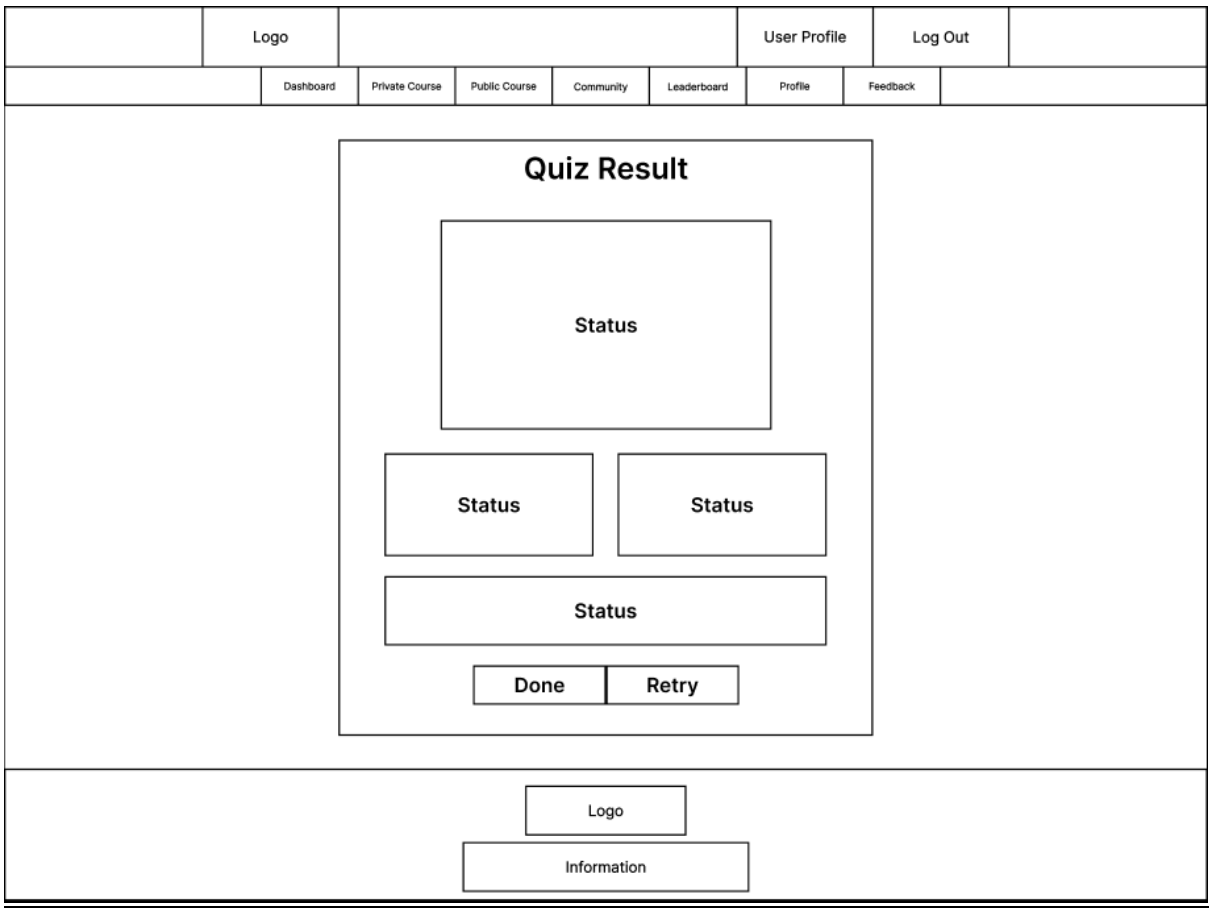

Figure 45: Quiz Result Page

The wireframe of this page is called Quiz Result, and the primary focus is to communicate the performance of the student. The findings are represented on several Status blocks. There is probably a big, central Status block with an indication of the overall result showing Pass or Fail and total score. Below it, there can be two smaller Status blocks presenting such key metrics as the number of correct answers and the time spent. Finally, the Status block may be a horizontal block, with detailed feedback, or a summary. The page will end with two action buttons, which is Done (to leave the results and continue with the course) and Retry (to take the quiz once more).

Leaderboard Page

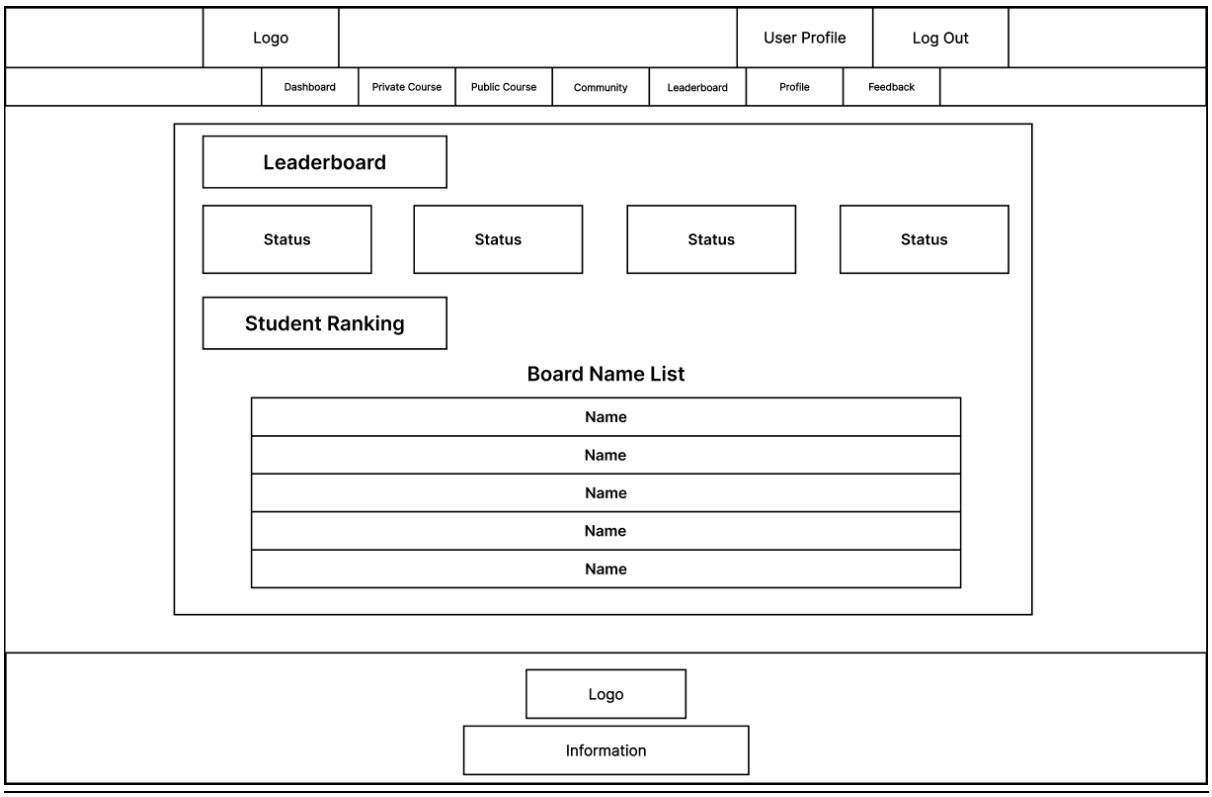

Figure 46: Leaderboard Page

This wireframe represent the Leaderboard and is supposed to be viewed by a registered user. The upper part shows key performance indicators in four recurrent Status blocks, presumably a summary of general student performance measures, progress or achievement. There is a section called Student Ranking, which is concerned with the aspect of competition and a list of board names comes next. The ranked students appear in this list and every repeating horizontal block marked by the word Name indicates an individual entry which contains the name of a student together with their corresponding score or rank.

Student Profile Page

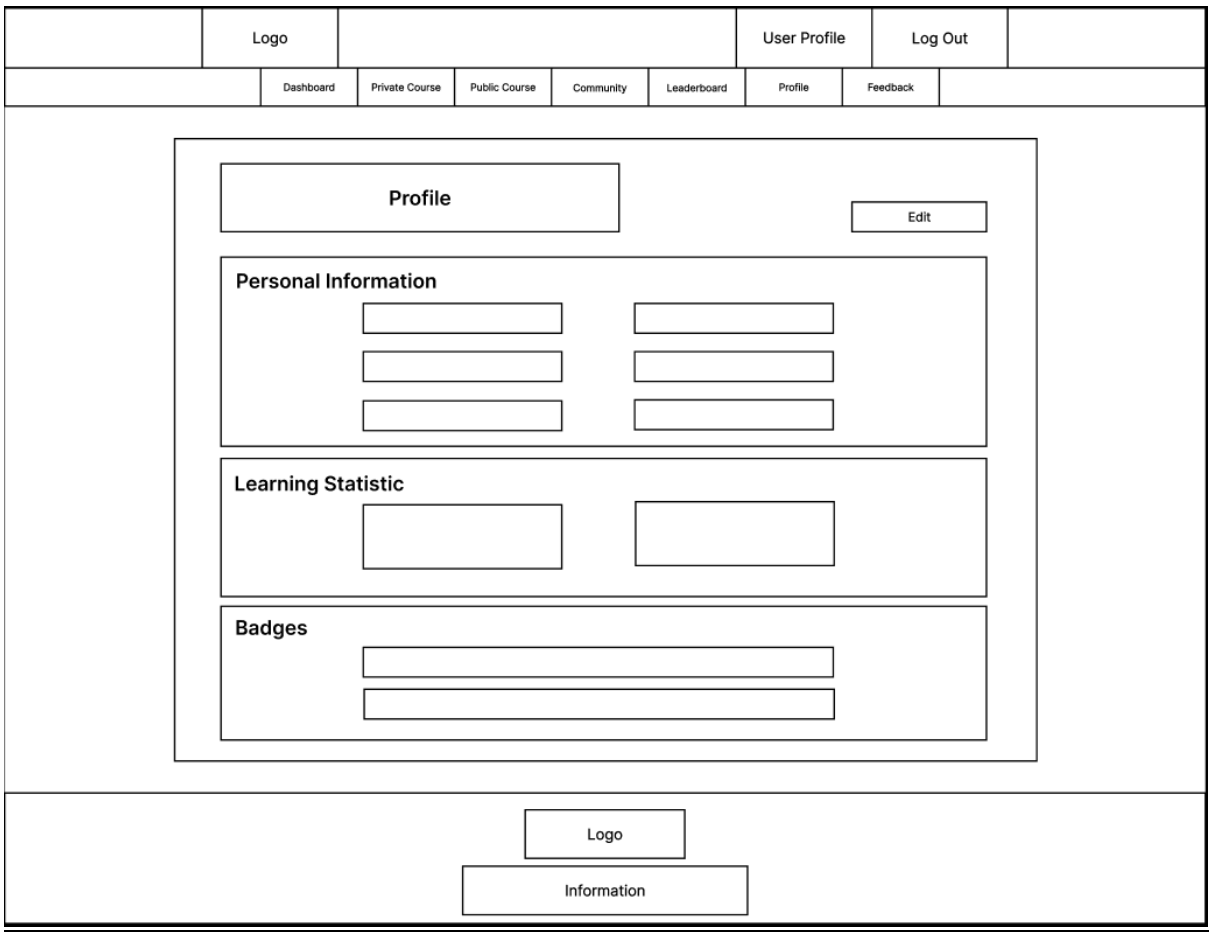

Figure 47: Student Profile Page

This wireframe is called Student Profile that is accompanied by an Edit button to enable the student to update their information. The text is divided into three parts. The section on Personal Information presents the biographical and contact information of the student in a multi-field and columnar format. The Learning Statistic section demonstrates the key performance indicators according to courses and activities of the student and embodies two statistic blocks. Lastly, the Badges section displays all the attained rewards and awards of the student on the platform, which appears as two repeated horizontal blocks.

Student Feedback Page

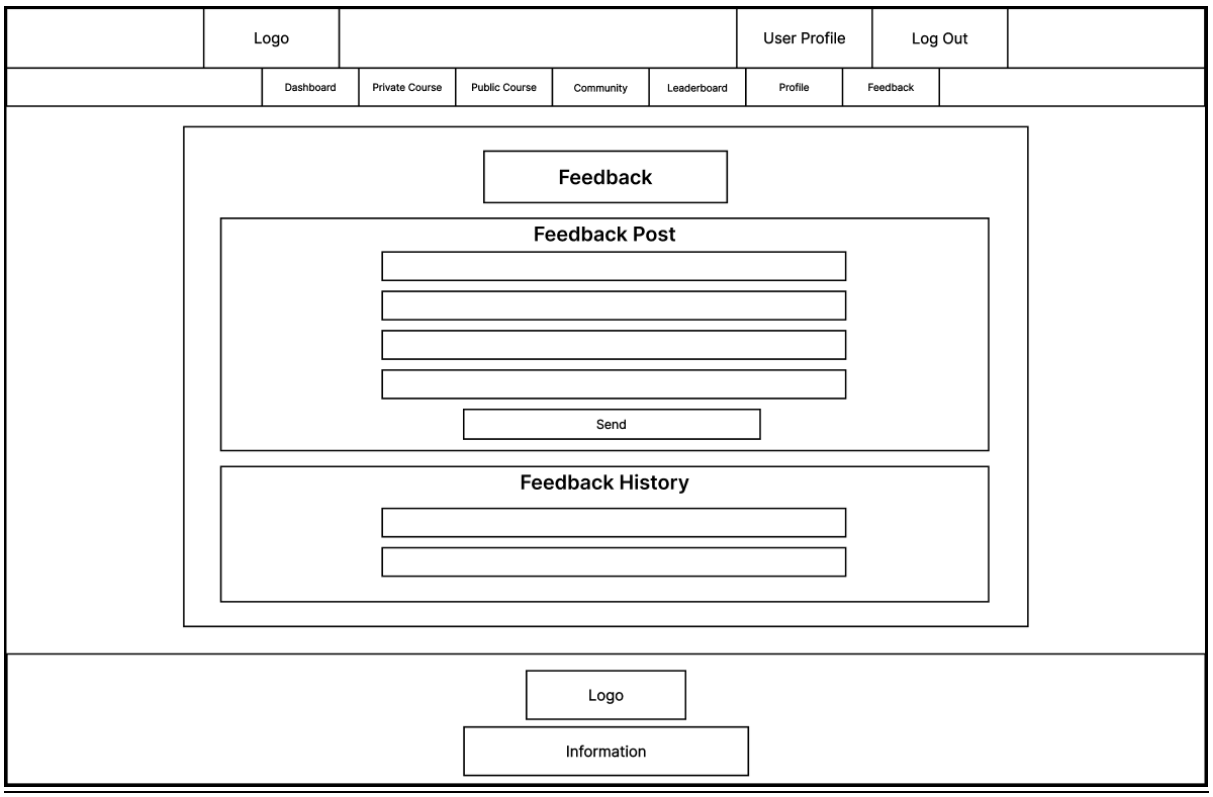

Figure 48: Student Feedback Page

This wireframe shows Feedback, and it is split into two major functional areas. The Feedback Post section is associated with posting new comments or reports, which has a few input fields probably subject, details, etc. A send button that allows submitting an entry. Under this is the Feedback History section which is utilized to show the status and contents of feedback that have been submitted before, and this is displayed through two repeating horizontal blocks.

3.2.4 Admin

Admin Dashboard Page

Figure 49: Admin Dashboard Page

The secondary navigation also is focused on the administrative task and is provided with links to Dashboard for Advertisement Management, User Management, Community, and Feedback. The primary content domain, which is called the Admin Dashboard, begins with the main performance indicators presented in four blocks, which are Total Students, Total Educators, Total Courses and Pending Feedback. The second significant section is Course Management, and it has a Search bar to find individual courses. A list of the courses appears in repeating blocks that are horizontal, and action buttons View to view or edit course details and Delete allow the administrative control of the course catalog.

Advertisment Management Page

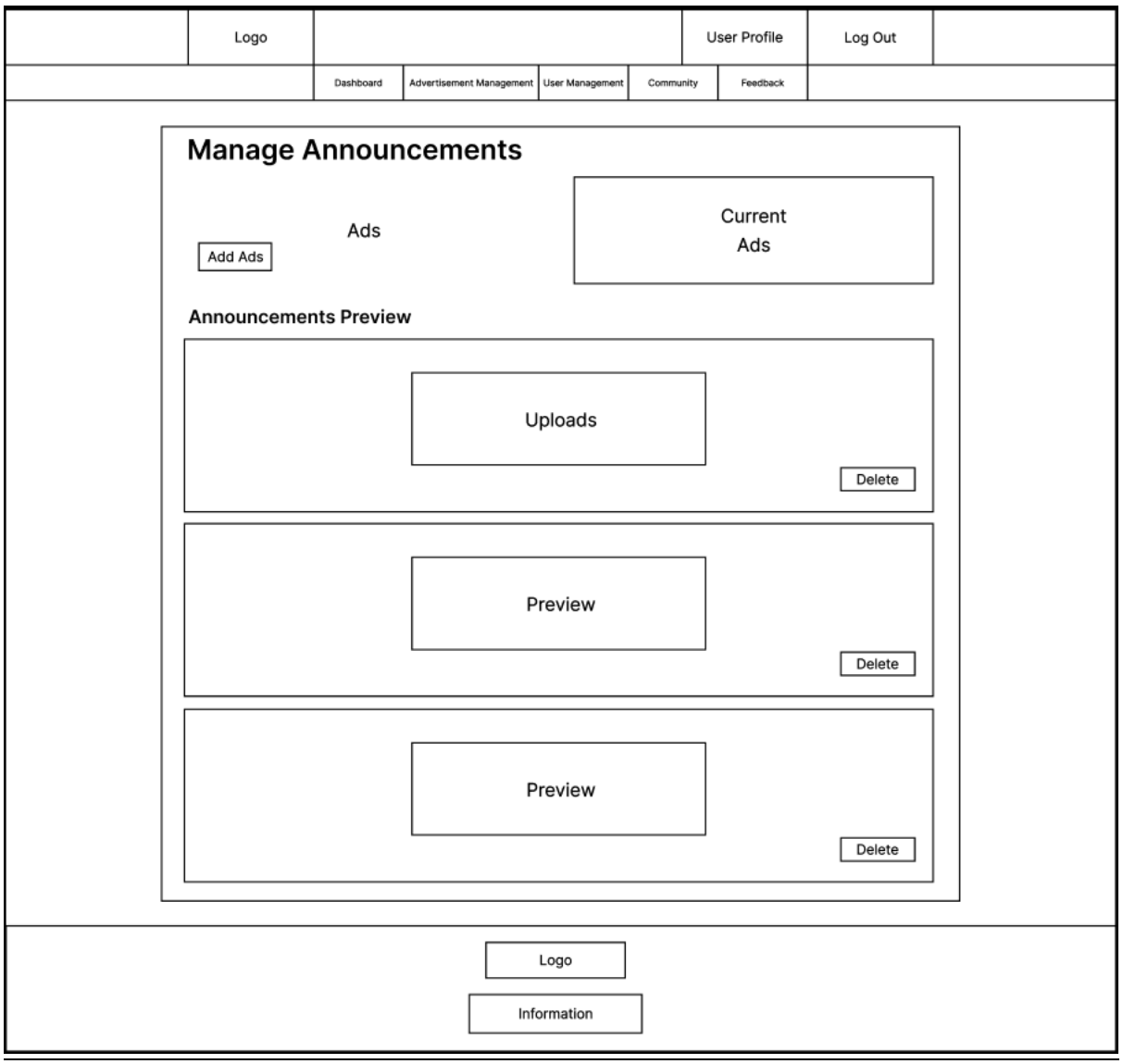

Figure 50: Advertisment Management Page

This wireframe shows the Manage Advertisement, and it is devoted to the control of advertising. The section titled Ads includes an Add Ads button to add new advertisement, and a big display box titled Current Ads, where there is probably a summary or a status of the ongoing campaigns. The second large category is Announcements Preview. In this section, the existing advertise is listed in blocks. The blocks give a preview of the content marked Uploads or Preview and a Delete button offering the administrator the option of deleting or retiring ads.

User Management Page

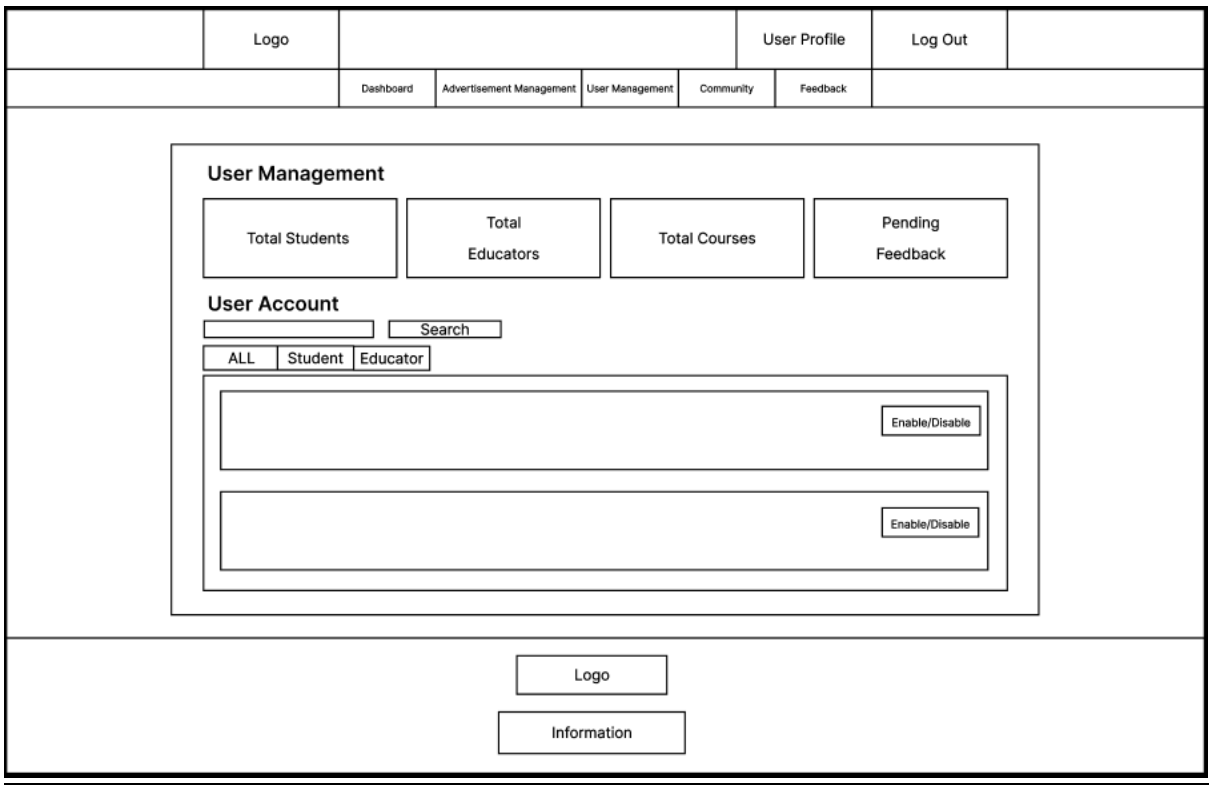

Figure 51: User Management Page

This wireframe shows User Management. Like the Admin Dashboard, it shows major metric blocks at the top, comprising of Total Students, Total Educators, Total Courses, and Pending Feedback. The User Account management section is at the centre of the page, and it has a Search bar where one can find a particular user. Filter on accounts is offered using ALL button, Student button and Educator button. The user accounts are displayed below the filters in repeating horizontal blocks with each account having an Enable/Disable button, thereby allowing the administrator to control the activated status of each of the user accounts.

Community Management Page

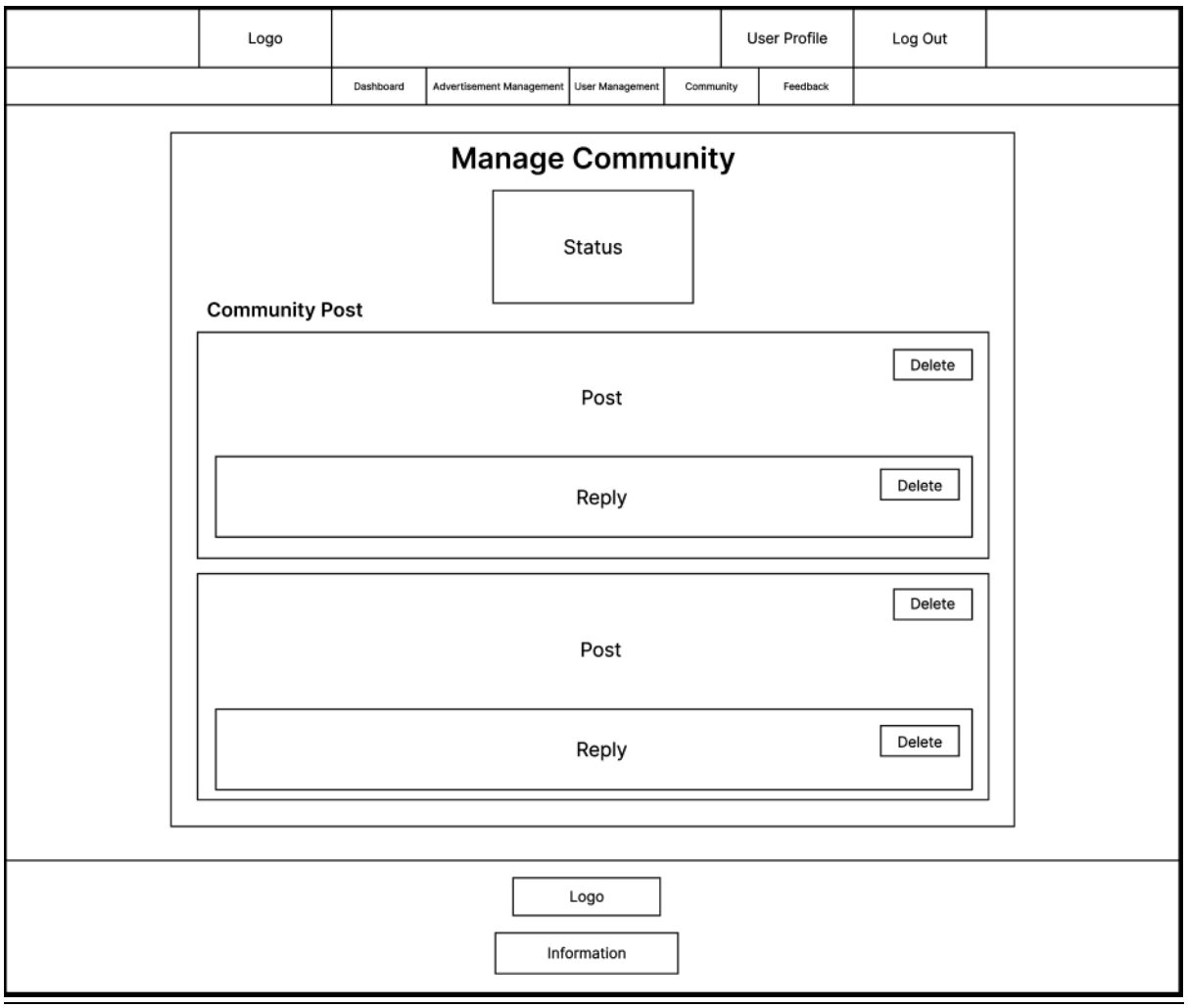

Figure 52: Community Management Page

This wireframe shows Manage Community and is reached through the navigation of the admin. There is a Status block at the top of the content area that probably provides important summary community information, including the total number of posts or unmoderated material. The primary contents of the page include the review and management of Community Post entries. The repeating blocks contain the posts and Reply entries related to them, which enables the administrator to check the discussion order. Both the main Post and individual Reply have a Delete button that promotes the administrator with a fine degree of control to delete any inappropriate or spam messages interacting with the community forum.

Manage Feedback Page

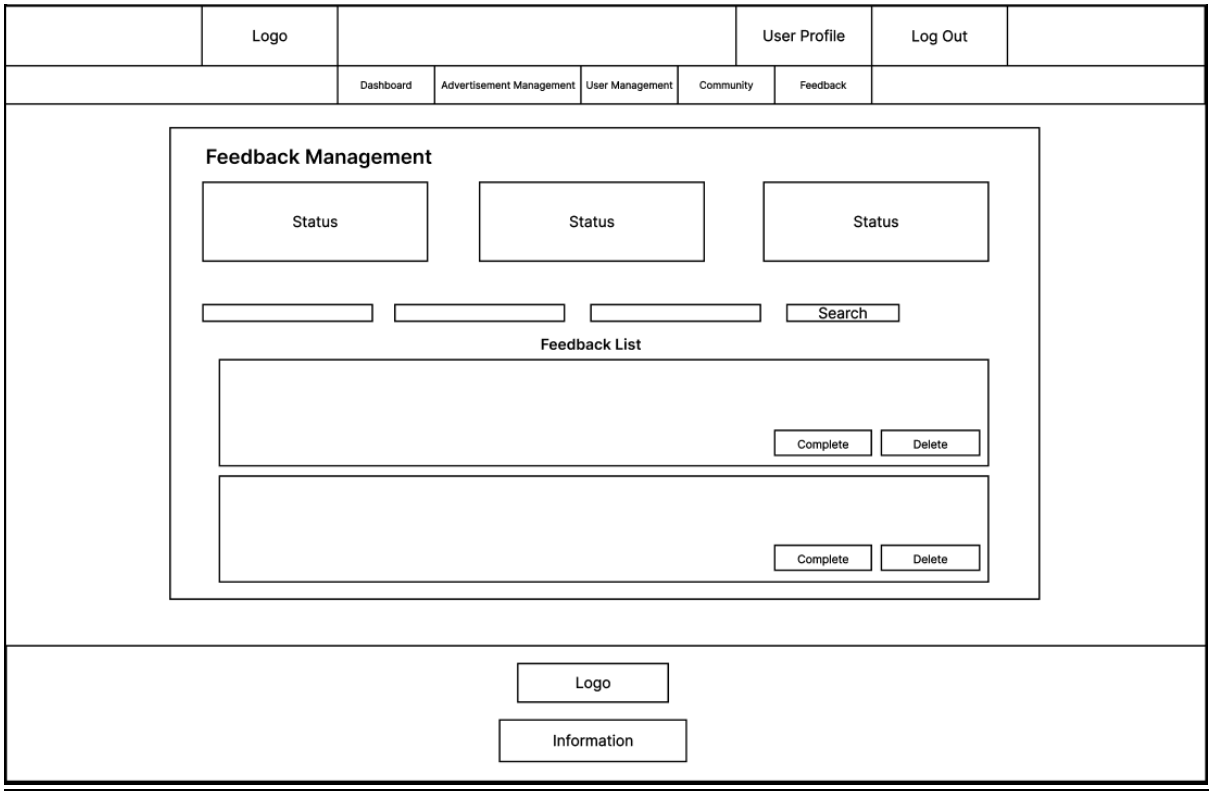

Figure 53: Manage Feedback Page

This wireframe shows Feedback Management which is devoted to managing all incoming user comments, suggestions, or problems. Three Status blocks are located at the top and show major measures regarding the feedback, which is total received, pending, resolved. Under the status indicators, two fields of input possibly to filter or sort by date or user appear, and this is followed by a Search button. The principal part is the Feedback List, that shows individual feedback in repeating horizontal rows. Every entry is assigned action buttons, Complete (to indicate the feedback as responded to) and Delete (to eliminate the entry permanently), providing the administrator with absolute access to the feedback pipeline.

3.3 Website Navigational Structure

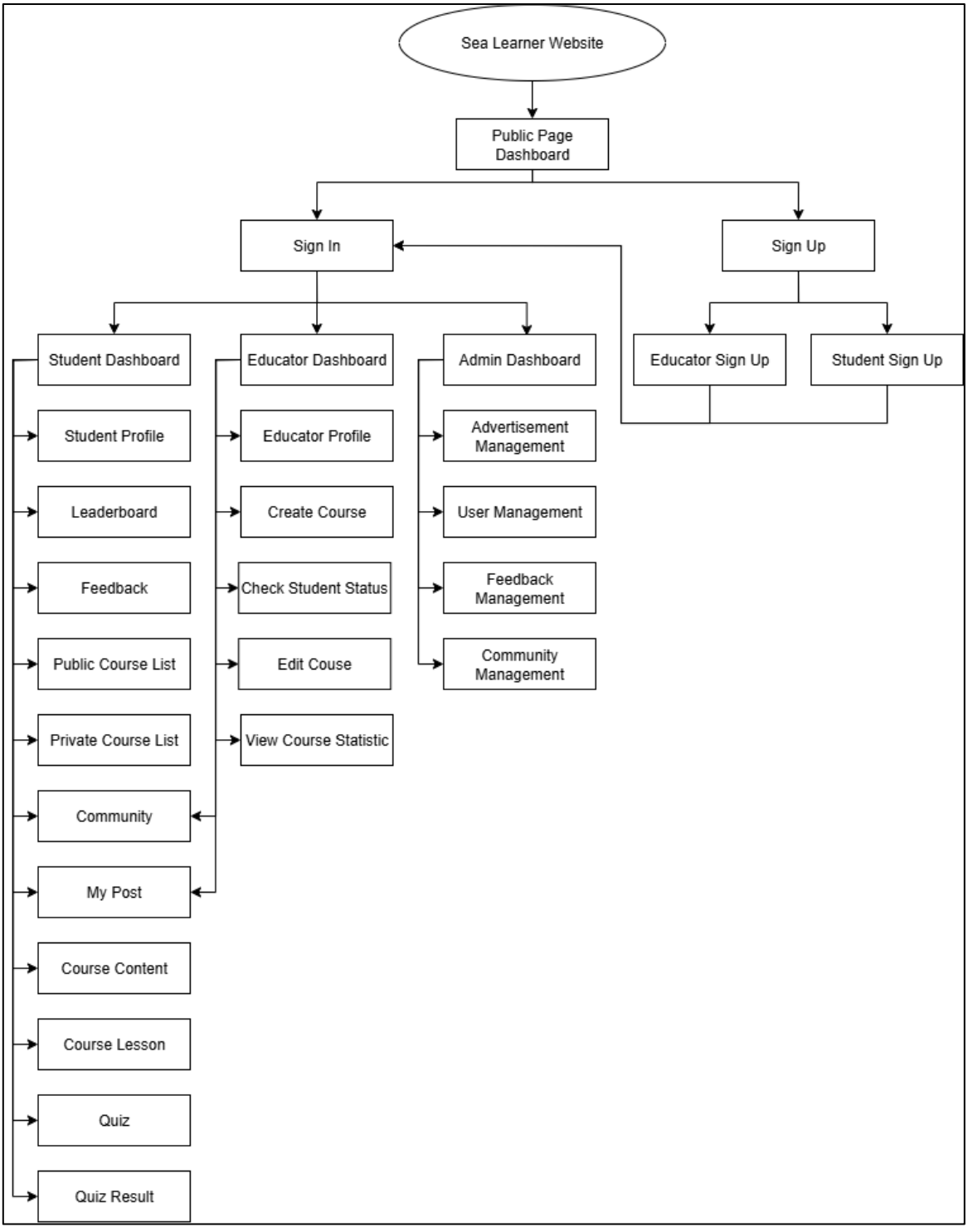

Figure 54: Website Navigational Structure

The Sea Learner site relies on a role-based navigation system to achieve a clear and well-organized user experience. The primary point of entry is the Public Page Dashboard, where one can Sign In or Sign Up with either a student or educator account. Once they sign in, they are

then taken to their own dashboards, Student Dashboard, Educator Dashboard or Admin Dashboard.

Student Dashboard consists of Profile, Leaderboard, Feedback, Public/Private Courses, Course Lessons, Quizzes, and Results. The Community Page is also available to students who want to participate in discussions and My Post Page where they can manage their own posts. The Educator Dashboard also offers Profile, Create/Edit Course, Check Student Status and View Course Statistics, the Community Page and the My Post Page to share and manage posts. Admin Dashboard is concerned with Advertisement, User, Feedback and Community Management that makes the platform run smoothly.

4.0 Implementation

4.1 CSS for SeaLearner Page Styling **OOI DUN TZI**

1. Body & HTML Base

```
body {
    font-family: 'Segoe UI', sans-serif;
    background: linear-gradient(135deg, #eaf0ff, #ffffff);
}

html::-webkit-scrollbar {
  display: none;
}
```

Figure 55: Body & HTML Base

Explanation:

In the body tyle there has created a consistent foundation by setting the font for the entire website to the clean and modern “Segoe UI”, and also the plain colour that using is a linear gradient background that able to transitions from a soft blue #eaf0ff to white. Therefore, this able to let the entire page have a soft and professional feel. Not only that, it also included a utility snippet on almost every page to hide the browser default scrollbar using html::-webkit-scrollbar.

2. "Frosted Glass" (Glass morphism) Containers

```
.dashboard-container {
    background-color: white;
    border-radius: 12px;
    padding: 40px;
    box-shadow: 0 2px 10px rgba(0,0,0,0.1);
    width: 99%;
    max-width: 1900px;
    margin: 0px auto 50px;
    background: rgba(255, 255, 255, 0.25);
    border-radius: 12px;
    box-shadow: 0 4px 25px rgba(0, 30, 255, 0.25);
    backdrop-filter: blur(10px);
    -webkit-backdrop-filter: blur(10px);
    border: 1px solid rgba(255, 255, 255, 0.3);
}
```

Figure 56: "Frosted Glass" (Glass morphism) Containers

Explanation:

The most important and most common style in the entire project is the main content containers, such as the sign in boxes, dashboard cards, and the popups to achieve a modern look where the panel seems to be made of frosted glass. Therefore, this effect is created by combining the four key properties, which is the first key will be the semi transparent background: rgba(255,255,255,0.25) allows the page gradient to show through. Next will be the second key that using backdrop-filter: blur(10px) to blur anything behind the container that creates the frosted look. Furthermore, the third key is the signature box-shadow: 0 4px 25px rgba(0,30,255,0.25) adds a soft blue glow that makes the panel seem to float. Lastly, a subtle border gives the glass a defined edge.

3. Primary Button Styling

```css
.btn {
  background-color: #001eff;
  color: white;
  border: none;
  border-radius: 6px;
  padding: 10px 20px;
  font-size: 16px;
  cursor: pointer;
  margin: 5px;
  transition: background-color 0.3s ease, color 0.3s
  ease, transform 0.25s ease, box-shadow 0.25s ease;
}

.btn:hover {
  transform: translateY(-2px);
  box-shadow: 0 6px 15px rgba(0, 30, 255, 0.25);
  opacity: 0.95;
}
```

Figure 57: Primary Button Styling

Explanation:

In this project there have a very consistent style for the main button, that using solid blue of #oo1eff and lift off the page when hovered and providing clear visual feedback to the user. Therefore, the style will be built on the transition property, which smoothly animated all the changes over 0.25-0.3 seconds. Next, for the hover the background colour and colour of the properties are inverted to get a very clear cue, and the most important effect is the transform:translateY(-2px) that lift the button up by 2 pixels. Because of that, the combination with the box-shadow that also shows on the hover will makes the button feel more interactive and three dimensional feels.

4. Standardized Form Styling

```css
label {
    display: block;
    color: #374151;
    font-weight: 600;
    margin-bottom: 5px;
    text-align:left;
}

input[type="text"], input[type="password"] {
    width: 100%;
    padding: 10px;
    border-radius: 6px;
    border: 1px solid #d1d5db;
    background-color: #f9fafb;
    max-width: none;
}
```

```css
input:focus, select:focus {
    outline: none;
    border-color: #001eff;
    box-shadow: 0 4px 25px rgba(0, 30, 255, 0.25);
    background: white;
}
```

Figure 58: Standardized Form Styling

Explanation:

Here, all the elements that are label have been set to display: block, and each label is separated into own lines and the labels placed above the relevant input field. Moreover, the labels are in font-weight:600 and left-aligned to ensure that the layout remains transparent to be easily readable, and the element such as an input, a selection, and a textarea are in the width:100% to enable them to be fully responsive and to occupy the entire width of the container. And that too, there is a state of focus to make it easier to use, using the project blue border-colour and a matching box shadow to indicate the active field.

5. Main Menu Bar (Site.master)

```
#menuBar {
    background: rgba(255, 255, 255, 0.25);
    border-radius: 12px;
    box-shadow: 0 4px 25px rgba(0, 30, 255, 0.25);
    backdrop-filter: blur(10px);
    -webkit-backdrop-filter: blur(10px);
    border: 1px solid rgba(255, 255, 255, 0.3);
    padding: 10px 0;
    margin-top: 60px;
}
#menuBar a {
    text-decoration: none;
    color: black;
    font-weight: 500;
    transition: all 0.3s ease;
    position: relative;
    margin: 0 12px;
}
#menuBar a:hover {
    color: #001eff;
    transform: translateY(-4px);
}
#menuBar a::after {
    content: '';
    position: absolute;
    width: 0%;
    height: 2px;
    bottom: -2px;
    left: 50%;
    background-color: #001eff;
    transition: all 0.3s ease;
    transform: translateX(-50%);
}
#menuBar a:hover::after {
    width: 60%;
}
```

Figure 59: Main Menu Bar (Site.master)

Explanation:

The primary navigation, which is the "#menuBar" has a frosted glass effect (semi-transparent background, backdrop-filter, blue box-shadow) and it seems like it is floating over the page contents. Moreover, menu links are very interactive: the links on the menu move up by 4 pixels when the mouse hovers over them (transform: translateY (-4px)) and they feel like they are responsive. In addition, the pseudo-element ::after is utilized in order to make an animated underline, this line has its opening width: 0% and goes to 60% on hover, which gives the impression that the underline is drawing itself.

4.2 Database Connectivity (SQL Queries)

4.2.1 Insert **OOI DUN TZI**

SECTION 1

Creating the Main User Record (Users Table)

```
string insertUserQuery = @"INSERT INTO Users
    (FullName, Email, Password, Role, Age, Gender, father, mother, Status)
    OUTPUT INSERTED.Id
    VALUES (@FullName, @Email, @Password, @Role, @Age, @Gender, @Father, @Mother, @Status)";


SqlCommand cmdUser = new SqlCommand(insertUserQuery, conn, transaction);
cmdUser.Parameters.AddWithValue("@FullName", txtFullName.Text.Trim());
cmdUser.Parameters.AddWithValue("@Email", txtEmail.Text.Trim());
cmdUser.Parameters.AddWithValue("@Password", txtPassword.Text.Trim());
cmdUser.Parameters.AddWithValue("@Role", rblRole.SelectedValue);
cmdUser.Parameters.AddWithValue("@Status", "active");


int age = 0;
string gender = "";

if (rblRole.SelectedValue == "student")...
else if (rblRole.SelectedValue == "educator")...

cmdUser.Parameters.AddWithValue("@Age", age);
cmdUser.Parameters.AddWithValue("@Gender", gender);

cmdUser.Parameters.AddWithValue("@Father", txtFatherName.Text.Trim());
cmdUser.Parameters.AddWithValue("@Mother", txtMotherName.Text.Trim());

int userId = Convert.ToInt32(cmdUser.ExecuteScalar());
```

Figure 60: Insert into Users

When every time user input the information and successfully passes the validation, the system will open a database connection and initiates a transaction to maintain the atomicity. Furthermore, the first operation performed is the insertion of user general information like the FullName, Email, Password, Age, and others as shown in the image above into the “Users” table, and this will be carried out through the parameterized SQL “INSERT” statement that includes “OUTPUT INSERTED.Id” to immediately retrieve the newly generated “UserId”. Moreover, before inserting the records to the system it will also need to identify the user role, which is when the user is classified as a student there will have a student specific field to fill in and this is also same as the educator, they will also have an educator specific field to filled. As a result, this will be able to ensure that each of the person will have a complete master record in the “Users” table to serve as the basis for adding specific role data.

SECTION2

Inserting Role-Specific Data (Student or Educator)

```
if (rblRole.SelectedValue == "student")
{
    string insertStudentQuery = @"INSERT INTO Student (UserId, School, InterestSubject)
                                  VALUES (@UserId, @School, @InterestSubject)";
    SqlCommand cmdStudent = new SqlCommand(insertStudentQuery, conn, transaction);
    cmdStudent.Parameters.AddWithValue("@UserId", userId);
    cmdStudent.Parameters.AddWithValue("@School", txtSchool.Text.Trim());
    cmdStudent.Parameters.AddWithValue("@InterestSubject", ddlSubject.SelectedValue);
    cmdStudent.ExecuteNonQuery();
}
else if (rblRole.SelectedValue == "educator")
{
    string insertEducatorQuery = @"INSERT INTO Educator (UserId, EducationQualification, GraduatedUniversity)
                                   VALUES (@UserId, @EducationQualification, @GraduatedUniversity)";
    SqlCommand cmdEducator = new SqlCommand(insertEducatorQuery, conn, transaction);
    cmdEducator.Parameters.AddWithValue("@UserId", userId);
    cmdEducator.Parameters.AddWithValue("@EducationQualification", ddlQualification.SelectedValue);
    cmdEducator.Parameters.AddWithValue("@GraduatedUniversity", txtUniversity.Text.Trim());
    cmdEducator.ExecuteNonQuery();
}
```

Figure 61: Student or Educator Insert

Once the main records have already successfully created and the “UserId” retrieved, the system will proceed to the insert role-specific details shows in the image above into either “Student” table or the “Educator” table, based on what have the user selected. Lastly, all supplementary details have store in the appropriate table, and each data will also properly link back to the main “Users” entry thorough the “UserId” as foreign key. As a result, this is a structure that maintains clear relational design and allows the application to manage and query user information efficiently based on the role type.

4.2.2 Select **CHEN XIN ZE**

Section 1: Input Validation Logic

The system checks that the user accessing the dashboard is an eligible educator before accessing any data in the database. This is validated in the “Page_Load” where the system will authenticate the user with the session to verify that the account is an educator. The educator ID that is needed in all further “SELECT” statements is also authenticated. When the educator ID is not present, the system tries to get it in the table of “Educator” with the current user ID. The system won’t run any database queries until the identity of the educator is verified and the corresponding “EducatorID” is acquired successfully. This will avoid unauthorized access and will guarantee that the accessed course data is that of the right educator.

Code Snippet: Validation

```
// Load educator ID if not already in session
if (Session["EducatorID"] == null)
{
    int userId = Convert.ToInt32(Session["UserId"]);
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
        SqlCommand cmd = new SqlCommand("SELECT Id FROM Educator WHERE UserId = @uid", conn);
        cmd.Parameters.AddWithValue("@uid", userId);
        object result = cmd.ExecuteScalar();

        // If educator exists, store ID in session
        if (result != null)
        {
            Session["EducatorID"] = Convert.ToInt32(result);
            Session["EducatorName"] = Session["FullName"];
        }
        else
        {
            // If educator not found, create a new record
            SqlCommand insert = new SqlCommand(
                "INSERT INTO Educator (UserId, EducationQualification) VALUES (@uid, 'Not Set'); SELECT SCOPE_IDENTITY();",
                conn
            );

            insert.Parameters.AddWithValue("@uid", userId);
            object newId = insert.ExecuteScalar();

            Session["EducatorID"] = Convert.ToInt32(newId);
            Session["EducatorName"] = Session["FullName"];
        }
    }
}
```

Figure 62: Input Validation Logic

Section 2: Executing the Select Query (Load Course List)

After validation, all courses made by the educator are retrieved by that system. This is done by a parameterized “SELECT” query to avoid SQL injection and secure database communication. The query will retrieve critical course data such as “course title”, “course type”, “number of lessons”, and “number of enrolled students”. The number of lessons in each course and the number of students who have enrolled in it are two more values to be computed by using subqueries. These values enable the educator to get a quick view of the status and level of participation of each course on their dashboard.

Code Snippet: SELECT query execution

```
string sql =
    @"SELECT c.id, c.title, c.courseType,
        ISNULL((SELECT COUNT(*) FROM lesson l WHERE l.courseId = c.id),0) AS LessonCount,
        ISNULL((SELECT COUNT(*) FROM studentCourseProgress scp WHERE scp.courseId = c.id),0) AS StudentCount
      FROM course c
      WHERE c.educatorId = @eid
      ORDER BY c.title";

using (SqlCommand cmd = new SqlCommand(sql, conn))
{
    cmd.Parameters.AddWithValue("@eid", _educatorId);
    using (SqlDataAdapter da = new SqlDataAdapter(cmd))
    {
        da.Fill(dt);
    }
}
```

Figure 63: Executing the SELECT Query

Section 3: Processing the Retrieved Results

Once the SQL query is executed the data is inserted into a “DataTable” object. This “DataTable” is used as an in-memory representation of the list of courses, each row represents a course of the educator. This system will then attach this “DataTable” to a “Repeater” control that will show the courses on the dashboard in a structured and friendly format. The system separates the data retrieval and the data binding operation to make sure that the backend logic is well organized and the user interface is responsive and dynamic.

Code Snippet: Take the SELECT result (dt) and displays it

```
// Bind the data to the Repeater for display
rptCourses.DataSource = dt;
rptCourses.DataBind();
```

Figure 64: Processing the Retrieved Results

Section 4: Error Handling and Empty Result Management

The “SELECT” operation is put in a controlled block of database connection to facilitate secure execution. When the educator does not have any created courses, the query just provides an empty result set, and the "Repeater" does not reveal any entries of the course without causing errors. Also, parameterized queries will aid in avoiding SQL injection and by default, the system will always verify the educator ID before executing the “SELECT” statement. This validation, safe querying, and controlled rendering combination will guarantee that the dashboard will display accurate information, and the system is secure and stable.

Code Snippet: Error handling

```
using (SqlDataAdapter da = new SqlDataAdapter(cmd))
{
    da.Fill(dt);
}
```

Figure 65: Error Handling and Empty Result Management

4.2.3 Update **YAP BOON SIONG**

SECTION 1: Input Validation Before Update

The system verifies the input of the user before any update is made to the database, and it is done to make sure that the information is complete and acceptable. The validation checks that were mandatory like full name, age, school, interest subject, and gender to be not empty before proceeding. Moreover, depending on whether a profile picture is uploaded or not, the system determines the file extension with Path.GetExtension() to verify that it is a right image file that is being saved. These checks are made to make sure that no invalid or incomplete data is handled.

Code Snippet: Validation

```
protected void btnSaveProfile_Click(object sender, EventArgs e)
{
    int userId = Convert.ToInt32(Session["UserId"]);
    string fullName = txtFullName.Text.Trim();
    string school = txtSchool.Text.Trim();
    string subject = txtInterestSubject.Text.Trim();
    string gender = ddlGender.SelectedValue;
    int.TryParse(txtAge.Text.Trim(), out int age);

    string fileName = null;
    if (fileUploadProfile.HasFile)
    {

        string extension = Path.GetExtension(fileUploadProfile.FileName);
        fileName = "profile_" + userId + "_" + DateTime.Now.Ticks + extension;

        string folderPath = Server.MapPath("~/Image/");
        if (!Directory.Exists(folderPath))
            Directory.CreateDirectory(folderPath);

        fileUploadProfile.SaveAs(Path.Combine(folderPath, fileName));
    }
```

Figure 66: Input Validation Before Update

SECTION 2: Updating the Student Record (Users & Student Table)

Once the input has passed the validation, the system modifies the general user data which is stored in the Users and Student table. SQL UPDATE parameterized query is a statement that guarantees the safe execution and eliminates SQL injection. The fields incorporated in the update are the full name of the user, age and sex of the user. When a new profile picture is posted, then the SQL update is also added to the ProfilePicture column, otherwise it leaves the old picture uploaded. For new updates for school and interest subject will be then update in Student table.

Code Snippet: Update Users / Update Student

```
using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();


    string queryUser = @"
        UPDATE Users
        SET FullName = @name, Age = @age, Gender = @gender
            " + (fileName != null ? ", ProfilePicture = @pic" : "") + @"
        WHERE Id = @uid";

    SqlCommand cmdUser = new SqlCommand(queryUser, conn);
    cmdUser.Parameters.AddWithValue("@name", fullName);
    cmdUser.Parameters.AddWithValue("@age", age);
    cmdUser.Parameters.AddWithValue("@gender", gender);
    cmdUser.Parameters.AddWithValue("@uid", userId);
    if (fileName != null)
        cmdUser.Parameters.AddWithValue("@pic", fileName);
    cmdUser.ExecuteNonQuery();


    string queryStudent = @"
        UPDATE Student
        SET School = @school, InterestSubject = @subject
        WHERE UserId = @uid";

    SqlCommand cmdStudent = new SqlCommand(queryStudent, conn);
    cmdStudent.Parameters.AddWithValue("@school", school);
    cmdStudent.Parameters.AddWithValue("@subject", subject);
    cmdStudent.Parameters.AddWithValue("@uid", userId);
    cmdStudent.ExecuteNonQuery();
```

Figure 67: Updating the Student Record (Users & Student Table)

SECTION 3: Session Update and Error Handling

Once the Users and Student tables are updated successfully, the system will update the corresponding session variables accordingly such that the full name and profile picture that have been updated are immediately reflected across the platform. It is also stable because the system uses safe error-handling techniques, in case of an error during the update with a file operation or an SQL query, the system will not allow any partial updates but instead will present a helpful error message. The combination of this method can guarantee a stable and convenient update process.

Code Snippet: Session Update & Error Handling Logic

```
Session["FullName"] = fullName;
if (fileName != null)
    Session["ProfilePicture"] = fileName;

lblMessage.Text = "Profile updated successfully!";
LoadProfile();
```

Figure 68: Session Update and Error Handling

4.2.4 Delete **CHEAH JUN HENG**

SECTION 1: Database Connection and Transaction Setup

The system establishes a secure connection to the database using the connection string from configuration and initiates a SQL transaction. This transaction ensures that all subsequent database operations either complete successfully as a single unit or are completely rolled back in case of any failure, maintaining database consistency and integrity throughout the deletion process.

Code Snippet: Connection and Transaction Initialization

```
1 reference
private bool DeleteCourseAndAllRelatedData(string courseId)
{
    string connStr = ConfigurationManager.ConnectionStrings["SeaLearnerConnection"].ConnectionString;
    using (SqlConnection conn = new SqlConnection(connStr))
    {
        conn.Open();
        SqlTransaction transaction = conn.BeginTransaction();
```

Figure 69: Delete Setup

SECTION 2: Cascading Deletion with Foreign Key Management

The system executes a carefully ordered sequence of parameterized SQL DELETE queries to remove all course-related data while respecting foreign key constraints. The deletion begins with the most dependent tables (StudentQuizProgress and Question), progresses through intermediate tables (Quiz, StudentCourseProgress, and Lesson), and finally removes the main course record. Each query uses parameterized commands with @CourseID to prevent SQL injection attacks.

Code Snippet: Ordered Deletion Queries Execution

```
        try
        {
            // Delete in correct order to respect foreign key constraints
            string[] deleteQueries = {
                // Delete from tables that reference other tables first
                "DELETE FROM StudentQuizProgress WHERE QuizId IN (SELECT Id FROM Quiz WHERE LessonId IN (SELECT Id FROM Lesson WHERE Cours
                "DELETE FROM Question WHERE QuizId IN (SELECT Id FROM Quiz WHERE LessonId IN (SELECT Id FROM Lesson WHERE CourseId = @Cou

                // Then delete from intermediate tables
                "DELETE FROM Quiz WHERE LessonId IN (SELECT Id FROM Lesson WHERE CourseId = @CourseID)",
                "DELETE FROM StudentCourseProgress WHERE CourseId = @CourseID",
                "DELETE FROM Lesson WHERE CourseId = @CourseID",

                // Finally delete the course
                "DELETE FROM Course WHERE Id = @CourseID"
            };

            foreach (string query in deleteQueries)
            {
                using (SqlCommand cmd = new SqlCommand(query, conn, transaction))
                {
                    cmd.Parameters.AddWithValue("@CourseID", courseId);
                    cmd.ExecuteNonQuery();
                }
            }

            transaction.Commit();
            return true;
        }
```

Figure 70: Ordered Deletion Queries

SECTION 3: Transaction Commitment and Error Handling

Upon successful execution of all deletion queries, the system commits the transaction, making all changes permanent. If any error occurs during the deletion process, the system rolls back the entire transaction, ensuring no partial deletions occur. The error is then re-thrown to be handled by the calling method, providing a robust safety mechanism against data corruption.

Code Snippet: Transaction Finalization

```
                transaction.Commit();
                return true;
            }
            catch (Exception)
            {
                transaction.Rollback();
                throw;
            }
        }
    }
```

Figure 71: Delete error handling

This structured approach guarantees that course deletion is performed safely and completely, with comprehensive error handling that maintains database integrity under all circumstances.

4.3 Explanation Form Validation and Key Source Code Features

4.3.1 Public/Core **OOI DUN TZI**

4.3.1.1 SiteMaster

1. Public vs Authenticated Pages

```
if (currentPage == "public_dashboard" || currentPage == "sign_in" ||
    currentPage == "sign_up" || currentPage == "forgot_password")
{
    navSignIn.Visible = true;
    navSignUp.Visible = true;
    navLogout.Visible = false;
    navUserProfile.Visible = false;
    menuBar.Visible = false;
}
else
{
    navSignIn.Visible = false;
    navSignUp.Visible = false;

    if (Session["UserId"] != null)...
    else...
}
```

Figure 72: Public vs Authenticated Pages

The difference between the SiteMaster with public and authenticated pages is that it is to manage what the element are visible to the user. For example, those pages like the sign in, sign up, forgot password or public dashboard will be displayed the Sign in and Sign Up button prominently for the user, in other side the Logout button, User Profile and the menu bar remain hidden. However, the same things that going happen is that the sign in and sign up button will be hidden in other then the page has mentioned. Therefore, this is to ensure that the user will have a clean header to see the elements that relevant to their current access level.

2. User Session Handling

```
if (Session["UserId"] != null)
{
    navUserProfile.Visible = true;
    navLogout.Visible = true;

    lblUserFullName.Text = Session["FullName"].ToString();

    string role = Session["Role"].ToString().ToLower();
    lblRole.Text = char.ToUpper(role[0]) + role.Substring(1).ToLower();
```

Figure 73: User Session Handling

In SiteMaster it will handles the user sessions by checking the UserId when user is logged in, through the existence Session[“UserId”] then if the session is active the profile and logout button will be displayed along with the users full name and role as the identity. Furthermore, the system will attempt to load the user profile picture as well, if the user hasn’t set the profile picture it will display the default image, and this will provide a personalized experience while maintaining the consistent session management.

3. Role-based Profile Links

```
if (role == "educator")
{
    lnkProfile.HRef = "~/educatorProfile.aspx";
}
else if (role == "student")
{
    lnkProfile.HRef = "~/studentProfile.aspx";
}
else
{
    lnkProfile.HRef = "~/profile.aspx";
}
```

Figure 74: Role-based Profile Links

Moreover, the system will redirect the users to their respective profile pages based on their roles and this is to ensure that each user will accesses to their appropriate profile interface that corresponds to their permission and responsibilities within the SiteMaster.

4. Dashboard Link Based on Role

```
switch (role)
{
    case "student":
        navDashboardLink.HRef = "~/StudentDashboard.aspx";
        break;
    case "educator":
        navDashboardLink.HRef = "~/educatordashboard.aspx";
        break;
    case "admin":
        navDashboardLink.HRef = "~/admin_dashboard.aspx";
        break;
    default:
        navDashboardLink.HRef = "~/public_dashboard.aspx";
        break;
}
```

Figure 75: Dashboard Link Based on Role

As well as the dashboard of each user will have the similar system like the profile page that mention earlier, which is the system will be depending on the user role and allows each user to be directed to their respective dashboard.

5. Menu Visibility

```
studentMenu.Visible = false;
educatorMenu.Visible = false;
adminMenu.Visible = false;
```

Figure 76: Menu Visibility

Menu visibility will be set based on role, which will prevent unauthorized users from accessing pages that are accessible to other roles. For example, the student, educator, admin will display their respective menu with different page options in it, and this is to maintain the clean interface while ensuring that the users have access only to the menus that are relevant to them and those settings will be shown below.

Student Menu Pages

```
if ((currentPage == "studentdashboard" ||
    currentPage == "feedback" ||
    currentPage == "leaderboard" ||
    currentPage == "privatecourse" ||
    currentPage == "publiccourse" ||
    currentPage == "studentprofile" ||
    currentPage == "studentcoursecontent" ||
    currentPage == "studentlesson" ||
    currentPage == "studentquiz" ||
    currentPage == "studentquizresult" ||
    (currentPage == "community" && userRole == "student")||
    (currentPage == "my_post" && userRole == "student")))
{
    studentMenu.Visible = true;
```

Figure 77: Student Menu Pages

Educator Menu Pages

```
else if ((currentPage == "educatordashboard" ||
          currentPage == "educatorprofile" ||
          currentPage == "createcourse" ||
          currentPage == "educatorcoursestudents" ||
          currentPage == "editcourse"||
          (currentPage == "community" && userRole == "educator")||
          (currentPage == "my_post" && userRole == "educator")))
{
    educatorMenu.Visible = true;
```

Figure 78: Educator Menu Pages

Admin Menu Pages

```
else if (currentPage == "admin_dashboard" ||
         currentPage == "adminmanage_ads" ||
         currentPage == "adminuser_management" ||
         currentPage == "admin_communitymanagement" ||
         currentPage == "admin_feedbackmanagement")
{
    adminMenu.Visible = true;
}
```

Figure 79: Admin Menu Pages

6. Logout Button

```
protected void Logout_Click(object sender, EventArgs e)
{
    Session.Clear();
    Session.Abandon();
    Response.Redirect("~/public_dashboard");
}
```

Figure 80: Logout Button

For the logout button it will have the functionality to clear all the session variables and ends the current session, and when the user logout it will redirect back to the dashboard. Therefore, this is done to ensure that sensitive session information is properly cleared and to prevent information leakage, also to ensure system security.

4.3.1.2 Sign In

1. Page Initialization (Page_Load)

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        Session.Clear();
    }
}
```

Figure 81: Page Initialization (Page_Load)

Explanation

This method will be runs automatically when every time the Sign In page has loads, and this is to checks whether the page us being loaded for the first time by using (!IsPostBack). If yes, then it will clear all the existing session data to ensures the user visits the sign in page will not showing the previous user session.

2. Retrieving and Validating User Input

```
string email = txtEmail.Text.Trim();
string password = txtPassword.Text.Trim();

if (string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password))
{
    lblError.Text = "Please enter both email and password.";
    lblError.ForeColor = System.Drawing.Color.Red;
    return;
}
```

Figure 82: Retrieving and Validating User Input

Explanation

In this section is that when the user clicked the sign in button it will retrieve the email and password that entered by the user, at the same time the .Trim method will remove all the extra spaces to proceeding the application check if all the information has filled. But if the field is empty there will show the error message to the user as a notification warning, and this will prevent unnecessary database queries and enforces basic form validation.

3. Verifying Credentials in the Users Table

```
string userQuery = "SELECT * FROM Users WHERE Email " +
    "= @Email AND Password = @Password";
SqlCommand userCmd = new SqlCommand(userQuery, conn);
userCmd.Parameters.AddWithValue("@Email", email);
userCmd.Parameters.AddWithValue("@Password", password);

SqlDataReader reader = userCmd.ExecuteReader();
```

Figure 83: Verifying Credentials in the Users Table

Explanation

The application compares the email and the password with the records in the Users table and the SQL injection attacks are prevented by using parameterized queries (SQL) for example the @Email and @Password. Not only that, the SqlDataReader will fetch the user details assuming that there is a record that will match it. In case no record is located, then the next step to verify the admin table will be the next step in the procedure.

5. Handling Active vs Inactive Accounts

```
string status = reader["Status"].ToString();

if (status.Equals("Inactive", StringComparison.
    OrdinalIgnoreCase))
{
    reader.Close();
    lblError.Text = "Your account has been blocked. " +
        "Please contact support.";
    lblError.ForeColor = System.Drawing.Color.Red;
    return;
}
```

Figure 84: Handling Active vs Inactive Accounts

Explanation

Furthermore, if a similar user is located, the application verifies the status of the account, and the account is marked as Inactive the login process is halted, and an error message is shown. As a result, this will guard against blocked users having access to the system.

6. Creating a User Session and Redirecting by Role

```
Session["UserId"] = reader["Id"].ToString();
Session["FullName"] = reader["FullName"].ToString();
Session["Email"] = reader["Email"].ToString();
Session["Role"] = reader["Role"].ToString().ToLower();
Session["ProfilePicture"] = reader["ProfilePicture"].ToString();

string role = reader["Role"].ToString().ToLower();
reader.Close();

if (role == "student")
    Response.Redirect("~/studentdashboard.aspx");
else if (role == "educator")
    Response.Redirect("~/educatordashboard.aspx");
```

Figure 85: Creating a User Session and Redirecting by Role

Explanation

After a valid active user sign in, his or her information is saved as Session variables to enable the application to identify the user on the other pages, and once the data about the session is stored the system verifies the role of the user. Which allow the learners and the teachers redirected to other dashboards with role-specific access and functionality

7. Checking the Admin Table if User Not Found

```
string adminQuery = "SELECT * FROM Admin WHERE Email = " +
    "@Email AND Password = @Password";
SqlCommand adminCmd = new SqlCommand(adminQuery, conn);
adminCmd.Parameters.AddWithValue("@Email", email);
adminCmd.Parameters.AddWithValue("@Password", password);

SqlDataReader adminReader = adminCmd.ExecuteReader();

if (adminReader.Read())
{
    Session["UserId"] = adminReader["Id"].ToString();
    Session["Email"] = adminReader["Email"].ToString();
    Session["Role"] = "admin";
    Session["FullName"] = "Administrator";

    adminReader.Close();

    Response.Redirect("~/admin_dashboard.aspx");
}
```

Figure 86: Checking the Admin Table if User Not Found

Explanation

In case no similar record exists in the Users table, the system conducts a second search on the admin table, and this enables the administrators to use different credentials to log in. As a result, if an account with the role of an administrator is authenticated, a session is established, and the user will be redirected to the Admin Dashboard.

8. Invalid Login Handling

```
lblError.Text = "Invalid email or password.";
lblError.ForeColor = System.Drawing.Color.Red;
```

Figure 87: Invalid Login Handling

Explanation

Lastly, if the email and password are not the match of something in any of the two tables, the application will show a generic error message and it will prevent attackers to be aware of the presence of an email in the system, which is a positive security measure.

4.3.1.3 Sign Up

1. Page Initialization & Password Field Handling

```
protected void Page_Load(object sender, EventArgs e)
{
}
0 references
protected void Page_PreRender(object sender, EventArgs e)
{
    txtPassword.Attributes["value"] = txtPassword.Text;
    txtConfirmPassword.Attributes["value"] = txtConfirmPassword.Text;
}
```

Figure 88: Page Initialization & Password Field Handling

Explanation

PageLoad is invoked when the page with Sign Up loads, and because of this there will no special startup is required in this instance. Moreover, the passage of password fields is usually cleared in ASP.net as a security measure, and the PagePreRender method will be recovers the password and confirm password value and then displays the page. Therefore, this will be able to makes the passwords typed visible in the time of validation errors, which make the procedure user-friendly.

2. Dynamic Role Selection (Student / Educator)

```
protected void rblRole_SelectedIndexChanged(object sender, EventArgs e)
{
    if (rblRole.SelectedValue == "student")
    {
        studentSection.Visible = true;
        educatorSection.Visible = false;

        studentSection.CssClass = "studentSection visible";
        educatorSection.CssClass = "educatorSection";
    }
    else if (rblRole.SelectedValue == "educator")
    {
        studentSection.Visible = false;
        educatorSection.Visible = true;

        studentSection.CssClass = "studentSection";
        educatorSection.CssClass = "educatorSection visible";
    }
    else
    {
        studentSection.Visible = false;
        educatorSection.Visible = false;
        studentSection.CssClass = "studentSection";
        educatorSection.CssClass = "educatorSection";
    }
}
```

Figure 89: Dynamic Role Selection (Student / Educator)

Explanation

The Sign Up page will enable the users to choose whether to be a student or an Educator, by providing the radio button and if the user has chosen the user that they want to be the appropriate form fields are displayed and the extravagant ones are concealed by the application. This forms a dynamic type where students will insert school and subject information whereas educators will insert qualification and university details.

3. Basic Input Validation Before Database Operations

```
if (txtPassword.Text != txtConfirmPassword.Text)
{
    lblMessage.Text = "Passwords do not match!";
    return;
}

if (string.IsNullOrWhiteSpace(txtFullName.Text) ||
    string.IsNullOrWhiteSpace(txtEmail.Text))
{
    lblMessage.Text = "Please fill in all required fields!";
    return;
}
```

Figure 90: Basic Input Validation Before Database Operations

Explanation

The application checks on the creation of an account with some basic validation, such as verifies the existence of a matching password and confirm password and also confirms that some critical fields like full name and email are not left blank. Therefore, if the requirement is not fit it will have a message to displayed and halting of the registration process. As a result, this eliminates the possibilities of sending wrong or incomplete data to the database.

5. Inserting the Main User Record

```
string insertUserQuery = @"INSERT INTO Users
    (FullName, Email, Password, Role, Age, Gender, father, mother, Status)
    OUTPUT INSERTED.Id
    VALUES (@FullName, @Email, @Password, @Role, @Age, @Gender, @Father,
    @Mother, @Status)";


SqlCommand cmdUser = new SqlCommand(insertUserQuery, conn, transaction);
cmdUser.Parameters.AddWithValue("@FullName", txtFullName.Text.Trim());
cmdUser.Parameters.AddWithValue("@Email", txtEmail.Text.Trim());
cmdUser.Parameters.AddWithValue("@Password", txtPassword.Text.Trim());
cmdUser.Parameters.AddWithValue("@Role", rblRole.SelectedValue);
cmdUser.Parameters.AddWithValue("@Status", "active");
```

Figure 91: Inserting the Main User Record

Explanation

General information is stored in Users table of all types of accounts. This SQL query adds the full name of the user, email, password, role, age, gender and answers to questions on security details (father and mother name) to the database. Furthermore, the automatically generated user ID is gotten using the OUTPUT INSERTED.Id Clause, and this ID will be required to add corresponding information in the Student or Educator tables.

6. Processing Age and Gender Based on Role

```
int age = 0;
string gender = "";

if (rblRole.SelectedValue == "student")
{
    int.TryParse(txtAge.Text, out age);
    gender = ddlGender.SelectedValue;
}
else if (rblRole.SelectedValue == "educator")
{
    int.TryParse(txtAgeEdu.Text, out age);
    gender = ddlGenderEdu.SelectedValue;
}
```

Figure 92: Processing Age and Gender Based on Role

Explanation

The form applies various fields to the students and educators, next the code identifies the chosen role and retrieves the appropriate age and gender values respectively. Therefore, the int.TryParse works as a safe way of converting age into a number without exposing the program to errors in case of incorrectly typed entries.

7. Inserting Student or Educator-Specific Data

```
string insertStudentQuery = @"INSERT INTO Student (UserId, School, InterestSubject)
                              VALUES (@UserId, @School, @InterestSubject)";

insertEducatorQuery = @"INSERT INTO Educator (UserId, EducationQualification, GraduatedUniversity)
                        VALUES (@UserId, @EducationQualification, @GraduatedUniversity)";
```

Figure 93: Inserting Student or Educator-Specific Data

Explanation

Once the general user information is inserted the system will verify the role chosen by the user. In case of a user being a student, the school and subject of interest are entered into Student table, and if the user is an educator, his/her qualification and university are entered in the Educator table. These tables give the user profile role-specific information.

8. Transaction Commit and Error Handling

```
    transaction.Commit();
    success = true;
}
catch (Exception ex)
{
    transaction.Rollback();
    lblMessage.Text = "Error: " + ex.Message;
    success = false;
}
```

Figure 94: Transaction Commit and Error Handling

Explanation

In case the both insert operations (Users + Student/Educator) are successful, then the transaction is committed, and the data is permanently saved. In case of any error made during any of the phases, the transaction is reversed back so that there will be no partially stored data in the database. This makes the sign up process reliable and corruption of the data is avoided.

9. Redirecting Upon Successful Registration

```
if (success)
{
    Response.Redirect("sign_in.aspx?message=AccountCreated");
}
```

Figure 95: Redirecting Upon Successful Registration

Explanation

Once the account has been created successfully, the user would be redirected to Sign In page with a query parameter that shows success. This offers easy navigation in which customers can instantly gain access to their new account.

4.3.1.4 Forgot Password

1. Multi-Step Form Navigation (Client-Side)

```
function nextStep(step) {
    document.querySelectorAll('[id^="step"]').forEach(div => div.classList.add('hidden'));
    document.getElementById('step' + step).classList.remove('hidden');
}

function prevStep(step) {
    document.querySelectorAll('[id^="step"]').forEach(div => div.classList.add('hidden'));
    document.getElementById('step' + step).classList.remove('hidden');
}
```

Figure 96: Multi-Step Form Navigation (Client-Side)

Explanation

The Forgot Password page works with the three steps of verifying which involves entering an email, answering security questions, which include the name of the father and the mother, and finally the new password. Furthermore, to control the direction of transition between these steps, JavaScript puts each of the steps in its own container (step1, step2, and step3). Therefore, by clicking the "Next" or "Back" user will be able to hide all steps and only show the one he/she currently is in which gives a progressive wizard like password recovery process that does not have to reload any pages.

2. Page Initialization

```
protected void Page_Load(object sender, EventArgs e)
{
}
```

Figure 97: Page Initialization

Explanation

The page does not need special startup when loaded, and all these are handled by clicking on buttons and using JavaScript step wise navigation. Consequently, Page_Load is still not filled.

3. Collecting User Input & Basic Validation

```
string email = txtEmail.Text.Trim();
string father = txtFather.Text.Trim();
string mother = txtMother.Text.Trim();
string newPassword = txtNewPassword.Text.Trim();
string confirmPassword = txtConfirmPassword.Text.Trim();

if (newPassword != confirmPassword)
{
    lblMessage.Text = "Passwords do not match!";
    return;
}
```

Figure 98: Collecting User Input & Basic Validation

Explanation

On the last step, when the user clicks on the Done button, the system will get the email, the security answers (father and mother name) and the new password values. The initial check is that the new password and the confirm password are the same, and in case they are not equal, the process of the update will be interrupted and a message about the error will appear.

4. Establishing Database Connection

```
string connString = ConfigurationManager.ConnectionStrings["SeaLearnerConnection"].ConnectionString;

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();
```

Figure 99: Establishing Database Connection

Explanation

The application reads out the database connection string in Web.config and establishes a SQL connection. So that the using block will make sure that even in case of error, the connection will be automatically closed. This will provide a safe and trustworthy communication with the database.

5. Verifying User Identity with Security Questions

```
string checkQuery = "SELECT COUNT(*) FROM Users WHERE Email = " +
    "@Email AND father = @Father AND mother = @Mother";
SqlCommand cmdCheck = new SqlCommand(checkQuery, conn);
cmdCheck.Parameters.AddWithValue("@Email", email);
cmdCheck.Parameters.AddWithValue("@Father", father);
cmdCheck.Parameters.AddWithValue("@Mother", mother);

int count = (int)cmdCheck.ExecuteScalar();
```

Figure 100: Verifying User Identity with Security Questions

Explanation

Instead of OTP or email link, the system identifies the user by setting two security questions stored in the system, namely, the name of the father and mother, to reset the password. The SQL query looks after a record which fully coincides with the given email and the two security answers. When the query gives out a count of one, the identity verification is said to be successful, the system gives an error message that shows the details provided are not correct.

6. Updating the Password After Successful Verification

```
string updateQuery = "UPDATE Users SET Password = @Password WHERE Email = @Email";
SqlCommand cmdUpdate = new SqlCommand(updateQuery, conn);
cmdUpdate.Parameters.AddWithValue("@Password", newPassword);
cmdUpdate.Parameters.AddWithValue("@Email", email);
cmdUpdate.ExecuteNonQuery();

lblMessage.ForeColor = System.Drawing.Color.Green;
lblMessage.Text = "Password successfully updated!";
```

Figure 101: Updating the Password After Successful Verification

Explanation

After the user is authenticated, the system changes his or her password in the Users table itself. To avoid SQL injection, the update is performed using parameterized SQL. On successfully resetting a password, a message appears in green that confirms the password is changed.

7. Handling Invalid Security Answers

```
else
{
    lblMessage.Text = "Invalid details! Please check your email, father, and mother name.";
}
```

Figure 102: Handling Invalid Security Answers

Explanation

In case the system fails to identify a corresponding user record, this message will be printed, and it will avert unauthorized passwords reset and directs the user to re-verify his/her input.

4.3.1.5 Community Page

1. Page Load: Authentication Check + Initial Post Loading

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["UserId"] == null)
    {
        Response.Redirect("sign_in.aspx");
        return;
    }

    if (!IsPostBack)
    {
        LoadPosts();
    }
}
```

Figure 103: Page Load: Authentication Check + Initial Post Loading

Explanation

On page loading, the system would first check whether the user is logged in by verifying that he is who he claims to be by the Session["UserId] looked up. However, in case of an empty session, it takes the user to the sign-in screen, and this makes sure that it is only those users who are authorized to access the community forum. Then, on page load-first page (no post back) the system executes LoadPosts() to acquire and show all the existing community posts.

2. Navigating to "My Posts" Page

```
protected void btnMyPosts_Click(object sender, EventArgs e)
{
    Response.Redirect("my_post.aspx");
}
```

Figure 104: Navigating to "My Posts" Page

Explanation

By clicking the My Posts button, the user will be redirected to my_post.aspx, which is a specific page that will list all the posts made by that user, and this aids in ease of control of self-content.

3. Creating a New Community Post

```
protected void btnAsk_Click(object sender, EventArgs e)
{
    string content = txtQuestion.Text.Trim();

    if (string.IsNullOrEmpty(content))
    {
        lblMessage.Text = "Please enter your question.";
        return;
    }

    using (SqlConnection conn = new SqlConnection(connString))
    {
        string query = "INSERT INTO CommunityPost (UserId, PostContent," +
            " PostDateTime) VALUES (@UserId, @PostContent, GETDATE())";
        SqlCommand cmd = new SqlCommand(query, conn);
        cmd.Parameters.AddWithValue("@UserId", Session["UserId"]);
        cmd.Parameters.AddWithValue("@PostContent", content);

        conn.Open();
        cmd.ExecuteNonQuery();
    }

    txtQuestion.Text = "";
    LoadPosts();
}
```

Figure 105: Creating a New Community Post

Explanation

This operation enables one to post a new post, and this will be achieved when the system read the text typed in the input box and ensure that it is not a blank text then it inserts a new record in the database storing the user ID, the content of the post and the current date/time. Once the post is inserted, the post textbox is emptied and LoadPosts is run to refresh the community post list.

4. Loading Posts and Including Replies

```
private void LoadPosts()
{
    DataTable dtPosts = new DataTable();

    using (SqlConnection conn = new SqlConnection(connString))
    {
        string query = @"SELECT CP.Id, CP.UserId, U.FullName, U.Role,
                        CP.PostContent, CP.PostDateTime
                FROM CommunityPost CP
                JOIN Users U ON CP.UserId = U.Id
                ORDER BY CP.PostDateTime DESC";

        SqlDataAdapter da = new SqlDataAdapter(query, conn);
        da.Fill(dtPosts);
    }

    // Add Replies column
    if (!dtPosts.Columns.Contains("Replies"))
    {
        dtPosts.Columns.Add("Replies", typeof(object));
    }

    foreach (DataRow row in dtPosts.Rows)
    {
        int postId = Convert.ToInt32(row["Id"]);
        row["Replies"] = GetReplies(postId);
    }

    rptPosts.DataSource = dtPosts;
    rptPosts.DataBind();
}
```

Figure 106: Loading Posts and Including Replies

Explanation

Using this method, all community posts are retrieved in the database with the name and role of the creator, and the posts are sorted in order of latest. Furthermore, a special column is added to the DataTable named as Replies this is done by the system calling GetReplies to load all of the replies of a given post. Last but not least, data is attached to an ASP. NET repeater control (rptPosts) to render the UI.

5. Retrieving Replies for Each Post

```
private List<object> GetReplies(int postId)
{
    List<object> replies = new List<object>();

    using (SqlConnection conn = new SqlConnection(connString))
    {
        string query = @"SELECT R.ReplyContent, R.ReplyDateTime, U.FullName
                         FROM Reply R
                         JOIN Users U ON R.UserId = U.Id
                         WHERE R.PostId = @PostId
                         ORDER BY R.ReplyDateTime ASC";
        SqlCommand cmd = new SqlCommand(query, conn);
        cmd.Parameters.AddWithValue("@PostId", postId);

        conn.Open();
        SqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
            replies.Add(new
            {
                FullName = reader["FullName"].ToString(),
                ReplyContent = reader["ReplyContent"].ToString(),
                ReplyDateTime = Convert.ToDateTime(reader["ReplyDateTime"])
            });
        }
    }

    return replies;
}
```

Figure 107: Retrieving Replies for Each Post

Explanation

To retrieve all replies of each post, in this approach, the Reply table is joined with the Users table to have the name of the person to have replied. Therefore, the responses are sent back as a list of anonymous objects with the name of the user, reply to messages and time. This list is subsequently shown beneath each posting within the Repeater.

6. Adding a Reply + Deleting a Post

```
protected void rptPosts_ItemCommand(object source, RepeaterCommandEventArgs e)
{
    int postId = Convert.ToInt32(e.CommandArgument);

    if (e.CommandName == "AddReply")
    {
        TextBox txtReply = (TextBox)e.Item.FindControl("txtReply");
        string replyText = txtReply.Text.Trim();

        if (replyText != "")
        {
            using (SqlConnection conn = new SqlConnection(connString))
            {
                string query = "INSERT INTO Reply (PostId, UserId, ReplyContent, " +
                    "ReplyDateTime) VALUES (@PostId, @UserId, @ReplyContent, GETDATE())";
                SqlCommand cmd = new SqlCommand(query, conn);
                cmd.Parameters.AddWithValue("@PostId", postId);
                cmd.Parameters.AddWithValue("@UserId", Session["UserId"]);
                cmd.Parameters.AddWithValue("@ReplyContent", replyText);

                conn.Open();
                cmd.ExecuteNonQuery();
            }

            LoadPosts();
        }
    }
```

```
if (e.CommandName == "DeletePost")
{
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();

        string deleteReplies = "DELETE FROM Reply WHERE PostId = @PostId";
        using (SqlCommand cmdReplies = new SqlCommand(deleteReplies, conn))
        {
            cmdReplies.Parameters.AddWithValue("@PostId", postId);
            cmdReplies.ExecuteNonQuery();
        }
        string deletePost = "DELETE FROM CommunityPost WHERE Id = " +
            "@PostId AND UserId = @UserId";
        using (SqlCommand cmdPost = new SqlCommand(deletePost, conn))
        {
            cmdPost.Parameters.AddWithValue("@PostId", postId);
            cmdPost.Parameters.AddWithValue("@UserId", Session["UserId"]);
            cmdPost.ExecuteNonQuery();
        }
    }

    LoadPosts();
}
```

Figure 108: Adding a Reply + Deleting a Post

Explanation

The Repeater control process has two major responses handled via its command logic in adding replies and deleting posts. On adding a reply, the system would take the text that was typed by the user and insert it into the table Reply and then the list of posts would be refreshed. In post deletion, the system will first delete all the replies to the post then the actual post, so that no one but the owner can do such deletion. The post list is re-freshed after the operation. In this process, there is secure content management and adequate control of user-generated content.This occurrence makes sure that only the author of a post is able to see the Delete button.

7. Show Delete Button Only to Post Owner

```
protected void rptPosts_ItemDataBound(object sender, RepeaterItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.Item || e.Item.ItemType == ListItemType.AlternatingItem)
    {
        LinkButton btnDelete = (LinkButton)e.Item.FindControl("btnDelete");
        DataRowView drv = (DataRowView)e.Item.DataItem;
        int postUserId = Convert.ToInt32(drv["UserId"]);
        int currentUserId = Convert.ToInt32(Session["UserId"]);

        if (postUserId == currentUserId)
        {
            btnDelete.Visible = true;
        }
    }
}
```

Figure 109: Show Delete Button Only to Post Owner

Explanation

The system will compare the ID of the creator of the post with the one logged in at the moment (postUserId) with the one of the currently logged-in user (id). In case they are identical, the delete button is displayed. Otherwise it will be hidden to any other user.

8. SweetAlert2 Confirmation

```
btn.addEventListener('click', function (event) {
    event.preventDefault();

    const postId = btn.getAttribute('commandargument');
    const uniqueID = btn.getAttribute('data-uniqueid');

    Swal.fire({
        html: `
        <div class="logout-container">
            <h4 style="margin-bottom:15px; color:black;">Confirm Delete</h4>
            <p style="margin-bottom:25px; color:#333;">
                Are you sure you want to delete this post?<br>
            </p>
            <div style="display:flex; justify-content:center; gap:20px;">
                <button id="confirmDelete" class="btn-delete">Delete</button>
                <button id="cancelDelete" class="btn-cancel">Cancel</button>
            </div>
        </div>
    `,
        showConfirmButton: false,
        showCancelButton: false,
        background: 'transparent',
        allowOutsideClick: false,
        allowEscapeKey: false,
    });

    setTimeout(() => {
        document.getElementById('confirmDelete').addEventListener('click', function () {
            __doPostBack(uniqueID, '');
        });

        document.getElementById('cancelDelete').addEventListener('click', function () {
            Swal.close();
        });
    }, 100);
});
```

Figure 110: SweetAlert2 Confirmation

Explanation

SweetAlert2 is used to show a confirmation modal before a post is deleted. This eliminates unintentional deletion. In case the user checks, then the JavaScript initiates ASP.NET postback process to perform the deletion on the server.

4.3.2 Student **YAP BOON SIONG**

Student Dashboard

1. Page Security and Initialization

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        // Ensure valid student session
        if (Session["UserId"] == null || Session["Role"] == null || Session["Role"].ToString().ToLower() != "student")
        {
            Response.Redirect("sign_in.aspx");
            return;
        }

        // Ensure StudentID exists
        if (Session["StudentID"] == null)
        {
            LoadStudentID();
        }

        if (Session["StudentID"] == null)
        {
            lblStudentName.Text = "Unknown Student";
            lblCoins.Text = "0";
            lblBadges.Text = "0";
            lblCoursesCompleted.Text = "0";
            return;
        }

        lblStudentName.Text = Session["FullName"]?.ToString() ?? "Unnamed";
        LoadStudentStats();
        LoadCourses();
    }
}
```

Figure 111: Code-Page Security and Initialization

Explanation

This section ensures that the page is secure since it has the user authentication as a student and has to redirect a user who is not authenticated to sign_in.aspx. It only works in page loading first page (!IsPostBack). It makes a call to LoadStudentID() to retrieve the database ID of the student, and it then makes a call to LoadStudentStats() and LoadCourses() to load all the elements of the dashboard.

2. Retrieving Student Stats

```
private void LoadStudentStats()
{
    int studentId = Convert.ToInt32(Session["StudentID"]);

    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
        SqlCommand cmd = new SqlCommand("SELECT Coins, BadgesEarned FROM Student WHERE Id=@id", conn);
        cmd.Parameters.AddWithValue("@id", studentId);
        SqlDataReader dr = cmd.ExecuteReader();

        if (dr.Read())
        {
            lblCoins.Text = dr["Coins"].ToString();
            lblBadges.Text = dr["BadgesEarned"].ToString();
        }
        dr.Close();

        SqlCommand cmd2 = new SqlCommand("SELECT COUNT(*) FROM StudentCourseProgress WHERE StudentId=@id AND Status='Completed'", conn);
        cmd2.Parameters.AddWithValue("@id", studentId);
        lblCoursesCompleted.Text = cmd2.ExecuteScalar().ToString();
    }
}
```

Figure 112: Code-Retrieving Student Stats

Explanation

The LoadStudentStats() procedure performs two major database functions. It initially queries the Student table and gets the current Coins and BadgesEarned of the student. Second, it sequentially queries StudentCourseProgress table with a count of courses completed by the student (Status= ‘Completed’). All these metrics are subsequently presented on the labels on the dashboard.

3. Loading and Displaying Courses

```
private void LoadCourses()
{
    int studentId = Convert.ToInt32(Session["StudentID"]);

    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();

        // In Progress Courses
        SqlDataAdapter daIncomplete = new SqlDataAdapter(
            @"SELECT c.Id, c.Title, e.EducationQualification AS EducatorName
              FROM Course c
              JOIN Educator e ON c.EducatorId=e.Id
              JOIN StudentCourseProgress p ON p.CourseId=c.Id
              WHERE p.StudentId=@sid AND p.Status='In Progress'", conn);
        daIncomplete.SelectCommand.Parameters.AddWithValue("@sid", studentId);
        DataTable dtIncomplete = new DataTable();
        daIncomplete.Fill(dtIncomplete);
        rptIncompleteCourses.DataSource = dtIncomplete;
        rptIncompleteCourses.DataBind();

        // Public
        SqlDataAdapter daPublic = new SqlDataAdapter(
            @"SELECT c.Id, c.Title, e.EducationQualification AS EducatorName
              FROM Course c
              JOIN Educator e ON c.EducatorId=e.Id
              WHERE c.CourseType='Public'", conn);
        DataTable dtPublic = new DataTable();
        daPublic.Fill(dtPublic);
        rptPublicCourses.DataSource = dtPublic;
        rptPublicCourses.DataBind();

        //Private Courses
        SqlDataAdapter daPrivate = new SqlDataAdapter(
            @"SELECT c.Id, c.Title, e.EducationQualification AS EducatorName,
              ISNULL(c.Coin, 0) AS CoinReward
              FROM Course c
```

Figure 113: Code-Loading and Displaying Courses

Explanation

The LoadCourses() procedure loads and splits all courses into four different lists including In Progress, Public, Private and Completed. It also utilizes four distinct SqlDataAdapter queries, mostly of the form of joining the Course and Educator tables and StudentCourseProgress where specific student lists are needed. The output of both queries is put into a DataTable and bound to individual Repeater controls like, rptPublicCourses and rptIncompleteCourses to show.

4. Course Enrollment Logic

```
protected void rptPrivateCourses_ItemCommand(object source, System.Web.UI.WebControls.RepeaterCommandEventArgs e)
{
    if (e.CommandName == "SubscribeCourse")
    {
        int studentId = Convert.ToInt32(Session["StudentID"]);
        int courseId = Convert.ToInt32(e.CommandArgument);

        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();


            SqlCommand checkExist = new SqlCommand(
                "SELECT COUNT(*) FROM StudentCourseProgress WHERE StudentId=@sid AND CourseId=@cid", conn);
            checkExist.Parameters.AddWithValue("@sid", studentId);
            checkExist.Parameters.AddWithValue("@cid", courseId);

            int exists = Convert.ToInt32(checkExist.ExecuteScalar());
            if (exists > 0)
            {
                Response.Write("<script>alert('You have already joined this course!');</script>");
                return;
            }


            SqlCommand getCoins = new SqlCommand("SELECT Coins FROM Student WHERE Id=@id", conn);
            getCoins.Parameters.AddWithValue("@id", studentId);
            int coins = Convert.ToInt32(getCoins.ExecuteScalar());


            SqlCommand getCost = new SqlCommand("SELECT ISNULL(Coin, 0) FROM Course WHERE Id=@cid", conn);
            getCost.Parameters.AddWithValue("@cid", courseId);
            int cost = Convert.ToInt32(getCost.ExecuteScalar());


            if (coins >= cost)
            {
```

Figure 114: Code-Course Enrollment Logic (private course)

```
protected void rptPublicCourses_ItemCommand(object source, System.Web.UI.WebControls.RepeaterCommandEventArgs e)
{
    if (e.CommandName == "StartCourse")
    {
        int studentId = Convert.ToInt32(Session["StudentID"]);
        int courseId = Convert.ToInt32(e.CommandArgument);

        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();

            //Step 1: Check if already joined this course
            SqlCommand checkExist = new SqlCommand(
                "SELECT COUNT(*) FROM StudentCourseProgress WHERE StudentId=@sid AND CourseId=@cid", conn);
            checkExist.Parameters.AddWithValue("@sid", studentId);
            checkExist.Parameters.AddWithValue("@cid", courseId);

            int exists = Convert.ToInt32(checkExist.ExecuteScalar());
            if (exists > 0)
            {
                Response.Write("<script>alert('You have already joined this course!');</script>");
                return;
            }

            //Step 2: Insert new progress record (auto join)
            SqlCommand enroll = new SqlCommand(
                "INSERT INTO StudentCourseProgress (StudentId, CourseId, Status) VALUES (@sid, @cid, 'In Progress')", conn);
            enroll.Parameters.AddWithValue("@sid", studentId);
            enroll.Parameters.AddWithValue("@cid", courseId);
            enroll.ExecuteNonQuery();

            //Step 3: Redirect to content page
            Response.Write("<script>alert('Course started successfully!');window.location='StudentCourseContent.aspx?courseId=" + courseId + "';</script>");
        }
    }
```

Figure 115: Code-Course Enrollment Logic (public course)

Explanation

The ItemCommand handlers deal with the enrollment of courses. In the case of Private Courses, the balance of the student in terms of coins is compared to the course price. In case it is adequate, it does an UPDATE query to subtract the coins and an INSERT query to add a new record with the status of In Progress. In the case of Public Courses, which are free, the code does not check the coin, instead directly executing the INSERT query in order to enroll the student and redirecting the student to the StudentCourseContent.aspx page to begin to take the course.

Public Course

1. Page Security and Setup

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {

        if (Session["UserId"] == null || Session["Role"]?.ToString().ToLower() != "student")
        {
            Response.Redirect("sign_in.aspx");
            return;
        }

        string keyword = Request.QueryString["keyword"];
        if (!string.IsNullOrEmpty(keyword))
        {
            txtSearch.Text = keyword;
            LoadCourses(keyword);
        }
        else
        {
            LoadCourses();
        }
    }
}
```

Figure 116: Code-Page Secutiry and Setup

Explanation

The main role of the Page_Load() method is security, only authenticated students should be allowed to pass. It first loads on the platform of initial load (!IsPostBack) the query string of the URL where a search keyword is tested. When there is a keyword usually a keyword used in a search performed before it enters the text into the search box and calls LoadCourses(). If there is none, all public courses are loaded.

2. Loading and Filtering Courses

```
private void LoadCourses(string keyword = "")
{
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
        string query = @"
            SELECT c.Id, c.Title, e.EducationQualification AS EducatorName
            FROM Course c
            JOIN Educator e ON c.EducatorId = e.Id
            WHERE c.CourseType = 'Public'";

        if (!string.IsNullOrEmpty(keyword))
            query += " AND (c.Title LIKE @kw OR e.EducationQualification LIKE @kw)";

        SqlCommand cmd = new SqlCommand(query, conn);
        if (!string.IsNullOrEmpty(keyword))
            cmd.Parameters.AddWithValue("@kw", "%" + keyword + "%");

        SqlDataAdapter da = new SqlDataAdapter(cmd);
        DataTable dt = new DataTable();
        da.Fill(dt);

        rptPublicCourses.DataSource = dt;
        rptPublicCourses.DataBind();
    }
```

Figure 117: Code-Loading and Filtering Courses

Explanation

The LoadCourses() function is used to load all the courses that are considered Public. The SQL query is used to retrieve the course information and the name of the educator. It is dynamically adding a WHERE clause which allows the result to be filtered by the course title or educator name with a search keyword , “LIKE” provided, making it SQL injection safe with parameterized queries. The data which has been retrieved is then attached to the rptPublicCourses Repeater control in which it can be displayed. The click event of the btnSearch_Click is just a kind of reload that calls LoadCourses() along with the currently written text in the search box.

3. Free Course Enrollment

```
protected void rptPublicCourses_ItemCommand(object source, System.Web.UI.WebControls.RepeaterCommandEventArgs e)
{
    if (e.CommandName == "StartCourse")
    {
        int studentId = Convert.ToInt32(Session["StudentID"]);
        int courseId = Convert.ToInt32(e.CommandArgument);

        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();


            SqlCommand checkExist = new SqlCommand(
                "SELECT COUNT(*) FROM StudentCourseProgress WHERE StudentId=@sid AND CourseId=@cid", conn);
            checkExist.Parameters.AddWithValue("@sid", studentId);
            checkExist.Parameters.AddWithValue("@cid", courseId);

            int exists = Convert.ToInt32(checkExist.ExecuteScalar());
            if (exists > 0)
            {
                Response.Write("<script>alert('You have already joined this course!');</script>");
                return;
            }


            SqlCommand enroll = new SqlCommand(
                "INSERT INTO StudentCourseProgress (StudentId, CourseId, Status) VALUES (@sid, @cid, 'In Progress')", conn);
            enroll.Parameters.AddWithValue("@sid", studentId);
            enroll.Parameters.AddWithValue("@cid", courseId);
            enroll.ExecuteNonQuery();


            Response.Write("<script>alert('Course started successfully!');window.location='StudentCourseContent.aspx?courseId=" + courseId + "';</script>");
        }
    }
}
```

Figure 118: Code-Course Enrollment

Explanation

This process processes the Start Course command of a course that is open. It verifies the StudentCourseProgress table first to avoid duplication of courses. In case the student is not already enrolled it will run an INSERT query to insert a new progress record with a status of ‘In Progress’. Once successfully enrolled, it loads a client side JavaScript that will issue an alert and redirects the browser to StudentCourseContent.aspx which will take the specific courseId to be used by the next page to know what content to load.

Public Course

1. Page Security and Setup

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {

        if (Session["UserId"] == null || Session["Role"] == null || Session["Role"].ToString().ToLower() != "student")
        {
            Response.Redirect("sign_in.aspx");
            return;
        }

        if (Session["StudentID"] == null)
        {
            LoadStudentID();
        }

        LoadCourses();
    }
}
```

Figure 119: Code-Page Security and Setup

Explanation

The Page_Load() approach is such that the user is confirmed that he is a student before moving on. In case the user session does not contain the StudentID which is important in the transactions. The user will call the helper method of LoadStudentID() to retrieve it in the database using the UserId. Lastly, it makes an invocation to LoadCourses() to show the list of available private courses.

2. Student ID Retrieval

```
private void LoadStudentID()
{
    int userId = Convert.ToInt32(Session["UserId"]);

    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
        SqlCommand cmd = new SqlCommand("SELECT Id FROM Student WHERE UserId = @uid", conn);
        cmd.Parameters.AddWithValue("@uid", userId);
        object result = cmd.ExecuteScalar();

        if (result != null)
        {
            Session["StudentID"] = Convert.ToInt32(result);
            Session["StudentName"] = Session["FullName"];
        }
    }
}
```

Figure 120: Code-Student ID Retrieval

Explanation

LoadStudentID() method is crucial in determining who is the user in a student specific table. It uses the generic UserId provided when one generally logs in to find a matching Id in the Student table. This particular StudentId is then stored in the Session["StudentID] variable that is utilized in every course enrollment and progressing tracking query.

3. Loading and Filtering Course

```
private void LoadCourses(string keyword = "", string coinFilter = "")
{
    try
    {
        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();

            string query = @"
                SELECT c.Id, c.Title, e.EducationQualification AS EducatorName,
                       ISNULL(c.Coin, 50) AS Coin
                FROM Course c
                JOIN Educator e ON c.EducatorId = e.Id
                WHERE c.CourseType = 'Private'";

            if (!string.IsNullOrEmpty(keyword))
                query += " AND (c.Title LIKE @kw OR e.EducationQualification LIKE @kw)";

            if (!string.IsNullOrEmpty(coinFilter))
                query += " AND ISNULL(c.Coin, 50) <= @coinFilter";

            SqlCommand cmd = new SqlCommand(query, conn);

            if (!string.IsNullOrEmpty(keyword))
                cmd.Parameters.AddWithValue("@kw", "%" + keyword + "%");

            if (!string.IsNullOrEmpty(coinFilter))
                cmd.Parameters.AddWithValue("@coinFilter", Convert.ToInt32(coinFilter));

            SqlDataAdapter da = new SqlDataAdapter(cmd);
            DataTable dt = new DataTable();
            da.Fill(dt);

            rptPrivateCourses.DataSource = dt;
            rptPrivateCourses.DataBind();
```

Figure 121: Code-Loading and Filtering Course

Explanation

The LoadCourses() procedure loads all courses that have been indicated as being private. The query contains the coin cost of the course, which default to 50 in case the cost is not defended in the database is NULL. It provides two filtering options, one being a keyword search on Title or EducatorName and the other being a coinFilter letting users see courses that are of a certain lower cost than a specific price. All the filters are dynamically appended to the SQL query and processed safely with parameterized statements. Its results are attached to the rptPrivateCourses Repeater.

4. Course Subscription (Coin Transaction)

```
protected void rptPrivateCourses_ItemCommand(object source, System.Web.UI.WebControls.RepeaterCommandEventArgs e)
{
    if (e.CommandName == "SubscribeCourse")
    {
        int studentId = Convert.ToInt32(Session["StudentID"]);
        int courseId = Convert.ToInt32(e.CommandArgument);

        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();


            SqlCommand checkExist = new SqlCommand(
                "SELECT COUNT(*) FROM StudentCourseProgress WHERE StudentId=@sid AND CourseId=@cid", conn);
            checkExist.Parameters.AddWithValue("@sid", studentId);
            checkExist.Parameters.AddWithValue("@cid", courseId);

            int exists = Convert.ToInt32(checkExist.ExecuteScalar());
            if (exists > 0)
            {
                Response.Write("<script>alert('You have already joined this course!');</script>");
                return;
            }


            SqlCommand getCoins = new SqlCommand("SELECT Coins FROM Student WHERE Id=@id", conn);
            getCoins.Parameters.AddWithValue("@id", studentId);
            int coins = Convert.ToInt32(getCoins.ExecuteScalar());


            SqlCommand getCost = new SqlCommand("SELECT ISNULL(Coin, 50) FROM Course WHERE Id=@cid", conn);
            getCost.Parameters.AddWithValue("@cid", courseId);
            int cost = Convert.ToInt32(getCost.ExecuteScalar());
```

Figure 122: Code-Course Subscription (Coin Transaction)

Explanation

This is the approach of the command “Subscribe”. It initially carries out an enrollment check. Unless enrolled, it makes two queries to get the current coin balance of the student and the coin cost of the course. Assuming the student has the sufficient number of coins, it processes the transaction: it runs an UPDATE query to subtract the price in the balance of the student, and then an INSERT query to register the student with the status In Progress. In case of inadequacy in funds, an alert is displayed.

Course Content

1. Page Setup and Validation

```
private void LoadCourseInfo(int courseId)
{
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
        SqlCommand cmd = new SqlCommand(@"
            SELECT c.Title, c.CourseType, ISNULL(c.Coin, 0) AS Coin,
                   c.Status, e.EducationQualification AS EducatorName
            FROM Course c
            JOIN Educator e ON c.EducatorId = e.Id
            WHERE c.Id = @id", conn);
        cmd.Parameters.AddWithValue("@id", courseId);
        SqlDataReader dr = cmd.ExecuteReader();

        if (dr.Read())
        {
            lblCourseTitle.Text = dr["Title"].ToString();
            lblEducator.Text = dr["EducatorName"].ToString();
            lblCourseType.Text = dr["CourseType"].ToString();
            lblStatus.Text = dr["Status"].ToString();
            lblCoin.Text = dr["Coin"].ToString();
        }
        else
        {
            lblError.Text = "Course not found.";
            lblError.Visible = true;
        }
        dr.Close();
    }
}
```

Figure 123: Code-Page Setup and Validation

Explanation

Security is dealt with in the PageLoad method by making sure that the user is a student. It is capable of strictly validating the courseId in the URL. In the case of a valid ID, it will save it to the session and will execute LoadCourseInfo(), LoadLessons(), and LoadQuizzes() to load the page content the first time (!IsPostBack). Selecting rptQuizzesItemCommand handler will be done on each page load to perform the quiz button clicks.

2. Loading Course Details and Lessons

```
private void LoadCourseInfo(int courseId)
{
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
        SqlCommand cmd = new SqlCommand(@"
            SELECT c.Title, c.CourseType, ISNULL(c.Coin, 0) AS Coin,
                   c.Status, e.EducationQualification AS EducatorName
            FROM Course c
            JOIN Educator e ON c.EducatorId = e.Id
            WHERE c.Id = @id", conn);
        cmd.Parameters.AddWithValue("@id", courseId);
        SqlDataReader dr = cmd.ExecuteReader();

        if (dr.Read())
        {
            lblCourseTitle.Text = dr["Title"].ToString();
            lblEducator.Text = dr["EducatorName"].ToString();
            lblCourseType.Text = dr["CourseType"].ToString();
            lblStatus.Text = dr["Status"].ToString();
            lblCoin.Text = dr["Coin"].ToString();
        }
        else
        {
            lblError.Text = "Course not found.";
            lblError.Visible = true;
        }
        dr.Close();
    }
}
```

Figure 124: Code-Loading Course Details

```
private void LoadLessons(int courseId)
{
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
        SqlDataAdapter da = new SqlDataAdapter(
            @"SELECT Id, LessonNumber, LessonTitle, ContentType, ContentFilePath
              FROM Lesson WHERE CourseId=@cid ORDER BY LessonNumber ASC", conn);
        da.SelectCommand.Parameters.AddWithValue("@cid", courseId);

        DataTable dt = new DataTable();
        da.Fill(dt);
        rptLessons.DataSource = dt;
        rptLessons.DataBind();
    }
}
```

Figure 125: Code-Loading Course Lessons

Explanation

These two are the methods that fetch and present the static course material. LoadCourseInfo() loads basic information (Title, Educator, Coin cost) of a Course table with the help of a SqlDataReader. LoadLessons() takes the list of lessons and their paths to files out of Lesson table in the order they appear and binds them to the rptLessons control.

3. Quiz Progress and Locking Logic

```
private void LoadQuizzes(int courseId)
{
    if (Session["StudentID"] == null || !int.TryParse(Session["StudentID"].ToString(), out int studentId))
    {
        Response.Redirect("sign_in.aspx");
        return;
    }

    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
        SqlCommand cmd = new SqlCommand(@"
            SELECT q.Id, q.QuizRewardCoins, l.LessonTitle, l.LessonNumber,
                   ISNULL(p.Status, 'NotStarted') AS Status
            FROM Quiz q
            JOIN Lesson l ON q.LessonId = l.Id
            LEFT JOIN StudentQuizProgress p ON p.QuizId = q.Id AND p.StudentId = @sid
            WHERE l.CourseId = @cid
            ORDER BY l.LessonNumber ASC", conn);
        cmd.Parameters.AddWithValue("@cid", courseId);
        cmd.Parameters.AddWithValue("@sid", studentId);

        SqlDataAdapter da = new SqlDataAdapter(cmd);
        DataTable dt = new DataTable();
        da.Fill(dt);

        dt.Columns.Add("ButtonText", typeof(string));
        dt.Columns.Add("ButtonClass", typeof(string));
        dt.Columns.Add("ButtonEnabled", typeof(bool));

        bool previousQuizCompleted = true;
        int completedCount = 0;

        foreach (DataRow row in dt.Rows)
```

Figure 126: Code-Quiz Progress and Locking Logic

Explanation

The most complicated method is loadQuizzes(). It retrieves all the quizzes of the course, along with the status of the student of any quiz with the help of a LEFT JOIN to the StudentQuizProgress table.

4. Course Completion and Rewards

```
protected void rptQuizzes_ItemCommand(object source, RepeaterCommandEventArgs e)
{
    if (e.CommandName == "OpenQuiz")
    {
        string quizId = e.CommandArgument.ToString();
        Response.Redirect($"StudentQuiz.aspx?quizId={quizId}");
    }
}

//Complete Course Button Click
0 references
protected void btnCompleteCourse_Click(object sender, EventArgs e)
{
    // Ensure student session is valid
    if (Session["StudentID"] == null)
    {
        lblError.Text = "Session expired. Please log in again.";
        lblError.Visible = true;
        return;
    }

    int studentId = Convert.ToInt32(Session["StudentID"]);
    int courseId = Convert.ToInt32(Session["SelectedCourseId"]);

    // Check if course progress is 100%
    if (ProgressPercent < 100)
    {
        lblError.Text = "You must complete all quizzes before finishing the course.";
        lblError.Visible = true;
        return;
    }

    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
```

Figure 127: Code-Course Completion and Rewards

Explanation

All the rptQuizzes_ItemCommand does is to redirect the student to the quiz page. The btnCompleteCourse_Click event is used to verify that the student has attained a 100% progress. Then it will do a two-step transaction where it sets the status in StudentCourseProgress to either Completed and the title of the earned badge and it sets the Student table to increase the total count of BadgesEarned completed the course.

Lesson Course

1. Page Setup and Validation

```
protected void Page_Load(object sender, EventArgs e)
{

    if (Session["SelectedCourseId"] != null && int.TryParse(Session["SelectedCourseId"].ToString(), out courseId))
    {

    }
    else
    {
        courseId = 0;
    }

    if (!IsPostBack)
    {

        if (Session["UserId"] == null || Session["Role"]?.ToString().ToLower() != "student")
        {
            Response.Redirect("sign_in.aspx");
            return;
        }

        string idParam = Request.QueryString["lessonId"];
        if (string.IsNullOrEmpty(idParam) || !int.TryParse(idParam, out int lessonId))
        {
            lblError.Text = "Invalid or missing lesson ID.";
            lblError.Visible = true;
            return;
        }

        LoadLessonContent(lessonId);
    }
}
```

Figure 128: Code-Page Setup and Validation

Explanation

The Page_Load method in the first instance, tries to retrieve the courseId in the session that is required to navigate and locate the next lesson. Then it ensures security by authenticating the role of the user. On the first load (!IsPostBack), it does a validation of the lessonId sent in the URL. In the case that the ID is valid, then it calls LoadLessonContent() to load and display the lesson.

2. Loading and Displaying Content

```
private void LoadLessonContent(int lessonId)
{
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();


        string lessonQuery = @"
            SELECT LessonNumber, LessonTitle, ContentType, ContentFilePath, ContentFile, CourseId
            FROM Lesson WHERE Id = @lessonId";

        SqlCommand cmdLesson = new SqlCommand(lessonQuery, conn);
        cmdLesson.Parameters.AddWithValue("@lessonId", lessonId);
        SqlDataReader dr = cmdLesson.ExecuteReader();

        if (dr.Read())
        {
            // Update Literals and class fields
            currentLessonNumber = Convert.ToInt32(dr["LessonNumber"]);
            litLessonNumber.Text = currentLessonNumber.ToString();
            litLessonTitle.Text = dr["LessonTitle"].ToString();
            litContentType.Text = dr["ContentType"].ToString();

            if (courseId == 0)
            {
                if (dr["CourseId"] != DBNull.Value)
                {
                    courseId = Convert.ToInt32(dr["CourseId"]);
                    Session["SelectedCourseId"] = courseId;
                }
            }

            string contentType = dr["ContentType"].ToString();
            string filePath = ResolveUrl(dr["ContentFilePath"]?.ToString() ?? "");
            string fileText = dr["ContentFile"]?.ToString() ?? "";
```

Figure 129: Code-Loading and Displaying Content

Explanation

Data handler is the LoadLessonContent() method. It queries the Lesson table to get the metadata of the lesson, the path of the content file, type of content and the textual content. It inserts the labels and the currentLessonNumber private field. The method then renders the content via an iframe tag in case the content is a video, PDF or Document. It then lastly calls the CheckForNextLesson() to find the navigation state.

3. Next Lesson Navigation

```
private void CheckForNextLesson(SqlConnection conn)
{
    //Look for the next sequential lesson
    string nextLessonQuery = @"
        SELECT TOP 1 Id AS NextLessonId
        FROM Lesson
        WHERE CourseId = @cid AND LessonNumber > @currentNum
        ORDER BY LessonNumber ASC";

    SqlCommand cmdNext = new SqlCommand(nextLessonQuery, conn);
    cmdNext.Parameters.AddWithValue("@cid", courseId);
    cmdNext.Parameters.AddWithValue("@currentNum", currentLessonNumber);

    object nextLessonIdObj = cmdNext.ExecuteScalar();

    if (nextLessonIdObj != null)
    {
        // Next lesson found
        btnNextLesson.Visible = true;
        btnNextLesson.CommandArgument = nextLessonIdObj.ToString();
    }
    else
    {
        // No more lessons in this course
        btnNextLesson.Visible = false;
    }
}


0 references
protected void btnNextLesson_Click(object sender, EventArgs e)
{
    Button btn = (Button)sender;
    string nextLessonId = btn.CommandArgument;

    // Redirect to the next lesson
    if (!string.IsNullOrEmpty(nextLessonId))
    {
        Response.Redirect($"StudentLesson.aspx?lessonId={nextLessonId}");
    }
}
```

Figure 130: Code-Next Lesson Navigation

Explanation

CheckForNextLesson() method employs SQL query which is used to locate the next lesson in the series of lessons (LessonNumber>@currentNum) of the current course. It involves ExecuteScalar() to obtain only Id with a lot of efficiency. When an ID is returned, a

btnNextLesson will be displayed and set to redirect the user to the lesson mentioned when it is clicked, and it will allow progressing sequentially.

4. Navigation Back

```
protected void btnBackToCourse_Click(object sender, EventArgs e)
{
    // Redirect back to the course content page
    if (courseId > 0)
    {
        Response.Redirect($"StudentCourseContent.aspx?courseId={courseId}");
    }
    else
    {
        Response.Redirect("StudentDashboard.aspx");
    }
}
```

Figure 131: Code-Navigation Back

Explanation

The btnBackToCourse_Click method takes courseId obtained in Page_Load to redirect student to the main course content view StudentCourseContent.aspx to view the progress and to take quizzes. In case the courseId is not available, it will default to maximum redirect to the dashboard.

Attempt Quiz

1. Page Setup and Quiz Loading

```
protected void Page_Load(object sender, EventArgs e)
{

    rptQuestions.ItemDataBound += rptQuestions_ItemDataBound;


    if (Session["UserId"] == null || Session["Role"]?.ToString().ToLower() != "student")
    {
        Response.Redirect("sign_in.aspx");
        return;
    }

    if (!IsPostBack)
    {

        string quizParam = Request.QueryString["quizId"];
        if (string.IsNullOrEmpty(quizParam) || !int.TryParse(quizParam, out int quizId))
        {
            lblError.Text = "Invalid or missing quiz ID.";
            lblError.Visible = true;
            return;
        }

        Session["CurrentQuizId"] = quizId;
        LoadQuiz(quizId);
    }
}
```

Figure 132: Code-Page Setup

```
private void LoadQuiz(int quizId)
{
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();

        SqlCommand cmdQuiz = new SqlCommand(
            @"SELECT q.Id, q.LessonId, q.QuizRewardCoins, q.TotalQuestions, l.LessonTitle
              FROM Quiz q
              JOIN Lesson l ON q.LessonId = l.Id
              WHERE q.Id=@qid", conn);
        cmdQuiz.Parameters.AddWithValue("@qid", quizId);
        SqlDataReader dr = cmdQuiz.ExecuteReader();

        if (dr.Read())
        {
            lblQuizTitle.Text = "Quiz for Lesson: " + dr["LessonTitle"].ToString();
            ViewState["QuizRewardCoins"] = dr["QuizRewardCoins"] == DBNull.Value ? 0 : Convert.ToInt32(dr["QuizRewardCoins"]);
            ViewState["QuizPassingGrade"] = 70f;
        }
        else
        {
            lblError.Text = "Quiz not found.";
            lblError.Visible = true;
            return;
        }

        int totalQuestionsFromDB = dr["TotalQuestions"] == DBNull.Value ? 0 : Convert.ToInt32(dr["TotalQuestions"]);
        dr.Close();

        SqlDataAdapter da = new SqlDataAdapter(
            "SELECT Id, QuestionText, OptionA, OptionB, OptionC, OptionD, CorrectAnswer FROM Question WHERE QuizId=@qid", conn);
        da.SelectCommand.Parameters.AddWithValue("@qid", quizId);

        DataTable dt = new DataTable();
        da.Fill(dt);

        rptQuestions.DataSource = dt;
        rptQuestions.DataBind();
        ViewState["QuizData"] = dt;

        if (totalQuestionsFromDB == 0 && dt.Rows.Count > 0)
        {
            UpdateTotalQuestions(quizId, dt.Rows.Count, conn);
        }
    }
}
```

Figure 133: Code-Quiz Loading

Explanation

The PageLoad() technique helps in assuring that the user is a student and quizid is correct as per the URL. The LoadQuiz() procedure then pulls quiz details and all questions and options in the quiz in the database. It saves the entire data of the question in ViewState["QuizData] to grade it after submission and hand the coins of reward in the quiz and a default passing grade is 70%.

2. Dynamic Question Rendering

```
protected void rptQuestions_ItemDataBound(object sender, RepeaterItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.Item || e.Item.ItemType == ListItemType.AlternatingItem)
    {
        DataRowView row = (DataRowView)e.Item.DataItem;
        RadioButtonList rbl = (RadioButtonList)e.Item.FindControl("rblOptions");

        if (rbl != null)
        {
            rbl.Items.Clear();

            if (!string.IsNullOrEmpty(row["OptionA"].ToString()))
                rbl.Items.Add(new ListItem(row["OptionA"].ToString(), "A"));
            if (!string.IsNullOrEmpty(row["OptionB"].ToString()))
                rbl.Items.Add(new ListItem(row["OptionB"].ToString(), "B"));
            if (!string.IsNullOrEmpty(row["OptionC"].ToString()))
                rbl.Items.Add(new ListItem(row["OptionC"].ToString(), "C"));
            if (!string.IsNullOrEmpty(row["OptionD"].ToString()))
                rbl.Items.Add(new ListItem(row["OptionD"].ToString(), "D"));
        }
    }
}
```

Figure 134: Code-Dynamic Question Rendering

Explanation

The rptQuestions_ItemDataBound event is critical towards dynamic generation of UI. Since every question is attached to the Repeater, the embedded RadioButtonList (rblOptions) and adds the options (A, B, C, D) as Listitem objects programmatically, only options with some text will be shown.

3. Submission and Grading

```
protected void btnSubmit_Click(object sender, EventArgs e)
{
    if (Session["UserId"] == null)
    {
        lblError.Text = "You must be logged in.";
        lblError.Visible = true;
        return;
    }

    foreach (RepeaterItem item in rptQuestions.Items)
    {
        RadioButtonList rbl = (RadioButtonList)item.FindControl("rblOptions");
        if (rbl == null || string.IsNullOrEmpty(rbl.SelectedValue))
        {
            lblError.Text = "Please answer all questions before submitting the quiz.";
            lblError.Visible = true;
            return;
        }
    }
    lblError.Visible = false;

    int userId = Convert.ToInt32(Session["UserId"]);
    int studentIdForFK = GetStudentIdFromUserId(userId);

    if (studentIdForFK <= 0)
    {
        lblError.Text = "Error: Cannot find a valid Student record for this user. Foreign Key conflict is likely.";
        lblError.Visible = true;
        return;
    }

    int quizId = Convert.ToInt32(Session["CurrentQuizId"]);
    DataTable dt = ViewState["QuizData"] as DataTable;
    float passingGrade = 70f;

    if (dt == null)
    {
        lblError.Text = "Something went wrong loading quiz data.";
        lblError.Visible = true;
        return;
    }

    int totalQuestions = dt.Rows.Count;
    int correctCount = 0;

    // Grade answers
    for (int i = 0; i < rptQuestions.Items.Count; i++)
    {
        RepeaterItem item = rptQuestions.Items[i];
        RadioButtonList rbl = (RadioButtonList)item.FindControl("rblOptions");
        string correctAnswer = dt.Rows[i]["CorrectAnswer"].ToString().Trim();

        if (rbl.SelectedValue.Equals(correctAnswer, StringComparison.OrdinalIgnoreCase))
            correctCount++;
    }


    double percentageScore = (totalQuestions == 0) ? 0 : (correctCount / (double)totalQuestions) * 100;
    bool passed = percentageScore >= passingGrade;


    if (passed)
    {

        InsertPassingProgress(quizId, studentIdForFK);
    }

    Response.Redirect($"StudentQuizResult.aspx?quizId={quizId}&score={correctCount}&total={totalQuestions}&passGrade={passingGrade}");
}
```

Figure 135: Code-Submission and Grading

Explanation

The btnSubmit_Click procedure initially checks the selection of answers to all questions. It then applies the QuizData in the ViewState and the choices made in the RadioButtonList controls of the repeater with which to grade the quiz. In case the student scores 70% or above (passed = true), helper method, which is called as InsertPassingProgress(), is used to insert the quiz to the database as Completed. Lastly the user is redirected to the StudentQuizResult.aspx page, to which all score information is being provided in the URL query.

Quiz Result

1. Page Setup and Validation

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Session["UserId"] == null || Session["Role"]?.ToString().ToLower() != "student")
    {
        Response.Redirect("sign_in.aspx");
        return;
    }


    int userId = Convert.ToInt32(Session["UserId"]);
    this.studentTableId = GetStudentIdFromUserId(userId);

    if (this.studentTableId <= 0)
    {
        lblError.Text = "Security Error: Student record not found. Cannot process results.";
        lblError.Visible = true;
        return;
    }
    // --- END CRITICAL FIX ---

    if (!ValidateQueryParams())
    {
        if (pnlResult != null) pnlResult.Visible = false;
        lblError.Text = "Invalid or missing quiz result data.";
        lblError.Visible = true;
        return;
    }

    if (!IsPostBack)
    {
        DisplayResult();
    }
}
```

Figure 136: Code-Page Setup and Validation

Explanation

The PageLoad() method initially verifies that it is a student. It involves an important step, which is to map the general Session[“UserId”] to the particular Id of the Student table, which is stored in studentTableId. This ID is required in order to update transactions such as the awarding of coins. Lastly, it confirms that the appropriate result parameters (quizId, score, total and passingGrade) were all appropriately passed in the query string prior to call DisplayResult().

2. Displaying Results and Awarding Coins

```
private void DisplayResult()
{
    if (total == 0) return;

    double percentageScore = (score / (double)total) * 100;
    bool passed = percentageScore >= passingGrade;

    litScore.Text = $"{score}/{total}";
    litPercentage.Text = $"{Math.Round(percentageScore, 2)}%";

    int rewardCoins = 0;
    int courseId = 0;

    // 1. Get Quiz/Lesson details
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
        string query = @"
            SELECT q.QuizRewardCoins, l.CourseId, l.LessonTitle
            FROM Quiz q
            JOIN Lesson l ON q.LessonId = l.Id
            WHERE q.Id = @quizId";

        using (SqlCommand cmd = new SqlCommand(query, conn))
        {
            cmd.Parameters.AddWithValue("@quizId", quizId);
            SqlDataReader dr = cmd.ExecuteReader();

            if (dr.Read())
            {
                rewardCoins = dr["QuizRewardCoins"] == DBNull.Value ? 0 : Convert.ToInt32(dr["QuizRewardCoins"]);
                courseId = dr["CourseId"] == DBNull.Value ? 0 : Convert.ToInt32(dr["CourseId"]);
                lblQuizTitle.Text = $"Quiz Results for: {dr["LessonTitle"].ToString()}";
                Session["SelectedCourseId"] = courseId;
            }
            dr.Close();

            btnBackToCourse.CommandArgument = courseId.ToString();
        }
    }
```

Figure 137: Code-Displaying Results

```
// 2. Apply styling and reward logic
HtmlGenericControl statusSection = statusIcon.Parent is HtmlGenericControl ? (HtmlGenericControl)statusIcon.Parent : n

if (passed)
{
    if (statusSection != null) statusSection.Attributes["class"] += " passed";
    statusIcon.InnerHtml = "☑";
    lblStatus.Text = "Congratulations! You passed!";

    if (rewardCoins > 0)
    {

        AddCoinsToStudent(this.studentTableId, rewardCoins);
        litRewardMessage.Text = $"You earned **{rewardCoins}** Coins!";


        pnlReward.Visible = true;
    }
    else
    {
        litRewardMessage.Text = "No coin reward assigned for this quiz.";
        pnlReward.Visible = true;
    }

    btnRetry.Visible = false;

}
else // Failed
{
    if (statusSection != null) statusSection.Attributes["class"] += " failed";
    statusIcon.InnerHtml = "❌";
    lblStatus.Text = "Oops! You did not meet the passing grade.";
    litRewardMessage.Text = $"The passing grade is {passingGrade}%. You scored {Math.Round(percentageScore, 2)}%.";
    pnlReward.Visible = true;

    btnRetry.Visible = true;
    btnRetry.CommandArgument = quizId.ToString();
}
```

Figure 138: Code-Awarding Coins

Explanation

DisplayResult() method determines the score and the passing condition. It queries the database to get the title of the lesson, course ID and amount of coin reward. In case the student succeeded, the status would be set to successful, the student coins balance would be updated by calling the AddCoinsToStudent() method to modify the balance of Coins in the student of the database and the Retry button would be disabled. In the event of the failure of the student, the status is registered as a failure, no coins are earned, and the Retry button is shown and the settings are set to resume the user to the quiz.

3. Helper Functions and Navigation

```
private void AddCoinsToStudent(int studentId, int coins)
{
    if (coins <= 0) return;

    using (SqlConnection conn = new SqlConnection(connString))
    {

        using (SqlCommand cmd = new SqlCommand("UPDATE Student SET Coins = ISNULL(Coins,0) + @coins WHERE Id=@sid", conn))
        {
            conn.Open();


            cmd.Parameters.AddWithValue("@coins", coins);
            cmd.Parameters.AddWithValue("@sid", studentId);

            cmd.ExecuteNonQuery();
        }
    }
}

0 references
protected void btnBackToCourse_Click(object sender, EventArgs e)
{
    Button btn = (Button)sender;
    string courseId = btn.CommandArgument;
    Response.Redirect($"StudentCourseContent.aspx?courseId={courseId}");
}

0 references
protected void btnRetry_Click(object sender, EventArgs e)
{
    Button btn = (Button)sender;
    string quizId = btn.CommandArgument;
    Response.Redirect($"StudentQuiz.aspx?quizId={quizId}");
}
```

Figure 139: Code-Helper Functions and Navigation

Explanation

The AddCoinsToStudent() helper will run the SQL command which will be used to safely add coins to the student. The handlers of the button clicks are the navigation, btnBackToCourse_Click takes them to the main content page and the handlers of the btnRetry_Click bring them back to the quiz page to retake the quiz.

Leaderboard

1. Page Setup and Data Retrieval

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        if (Session["StudentID"] == null)
        {
            Response.Redirect("sign_in.aspx");
            return;
        }

        LoadLeaderboard();
    }
}

1 reference
private void LoadLeaderboard()
{
    int currentId = Convert.ToInt32(Session["StudentID"]);

    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();

        string query = @"
            SELECT s.Id AS StudentId, u.FullName, s.BadgesEarned,
                   (SELECT COUNT(*) FROM StudentCourseProgress p
                    WHERE p.StudentId = s.Id AND p.Status='Completed') AS CoursesCompleted,
                   s.School, s.InterestSubject
            FROM Student s
            JOIN Users u ON s.UserId = u.Id
            ORDER BY s.BadgesEarned DESC, CoursesCompleted DESC";

        SqlDataAdapter da = new SqlDataAdapter(query, conn);
        DataTable dt = new DataTable();
        da.Fill(dt);

        rptLeaderboard.DataSource = dt;
        rptLeaderboard.DataBind();

        //Top Stats Section
        lblTotalStudents.Text = dt.Rows.Count.ToString();

        for (int i = 0; i < dt.Rows.Count; i++)
        {
            if (Convert.ToInt32(dt.Rows[i]["StudentId"]) == currentId)
            {
                lblRank.Text = (i + 1).ToString();
                lblBadges.Text = dt.Rows[i]["BadgesEarned"].ToString();
                lblCourses.Text = dt.Rows[i]["CoursesCompleted"].ToString();
                break;
            }
        }
    }
```

Figure 140: Page Setup and Data Retrieval

Explanation

The PageLoad() method makes sure that the user is logged in and calls LoadLeaderboard() once. This procedure performs a complicated SQL query, which returns all the student information, combining the Student and the Users table. It determines the CoursesCompleted through a sub query. BadgesEarned (primary rank metric) is used to rank the results. This is then connected to the rptLeaderboard Repeater and another loop is formed to remove the rank and statistics of the currently logged in user which is then displayed in a special summary section.

2. Repeater Item Customization

```
protected void rptLeaderboard_ItemDataBound(object sender, RepeaterItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.Item || e.Item.ItemType == ListItemType.AlternatingItem)
    {
        DataRowView drv = (DataRowView)e.Item.DataItem;
        int rank = e.Item.ItemIndex + 1;
        int currentId = Convert.ToInt32(Session["StudentID"]);
        bool isCurrentUser = Convert.ToInt32(drv["StudentId"]) == currentId;

        Literal litYou = (Literal)e.Item.FindControl("litYou");
        Literal litBadgeLabel = (Literal)e.Item.FindControl("litBadgeLabel");
        Literal litSchool = (Literal)e.Item.FindControl("litSchool");
        Literal litSubject = (Literal)e.Item.FindControl("litSubject");
        Panel pnlRow = (Panel)e.Item.FindControl("pnlRow");

        // Highlight current user
        if (isCurrentUser)
        {
            pnlRow.CssClass += " highlighted";
            litYou.Text = " <span style='color:#001eff;'>(You)</span>";
        }

        // Top 3 badge tags
        if (rank == 1)
            litBadgeLabel.Text = " <span class='badge-label champion'>🏆 Champion</span>";
        else if (rank == 2)
            litBadgeLabel.Text = " <span class='badge-label runner'>🥈 Runner-up</span>";
        else if (rank == 3)
            litBadgeLabel.Text = " <span class='badge-label third'>🥉 Third Place</span>";

        // Student details
        litSchool.Text = string.IsNullOrEmpty(drv["School"].ToString()) ? "–" : drv["School"].ToString();
        litSubject.Text = string.IsNullOrEmpty(drv["InterestSubject"].ToString()) ? "–" : drv["InterestSubject"].ToString();
    }
}
```

Figure 141: Code-Repeater Item Customization

Explanation

The rptLeaderboard_ItemDataBound event personalizes the presentation of every row of students. It takes the rank of the item using item index and dynamically:

- HIghlight row of the student being logged in.
- Attaches special emoji labels to the 1st, 2nd, and 3rd-ranked students.
- Fills more fields such as School and Interest Subject.

3. UI Styling Helper

```
public string GetAvatarColor(int rank)
{
    switch (rank)
    {
        case 1: return "background:linear-gradient(135deg,#facc15,#fbbf24);";
        case 2: return "background:linear-gradient(135deg,#d1d5db,#9ca3af);";
        case 3: return "background:linear-gradient(135deg,#fcd34d,#fb923c);";
        default: return "background:linear-gradient(135deg,#001eff,#4f46e5);";
    }
}
```

Figure 142: Code-UI Styling Helper

Explanation

The GetAvatarColor() public helper method is probably calls directly on the.aspx page to add unique CSS styles gradients to the avatar or rank icon of the student in the top 3 spots to make the leaders of the board more visually distinctive.

Feedback

1. Page Initialization and Session Validation

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        //Ensure valid student session
        if (Session["UserId"] == null || Session["Role"] == null || Session["Role"].ToString().ToLower() != "student")
        {
            Response.Redirect("sign_in.aspx");
            return;
        }

        //Ensure StudentID exists
        if (Session["StudentID"] == null)
        {
            int userId = Convert.ToInt32(Session["UserId"]);
            using (SqlConnection conn = new SqlConnection(connStr))
            {
                conn.Open();
                SqlCommand cmd = new SqlCommand("SELECT Id FROM Student WHERE UserId = @uid", conn);
                cmd.Parameters.AddWithValue("@uid", userId);
                object result = cmd.ExecuteScalar();
                if (result != null)
                    Session["StudentID"] = Convert.ToInt32(result);
                else
                {
                    Response.Redirect("sign_in.aspx");
                    return;
                }
            }
        }

        LoadFeedbackHistory();
    }
}
```

Figure 143: Code-Page Initialization and Session Validation

Explanation

Page_Load() method is mostly focused on the security and integrity of the sessions. During the first load, the (!IsPostBack) does a verification to confirm that the user is a registered student. It does a significant check in case the Session["StudentID"]. The ID of the student table is not present, it will make a dynamic query to the database with the help of the Session["UserId”] so that it can fetch and establish the appropriate StudentID. After validation has been accomplished it makes a call to LoadFeedbackHistory().

2. Feedback Submission

```
protected void btnSubmit_Click(object sender, EventArgs e)
{
    if (string.IsNullOrEmpty(ddlCategory.SelectedValue) ||
        string.IsNullOrEmpty(ddlPriority.SelectedValue) ||
        string.IsNullOrEmpty(txtSubject.Text.Trim()) ||
        string.IsNullOrEmpty(txtDescription.Text.Trim()))
    {
        lblMessage.ForeColor = System.Drawing.Color.Red;
        lblMessage.Text = "Please fill in all fields before submitting.";
        return;
    }

    int studentId = Convert.ToInt32(Session["StudentID"]);

    using (SqlConnection conn = new SqlConnection(connStr))
    {
        conn.Open();
        SqlCommand cmd = new SqlCommand(@"
            INSERT INTO Feedback (StudentId, Category, Priority, Subject, Description, Status, CreatedAt)
            VALUES (@sid, @cat, @pri, @sub, @desc, 'Pending', GETDATE())", conn);

        cmd.Parameters.AddWithValue("@sid", studentId);
        cmd.Parameters.AddWithValue("@cat", ddlCategory.SelectedValue);
        cmd.Parameters.AddWithValue("@pri", ddlPriority.SelectedValue);
        cmd.Parameters.AddWithValue("@sub", txtSubject.Text.Trim());
        cmd.Parameters.AddWithValue("@desc", txtDescription.Text.Trim());
        cmd.ExecuteNonQuery();
    }

    lblMessage.ForeColor = System.Drawing.Color.Green;
    lblMessage.Text = "Feedback submitted successfully!";
    txtSubject.Text = "";
    txtDescription.Text = "";
    ddlCategory.SelectedIndex = 0;
    ddlPriority.SelectedIndex = 0;

    LoadFeedbackHistory();
}
```

Figure 144: Code-Feedback Submission

Explanation

The first step in the btnSubmit_Click is to ensure that all the necessary input fields have been validated. In case validation is successful, it inserts the new record in Feedback database table using the Session["StudentID]. The default status is set to ‘Pending’ and the time CreatedAt is recorded. Lastly, it replaces the user interface with a success message, disables the fields on the form, and replenishes the history on the display.

3. Displaying History

```
private void LoadFeedbackHistory()
{
    int studentId = Convert.ToInt32(Session["StudentID"]);

    using (SqlConnection conn = new SqlConnection(connStr))
    {
        conn.Open();
        SqlDataAdapter da = new SqlDataAdapter(@"
            SELECT Category, Priority, Subject, Description, Status, CreatedAt
            FROM Feedback
            WHERE StudentId=@id
            ORDER BY CreatedAt DESC", conn);
        da.SelectCommand.Parameters.AddWithValue("@id", studentId);
        DataTable dt = new DataTable();
        da.Fill(dt);

        rptFeedbackHistory.DataSource = dt;
        rptFeedbackHistory.DataBind();
    }
}
```

Figure 145: Code-Displaying History

Explanation

The LoadFeedbackHistory() procedure retrieves all the prior feedbacks of the present student (@id = studentId) in Feedback table. It ranks the results in order of CreatedAt such that the newest submissions are first. This data has been bound to the rptFeedbackHistory control that shows the history of submission of the student on the page.

Student Profile

1. Page Initialization and Data Loading

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {

        if (Session["UserId"] == null || Session["Role"] == null || Session["Role"].ToString().ToLower() != "student")
        {
            Response.Redirect("sign_in.aspx");
            return;
        }

        LoadProfile();
        LoadStatsAndBadges();
    }
}
```

Figure 146: Code-Page Initialization and Data Loading

Explanation

Page_Load() method is executed during the first request. It carries out two important tasks. First is strictly authenticate that the user is a student who is logged in. Second, it loads the page by calling LoadProfile() to get personal data and LoadStatsAndBadges() to get achievements, which is the initial time the page is loaded.

2. Profile Data Retrieval

```
3 references
private void LoadProfile()
{
    int userId = Convert.ToInt32(Session["UserId"]);

    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();
        string query = @"
            SELECT
                s.Id AS StudentId,
                u.FullName,
                u.Email,
                u.Age,
                u.Gender,
                u.ProfilePicture,
                s.School,
                s.InterestSubject,
                s.Coins,
                s.BadgesEarned
            FROM Users u
            JOIN Student s ON u.Id = s.UserId
            WHERE u.Id = @uid";

        SqlCommand cmd = new SqlCommand(query, conn);
        cmd.Parameters.AddWithValue("@uid", userId);
        SqlDataReader dr = cmd.ExecuteReader();

        if (dr.Read())
        {

            lblUserName.Text = dr["FullName"].ToString();
            lblStudentId.Text = dr["StudentId"].ToString();
            txtFullName.Text = dr["FullName"].ToString();
            txtEmail.Text = dr["Email"].ToString();
            txtSchool.Text = dr["School"].ToString();
            txtInterestSubject.Text = dr["InterestSubject"].ToString();
            txtAge.Text = dr["Age"].ToString();

            string gender = dr["Gender"].ToString();
            if (!string.IsNullOrEmpty(gender) && ddlGender.Items.FindByValue(gender) != null)
                ddlGender.SelectedValue = gender;

            lblCoins.Text = dr["Coins"].ToString();
            lblBadges.Text = dr["BadgesEarned"].ToString();


            string profilePic = dr["ProfilePicture"].ToString();
            imgProfile.ImageUrl = string.IsNullOrEmpty(profilePic)
                ? ResolveUrl("~/Image/default_profile2.png")
                : ResolveUrl("~/Image/" + profilePic);


            Session["StudentID"] = dr["StudentId"].ToString();
            Session["FullName"] = dr["FullName"].ToString();
            Session["ProfilePicture"] = dr["ProfilePicture"].ToString();
        }
        dr.Close();
    }
}
```

Figure 147: Code-Profile Data Retrieval

Explanation

LoadProfile() method loads all the profile data by uniting the Students and the Users tables based on the current UserId. It fills in all display controls such as the setting of the imgProfile.ImageUrl with a default fallback. It also updates the vital session variables (StudentID, FullName, ProfilePicture) to the most recent data.

3. Stats and Badges Retrieval

```
private void LoadStatsAndBadges()
{
    int studentId = 0;
    int userId = Convert.ToInt32(Session["UserId"]);

    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();

        SqlCommand getSid = new SqlCommand("SELECT Id FROM Student WHERE UserId = @uid", conn);
        getSid.Parameters.AddWithValue("@uid", userId);
        object sidObj = getSid.ExecuteScalar();
        if (sidObj != null)
            studentId = Convert.ToInt32(sidObj);

        SqlDataAdapter da = new SqlDataAdapter(@"
            SELECT p.BadgeEarned, c.Title AS CourseTitle
            FROM StudentCourseProgress p
            JOIN Course c ON p.CourseId = c.Id
            WHERE p.StudentId = @sid AND p.BadgeEarned IS NOT NULL", conn);

        da.SelectCommand.Parameters.AddWithValue("@sid", studentId);
        DataTable dt = new DataTable();
        da.Fill(dt);

        rptBadges.DataSource = dt;
        rptBadges.DataBind();
    }
}
```

Figure 148: Code-Stats and Badges Retrieval

Explanation

This is a first step of making sure it has Student.Id. It then queries StudentCourseProgress table, joined with Course, to get all records that had badge earned is ‘not null’ on a BadgeEarned. The list of earned badges and the titles of the courses associated with these badges are attached to the rptBadges repeater.

4. Profile Update

```
protected void btnSaveProfile_Click(object sender, EventArgs e)
{
    int userId = Convert.ToInt32(Session["UserId"]);
    string fullName = txtFullName.Text.Trim();
    string school = txtSchool.Text.Trim();
    string subject = txtInterestSubject.Text.Trim();
    string gender = ddlGender.SelectedValue;
    int.TryParse(txtAge.Text.Trim(), out int age);

    string fileName = null;
    if (fileUploadProfile.HasFile)
    {

        string extension = Path.GetExtension(fileUploadProfile.FileName);
        fileName = "profile_" + userId + "_" + DateTime.Now.Ticks + extension;

        string folderPath = Server.MapPath("~/Image/");
        if (!Directory.Exists(folderPath))
            Directory.CreateDirectory(folderPath);

        fileUploadProfile.SaveAs(Path.Combine(folderPath, fileName));
    }

    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();


        string queryUser = @"
            UPDATE Users
            SET FullName = @name, Age = @age, Gender = @gender
                " + (fileName != null ? ", ProfilePicture = @pic" : "") + @"
            WHERE Id = @uid";

        SqlCommand cmdUser = new SqlCommand(queryUser, conn);
        cmdUser.Parameters.AddWithValue("@name", fullName);
        cmdUser.Parameters.AddWithValue("@age", age);
        cmdUser.Parameters.AddWithValue("@gender", gender);
        cmdUser.Parameters.AddWithValue("@uid", userId);
        if (fileName != null)
            cmdUser.Parameters.AddWithValue("@pic", fileName);
        cmdUser.ExecuteNonQuery();


        string queryStudent = @"
            UPDATE Student
            SET School = @school, InterestSubject = @subject
            WHERE UserId = @uid";

        SqlCommand cmdStudent = new SqlCommand(queryStudent, conn);
        cmdStudent.Parameters.AddWithValue("@school", school);
        cmdStudent.Parameters.AddWithValue("@subject", subject);
        cmdStudent.Parameters.AddWithValue("@uid", userId);
        cmdStudent.ExecuteNonQuery();


        Session["FullName"] = fullName;
        if (fileName != null)
            Session["ProfilePicture"] = fileName;

        lblMessage.Text = "Profile updated successfully!";
        LoadProfile();
    }
}
```

Figure 149: Code-Profile Update

Explanation

This handler deals with the profile updating process. It does the file upload which generates a new name and stores the image and does two different SQL UPDATE query. One on the Users table and one on the Student table. It also updates the session and reloads the profile data in order to display the changes instantly.

5. UI Managment

```
protected void btnEdit_Click(object sender, EventArgs e)
{
    txtFullName.ReadOnly = false;
    txtAge.ReadOnly = false;
    txtSchool.ReadOnly = false;
    txtInterestSubject.ReadOnly = false;
    ddlGender.Enabled = true;
    fileUploadProfile.Visible = true;

    btnEdit.Visible = false;
    btnSaveProfile.Visible = true;
    btnCancel.Visible = true;
    lblMessage.Text = "";
}

0 references
protected void btnCancel_Click(object sender, EventArgs e)
{
    txtFullName.ReadOnly = true;
    txtAge.ReadOnly = true;
    txtSchool.ReadOnly = true;
    txtInterestSubject.ReadOnly = true;
    ddlGender.Enabled = false;
    fileUploadProfile.Visible = false;

    btnEdit.Visible = true;
    btnSaveProfile.Visible = false;
    btnCancel.Visible = false;

    LoadProfile();
    lblMessage.Text = "Changes canceled.";
    lblMessage.ForeColor = System.Drawing.Color.Gray;
}
```

Figure 150: Code-UI Managment

Explanation

The btnEdit_Click and btnCancel_Click functions govern the form option between read-only and edit mode, where btnCancel_Click is essential in that it calls the LoadProfile() that restores the form fields to the previous database values that were saved last.

4.3.3 Educator **CHEN XIN ZE**

Create Course:

1. Page Initialization (Page_Load):

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        // Clears old lesson and question data
        lessonList.Clear();
        tempQuestionList.Clear();

        // Displays lesson list in the table (GridView)
        BindLessonGrid();

        // Updates the total number of lessons label
        UpdateLessonCountLabel();

        // Refreshes the dropdown list for selecting quiz lessons
        BindQuizLessonDropdown();

        // Displays the temporary question list in the table
        BindTempQuestionGrid();
    }
}
```

Figure 151: Page Initialization

The system clears all the temporary lesson and quiz data whenever the page loads first time to prevent any remaining values of the course creation before. It then binds the lesson list, sets the number of lessons, loads any lesson titles that are available into the quiz dropdown and shows any quiz questions in the quiz that are temporary. This makes the form clean at all times and prevents unwanted repetition of lessons or quiz questions.

2. Course Type Selection (Show Coin Field):

```
// Show or hide the coin input based on course type
coinDiv.Visible = ddlCourseType.SelectedValue == "Private";
```

Figure 152: Course Type Selection

When the educator switches the course type, the system will automatically reveal or conceal the input field of the “Coin”. This makes the form simpler because only private courses will need a coin value and other public courses will not be required to have unnecessary inputs.

3. Adding a Lesson with Validation:

```
if (fuContentFile.HasFile && fuContentFile.PostedFile.ContentLength > 52428800) // If greater than 50MB
{
    lblStatus.ForeColor = System.Drawing.Color.Red; // Display red error message
    lblStatus.Text = "Lesson File: Size exceeds 50MB limit.";
    return; // Stop execution
}
string lessonTitle = txtLessonTitle.Text.Trim(); // Get lesson title and ensure it's not empty
if (string.IsNullOrEmpty(lessonTitle))
{
    lblStatus.ForeColor = System.Drawing.Color.Red;
    lblStatus.Text = "Please enter a lesson title.";
    return;
}
```

Figure 153: Adding a Lesson with Validation

The system checks the input of the educator before inserting a lesson. It verifies that the uploaded lesson file is not larger than 50MB and that it includes a lesson title. These validations eliminate incompleteness in lessons and server overworking due to files of unnecessary size.

4. Binding Lesson List to the "GridView":

```
gvLessons.DataSource = lessonList; // Use lessonList as the data source
gvLessons.DataBind();
```

Figure 154: Binding Lesson List to the "GridView"

Lessons which are added by the educator are stored temporarily in "lessonList". All lessons would be shown immediately, without any information being stored on the database. This will enable the educator to check and revise lessons before submission.

5. Editing an Existing Lesson:

```
lessonToUpdate.LessonTitle = txtEditTitle.Text.Trim();
lessonToUpdate.ContentFile = txtEditContent.Text.Trim();
```

Figure 155: Editing an Existing Lesson

The system retrieves the lesson object of the temporary list and updates the title and content of the lesson when the educator is editing the lesson. This will enable the easy adjustments of lessons until they ultimately get into the database when creating a course.

6. Quiz Status Display in "GridView":

```
// If the lesson has a quiz and at least one question
if (lesson.LessonQuiz != null && lesson.LessonQuiz.Questions.Count > 0)
{
    // Show "Yes" with question count
    lblQuizStatus.Text = $"Yes ({lesson.LessonQuiz.Questions.Count} Qs)";
    lblQuizStatus.ForeColor = System.Drawing.Color.Green;
}
else
{
    // Otherwise, show "No"
    lblQuizStatus.Text = "No";
    lblQuizStatus.ForeColor = System.Drawing.Color.Gray;
}
```

Figure 156:Quiz Status Display in "GridView"

This code snippet examines the presence of a lesson quiz and shows the status of the quiz in the list of the lessons. It enables educator to easily identify lessons with existing quizzes and the number of questions in the quizzes.

7. Editing or Deleting a Quiz:

```
if (e.CommandName == "EditQuiz")
{
    // Load quiz data into the quiz form
    ddlQuizLesson.SelectedValue = lesson.LessonNumber.ToString();
    txtQuizRewardCoins.Text = lesson.LessonQuiz?.QuizRewardCoins?.ToString() ?? "";

    // Copy the existing quiz questions into the temporary list for editing
    tempQuestionList = (lesson.LessonQuiz != null) ? new List<Question>(lesson.LessonQuiz.Questions) : new List<Question>();
    BindTempQuestionGrid(); // Show questions in the grid

    // Mark that we are editing this lesson's quiz
    ViewState["QuizEditLessonNumber"] = lessonNumber;

    // Update the UI to reflect edit mode
    btnSaveQuiz.Text = "Update Quiz";
    ddlQuizLesson.Enabled = false; // Don't allow changing lesson while editing quiz
    lblStatus.Text = $"Editing quiz for: {lesson.LessonTitle}";
    lblStatus.ForeColor = System.Drawing.Color.Blue;
}
```

Figure 157: Editing or Deleting a Quiz

Every time the educator decides to edit a quiz, the system will open up the quiz questions into a temporary list and will show them in the quiz editor. This enables them to edit quiz completely without compromising other lessons until the educator saves the changes.

8. Adding a Quiz Question with Validation:

```
// Validate required fields before adding the question
if (string.IsNullOrWhiteSpace(txtQuestionText.Text) ||
    string.IsNullOrWhiteSpace(txtOptionA.Text) ||
    string.IsNullOrWhiteSpace(txtOptionB.Text) ||
    ddlCorrectAnswer.SelectedValue == "")
{
    // Show error if any required input is missing
    lblQuizStatus.ForeColor = System.Drawing.Color.Red;
    lblQuizStatus.Text = "Please fill in Question Text, Option A, Option B, and select a Correct Answer.";
    return;
}
```

Figure 158: Adding a Quiz Question with Validation

The system checks the text of the question, options A and B as well as the correct answer before it is added to the quiz. This minimizes the mistakes and makes sure that all the questions are filled in before they are included in the quiz list.

9. Saving the Quiz to a Lesson:

```
// Attach quiz to the lesson
lessonToUpdate.LessonQuiz = newQuiz;

// Clear form and temp list
ClearQuestionForm();
tempQuestionList.Clear();
BindTempQuestionGrid();
txtQuizRewardCoins.Text = "";
ddlQuizLesson.SelectedIndex = 0;

// Reset edit mode if we were in it
if (ViewState["QuizEditLessonNumber"] != null)
{
    ViewState["QuizEditLessonNumber"] = null;
    btnSaveQuiz.Text = "Save Quiz to Lesson";
    ddlQuizLesson.Enabled = true;
    lblStatus.Text = "Quiz updated successfully!";
    lblStatus.ForeColor = System.Drawing.Color.Green;
}
else
{
    lblStatus.Text = "Quiz saved to lesson successfully!";
    lblStatus.ForeColor = System.Drawing.Color.Green;
}

lblQuizStatus.Text = "";
BindLessonGrid(); // Rebind lesson grid to update quiz status
```

Figure 159: Saving the Quiz to a Lesson

When the educator completes the quiz, the system links the quiz object (including all the questions in it) to the lesson chosen. It then changes the list of lessons to indicate the new status of the quiz and removes temporary editing information. This ensures that there is clean flow of data and eliminates duplication.

10. Validating Course Creation Before Saving:

```
// Ensure course title, type, and at least one lesson are provided
if (string.IsNullOrWhiteSpace(txtCourseTitle.Text) || ddlCourseType.SelectedValue == "" || lessonList.Count == 0)
{
    lblStatus.ForeColor = System.Drawing.Color.Red;
    lblStatus.Text = "Please provide a Course Title, Type, and add at least one Lesson.";
    return;
}
```

Figure 160: Validating Course Creation Before Saving

The system checks that a course title, type and at least one lesson have been entered before the entire course is saved to the database. This will eliminate the possibility of having incomplete courses being stored and will also guarantee that all the created courses meet the minimum requirement of the structure.

11. Saving Course, Lessons, and Quizzes in One Transaction:

```
transaction = con.BeginTransaction(); // Start a transaction
```

Figure 161: Saving Course, Lessons, and Quizzes in One Transaction

```
using (SqlCommand cmdCourse = new SqlCommand(insertCourseQuery, con, transaction))
{
    cmdCourse.Parameters.AddWithValue("@Title", txtCourseTitle.Text.Trim());
    cmdCourse.Parameters.AddWithValue("@EducatorId", educatorId);
    cmdCourse.Parameters.AddWithValue("@CourseType", ddlCourseType.SelectedValue);
    cmdCourse.Parameters.AddWithValue("@CoursePicture", DBNull.Value); // Placeholder
    cmdCourse.Parameters.AddWithValue("@Status", "Active");
    cmdCourse.Parameters.AddWithValue("@Coin", coin);
    courseId = (int)cmdCourse.ExecuteScalar(); // Get the newly created Course ID
}
```

Figure 162: Saving Course, Lessons, and Quizzes in One Transaction

```
using (SqlCommand cmdLesson = new SqlCommand(insertLessonQuery, con, transaction))
{
    cmdLesson.Parameters.AddWithValue("@CourseId", courseId);
    cmdLesson.Parameters.AddWithValue("@LessonNumber", lesson.LessonNumber);
    cmdLesson.Parameters.AddWithValue("@LessonTitle", lesson.LessonTitle);
    cmdLesson.Parameters.AddWithValue("@ContentType", (object)lesson.ContentType ?? DBNull.Value);
    cmdLesson.Parameters.AddWithValue("@ContentFilePath", (object)lesson.ContentFilePath ?? DBNull.Value);
    cmdLesson.Parameters.AddWithValue("@ContentFile", (object)lesson.ContentFile ?? DBNull.Value);
    lessonId = (int)cmdLesson.ExecuteScalar(); // Get the new Lesson ID
}
```

Figure 163: Saving Course, Lessons, and Quizzes in One Transaction

```
using (SqlCommand cmdQuiz = new SqlCommand(insertQuizQuery, con, transaction))
{
    cmdQuiz.Parameters.AddWithValue("@LessonId", lessonId);
    cmdQuiz.Parameters.AddWithValue("@QuizRewardCoins", (object)lesson.LessonQuiz.QuizRewardCoins ?? DBNull.Value);
    cmdQuiz.Parameters.AddWithValue("@TotalQuestions", lesson.LessonQuiz.Questions.Count);
    quizId = (int)cmdQuiz.ExecuteScalar(); // Get the new Quiz ID
}
```

Figure 164: Saving Course, Lessons, and Quizzes in One Transaction

```
using (SqlCommand cmdQuestion = new SqlCommand(insertQuestionQuery, con, transaction))
{
    cmdQuestion.Parameters.AddWithValue("@QuizId", quizId);
    cmdQuestion.Parameters.AddWithValue("@QuestionText", question.QuestionText);
    cmdQuestion.Parameters.AddWithValue("@OptionA", (object)question.OptionA ?? DBNull.Value);
    cmdQuestion.Parameters.AddWithValue("@OptionB", (object)question.OptionB ?? DBNull.Value);
    cmdQuestion.Parameters.AddWithValue("@OptionC", (object)question.OptionC ?? DBNull.Value);
    cmdQuestion.Parameters.AddWithValue("@OptionD", (object)question.OptionD ?? DBNull.Value);
    cmdQuestion.Parameters.AddWithValue("@CorrectAnswer", question.CorrectAnswer);
    cmdQuestion.ExecuteNonQuery();
}
```

Figure 165: Saving Course, Lessons, and Quizzes in One Transaction

```
// Commit all database operations if no errors occur
transaction.Commit();
```

Figure 166: Saving Course, Lessons, and Quizzes in One Transaction

The whole course creation including course details, lessons, quizzes, quiz questions, are all performed within one SQL transaction. This is in case any section of the saving process fails; the entire process is reversed. It assures the integrity of the database and it avoids half-baked or wrong format course entries.

12. Transaction Rollback on Error:

```
// If an error occurs, roll back all changes
try
{
    if (transaction != null)
        transaction.Rollback();
}
```

Figure 167: Transaction Rollback on Error

In case of any unforeseen error in course creation, the system undoes all the operations that have been done to the database so far. This eliminates the possibility of incomplete records being stored and makes the database consistent.

Edit Course:

Edit Course has same logic as the Create Course, but has some difference as mentioned below.

1. Loading Existing Course Data:

```
if (Request.QueryString["id"] != null && int.TryParse(Request.QueryString["id"], out int courseId))
{
    // Store the CourseId to use it in the Update button click
    ViewState["CourseId"] = courseId;

    // Load all existing data from the DB into the static lists
    LoadCourseData(courseId);
}
```

Figure 168: Loading Existing Course Data

The Edit Course page unlike the Create Course page will need to load a course that exists based on the course ID that is passed along the URL. The system authenticates the ID and it is stored in "ViewState" which then calls "LoadCourseData" to retrieve all course information, lessons, quizzes and questions stored in the database. This enables the educators to revise their courses that they have created.

2. Verifying Educator Ownership Before Editing:

```
// 2. Load Course Details and verify ownership
string courseQuery = "SELECT Title, CourseType, Coin FROM Course WHERE Id = @CourseId AND EducatorId = @EducatorId";
```

Figure 169: Verifying Educator Ownership Before Editing

To promote security, the system verifies that the present educator who has been logged in is the owner of the course. In case the course is not part of the user, it is not allowed to edit. This ensures that unauthorized users will not alter another educator course.

3. Loading Lessons from the Database:

```
// 3. Load Lessons
string lessonQuery = "SELECT Id, LessonNumber, LessonTitle, ContentType, ContentFilePath, ContentFile " +
                     "FROM Lesson WHERE CourseId = @CourseId ORDER BY LessonNumber";
```

Figure 170: Loading Lessons from the Database

In contrast to Create Course where the lessons are generated afresh in memory, Edit Course retrieves the existing lessons in the database and saves them into "lessonList". This enables the educator to update, rename or delete existing lessons.

4. Loading Quizzes and Questions:

```
// 4. Load Quizzes and Questions
string quizQuery = @"
    SELECT
        q.Id AS QuizId, q.LessonId, q.QuizRewardCoins,
        qn.Id AS QuestionId, qn.QuestionText,
        qn.OptionA, qn.OptionB, qn.OptionC, qn.OptionD, qn.CorrectAnswer
    FROM Quiz q
    LEFT JOIN Question qn ON q.Id = qn.QuizId
    WHERE q.LessonId IN (SELECT Id FROM Lesson WHERE CourseId = @CourseId)
    ORDER BY q.LessonId, qn.Id"; // Order by the Question's primary key
```

Figure 171: Loading Quizzes and Questions

To edit, the system needs to re-build every quiz of a lesson, its reward settings, and questions. This query loads all and puts quizzes into memory as they should be in the correct lesson so that they can be fully edited as though the educator had just created them.

5. Updating the Course (Delete & Re-Insert Strategy):

```
// Delete StudentQuizProgress first
string deleteProgressQuery = @"
    DELETE sqp FROM StudentQuizProgress sqp
    INNER JOIN Quiz q ON sqp.QuizId = q.Id
    INNER JOIN Lesson l ON q.LessonId = l.Id
    WHERE l.CourseId = @CourseId";
using (SqlCommand delProgCmd = new SqlCommand(deleteProgressQuery, con, transaction))
{
    delProgCmd.Parameters.AddWithValue("@CourseId", courseId);
    delProgCmd.ExecuteNonQuery();
}

// Delete Questions
string deleteQuestionsQuery = @"
    DELETE qn FROM Question qn
    INNER JOIN Quiz q ON qn.QuizId = q.Id
    INNER JOIN Lesson l ON q.LessonId = l.Id
    WHERE l.CourseId = @CourseId";
using (SqlCommand delQnCmd = new SqlCommand(deleteQuestionsQuery, con, transaction))
{
    delQnCmd.Parameters.AddWithValue("@CourseId", courseId);
    delQnCmd.ExecuteNonQuery();
}

// Delete Quizzes
string deleteQuizQuery = @"
    DELETE q FROM Quiz q
    INNER JOIN Lesson l ON q.LessonId = l.Id
    WHERE l.CourseId = @CourseId";
using (SqlCommand delQuizCmd = new SqlCommand(deleteQuizQuery, con, transaction))
{
    delQuizCmd.Parameters.AddWithValue("@CourseId", courseId);
    delQuizCmd.ExecuteNonQuery();
}

// Delete Lessons
string deleteLessonQuery = "DELETE FROM Lesson WHERE CourseId = @CourseId";
using (SqlCommand delLessonCmd = new SqlCommand(deleteLessonQuery, con, transaction))
{
    delLessonCmd.Parameters.AddWithValue("@CourseId", courseId);
    delLessonCmd.ExecuteNonQuery();
}
```

Figure 172: Updating the Course (Delete & Re-Insert Strategy)

```
// 3a. Insert Lesson
string insertLessonQuery = @"
    INSERT INTO Lesson (CourseId, LessonNumber, LessonTitle, ContentType, ContentFilePath, ContentFile)
    OUTPUT INSERTED.Id
    VALUES (@CourseId, @LessonNumber, @LessonTitle, @ContentType, @ContentFilePath, @ContentFile)";

int lessonId;
using (SqlCommand cmdLesson = new SqlCommand(insertLessonQuery, con, transaction))
{
    cmdLesson.Parameters.AddWithValue("@CourseId", courseId); // Use existing CourseId
    cmdLesson.Parameters.AddWithValue("@LessonNumber", lesson.LessonNumber);
    cmdLesson.Parameters.AddWithValue("@LessonTitle", lesson.LessonTitle);
    cmdLesson.Parameters.AddWithValue("@ContentType", (object)lesson.ContentType ?? DBNull.Value);
    cmdLesson.Parameters.AddWithValue("@ContentFilePath", (object)lesson.ContentFilePath ?? DBNull.Value);
    cmdLesson.Parameters.AddWithValue("@ContentFile", (object)lesson.ContentFile ?? DBNull.Value);
    lessonId = (int)cmdLesson.ExecuteScalar(); // Get the new Lesson ID
}
```

Figure 173: Updating the Course (Delete & Re-Insert Strategy)

```
// 3c. Insert Quiz
string insertQuizQuery = @"
    INSERT INTO Quiz (LessonId, QuizRewardCoins, TotalQuestions)
    OUTPUT INSERTED.Id
    VALUES (@LessonId, @QuizRewardCoins, @TotalQuestions)";

int quizId;
using (SqlCommand cmdQuiz = new SqlCommand(insertQuizQuery, con, transaction))
{
    cmdQuiz.Parameters.AddWithValue("@LessonId", lessonId);
    cmdQuiz.Parameters.AddWithValue("@QuizRewardCoins", (object)lesson.LessonQuiz.QuizRewardCoins ?? DBNull.Value);
    cmdQuiz.Parameters.AddWithValue("@TotalQuestions", lesson.LessonQuiz.Questions.Count);
    quizId = (int)cmdQuiz.ExecuteScalar(); // Get the new Quiz ID
}
```

Figure 174: Updating the Course (Delete & Re-Insert Strategy)

```
string insertQuestionQuery = @"
    INSERT INTO Question (QuizId, QuestionText, OptionA, OptionB, OptionC, OptionD, CorrectAnswer)
    VALUES (@QuizId, @QuestionText, @OptionA, @OptionB, @OptionC, @OptionD, @CorrectAnswer)";

using (SqlCommand cmdQuestion = new SqlCommand(insertQuestionQuery, con, transaction))
{
    cmdQuestion.Parameters.AddWithValue("@QuizId", quizId);
    cmdQuestion.Parameters.AddWithValue("@QuestionText", question.QuestionText);
    cmdQuestion.Parameters.AddWithValue("@OptionA", (object)question.OptionA ?? DBNull.Value);
    cmdQuestion.Parameters.AddWithValue("@OptionB", (object)question.OptionB ?? DBNull.Value);
    cmdQuestion.Parameters.AddWithValue("@OptionC", (object)question.OptionC ?? DBNull.Value);
    cmdQuestion.Parameters.AddWithValue("@OptionD", (object)question.OptionD ?? DBNull.Value);
    cmdQuestion.Parameters.AddWithValue("@CorrectAnswer", question.CorrectAnswer);
    cmdQuestion.ExecuteNonQuery();
}
```

Figure 175: Updating the Course (Delete & Re-Insert Strategy)

Unlike Create Course, Edit Course has to overwrite all the lessons, quizzes, and questions that already exist instead of adding them.

To provide a consistency in data, the system:

1. Deletes all records that the depend on it. (student progress à questions à quizzes à lessons)
2. Updates the changed structure according to the changes made by the educator.

This procedure ensures that the course is updated precisely by the newest changes without any remaining or old records within the database.

Track Student Progression:

1. Loading the Course Title from the Database:

```
// Gets the course title from the database using the CourseId
using (SqlConnection conn = new SqlConnection(connStr))
{
    conn.Open();
    SqlCommand cmd = new SqlCommand("SELECT Title FROM Course WHERE Id=@cid", conn);
    cmd.Parameters.AddWithValue("@cid", courseId);

    // If no title found, show "Untitled Course"
    lblCourseTitle.Text = cmd.ExecuteScalar()?.ToString() ?? "Untitled Course";
}
```

Figure 176: Loading the Course Title from the Database

The page will fetch the course title depending on the “CourseId” and show it on the top of the interface. This will enable educator to check with ease which student has progressed on which course they are examining.

2. Retrieving Student Progress with JOIN Queries:

```
// Base SQL query: joins StudentCourseProgress, Student, and Users tables
string sql = @"
    SELECT u.FullName, u.Email,
           ISNULL(scp.Status, 'Incomplete') AS Status
    FROM StudentCourseProgress scp
    JOIN Student s ON scp.StudentId = s.Id
    JOIN Users u ON s.UserId = u.Id
    WHERE scp.CourseId = @cid";
```

Figure 177: Retrieving Student Progress with JOIN Queries

This SQL query is used to retrieve the names of the students, emails, and status of course completion through three tables by using “JOIN” operations. It consolidates all pertinent student data in a single set of data, which enables the educator to track student progress.

3. Dynamic Progress Filtering (Completed and Incomplete):

```
// Filter based on button clicked
if (statusFilter == "Completed")
    sql += " AND scp.Status = 'Completed'";
else
    sql += " AND (scp.Status IS NULL OR scp.Status <> 'Completed')";
```

Figure 178: Dynamic Progress Filtering (Completed and Incomplete)

The system dynamically changes the SQL query depending on the choice of the educator. This makes it possible to filter between students who have already completed the course and those who are still in progress. It makes the track page flexible and interactive without moving on to another page.

4. Binding Student Data to the UI:

```
// Run the query and fill results into a DataTable
SqlDataAdapter da = new SqlDataAdapter(sql, conn);
da.SelectCommand.Parameters.AddWithValue("@cid", courseId);
DataTable dt = new DataTable();
da.Fill(dt);

// Bind the student list to the Repeater (rptStudents)
rptStudents.DataSource = dt;
rptStudents.DataBind();
```

Figure 179: Binding Student Data to the UI

The data of the progress retrieved is inserted in a “DataTable” and then bound to the “Repeater” component, and as a result the display is updated immediately. This will guarantee that educators will have the latest student progress in the course they have selected.

5. Showing "No Data Found" Messages:

```
// Show a message if no students found
lblNoData.Visible = dt.Rows.Count == 0;
lblNoData.Text = dt.Rows.Count == 0
    ? "No students found for this category."
    : "";
```

Figure 180: Showing "No Data Found" Messages

In case no students fit the chosen filter (Completed or Incomplete), the system does not show a blank list, but presents a helpful message. This enhances clarity and does not confuse the educator.

6. Switching Between Tabs (Incomplete or Completed):

```
protected void btnIncomplete_Click(object sender, EventArgs e)
{
    // When "Incomplete" tab clicked:
    btnIncomplete.CssClass = "tab-button active";
    btnComplete.CssClass = "tab-button";

    int courseId = Convert.ToInt32(Request.QueryString["CourseId"]);

    // Reload students with Incomplete status
    LoadStudents(courseId, "Incomplete");
    lblStatusMessage.Text = "Showing students with incomplete progress.";
}
```

Figure 181: Switching Between Tabs (Incomplete or Completed)

```
protected void btnComplete_Click(object sender, EventArgs e)
{
    // When "Completed" tab clicked:
    btnIncomplete.CssClass = "tab-button";
    btnComplete.CssClass = "tab-button active";

    int courseId = Convert.ToInt32(Request.QueryString["CourseId"]);

    // Reload students with Completed status
    LoadStudents(courseId, "Completed");
    lblStatusMessage.Text = "Showing students who have completed this course.";
}
```

Figure 182: Switching Between Tabs (Incomplete or Completed)

In case the educator clicks either of the tabs, the UI changes the button styles to reflect which tab is used. The student list is reloaded into the system with the correct filter. This offers a convenient interface to monitor progression.

Educator profile:

1. **Loading Educator Profile Information:**

```
// Get educator profile info from Users and Educator tables
string sql = @"
    SELECT
        e.Id AS EducatorId,
        u.FullName, u.Age, u.Gender, u.ProfilePicture, u.Email,
        e.EducationQualification, e.GraduatedUniversity
    FROM Users u
    JOIN Educator e ON u.Id = e.UserId
    WHERE u.Id = @uid";
```

Figure 183: Loading Educator Profile Information

This query joins both “Educator” and “Users” tables and loads all information that is specific to educator. It extracts personal information of the educator, their academic qualification, university and profile picture. A combination of this data enables the profile page to present all the educator information in a single place.

2. **Displaying Profile Picture (With Default Fallback):**

```
// Load profile picture
string profilePic = dr["ProfilePicture"]?.ToString();
imgProfile.ImageUrl = string.IsNullOrEmpty(profilePic)
    ? ResolveUrl("~/Image/default_profile2.png")
    : ResolveUrl("~/Image/" + profilePic);
```

Figure 184: Displaying Profile Picture (With Default Fallback)

In case the educator does not post any profile picture, the system uses a default image. Otherwise, it shows the picture uploaded by the educator. This makes the UI appearance complete at any given time.

3. Switching Profile into Edit Mode:

```
protected void btnEdit_Click(object sender, EventArgs e)
{
    // Make profile fields editable
    txtFullName.ReadOnly = false;
    txtAge.ReadOnly = false;
    txtUniversity.ReadOnly = false;
    ddlGender.Enabled = true;
    ddlQualification.Enabled = true;
    fileUploadProfile.Visible = true;

    // Show Save/Cancel buttons
    btnEdit.Visible = false;
    btnSave.Visible = true;
    btnCancel.Visible = true;
    lblMsg.Text = "";
}
```

Figure 185: Switching Profile into Edit Mode

The system will turn the profile form into an editable mode when the educator clicks the “Edit Profile” button. Input fields are activated and the picture profile upload control is visible. This brings clear distinction between viewing and editing states.

4. Uploading a New Profile Picture:

```
// Generate a unique file name for uploaded profile picture
string extension = Path.GetExtension(fileUploadProfile.FileName);
fileName = "educator_" + userId + "_" + DateTime.Now.Ticks + extension;

string folderPath = Server.MapPath("~/Image/");
if (!Directory.Exists(folderPath))
    Directory.CreateDirectory(folderPath);

// Save the uploaded file to Image folder
fileUploadProfile.SaveAs(Path.Combine(folderPath, fileName));
fileUploadedSuccessfully = true;
```

Figure 186: Uploading a New Profile Picture

When a new profile picture is uploaded a unique filename will be created based on the ID and time of the educator. This will avoid conflicts on files and make sure everything that is uploaded overwrites nothing accidentally. The file is then stored in the image directory of the server.

5. Updating Educator & User Tables (Within a Transaction):

```
// Update Users table
string sqlUser = @"UPDATE Users
          SET FullName=@name, Age=@age, Gender=@gender"
                  + (fileUploadedSuccessfully ? ", ProfilePicture=@pic" : "")
                  + " WHERE Id=@uid";
```

Figure 187: Updating Educator & User Tables (Within a Transaction)

```
// Update Educator table
string sqlEdu = @"UPDATE Educator
          SET EducationQualification=@qual, GraduatedUniversity=@uni
          WHERE UserId=@uid";
```

Figure 188: Updating Educator & User Tables (Within a Transaction)

The profile updates are saved in both “Educator” and “Users” tables. A transaction makes sure that all the updates, such as the optional profile picture, are reviewed either successfully or unsuccessfully as a whole. This prevents partial updates (name updated and qualification not updated).

6. Teaching Statistics (Total Courses, Students, Completions):

```
// Count courses created
SqlCommand cmdCourses = new SqlCommand("SELECT COUNT(*) FROM Course WHERE EducatorId=@eid", conn);
cmdCourses.Parameters.AddWithValue("@eid", educatorId);
lblCoursesCreated.Text = cmdCourses.ExecuteScalar()?.ToString() ?? "0";

// Count total students enrolled
SqlCommand cmdStudents = new SqlCommand(@"
    SELECT COUNT(DISTINCT scp.StudentId)
    FROM StudentCourseProgress scp
    INNER JOIN Course c ON scp.CourseId = c.Id
    WHERE c.EducatorId = @eid", conn);
cmdStudents.Parameters.AddWithValue("@eid", educatorId);
lblTotalStudents.Text = cmdStudents.ExecuteScalar()?.ToString() ?? "0";

// Count completed courses
SqlCommand cmdCompleted = new SqlCommand(@"
    SELECT COUNT(*)
    FROM StudentCourseProgress scp
    INNER JOIN Course c ON scp.CourseId = c.Id
    WHERE c.EducatorId = @eid AND scp.Status = 'Completed'", conn);
cmdCompleted.Parameters.AddWithValue("@eid", educatorId);
lblCompletions.Text = cmdCompleted.ExecuteScalar()?.ToString() ?? "0";
```

Figure 189: Teaching Statistics (Total Courses, Students, Completions)

In the educator dashboard, there are three statistics will be showing which is:

1. Total Students
2. Courses Created
3. Student Course Completed

These queries provide the summarization of the educator activity by combining the required tables and summing up the appropriate records.

4.3.4 Admin **CHEAH JUN HENG**

Admin Dashboard

1. Form Validation Features

Search Input Validation

```
case "CourseID":
    if (int.TryParse(SearchValue, out int courseId))
    {
        whereSql += "Id = @SearchValue";
        parameters.Add("@SearchValue", courseId);
    }
    else
    {
        whereSql += "1 = 0";
    }
    break;
case "EducatorID":
    if (int.TryParse(SearchValue, out int educatorId))
    {
        whereSql += "EducatorId = @SearchValue";
        parameters.Add("@SearchValue", educatorId);
    }
    else
    {
        whereSql += "1 = 0";
    }
    break;
```

Figure 190: Search Input Validation Code

When filtering by CourseID or EducatorID, the code validates that the search value is a valid integer using int.TryParse(). If invalid, it returns no results (1 = 0) instead of throwing an error.

2. State Management Features

View State Properties for Filter Persistence

```
private string FilterType
{
    get { return ViewState["FilterType"] as string ?? "All"; }
    set { ViewState["FilterType"] = value; }
}

9 references
private string SearchValue
{
    get { return ViewState["SearchValue"] as string ?? ""; }
    set { ViewState["SearchValue"] = value; }
}
```

Figure 191: Filter Code

These properties persist filter values across post backs using View State, maintaining the user's search criteria.

Page Number Persistence

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        currentPage = 1;
        ViewState["CurrentPage"] = currentPage;
        FilterType = "All";
        SearchValue = "";
        LoadDashboardStats();
        LoadCourses();
    }
    else
    {
        currentPage = (int)(ViewState["CurrentPage"] ?? 1);
    }

    // Update the search box with persisted value
    if (!IsPostBack)
    {
        txtSearchValue.Text = SearchValue;
        ddlFilterType.SelectedValue = FilterType;
    }
}
```

Figure 192: View State Post Back

The current page number is stored in ViewState to maintain pagination state across postbacks.

3. Error Handling and User Feedback

Graceful Error Handling

```
    catch (Exception ex)
    {
        ShowMessage("Error loading courses. Please try again.", "error");
        gvCourses.DataSource = null;
        gvCourses.DataBind();
    }
}
```

Figure 193: Graceful Error Handling

Different error handling strategies - silent failure for stats (non-critical) and user-friendly messages for course loading (critical).

User Message System

```
    private void ShowMessage(string message, string type)
    {
        pnlMessage.Visible = true;
        lblMessage.Text = message;

        if (type == "success")
        {
            pnlMessage.CssClass = "message-container message-success";
        }
        else
        {
            pnlMessage.CssClass = "message-container message-error";
        }

        // Auto-hide success messages after 5 seconds
        if (type == "success")
        {
            ScriptManager.RegisterStartupScript(this, this.GetType(), "hideMessage",
                "setTimeout(function() { document.getElementById('" + pnlMessage.ClientID + "').style.display = 'none'; }, 5000);", true);
        }
    }
}
```

Figure 194: User Message System

Provides consistent user feedback with different styling for success/error messages and auto-hides success messages.

4. Pagination Logic

```
private void LoadCourses()
{
    string connString = ConfigurationManager.ConnectionStrings["SeaLearnerConnection"].ConnectionString;

    try
    {
        var (whereSql, parameters) = BuildWhereClauseWithParameters();

        // Get total count for pagination
        int totalRecords = GetTotalCourseCount(whereSql, parameters);
        int totalPages = (int)Math.Ceiling((double)totalRecords / pageSize);

        // Update pagination controls
        lblCurrentPage.Text = currentPage.ToString();
        lblTotalPages.Text = totalPages.ToString();
        btnPrev.Enabled = currentPage > 1;
        btnNext.Enabled = currentPage < totalPages;

        // Calculate pagination
        int startIndex = (currentPage - 1) * pageSize;

        using (SqlConnection conn = new SqlConnection(connString))
        {
            string query = $@"
                SELECT Id, EducatorId, CourseType, Status, Title
                FROM Course
                {whereSql}
                ORDER BY Id
                OFFSET {startIndex} ROWS
                FETCH NEXT {pageSize} ROWS ONLY";
```

Figure 195: Pagination Logic

Implements efficient server-side pagination using SQL Server's OFFSET-FETCH feature to only retrieve the needed records.

5. Dynamic Filtering System

```
private (string whereSql, System.Collections.Generic.Dictionary<string, object> parameters) BuildWhereClauseWithParameters()
{
    var parameters = new System.Collections.Generic.Dictionary<string, object>();

    if (string.IsNullOrEmpty(SearchValue) || FilterType == "All")
        return ("", parameters);

    string whereSql = "WHERE ";
    switch (FilterType)...

    return (whereSql, parameters);
}
```

Figure 196: Filtering

Creates dynamic SQL WHERE clauses safely using parameters, supporting multiple filter types without SQL injection vulnerabilities. These features work together to create a robust, user-friendly admin interface with proper validation, state management and error handling.

Admin Community

1. Data Management Features

```
protected void rptPosts_ItemCommand(object source, RepeaterCommandEventArgs e)
{
    if (e.CommandName == "DeletePost")
    {
        int postId = Convert.ToInt32(e.CommandArgument);

        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();

            // Step 1: Delete all replies linked to this post
            string deleteReplies = "DELETE FROM Reply WHERE PostId = @PostId";
            using (SqlCommand cmdReplies = new SqlCommand(deleteReplies, conn))
            {
                cmdReplies.Parameters.AddWithValue("@PostId", postId);
                cmdReplies.ExecuteNonQuery();
            }

            // Step 2: Delete the post itself
            string deletePost = "DELETE FROM CommunityPost WHERE Id = @PostId";
            using (SqlCommand cmdPost = new SqlCommand(deletePost, conn))
            {
                cmdPost.Parameters.AddWithValue("@PostId", postId);
                cmdPost.ExecuteNonQuery();
            }
        }

        // Reload data without showing success message
        LoadPosts();
    }
}
```

Figure 197: Admin Dashboard Data Management Features

Manually implements cascading deletion by first removing child records (replies) before deleting the parent record (post) to maintain referential integrity.

2. Dynamic Data Structure

```
private void LoadPosts()
{
    DataTable dtPosts = new DataTable();

    using (SqlConnection conn = new SqlConnection(connString))
    {
        string query = @"SELECT CP.Id, CP.UserId, U.FullName, U.Role, CP.PostContent, CP.PostDateTime
                FROM CommunityPost CP
                JOIN Users U ON CP.UserId = U.Id
                ORDER BY CP.PostDateTime DESC";

        SqlDataAdapter da = new SqlDataAdapter(query, conn);
        da.Fill(dtPosts);
    }

    // Add Replies column
    if (!dtPosts.Columns.Contains("Replies"))
    {
        dtPosts.Columns.Add("Replies", typeof(object));
    }

    foreach (DataRow row in dtPosts.Rows)
    {
        int postId = Convert.ToInt32(row["Id"]);
        row["Replies"] = GetReplies(postId);
    }

    rptPosts.DataSource = dtPosts;
    rptPosts.DataBind();

    // Show no posts message if empty
    pnlNoPosts.Visible = dtPosts.Rows.Count == 0;

    // Update statistics after operations
    LoadStatistics();
}
```

Figure 198: Admin Community Dynamic Data Structure

Dynamically adds a "Replies" column to the Data Table to store nested reply to data and creating a hierarchical data structure.

3. Nested Data Retrieval

```
1 reference
private List<object> GetReplies(int postId)
{
    List<object> replies = new List<object>();

    using (SqlConnection conn = new SqlConnection(connString))
    {
        string query = @"SELECT R.Id as ReplyId, R.PostId, R.ReplyContent, R.ReplyDateTime, U.FullName
                         FROM Reply R
                         JOIN Users U ON R.UserId = U.Id
                         WHERE R.PostId = @PostId
                         ORDER BY R.ReplyDateTime ASC";
        SqlCommand cmd = new SqlCommand(query, conn);
        cmd.Parameters.AddWithValue("@PostId", postId);

        conn.Open();
        SqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
            replies.Add(new
            {
                ReplyId = reader["ReplyId"],
                PostId = reader["PostId"],
                FullName = reader["FullName"].ToString(),
                ReplyContent = reader["ReplyContent"].ToString(),
                ReplyDateTime = Convert.ToDateTime(reader["ReplyDateTime"])
            });
        }
    }

    return replies;
}
```

Figure 199: Nested Data Retrieval

Uses anonymous objects to create flexible data structures for replies, allowing easy binding to nested repeaters.

4. Control Identification Pattern

```
protected void btnDeletePost_PreRender(object sender, EventArgs e)
{
    LinkButton btn = (LinkButton)sender;
    btn.Attributes["data-uniqueid"] = btn.UniqueID;
}

0 references
protected void btnDeleteReply_PreRender(object sender, EventArgs e)
{
    LinkButton btn = (LinkButton)sender;
    btn.Attributes["data-uniqueid"] = btn.UniqueID;
}
```

Figure 200: Control Identification Pattern

Stores the UniqueID in custom attributes during PreRender to help with control identification in complex nested repeater scenarios.

Admin Feedback Management

1. Safe Data Handling

```
private string GetSafeString(object value, string defaultValue = "")
{
    if (value == null || value == DBNull.Value)
        return defaultValue;
    return value.ToString();
}
```

Figure 201: Admin Feedback Safe Data Handling

Provides a reusable method to safely handle database null values with configurable default values.

2. Client-Server Communication Pattern

```
public partial class admin_feedbackmanagement : System.Web.UI.Page
{
    0 references
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            LoadFeedbackStats();
            LoadFeedback();
        }

        // Handle postback commands
        if (Request["__EVENTTARGET"] == "DeleteFeedback")
        {
            string feedbackId = Request["__EVENTARGUMENT"];
            DeleteFeedback(feedbackId);
        }

        if (Request["__EVENTTARGET"] == "CompleteFeedback")
        {
            string feedbackId = Request["__EVENTARGUMENT"];
            CompleteFeedback(feedbackId);
        }
    }
```

Figure 202: Client-Server Communication Pattern

Uses ASP.NET's built-in event target mechanism to handle custom client-side actions for delete and complete operations.’

3. JavaScript Integration for User Confirmation

```
        Button btnComplete = new Button();
        btnComplete.Text = "Complete";
        btnComplete.CssClass = "btn-sm btn-success";
        btnComplete.OnClientClick = $"completeFeedback('{id}'); return false;";
        actions.Controls.Add(btnComplete);
    }

    // Delete button
    Button btnDelete = new Button();
    btnDelete.Text = "Delete";
    btnDelete.CssClass = "btn-sm btn-danger";
    btnDelete.OnClientClick = $"confirmDelete('{id}', '{subject.Replace("'", "''")}'); return
    actions.Controls.Add(btnDelete);

    footer.Controls.Add(actions);
    card.Controls.Add(footer);

    pnlFeedback.Controls.Add(card);
}
```

Figure 203: Java Script Confirmation

Integrates JavaScript functions for client-side interactions with proper string escaping to prevent JavaScript errors.

Admin Manage Ads

1. Custom Data Formatting & Parsing

```
private void CreateAnnouncementCard(string id, string content, string imagePath, DateTime? startDate, DateTime? endDate)
{
    string title = "Announcement";
    string announcementType = "promotion"; // default type
    string announcementContent = content;

    // Parse content format: "Title\n\nContent\n\nType"
    string[] sections = content.Split(new[] { "\n\n" }, StringSplitOptions.RemoveEmptyEntries);

    if (sections.Length >= 1)
    {
        title = sections[0].Trim();

        if (sections.Length >= 2)
        {
            // Check if last section is a valid type
            string lastSection = sections[sections.Length - 1].Trim().ToLower();

            // If last section is a valid type, use it and remove it from content
            if (lastSection == "promotion" || lastSection == "lesson" || lastSection == "partner")
            {
                announcementType = lastSection;

                // Rebuild content without the type section
                if (sections.Length > 2)
                {
                    announcementContent = string.Join("\n\n", sections, 1, sections.Length - 2);
                }
                else
                {
                    announcementContent = ""; // Only title and type, no content
                }
            }
            else
            {
                // Last section is not a valid type, so use all sections after title as content
                announcementContent = string.Join("\n\n", sections, 1, sections.Length - 1);
                announcementType = "promotion"; // default type
            }
        }
```

Figure 204: Data Formatting & Parsing

Implements a custom data parsing system that extracts title, content, and type from a structured string format using \n\n as delimiters.

2. Base64 Image Upload Handling

```
string fileName = hdnFileName.Value;
string fileData = hdnFileData.Value;

if (!string.IsNullOrEmpty(fileName) && !string.IsNullOrEmpty(fileData))
{
    try
    {
        string fileExtension = Path.GetExtension(fileName).ToLower();

        // Validate file type
        if (fileExtension != ".jpg" && fileExtension != ".jpeg" && fileExtension != ".png" && fileExtension != ".gif")
        {
            ScriptManager.RegisterStartupScript(this, this.GetType(), "showError", "alert('Please upload only JPG, PNG, or
            return;
        }

        // Ensure Image directory exists
        string imageDir = Server.MapPath("~/Image/");
        if (!Directory.Exists(imageDir))
        {
            Directory.CreateDirectory(imageDir);
            System.Diagnostics.Debug.WriteLine($"Created directory: {imageDir}");
        }

        // Generate unique filename to avoid conflicts
        string uniqueFileName = Guid.NewGuid().ToString() + fileExtension;
        string fullPath = Path.Combine(imageDir, uniqueFileName);

        // Convert base64 string to image file
        byte[] imageBytes = Convert.FromBase64String(fileData.Split(',')[1]); // Remove  prefix
        File.WriteAllBytes(fullPath, imageBytes);

        // Set the image path for database - store only filename
        imagePath = uniqueFileName;

        System.Diagnostics.Debug.WriteLine($"Image saved successfully: {fullPath}");
        System.Diagnostics.Debug.WriteLine($"Database will store filename: {imagePath}");
    }
```

Figure 205: Image Upload Handling

Processes base64-encoded images from client-side uploads, validates file types, generates unique filenames, and saves to the server.

3. Comprehensive File Cleanup

```
if (rowsAffected > 0)
{
    // Delete the image file from Image folder
    if (imagePathObj != null && imagePathObj != DBNull.Value)
    {
        string imagePath = imagePathObj.ToString();
        if (!string.IsNullOrEmpty(imagePath))
        {
            string physicalPath = Server.MapPath("~/Image/" + imagePath);
            if (File.Exists(physicalPath))
            {
                File.Delete(physicalPath);
                System.Diagnostics.Debug.WriteLine($"Deleted image file: {physicalPath}");
            }
        }
    }
```

Figure 206: File Cleanup

Implements complete cleanup by deleting both database records and associated image files to prevent orphaned files.

Admin View Course

1. Query String Parameter Validation

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        if (Request.QueryString["courseId"] != null)
        {
            string courseId = Request.QueryString["courseId"];
            LoadCourseDetails(courseId);
            LoadCourseLessons(courseId);
            LoadCourseQuizzes(courseId);
        }
        else
        {
            ShowMessage("No course ID provided.", "error");
        }
    }
}
```

Figure 207: Query String Parameter Validation

Validates the presence of required query string parameters and provides user feedback when missing.

2. Hierarchical Data Loading Pattern

```
0 references
protected void rptQuizzes_ItemDataBound(object sender, RepeaterItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.Item || e.Item.ItemType == ListItemType.AlternatingItem)
    {
        RepeaterItem item = e.Item;
        DataRowView rowView = (DataRowView)item.DataItem;

        string quizId = rowView["QuizId"].ToString();

        // Find the questions repeater in the current quiz item
        Repeater rptQuestions = (Repeater)item.FindControl("rptQuestions");

        if (rptQuestions != null)
        {
            LoadQuizQuestions(quizId, rptQuestions);
        }
    }
}
```

Figure 208: Hierarchical Data Loading Pattern

Implements a parent-child data loading pattern where quizzes are loaded first, then questions for each quiz are loaded dynamically during the ItemDataBound event.

3. Graceful Fallback for Missing Data

```
if (row["CoursePicture"] != DBNull.Value && !string.IsNullOrEmpty(row["CoursePicture"].ToString()))
{
    imgCourse.ImageUrl = row["CoursePicture"].ToString();
}
else
{
    imgCourse.ImageUrl = "~/Image/default-course.jpg";
    imgCourse.AlternateText = "Default Course Image";
}
```

Figure 209: Replace Missing Data

Provides fallback default images and alt text when course images are not available.

Admin User Management

1. Custom Event Handling via __EVENTTARGET

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        LoadDashboardStats();
        LoadUsers();
    }

    // Handle status change commands
    if (Request["__EVENTTARGET"] == "DisableUser")
    {
        string userId = Request["__EVENTARGUMENT"];
        UpdateUserStatus(userId, "Inactive");
    }
    else if (Request["__EVENTTARGET"] == "EnableUser")
    {
        string userId = Request["__EVENTARGUMENT"];
        UpdateUserStatus(userId, "Active");
    }
}
```

Figure 210: Custom Event Handling

Uses ASP.net built-in __EVENTTARGET and __EVENTARGUMENT mechanism to handle custom client-side events for user status management.

2. Dynamic UI Generation with Security

```
private void CreateUserCard(string id, string fullName, string email, string role, string age, string gender, string status)
{
    Panel userCard = new Panel();
    userCard.CssClass = "user-card";

    // Main content
    Panel mainContent = new Panel();
    mainContent.CssClass = "user-main-content";

    // Header
    Panel header = new Panel();
    header.CssClass = "user-header";

    // Avatar
    Panel avatar = new Panel();
    avatar.CssClass = "user-avatar";
    avatar.Controls.Add(new LiteralControl(fullName.Substring(0, 1).ToUpper()));
    header.Controls.Add(avatar);

    // User info
    Panel userInfo = new Panel();
    userInfo.CssClass = "user-info";

    Label lblName = new Label();
    lblName.Text = $"<div class='user-name'>{Server.HtmlEncode(fullName)}</div>";
    userInfo.Controls.Add(lblName);

    Label lblRole = new Label();
    string roleClass = role.ToLower() == "student" ? "student-role" : "educator-role";
    lblRole.Text = $"<span class='user-role {roleClass}'>{Server.HtmlEncode(role)}</span>";
    userInfo.Controls.Add(lblRole);
```

Figure 211: Dynamic UI Generation

Dynamically builds the entire user interface programmatically with proper HTML encoding to prevent XSS attacks.

3. Conditional UI Rendering

```
// Status text
Label lblStatus = new Label();
lblStatus.Text = $"<div class='status-text {(status == "Active" ? "" : "disabled")}'>{status}</div>";
statusSection.Controls.Add(lblStatus);

// Action buttons
Panel actions = new Panel();
actions.CssClass = "user-actions";

if (status == "Active")
{
    Button btnDisable = new Button();
    btnDisable.Text = "Disable";
    btnDisable.CssClass = "disable-btn";
    btnDisable.OnClientClick = $"return confirmDisable('{id}', '{fullName.Replace("'", "\\'")}');";
    actions.Controls.Add(btnDisable);
}
else
{
    Button btnEnable = new Button();
    btnEnable.Text = "Enable";
    btnEnable.CssClass = "enable-btn";
    btnEnable.OnClientClick = $"return confirmEnable('{id}', '{fullName.Replace("'", "\\'")}');";
    actions.Controls.Add(btnEnable);
}
```

Figure 212: Conditional UI Rendering

Dynamically generates appropriate action buttons and applies conditional CSS classes based on user status.

4. Real-time Statistics Synchronization

```
private void UpdateUserStatus(string userId, string status)
{
    string connString = ConfigurationManager.ConnectionStrings["SeaLearnerConnection"].ConnectionString;

    try
    {
        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();

            string query = "UPDATE Users SET Status = @Status WHERE Id = @Id";
            SqlCommand cmd = new SqlCommand(query, conn);
            cmd.Parameters.AddWithValue("@Status", status);
            cmd.Parameters.AddWithValue("@Id", userId);

            int rowsAffected = cmd.ExecuteNonQuery();

            if (rowsAffected > 0)
            {
                ShowSuccess($"User status updated to {status} successfully!");
                LoadDashboardStats();
                LoadUsers();
            }
            else
            {
                ShowError("User not found or update failed.");
            }
        }
    }
```

Figure 213: Real-time Statistics

Ensures dashboard statistics and user lists remain synchronized after any status changes.

5.0 User Guidance **CHEAH JUN HENG**

Public Page

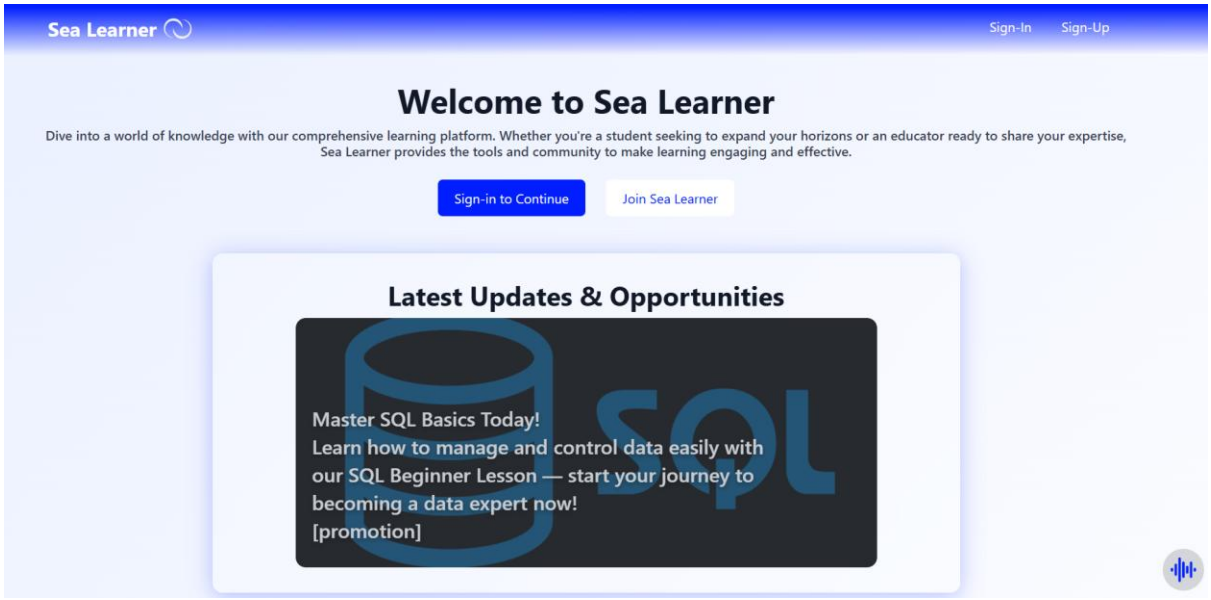

Figure 214: Public Page

This Public Page is for public view, so they will have the sign in or join sea learner (sign up) button to let user to sign in to their page or create their account.

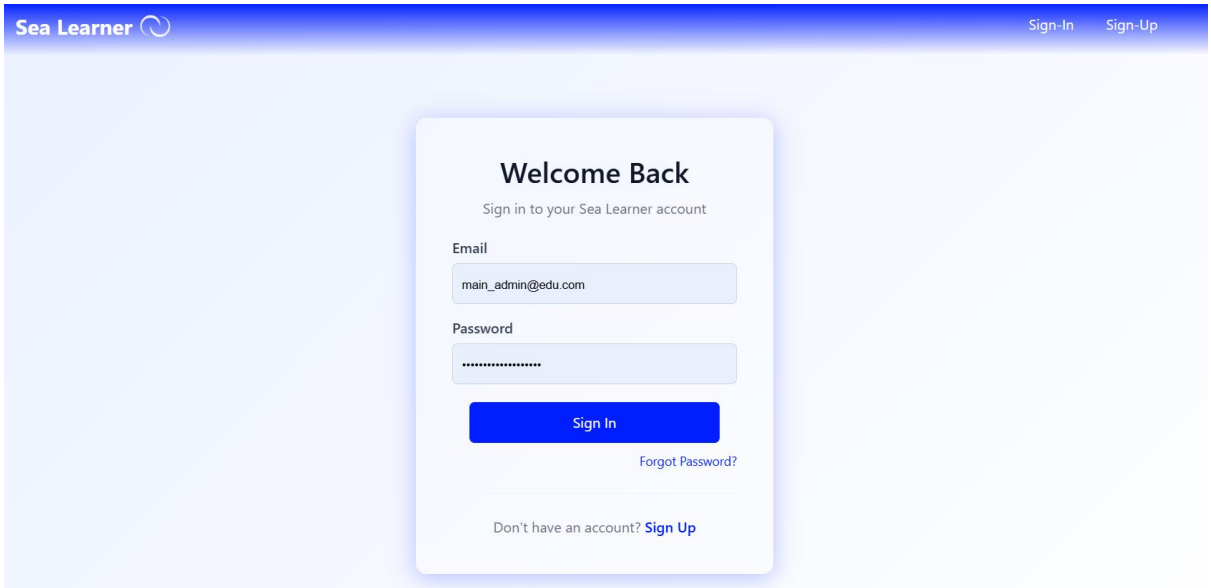

Figure 215: Sign In Page

This is the Log in Page that user insert their email and password to sign in.

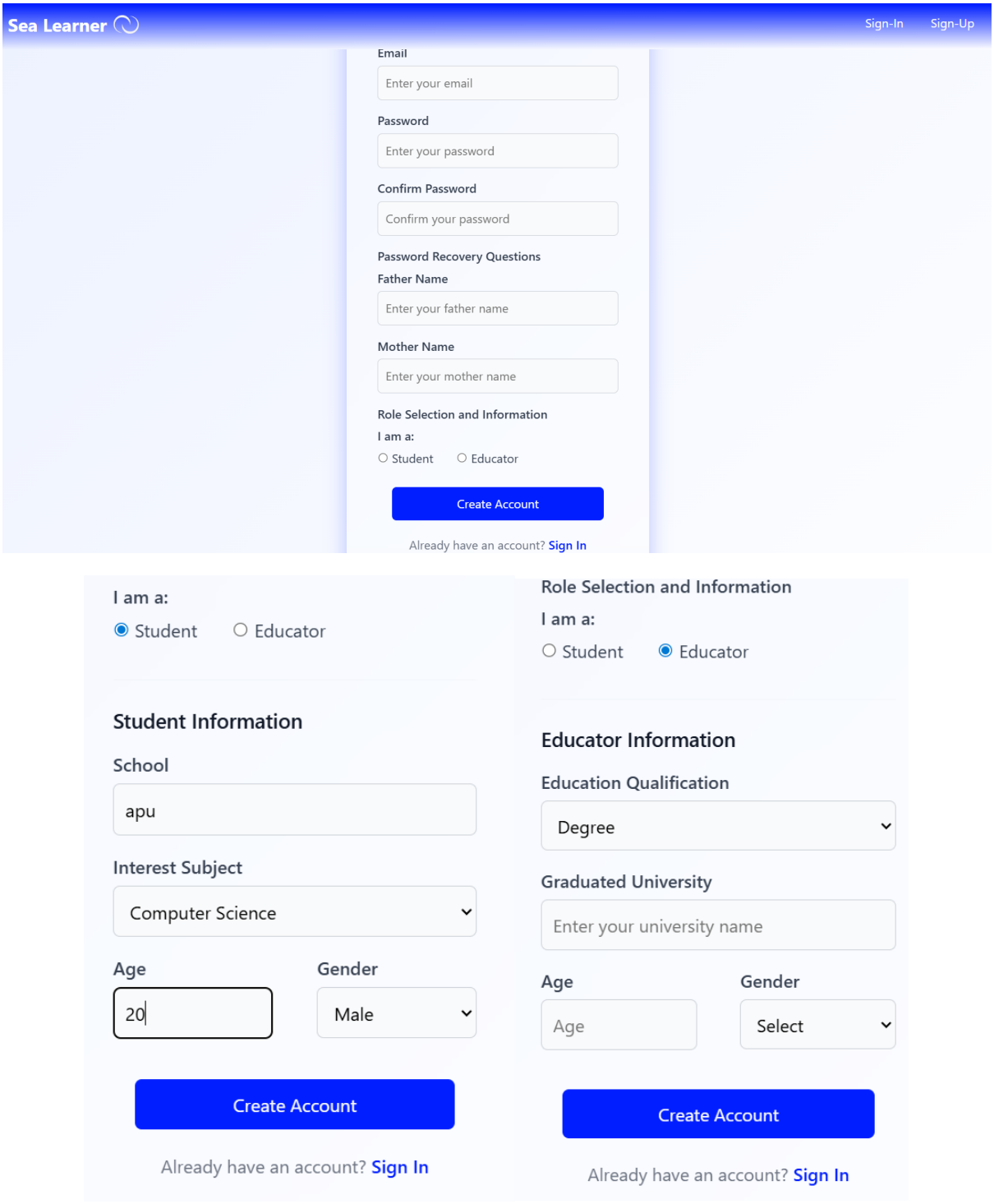

Figure 216: Sign Up Page

This is the Sign-Up Page, so user must insert all information otherwise can't create their account. So, Student and Educator have different information to insert so after insert the correct information and then press the create account button to create their account and go back to the sign in page.

Admin

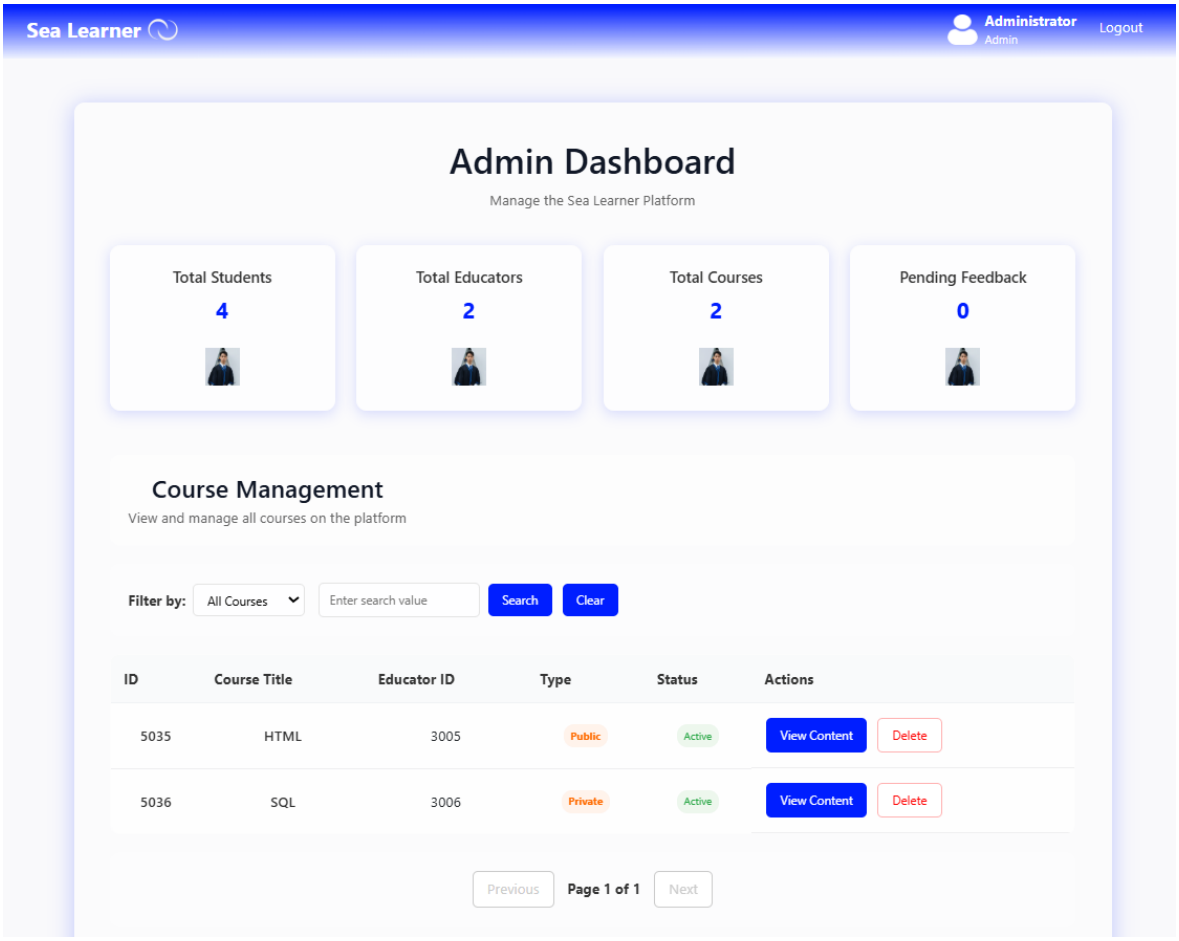

Figure 217: Admin Dashboard Page

This is the admin when login will see the first page dashboard. In this admin dashboard, admin can see the total students, total educator, total course and the how many that the pending feedback remain.

Below the course management, can see what courses it has currently. So, there is the filter that the admin can find the course easily and then the course admin can delete the course or view the course to see what content inside the course.

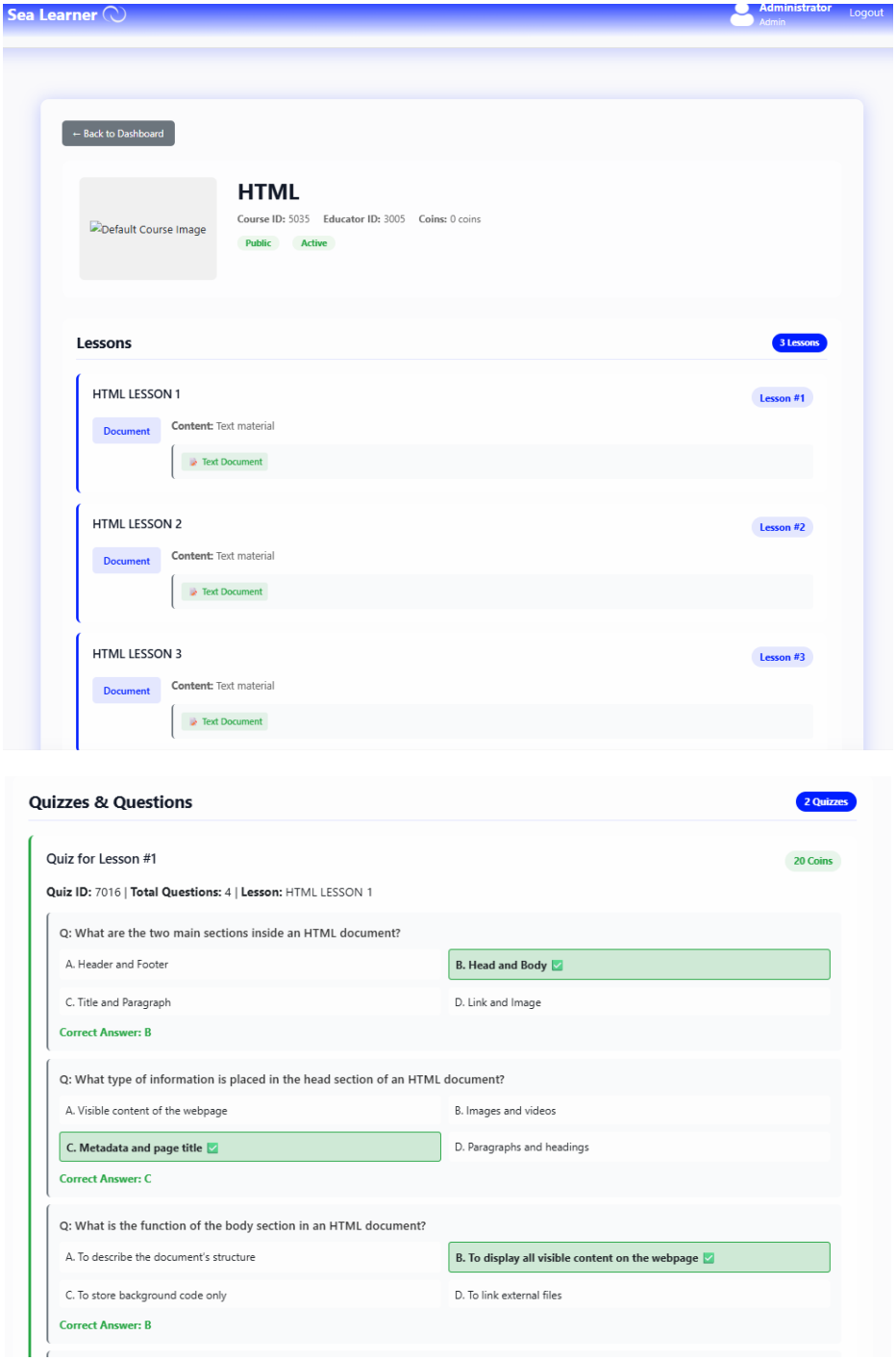

Figure 218: Admin View Couse Page

So, after admin click the view course will come to this page, this page can view the course lesson and the quiz that the educator created. After admin view finish this page and then press the back to dashboard button to go back to admin page.

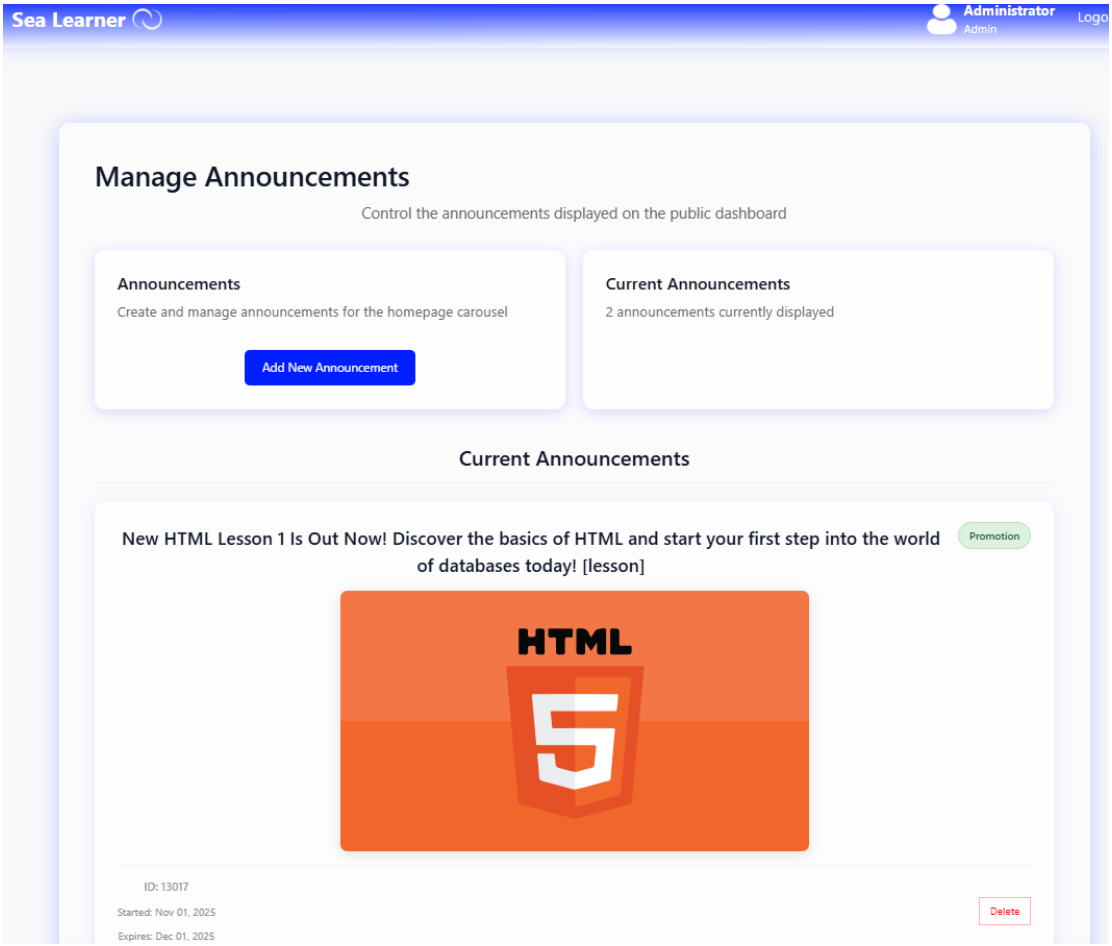

Figure 219: Admin Manage Announcement Page

In this admin manage announcement page, admin can create the new announcement but clicking the button and can view the current announcement that will show on the public page. If admin wants to delete the announcement also can click the delete button to delete the announcement.

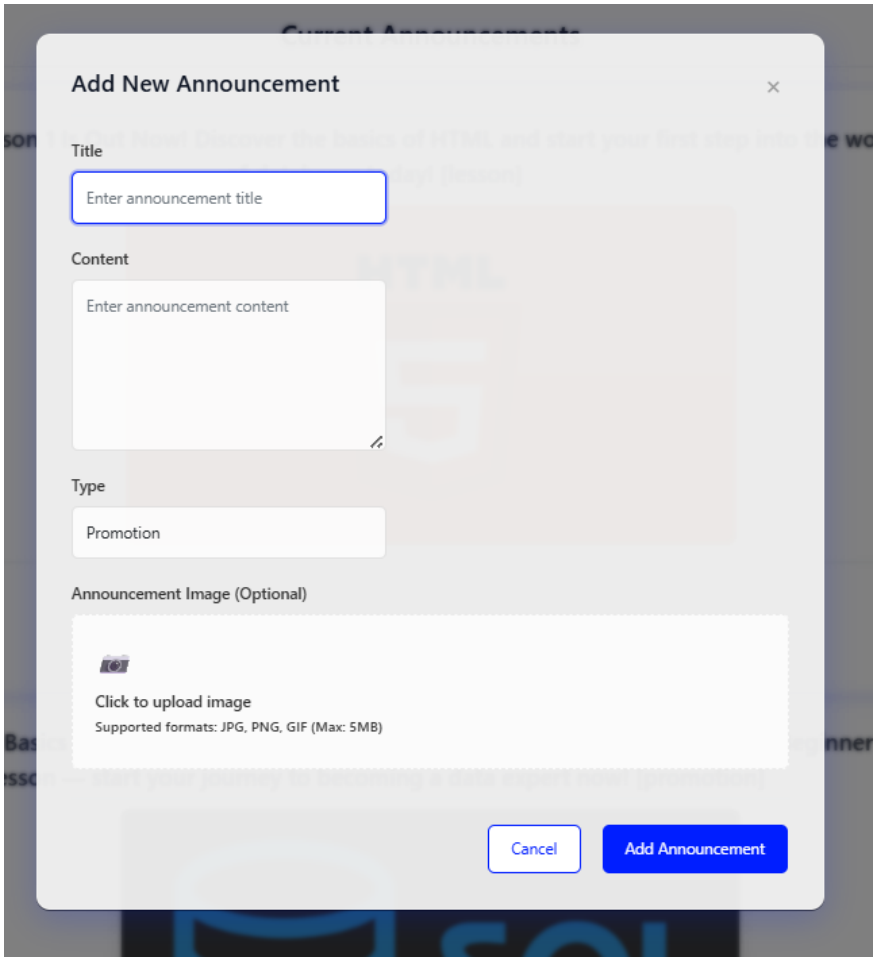

Figure 220: Admin Add New Announcement Small Page

So, after admin click the add new announcement will pop out this page. After admin insert all the following information or see whether that admin wants to put the picture, so this picture is optional. After filling in all the information can click the add announcement button to create it. If admin don't want to create the announcement already can click the cancel button to go back.

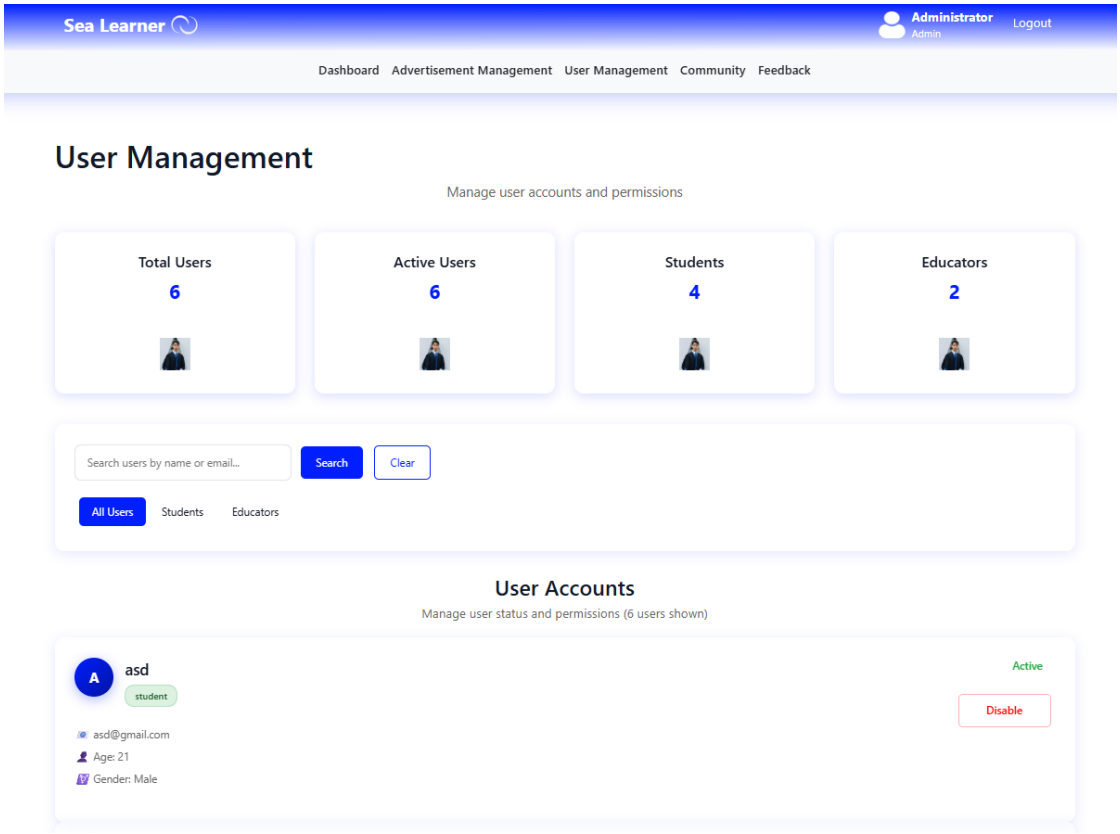

Figure 221: Admin User Management Page

This is the Admin User Management Page, so admin can see the total users, active users and the user divide into two that is students and educator to let admin to easily see how many users of it. Below there will also have a small filter function to let admin to search by the name or email and then click the search button to search or cancel button to undo and have a students and educator filter button.

Below the is to view more information of the user accounts, so admin can see the user information. There is still have a function that is admin can disable their account or disable their account by clicking the disabled button. If is unactive so will appear able button.

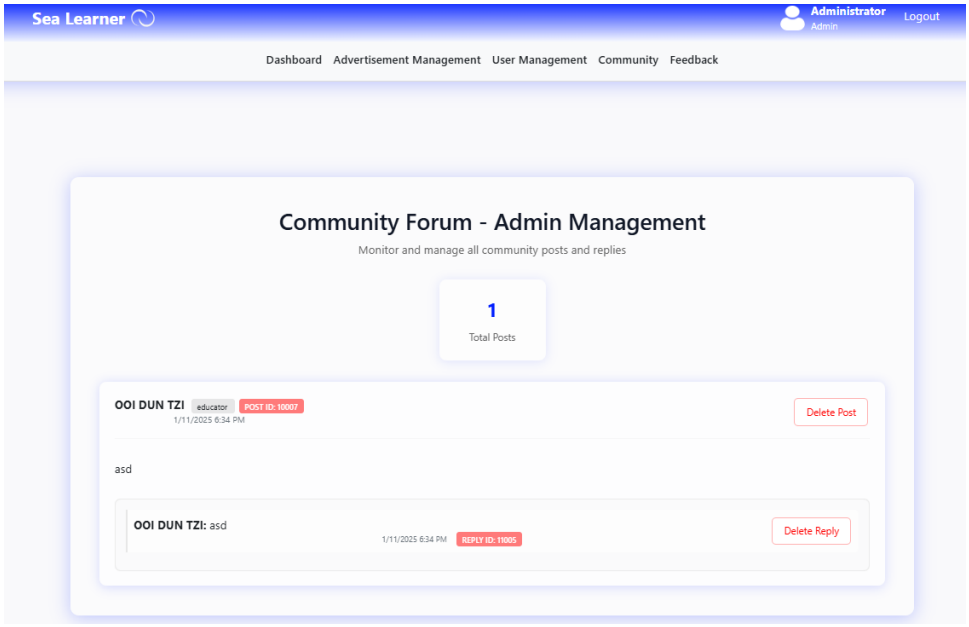

Figure 222: Admin Manage Community Page

In this Admin Manage Community Page, admin can see the total forum that users posted and writing about. Admin can click the delete post or delete the post reply by clicking the button if admin wants to delete it.

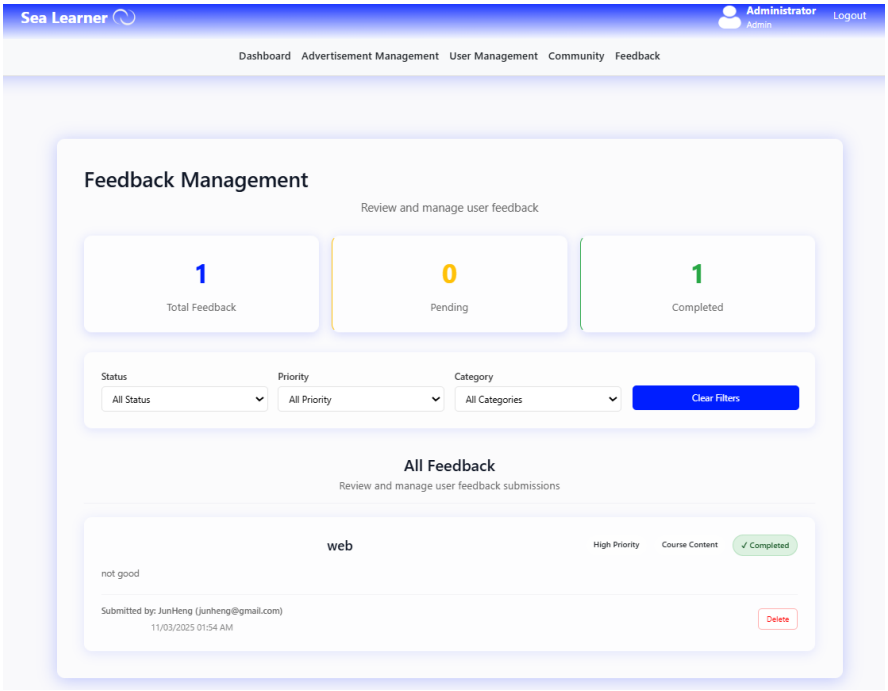

Figure 223: Admin Manage Feedback Page

In this page admin can see the total feedback, pending and the completed feedback that already done. Below also will have a filter button the find the feedback easily. So, in this page mainly is to see the student given what feedback so that the admin can improve after done view the

feedback can click done so will mark as completed. If admin want to delete the feedback also can click the delete button to delete it.

Student

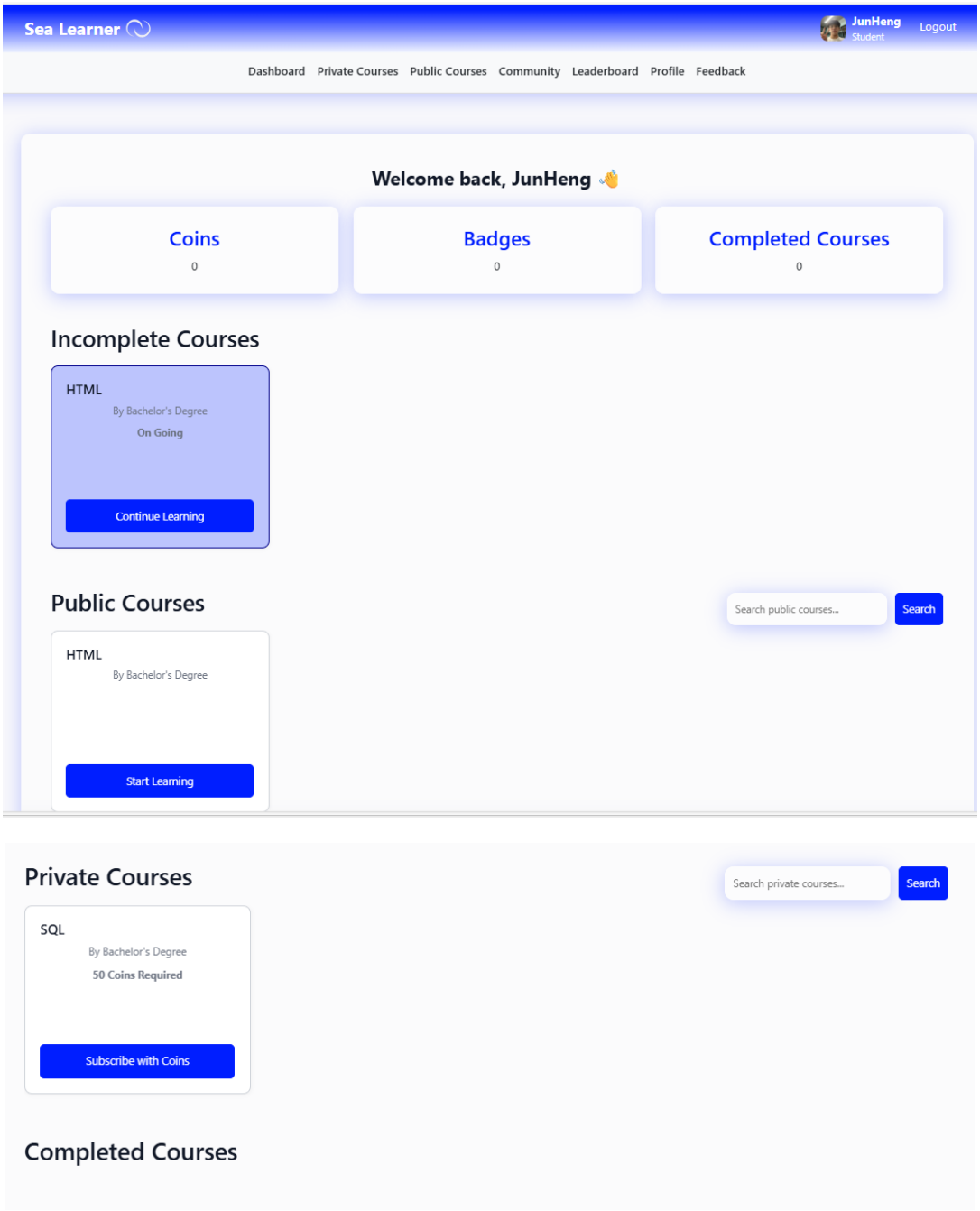

Figure 224: Student Dashboard Page

This is the student main page that when student's login will show this page first. In this page will show the coins, badges and completed courses. Below there will show the incomplete courses, public courses and private courses. So, the incomplete courses if student not yet

finish their course press the continue learning button and will go direct to the courses page and also the public courses and private course will show out some courses, so student can see whether which courses they want and just press start learning and will show at the incomplete courses.

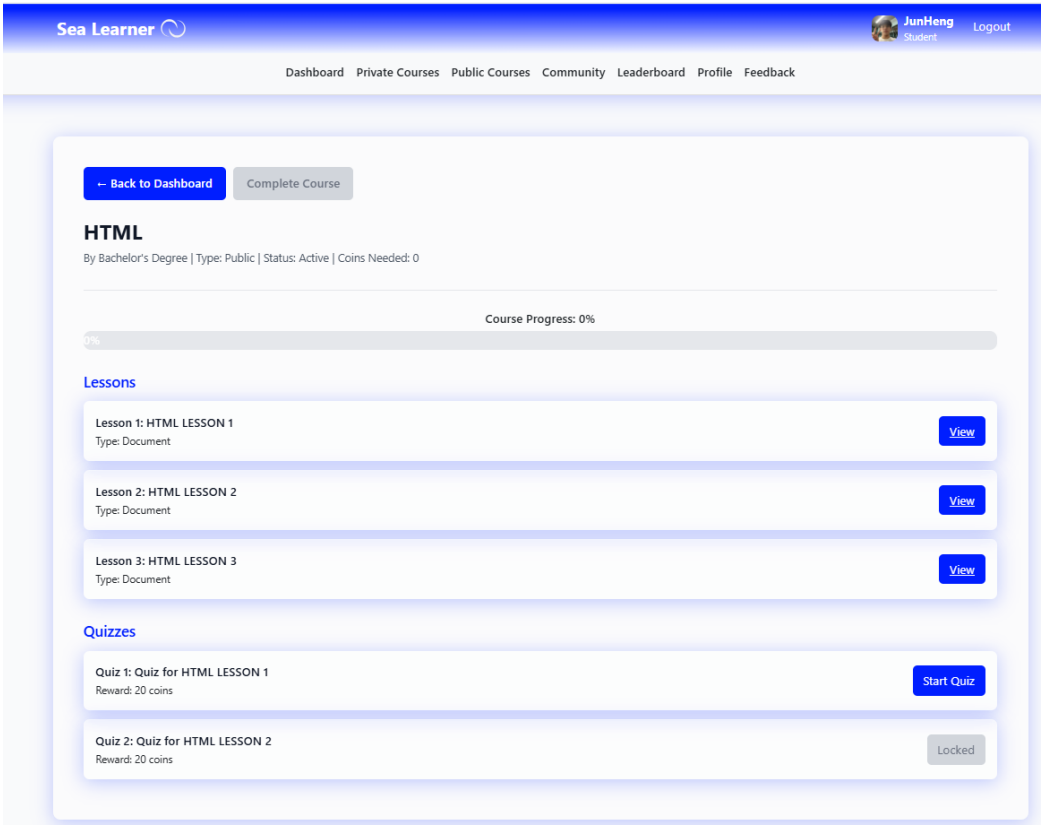

Figure 225: Student Course Page

This is the page that student press continue will go to this page, here mainly will display the course overview and the lesson and quizzes. At the lesson part student will need to press the view only will go to the content. At the quizzes part student may take the quiz by clicking the start quiz so after finish only the next quiz will be able to enter.

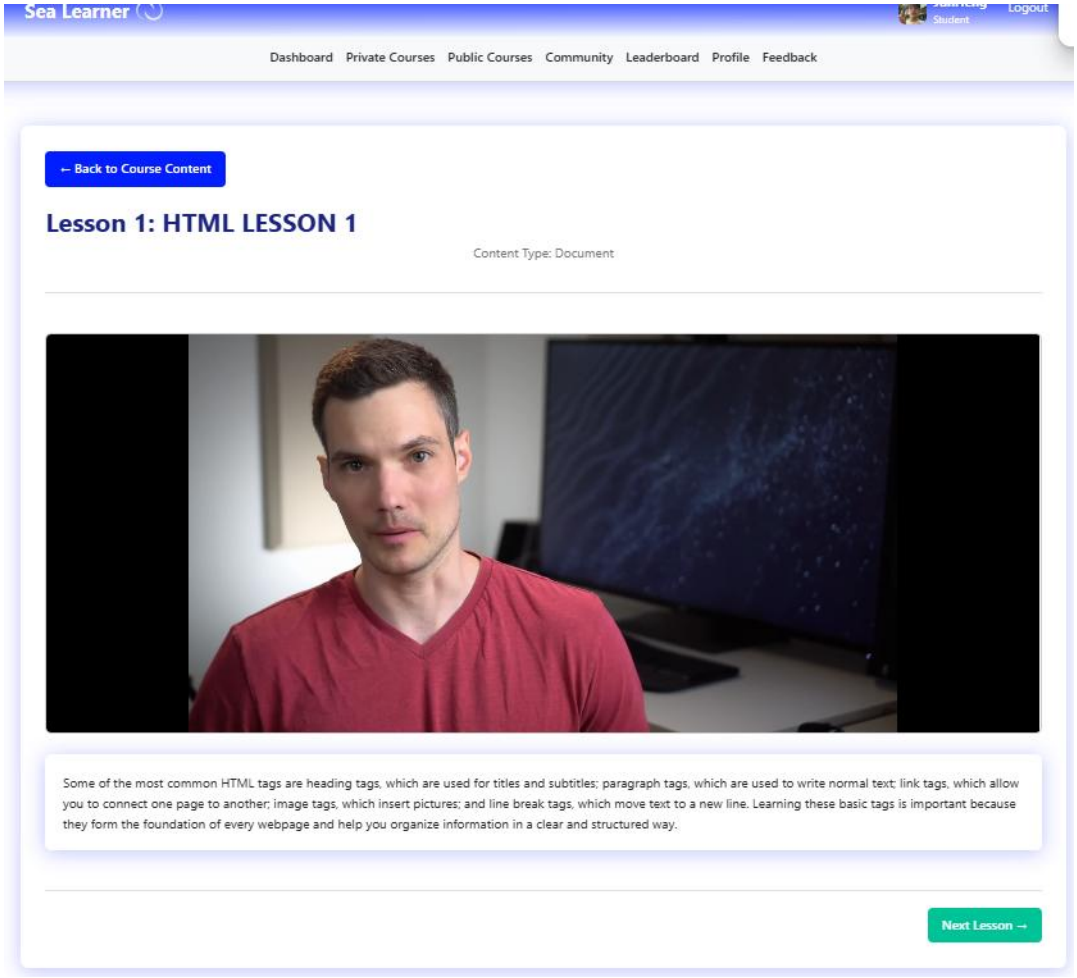

Figure 226: Student View Lesson Page

So this page is the enter lesson page student can see the content and next lesson by clicking the button. If user want to exit press the back button.

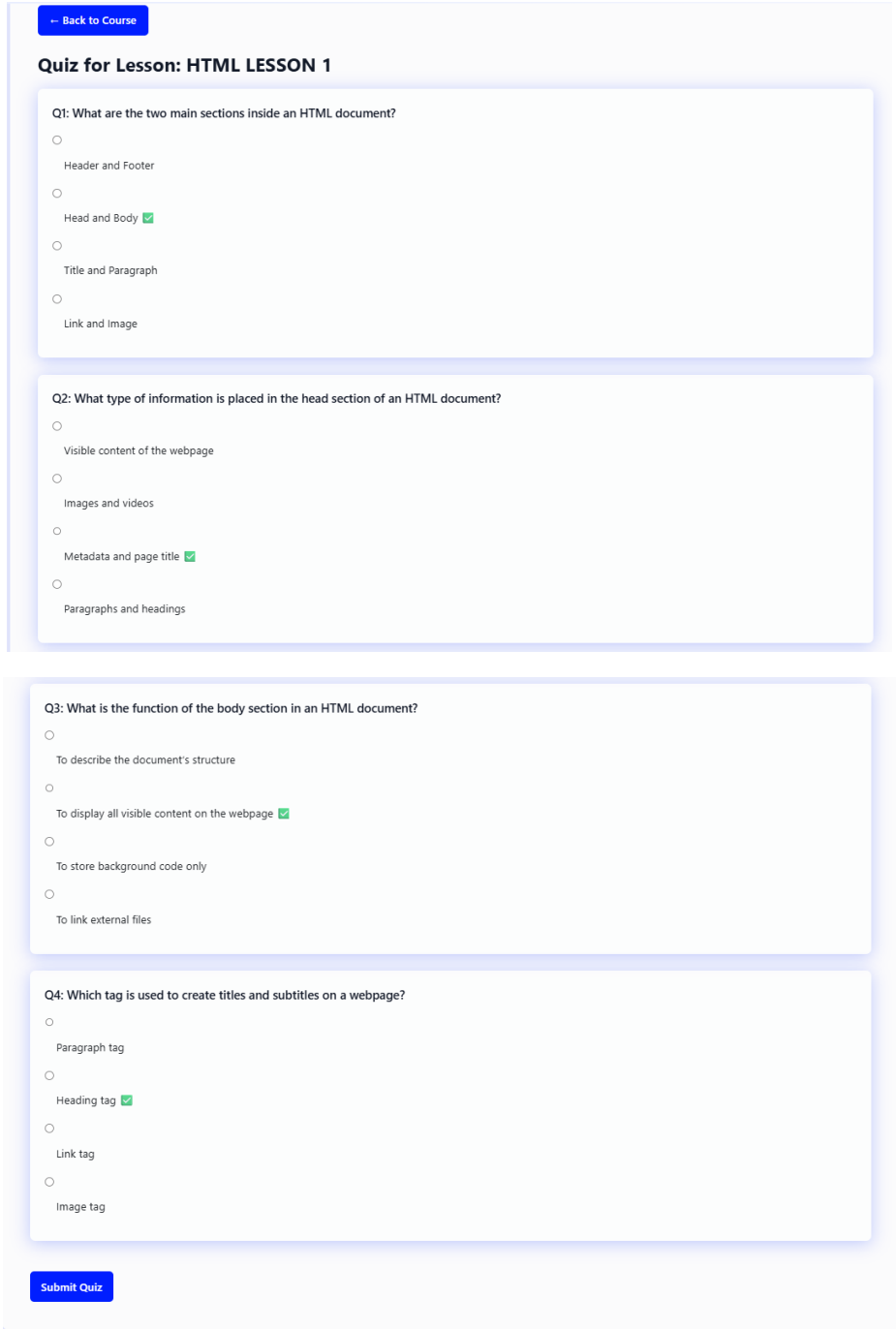

Figure 227: Student Quiz Page

In this page is the student take the quiz will enter this page so student may choose the option for their answer and then submit the quiz.

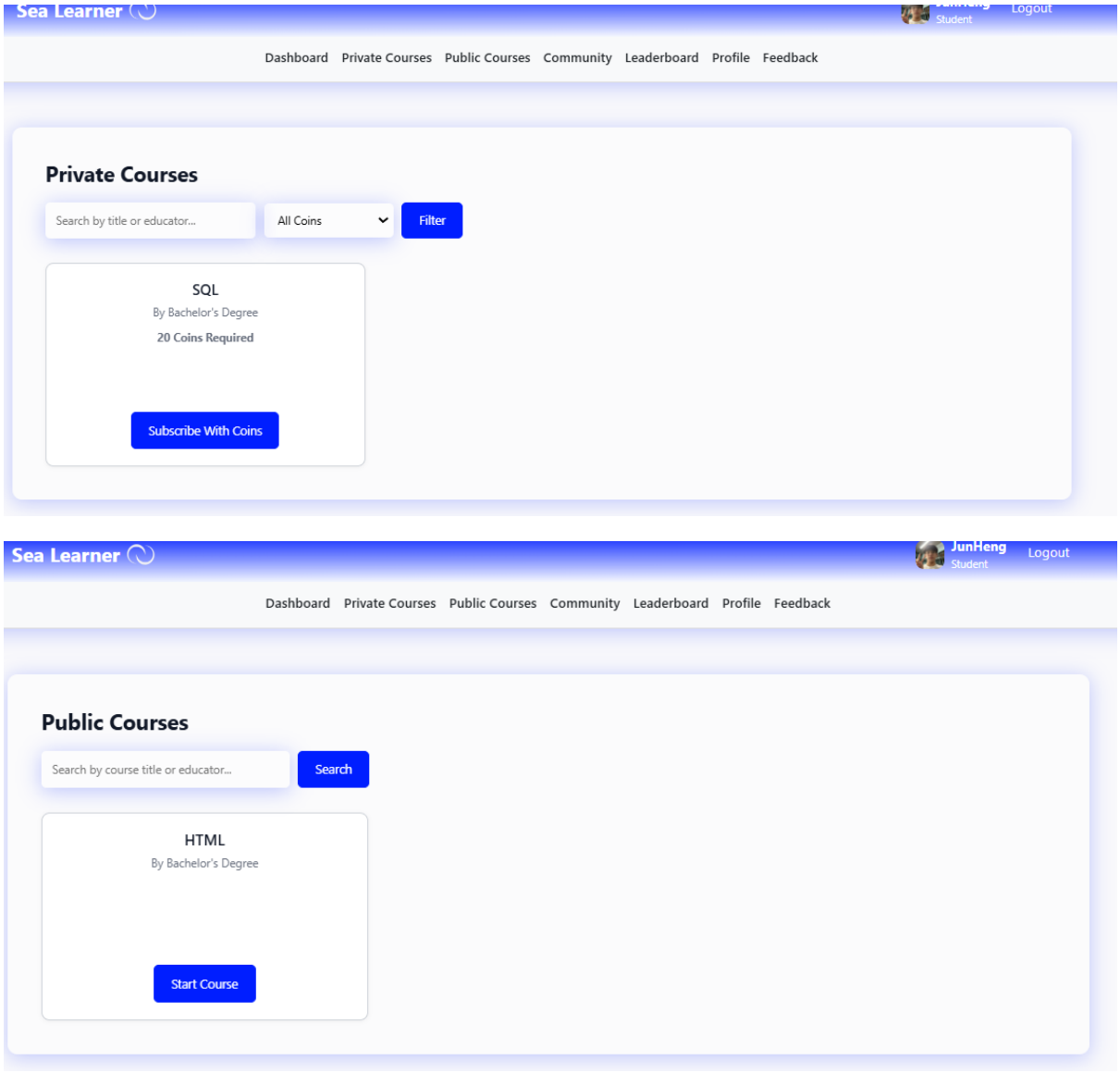

Figure 228: Student All Courses Page

Here will show all the courses private or public in each dashboard, so student can choose their course to start learning and will have a search button to search for the course easily and fast.

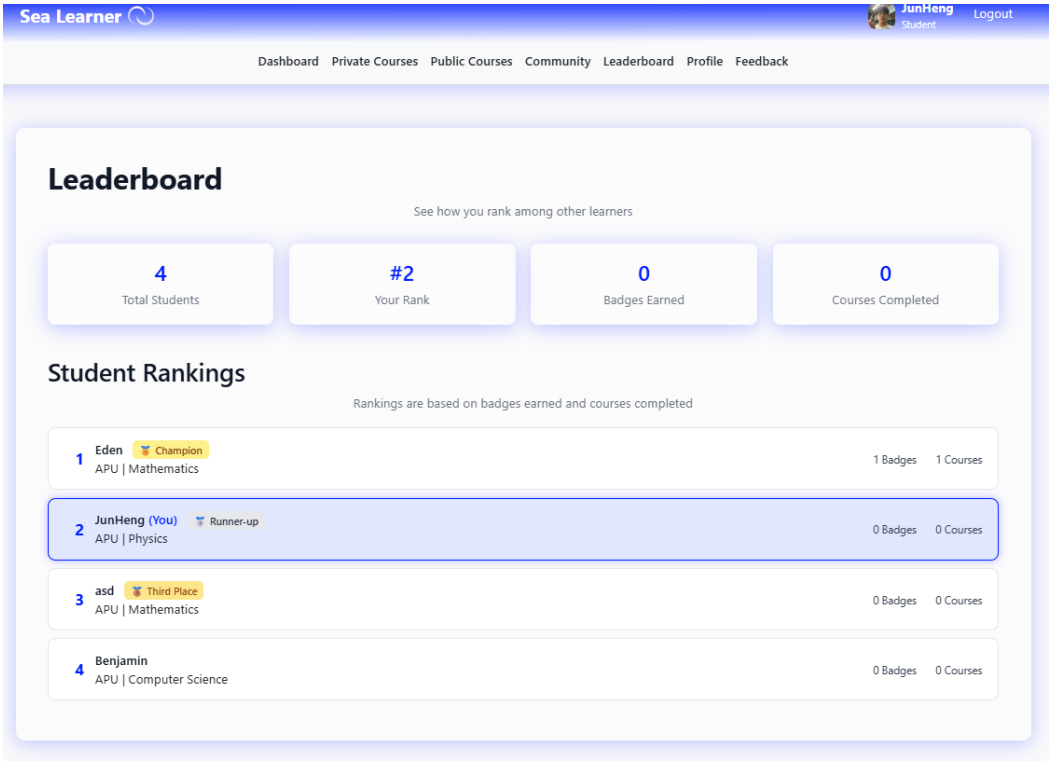

Figure 229: Student Leaderboard Page

This is the student leaderboard page that student can see their ranking according to the badges and how many courses that completed.

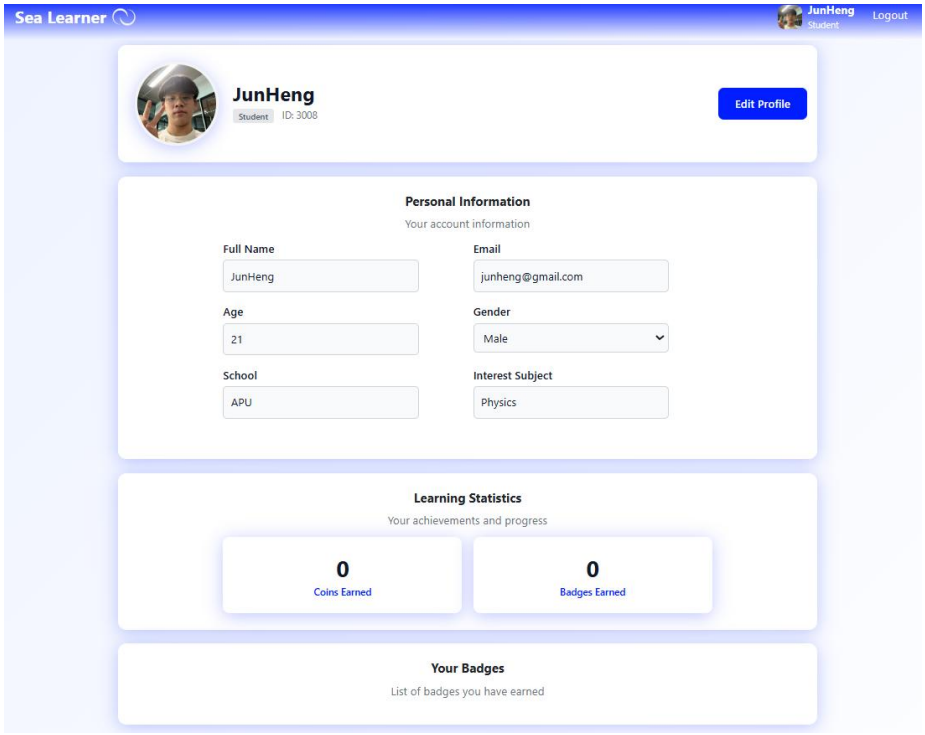

Figure 230: Student Profile Page

This student profile page is displaying student personal information, learning statistics and badges they awarded.

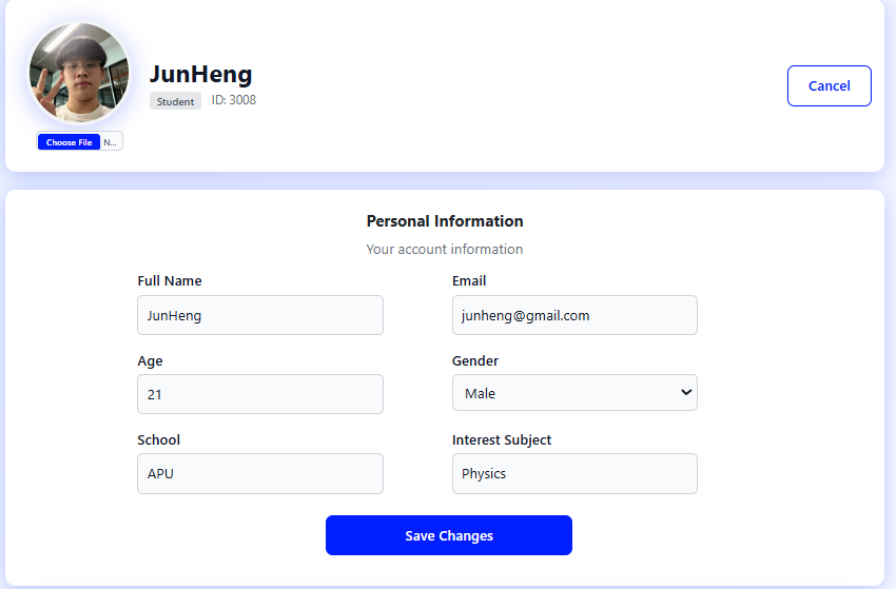

Figure 231: Student Edit Profile

So if students want to change their personal information can click the edit button and then start changing their information and then click on the save button so their information will be updated.

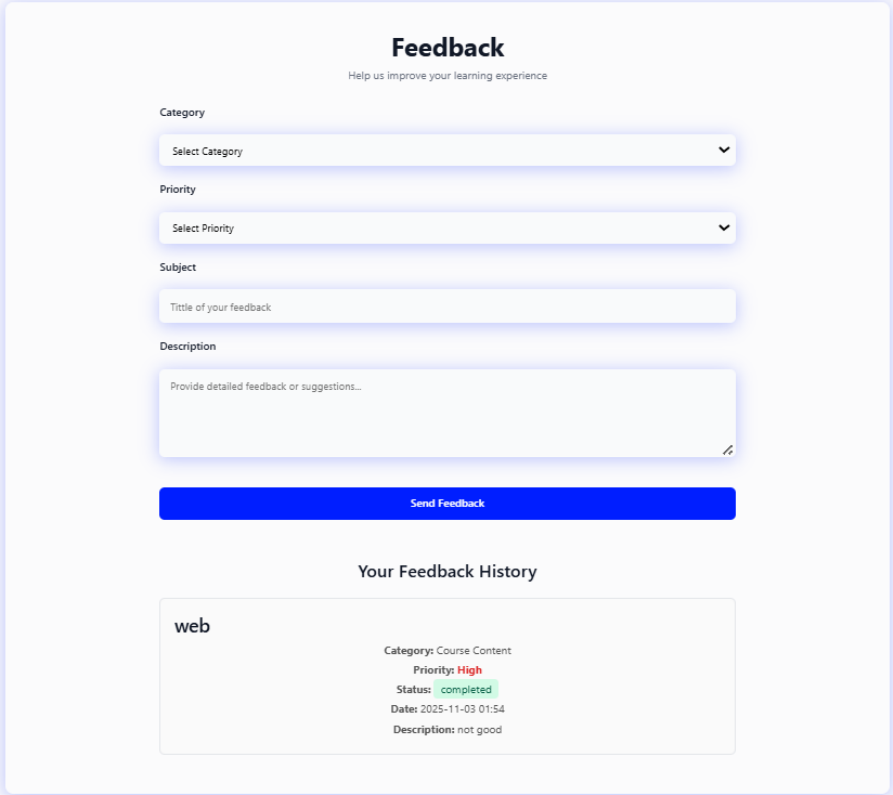

Figure 232: Student Give Feedback Page

Student can give feedback on feedback page. Select the category, priority, subject and write some description and then click the send feedback so that admin can receive it. Below can see their feedback history.

Educator

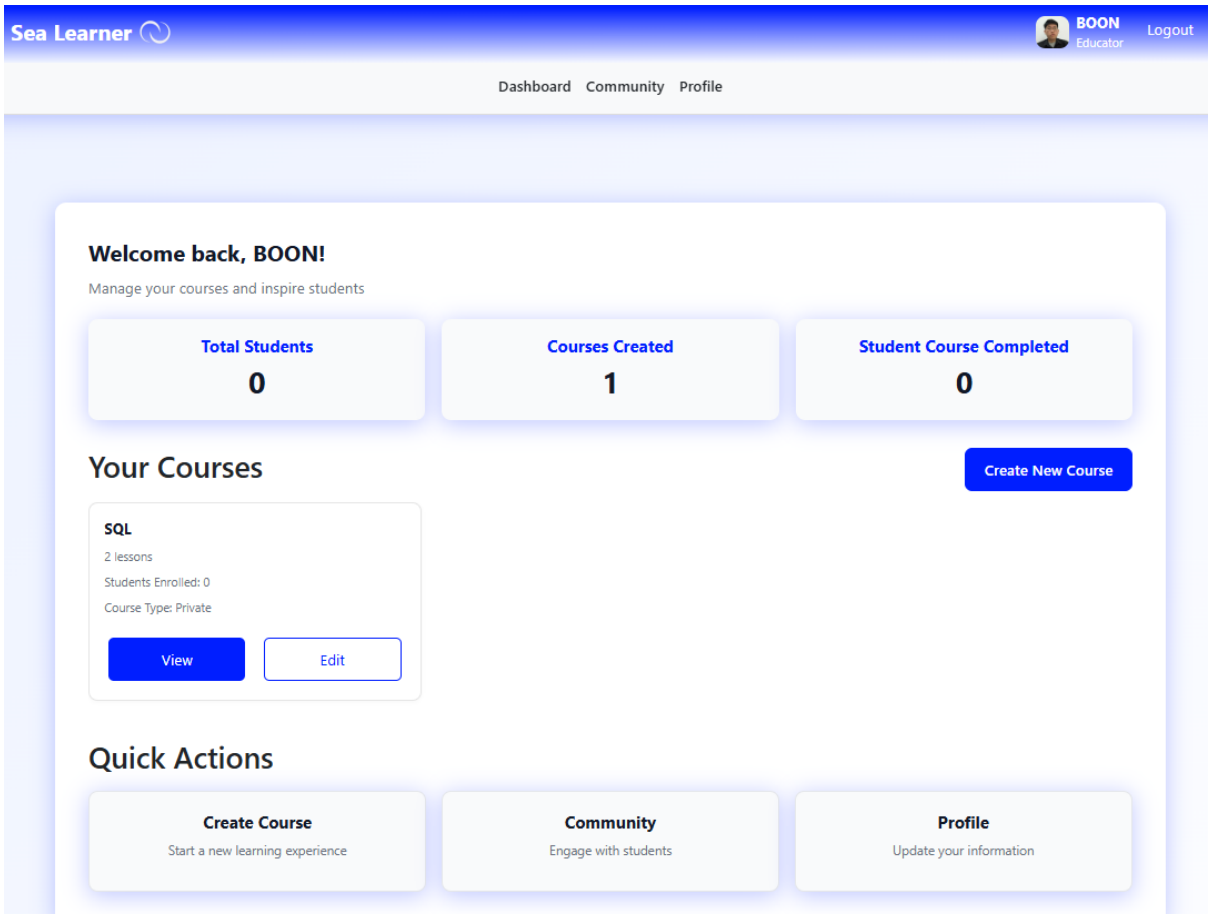

Figure 233: Educator Main Page

This is educator dashboard page, which is main page of the educator. After educator login, it will show the “Total Students”, “Courses Created”, and “Student Course Completed” that by the educator. Educator can click the “Create New Course” to create a course. Under “Your Courses” will show the courses that educator has created before, they can click the “View” to view student course progression, and the “Edit” button is for edit or update the current course. The bottom of the main page is “Quick Actions”, which educator can click then go to that page quickly. Educator able to participate in the community forum to ask question and reply to other user's question. Educator also able to update their profile.

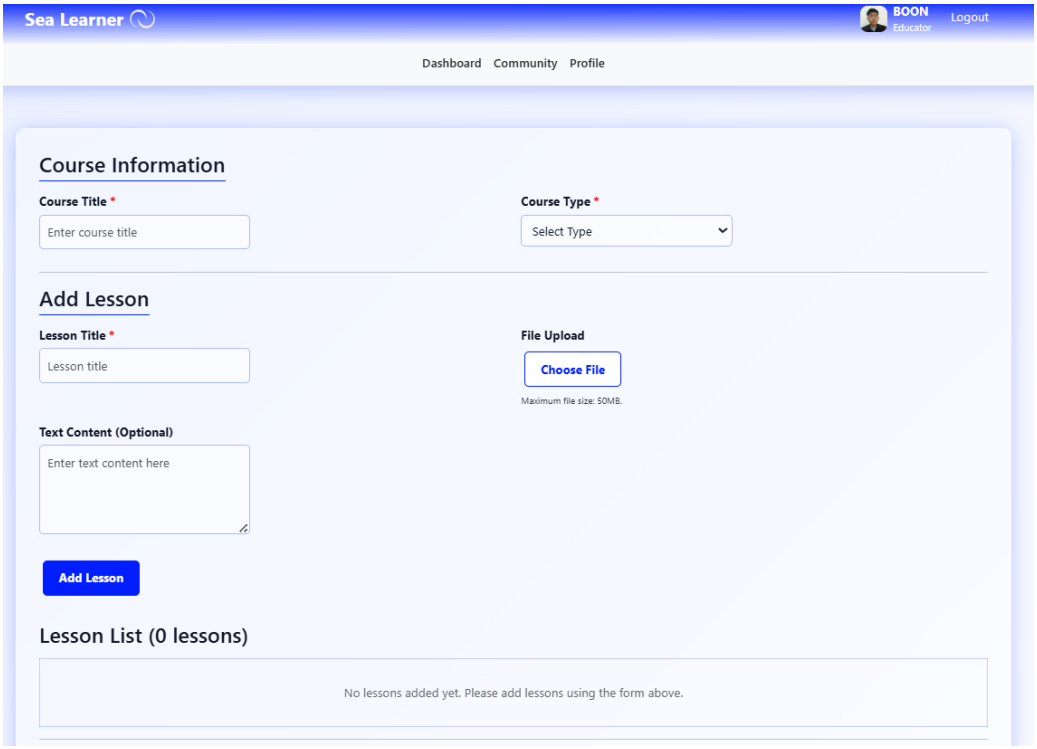

Figure 234: Educator Create Course Page (1)

In the create course page, to create a course educator must at least fill in the “Course Title”, “Course Type”, “Lesson Title”, and after that click the button “Add Lesson” then only can create the course. The “Course Type” contain two types which is “Public” and “Private”, if educator select the “Private”, an extra input is required “Course Coin” to set the course cost. After educator click the “Add Lesson” the according lesson will appear in the “Lesson List”, so that educator can make change if fill in wrongly. The “Add Lesson” section accept for uploading a file.

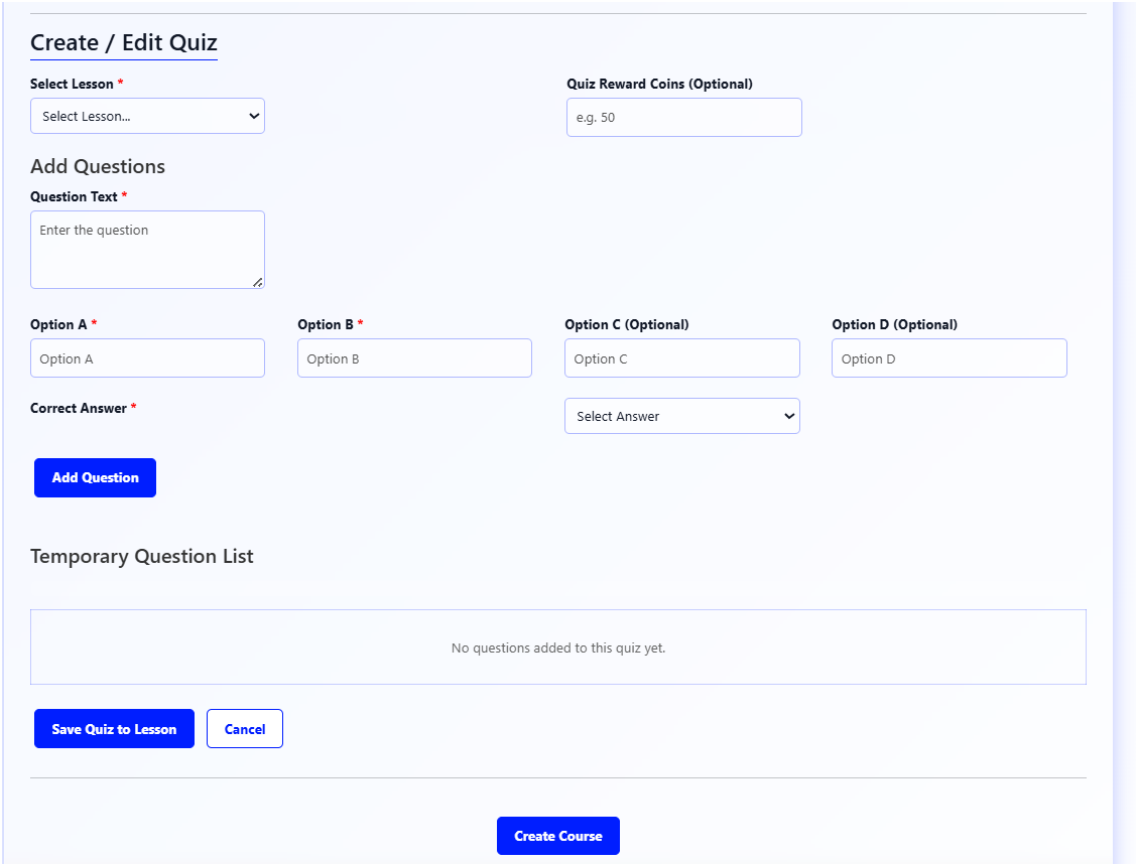

Figure 235: Educator Create Course Page (2)

After educator click the button “Add Leson”, they will be able to add question right now. Select the lesson for adding quiz under which lesson educator wants to add, for the “Quiz Reward Coins” is optional, which is when student complete it how much they will get. After that, enter the question in the “Question Text”, enter the “Option A” and “Option B”, for the “Option C” and “Option D” is optional, and define the correct answer, then click the “Add Question”. After clicked the “Add Question” the question will appear in the “Temporary Question List”, they can double check and edit if needed before clicking the “Save Quiz to Lesson”. Once everything is correct and checked, educator can now click the “Create Course” to publish the course.

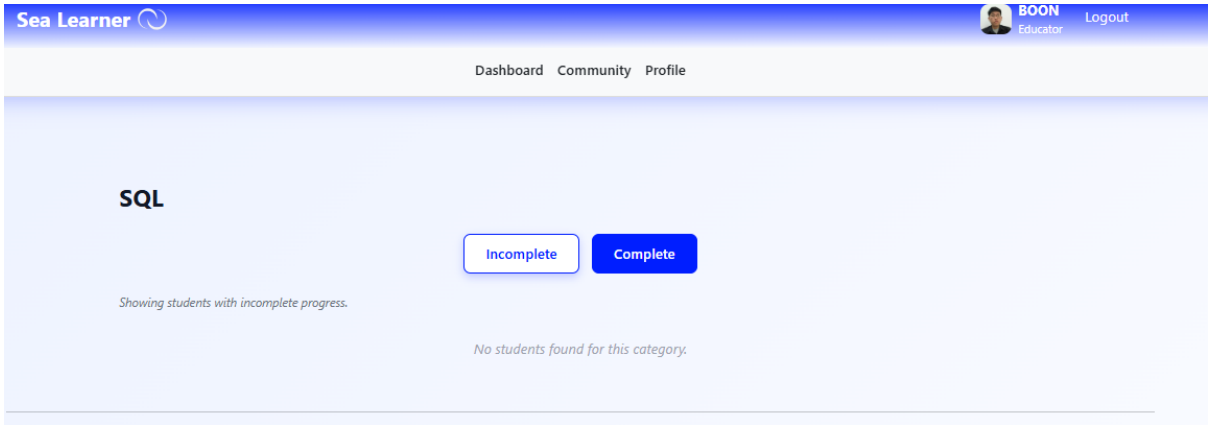

Figure 236: Educator View Student Progress

After clicking the "View" button, it will go to this page which is view student progression page. Educator can either click the "Incomplete" or "Complete" to see the specific student progression.

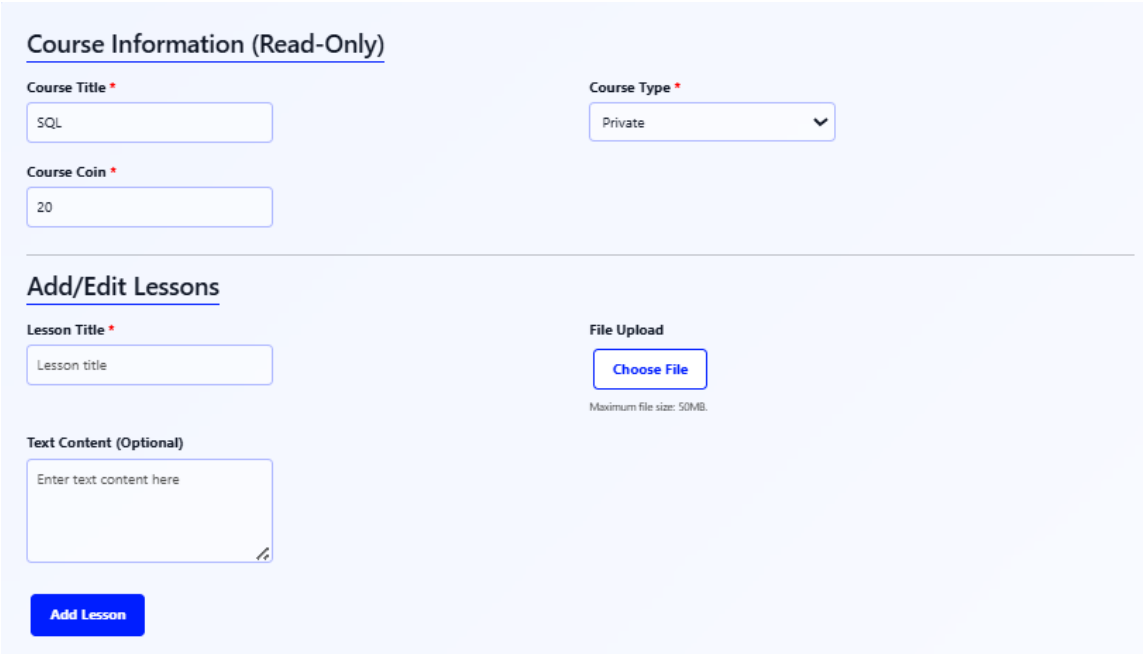

Figure 237: Educator Edit Course Page (1)

After user click the "Edit" button, it will go to this page, which is edit course page. This edit course page is same as the page "Create Course", only different is "Course Title" and "Course Type" cannot be changed.

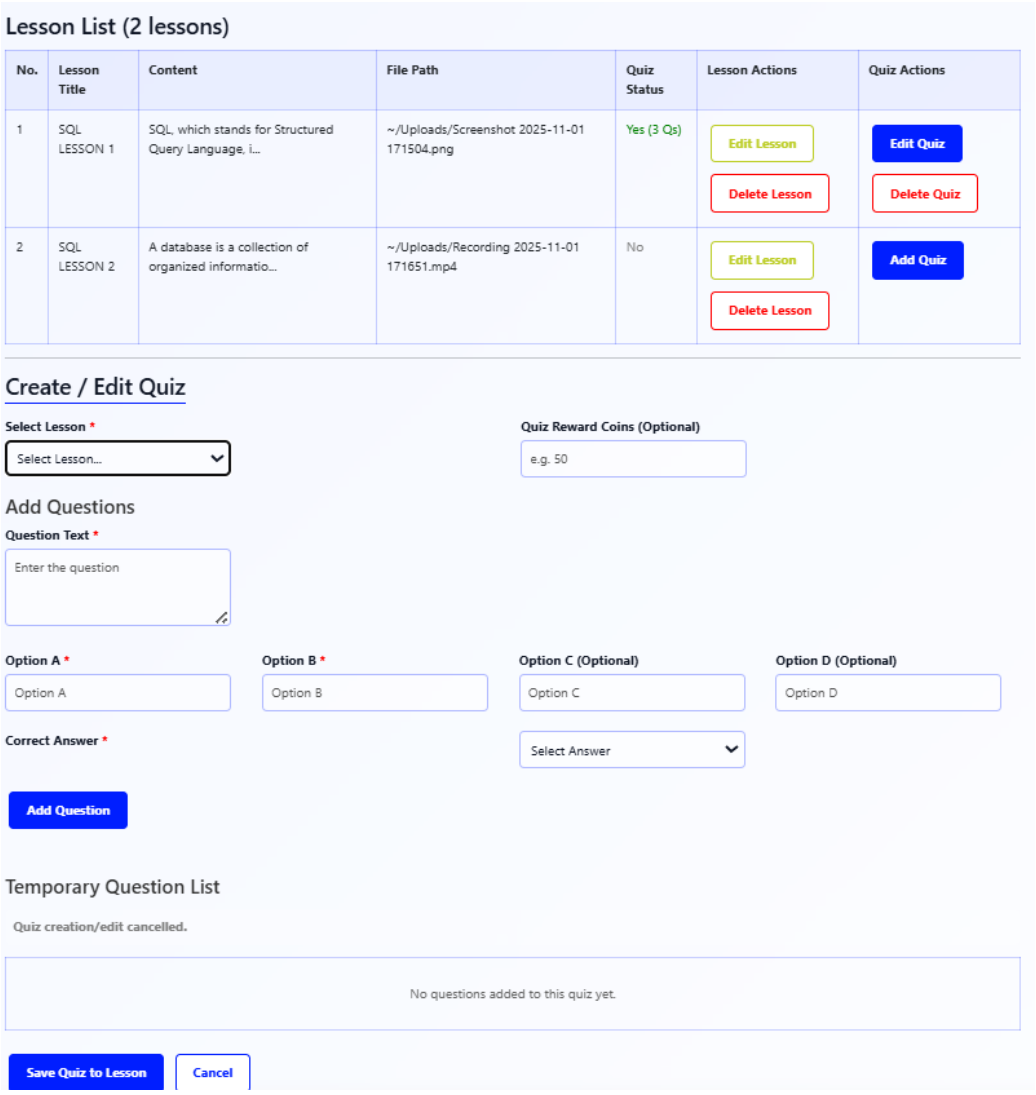

Figure 238: Educator Edit Course Page (2)

This section also same as the “Create Course” page, educator can edit or update the lesson and question accordingly.

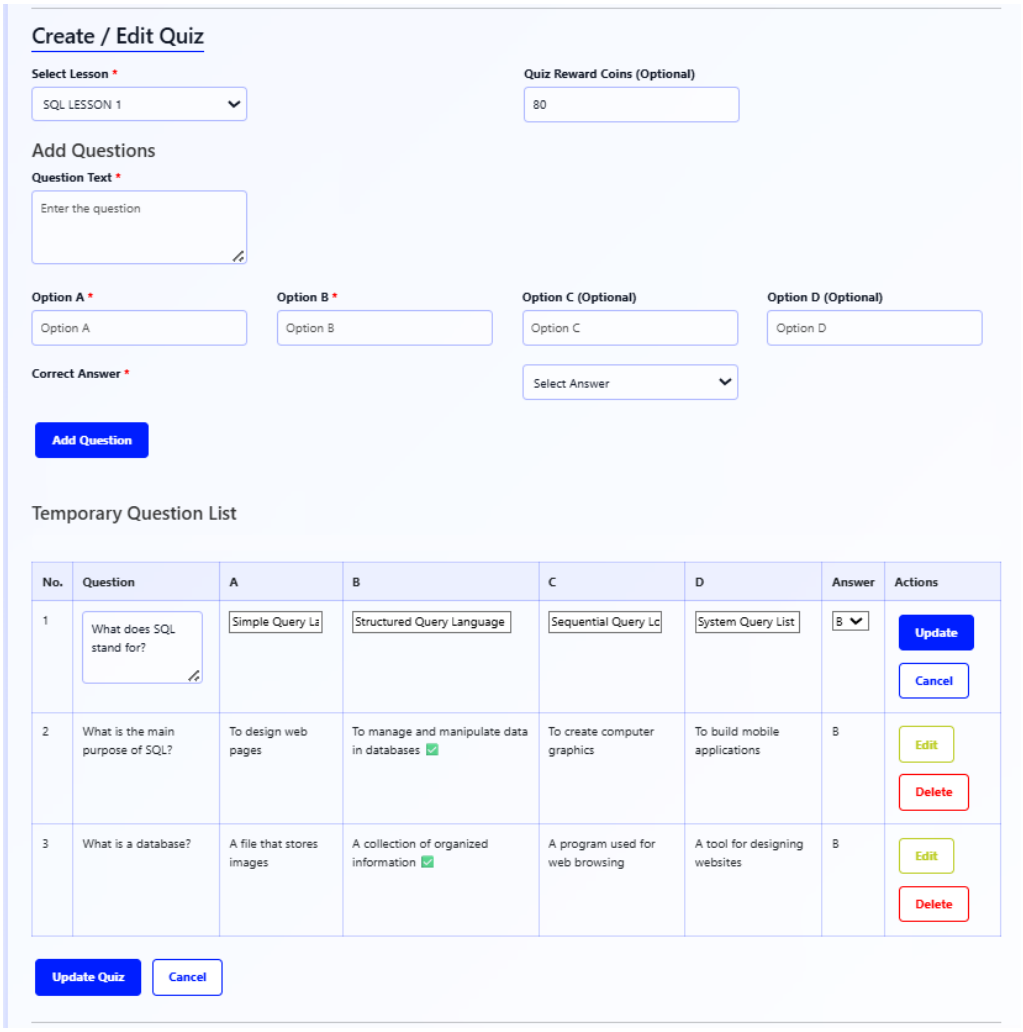

Figure 239 Educator Edit Course Page (3: Quiz)

This section also same as the “Create Course” page, educator can edit or update the lesson and question accordingly.

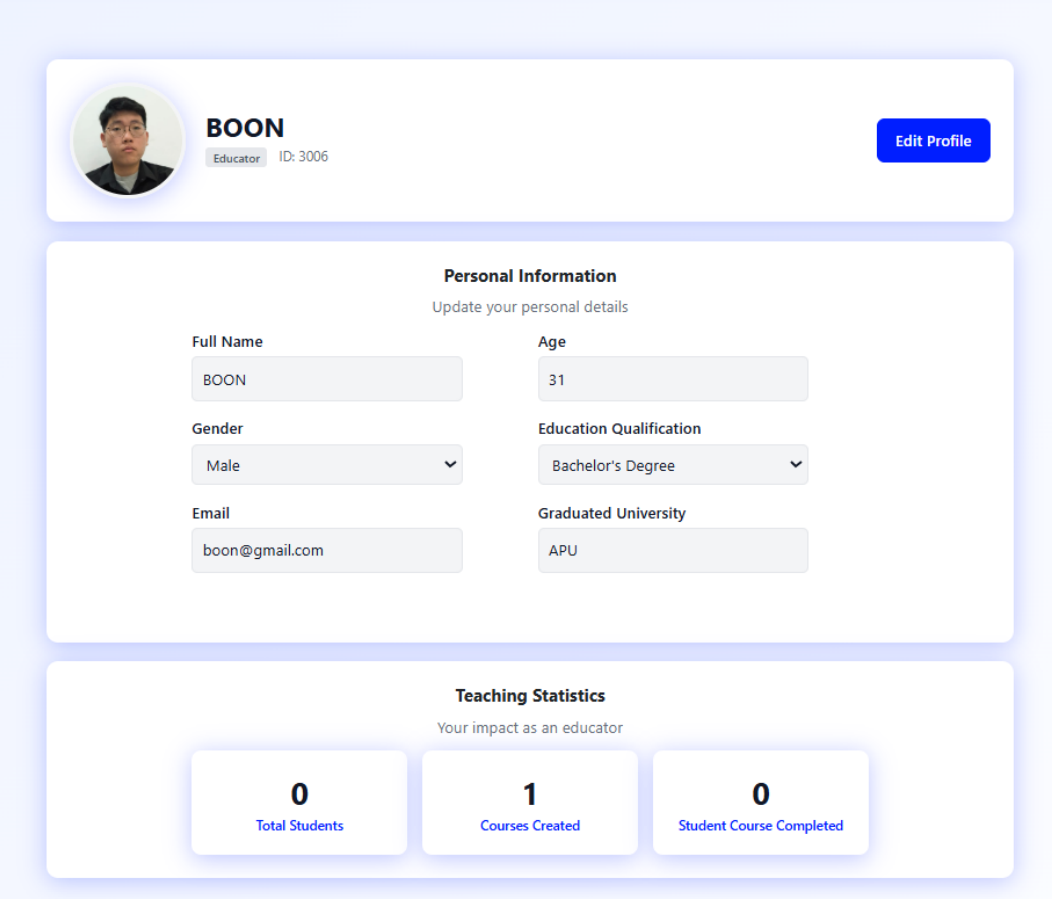

Figure 240: Educator Profile Page

Educator can edit the profile by clicking the “Profile”. Once click the “Edit Profile”, they can change their information accordingly, except for the “Email” is unchangeable, then once finish click the “Save Changes”. Bottom of the “Profile” is “Teaching Statistic”, which is same as the main page, this feature is to target the goal of user-friendly web site.

This is educator profile, so the personal information and the edit profile is same as the student. Only the statistic is different in educator side can view the total student they have, courses created and student course completed.

Community Forum (Student and Educator)

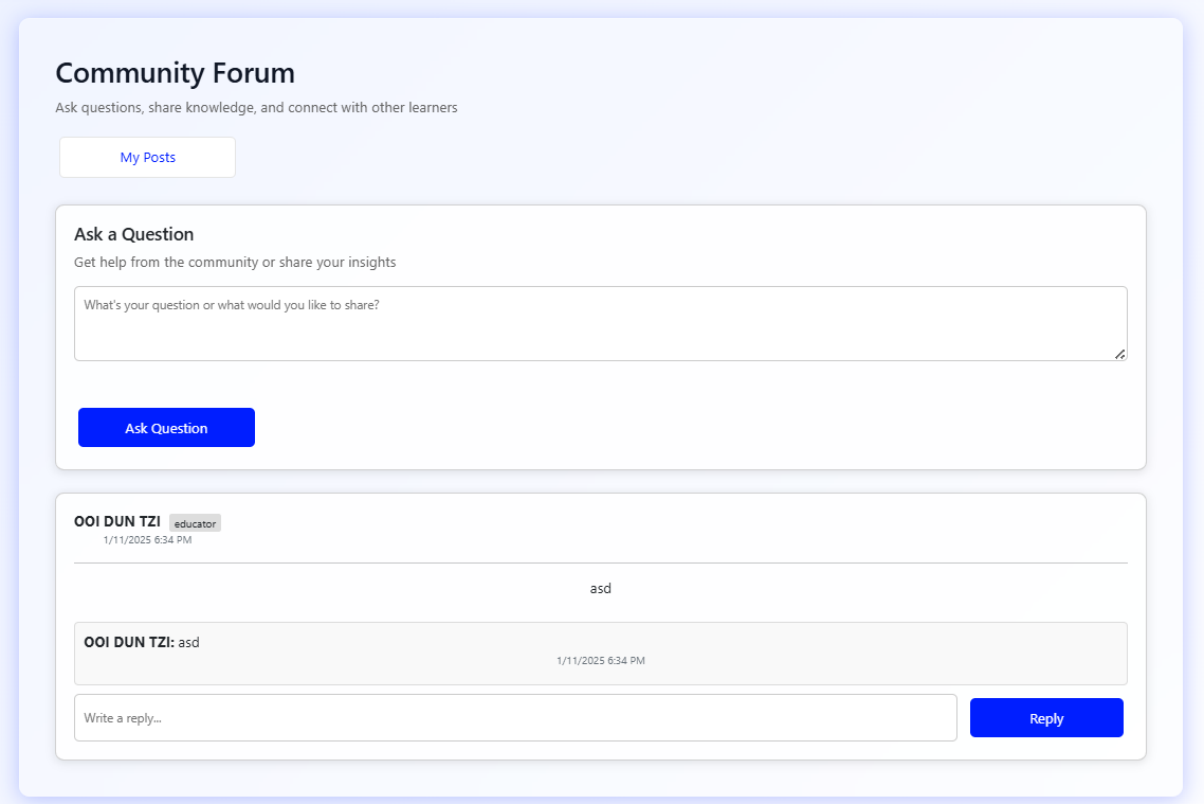

Figure 241: Community Forum Page

This page is “Community Forum” page, both of student and educator can participate to the community forum to ask a question and reply to other user’s question. They can delete their previous asked question through “My Posts”.

This Community forum is for student and educator to let them have a chat. User can add the question they want, view others user post and reply to their post.

6.0 Conclusion **CHEAH JUN HENG**

Sea Learner is a comprehensive web-based educational platform designed to function as a complete learning ecosystem, connecting administrators, educators, and students. The platform enables educators to create, manage, and publish multimedia-rich courses, while students can enrol in public or private courses, complete lessons and quizzes, and earn coins and badges through a gamified reward system. Administrators oversee platform integrity by managing users, moderating content, and handling feedback. Key features include a community forum for interaction, a leaderboard to foster competition, and role-based dashboards tailored to each user type. The project successfully integrates front-end design with back-end functionality, utilizing ASP.NET, SQL Server, and modern CSS techniques to deliver a responsive and engaging user experience.

Throughout the development of Sea Learner, the team gained valuable insights into full-stack web application development. Key lessons included the importance of role-based access control to ensure secure and organized user experiences, the effective use of SQL transactions to maintain database integrity during complex operations like course creation and deletion, and the implementation of responsive UI design using CSS features like glass morphism and dynamic form validation. Collaboration and version control were also critical in managing a multi-developer project, while iterative testing helped identify and resolve issues related to user session management, file uploads, and quiz functionality.

To further improve Sea Learner, several enhancements can be considered. These include integrating real-time chat or video conferencing features to support live interactions between educators and students, expanding gamification with more badge types and achievement levels, and introducing mobile app compatibility for on-the-go learning. Additional features such as course recommendations based on user behaviour, advanced analytics for educators and support for multiple languages would also enhance accessibility and engagement. Finally, incorporating AI-driven tools for automated feedback and content moderation could further streamline platform management and enrich the learning experience.

8.0 Appendix

8.1 Proposal Report

Proposal Report

Design Report

9.0 Workload Matrix

Content	OOI DUN TZI TP071308	YAP BOON SIONG TP070171	CHEAH JUN HENG TP071767	CHEN XIN ZE TP081210
1.0 Introduction	/			
2.0 Requirement Specification				/
3.0 Design and Modelling		/		
4.0 Implementation				
4.1 CSS for SeaLearner Page Styling	/			
4.2 Database Connectivity				
4.2.1 Insert	/			
4.2.2 Select				/
4.2.3 Update		/		
4.2.4 Delete			/	
4.3 Explanation Form Validation and Key Source Code Features				
4.3.1 Public/Core	/			
4.3.2 Student		/		
4.3.3 Educator				/
4.3.4 Admin			/	
5.0 User Guidance			/	
6.0 Conclusion			/	
Total Workload (%)	**25%**	**25%**	**25%**	**25%**