Bilkent University

Department Of Computer Engineering

CS-353 Database Systems

2024-2025 Spring

Group 10

Final Report

26.05.2025

Instructor: Özgür Ulusoy

Teaching Assistant: Hasan Alp Caferoğlu

Group Members:

Name	**ID**	**Section**
Emre Furkan Akyol	22103352	2
Cem Apaydın	21802270	1
Ayça Candan Ataç	22203501	2
İbrahim Çaycı	22103515	1
Mustafa Özkan İr	22103267	2

Table Of Contents

1. Brief Description of the Application

LearnHub is an online education and certification platform that supports multiple roles and essential functionalities for each of them. There are three types of users: students, instructors, and admins. Firstly, the system provides myriad functionalities to students, such as exploring and enrolling in various courses, viewing content and completing different types of tasks, tracking their progress, providing comments on the content, giving feedback on the courses, and obtaining certifications for successfully completed courses. Similarly, instructors can create courses, manage the course content, prepare exams, quizzes, and assignments, and view student evaluations about the course. Finally, the admins can approve the recently created courses, view various statistics about the platform with the help of general and range-specific reports, such as the most popular or most completed courses, monthly registration counts and enrollment statistics. The system also includes a financial aid program to grant students free access to paid courses. The users will be able to see an overview of their courses, certificates, feedback etc., in their profiles. Lastly, all the users will be notified of the events that require attention. The system automatically generates notifications for course status changes, financial aid updates, enrollment confirmations, course completions, feedback submissions, grading results, new student enrollments, and certificate awards, ensuring all user types stay informed throughout the learning process.

2. Contribution of Each Group Member

2.1. Emre Furkan Akyol

Contributed to non-functional requirements in the proposal report. Wrote half of the SQL queries, which are instructor-related SQL queries, in the design report. During the implementation, focused on the backend development of instructor operations. Also, implemented the complete notification logic with 6 different triggers, frontend, and backend. Built postman api calls for the instructor and for the notification backend. Contributed to resolving frontend components' GUI problems, such as fixing orientation in the Online Degrees Page. Checked the Docker integration and specified the configurations in the final report.

2.2. Cem Apaydın

Wrote the "why-how a database is going to be used" part in the proposal report. Generated some of the frontend before the design report and made the rest of the pages in Figma to create all mock-ups in the design report. Implemented almost all of the frontend and linked existing backend functions to the frontend. Added some missing functions to the backend for the linking and handled storing the role and user id of the currently logged-in user in the local storage for simplicity.

2.3. Ayça Candan Ataç

Helped with the design of the ER diagrams and wrote the table schemas in reports. Implemented the report generation and user deletion operations for admins on both the backend and the frontend. Implemented the register, login, logout, forgot password and change password functions for authentication and authorization. Designed separate profiles for instructors, admins and students. Helped fix some small problems on the frontend and database design; added triggers and views.

2.4. İbrahim Çaycı

Wrote a part of functional requirements in the proposal report. Wrote half of SQL queries of web app pages in the design report. Created APIs, triggers, and implemented tables for course management, including support for tasks, assignments, assessments, documents, and visual materials. Designed course, section, content (including task, document, visual material, assessment, assignment) creation APIs for instructor. Developed backend logic and APIs for critical student interactions such as enrollment, content submission, grading, and comment functionalities with interaction between tables. Implemented complex SQL views and triggers for maintaining order numbers, progress tracking, and instructor metadata like course count. Built detailed API endpoints for course content pages, including uncompleted content summaries and dynamic question retrieval. Contributed to student pages' backend part, such as home page,

my learning, online degrees APIs. Lastly, implemented frontend for grading and integrated with backend and managed styling and logic for documents,visual materials, optional questions.

2.5. Mustafa Özkan İr

Worked as a full-stack developer throughout the term project. Implemented the complete financial aid feature, including backend and frontend for instructors and students. Developed certificate functionalities for students. Built MyLearning page and MyCourses pages for instructors and students. Implemented Course Approvals backend and frontend for the admin users. Contributed to the core logic for enrollment and payment processes. Also worked on proper integration of the backend and frontend for various pages throughout the system. Additionally, initialized the backend and its connection to the database, also providing Docker setup for deployment and development.

3. Final E/R

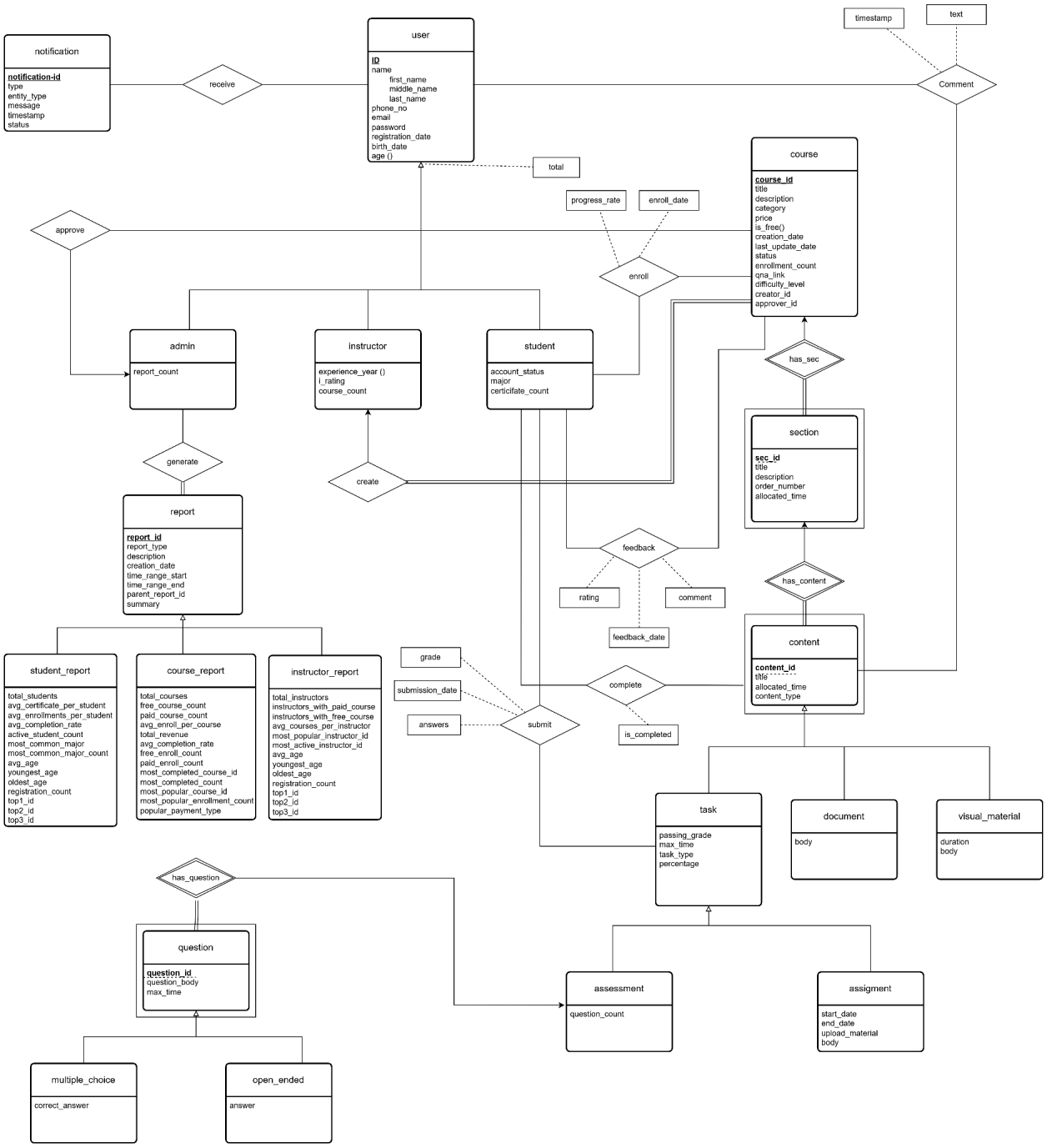
For high-resolution diagrams:

https://drive.google.com/file/d/1RqGTpdd39KD4W69ER-8BEV0qu-2jXui5/view?usp=sharing

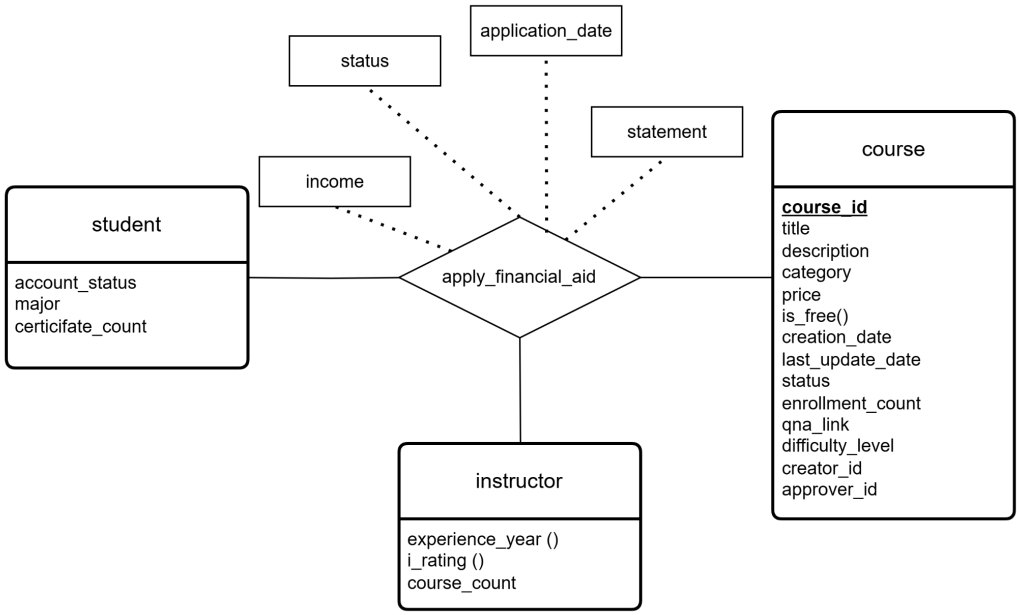
3.1 Changes to the E/R Diagram

- The report entity and its sub entities are updated to store our reports.
- The admin-report relation is converted to many to many from one to many because we do not store duplicate reports when two admins generate the same ranged report.
- Course entity is updated according to the new implementation.
- Financial Aid logic is simplified by using ternary relation instead of aggregation.

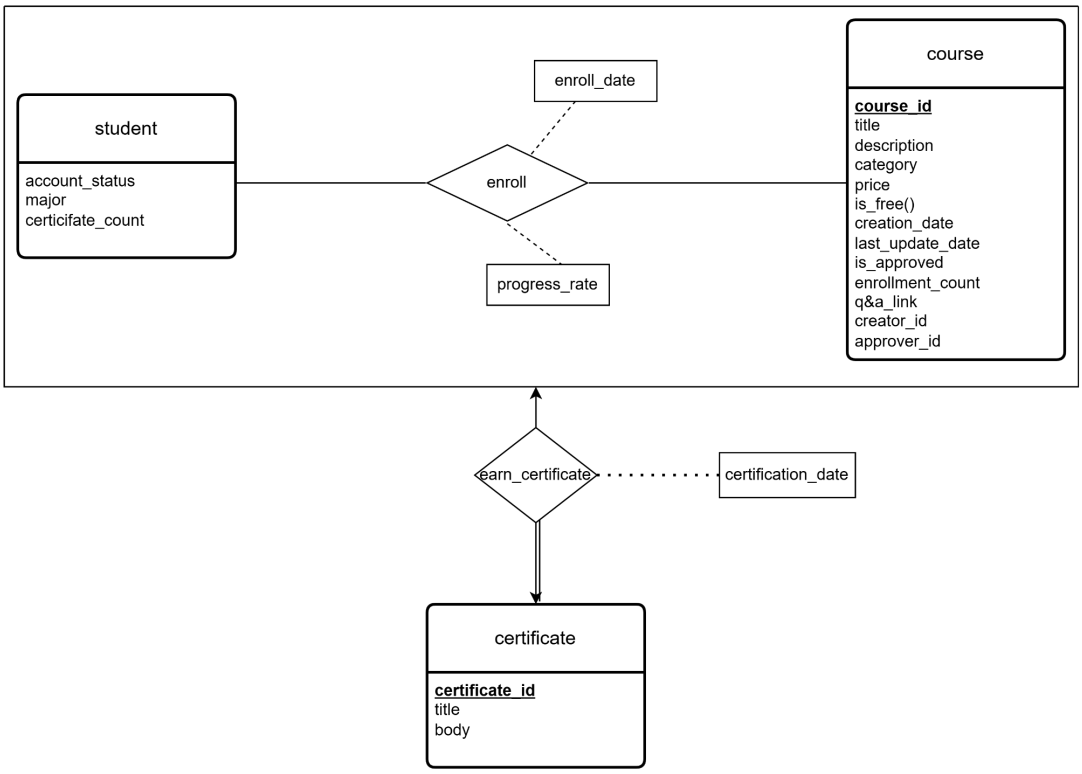
3.2 Main E/R Diagram



3.3 Financial Aid E/R Diagram



3.4 Certificate E/R Diagram



4. Final List of Tables

4.1. User

Relation: user(id, first_name, middle_name, last_name, phone_no, email, password, registration_date, birth_date, role)

Primary Key: id

Foreign Keys: —

Description: Stores general information and credentials for all users. The role field specifies whether a user is a student, instructor, or admin.

4.2. Student

Relation: student(id, major, account_status, certificate_count)

Primary Key: id

Foreign Keys: id → user(id)

Description: Contains additional student-specific attributes and tracks the number of certificates earned.

4.3. Instructor

Relation: instructor(id, i_rating, course_count)

Primary Key: id

Foreign Keys: id → user(id)

Description: Stores instructor-specific data such as average rating and the number of courses created.

4.4. Admin

Relation: admin(id, report_count)

Primary Key: id

Foreign Keys: id → user(id)

Description: Represents administrative users who oversee course approval and reporting functions.

4.5. Notification

Relation: notification(notification_id, type, entity_type, entity_id, message, timestamp, status)

Primary Key: notification_id

Foreign Keys: —

Description: Stores system-generated messages targeted to users or entities. Supports status tracking (e.g., unread, read, archived).

4.6. Receive

Relation: receive(notification_id, id, read_at)

Primary Key: (notification_id, id)

Foreign Keys: notification_id → notification(notification_id)

id → user(id)

Description: Tracks which users have received and read specific notifications.

4.7. Course

Relation: course(course_id, title, description, category, price, creation_date, last_update_date, status, enrollment_count, qna_link, difficulty_level, creator_id, approver_id)

Primary Key: course_id

Foreign Keys: creator_id → instructor(id)

approver_id → admin(id)

Description: Central entity for all educational content. Includes metadata and workflow status for admin approvals.

4.8. Section

Relation: section(course_id, sec_id, title, description, order_number, allocated_time)

Primary Key: (course_id, sec_id)

Foreign Keys: course_id → course(course_id)

Description: Courses are divided into sections with sequencing and time allocation.

4.9. Content

Relation: content(course_id, sec_id, content_id, title, order_number, allocated_time, content_type)

Primary Key: (course_id, sec_id, content_id)

Foreign Keys: (course_id, sec_id) → section(course_id, sec_id)

Description: Generic content entity with type-discrimination (e.g., task, document, visual_material).

4.10. Task

Relation: task(course_id, sec_id, content_id, passing_grade, max_time, task_type, percentage)

Primary Key: (course_id, sec_id, content_id)

Foreign Keys: (course_id, sec_id, content_id) → content(course_id, sec_id, content_id)

Description: Abstract superclass for assignment and assessment, defines grading criteria and duration.

4.11. Assessment

Relation: assessment(course_id, sec_id, content_id, question_count)

Primary Key: (course_id, sec_id, content_id)

Foreign Keys: (course_id, sec_id, content_id) → task(course_id, sec_id, content_id)

Description: Represents quizzes or exams with a fixed number of questions.

4.12. Assignment

Relation: assignment(course_id, sec_id, content_id, start_date, end_date, upload_material, body)

Primary Key: (course_id, sec_id, content_id)

Foreign Keys: (course_id, sec_id, content_id) → task(course_id, sec_id, content_id)

Description: Assignments with submission deadlines and file type restrictions.

4.13. Document

Relation: document(course_id, sec_id, content_id, body)

Primary Key: (course_id, sec_id, content_id)

Foreign Keys: (course_id, sec_id, content_id) → content(course_id, sec_id, content_id)

Description: Stores file-based instructional material.

4.14. Visual Material

Relation: visual_material(course_id, sec_id, content_id, duration, body)

Primary Key: (course_id, sec_id, content_id)

Foreign Keys: (course_id, sec_id, content_id) → content(course_id, sec_id, content_id)

Description: Stores video-based instructional material.

4.15. Question

Relation: question(course_id, sec_id, content_id, question_id, question_body, max_time)

Primary Key: (course_id, sec_id, content_id, question_id)

Foreign Keys: (course_id, sec_id, content_id) → assessment(course_id, sec_id, content_id)

Description: Table for both multiple-choice and open-ended questions with options.

4.16. Multiple Choice

Relation: multiple_choice(course_id, sec_id, content_id, question_id, correct_answer)

Primary Key: (course_id, sec_id, content_id, question_id)

Foreign Keys: (course_id, sec_id, content_id, question_id) → question(course_id, sec_id, content_id, question_id)

Description: Stores correct multiple-choice question answer using labeled options (A–E).

4.17. Open Ended

Relation: open_ended(course_id, sec_id, content_id, question_id, answer)

Primary Key: (course_id, sec_id, content_id, question_id)

Foreign Keys: (course_id, sec_id, content_id, question_id) → question(course_id, sec_id, content_id, question_id)

Description: Stores correct open-ended question answer.

4.18. Enroll

Relation: enroll(course_id, student_id, enroll_date, progress_rate)

Primary Key: (course_id, student_id)

Foreign Keys: course_id → course(course_id)

student_id → student(id)

Description: Tracks course enrollment, enrollment date and progress of students.

4.19. Submit

Relation: submit(course_id, sec_id, content_id, student_id, grade, submission_date, answers)

Primary Key: (course_id, sec_id, content_id, student_id)

Foreign Keys: (course_id, sec_id, content_id) → task(course_id, sec_id, content_id)

student_id → student(id)

Description: Records assignment or assessment submissions, including grades and responses.

4.20. Complete

Relation: complete(course_id, sec_id, content_id, student_id, is_completed)

Primary Key: (course_id, sec_id, content_id, student_id)

Foreign Keys: (course_id, sec_id, content_id) → content(course_id, sec_id, content_id)

student_id → student(id)

Description: Tracks whether a student has completed a piece of content.

4.21. Feedback

Relation: feedback(course_id, student_id, rating, comment, feedback_date)

Primary Key: (course_id, student_id)

Foreign Keys: course_id → course(course_id)

student_id → student(id)

Description: Students submit reviews and ratings for courses.

4.22. Comment

Relation: comment(course_id, sec_id, content_id, user_id, text, timestamp)

Primary Key: (course_id, sec_id, content_id, user_id, timestamp)

Foreign Keys: (course_id, sec_id, content_id) → content(course_id, sec_id, content_id)

user_id → user(id)

Description: Enables discussions on content with timestamped comments.

4.23. Apply Financial Aid

Relation: apply_financial_aid(course_id, student_id, income, statement, application_date, status, evaluator_id)

Primary Key: (course_id, student_id)

Foreign Keys: course_id → course(course_id)

student_id → student(id)

evaluator_id → instructor(id)

Description: Students apply for financial aid on paid courses. Instructors review and update status.

4.24. Certificate

Relation: certificate(certificate_id, title, body)

Primary Key: certificate_id

Foreign Keys: —

Description: Certificates awarded to students upon course completion.

4.25. Earn Certificate

Relation: earn_certificate(student_id, course_id, certificate_id, certification_date)

Primary Key: (student_id, course_id, certificate_id)

Foreign Keys: (student_id, course_id) → enroll(student_id, course_id)

certificate_id → certificate(certificate_id)

Description: Links students to earned certificates after meeting course criteria.

4.26. Report

Relation: report(report_id, report_type, description, creation_date, time_range_start, time_range_end, parent_report_id, summary)

Primary Key: report_id

Foreign Keys: parent_report_id → report(report_id)

Description: Base report entity supporting different types and time-scoped analytics.

4.27. Student Report

Relation: student_report(report_id, total_students, avg_certificate_per_student, avg_enrollments_per_student, avg_completion_rate, active_student_count, most_common_major, most_common_major_count, avg_age, youngest_age, oldest_age, monthly_reg_count, top1_id, top2_id, top3_id)

Primary Key: report_id

Foreign Keys: report_id → report(report_id)

top1_id, top2_id, top3_id → student(id)

Description: Aggregated statistics on students, including enrollment rates, majors, and top performers.

4.28. Instructor Report

Relation: instructor_report(report_id, total_instructors, instructors_with_paid_course, instructors_with_free_course, avg_courses_per_instructor, most_popular_instructor_id, most_active_instructor_id, avg_age, youngest_age, oldest_age, registration_count, top1_id, top2_id, top3_id)

Primary Key: report_id

Foreign Keys: report_id → report(report_id)

top1_id, top2_id, top3_id → instructor(id)

most_popular_instructor_id → instructor(id)

most_active_instructor_id → instructor(id)

Description: Instructor engagement and performance analytics.

4.29. Course Report

Relation: course_report(report_id, total_courses, free_course_count, paid_course_count, avg_enroll_per_course, total_revenue, avg_completion_rate, free_enroll_count, paid_enroll_count, most_completed_course_id, most_completed_count, most_popular_course_id, most_popular_enrollment_count, popular_payment_type, ext_stats)

Primary Key: report_id

Foreign Keys: report_id → report(report_id)

most_completed_course_id → course(course_id)

most_popular_course_id → course(course_id)

Description: Revenue, enrollments, and popularity metrics for courses.

4.30. Admin Report

Relation: admin_report(admin_id, report_id)

Primary Key: (admin_id, report_id)

Foreign Keys: admin_id → admin(id)

report_id → report(report_id)

Description: Tracks which reports were generated by which admin.

5. Implementation Details

5.1. Backend and Database Connection

Our application's backend architecture leveraged Flask, a minimalist yet powerful Python web framework that enabled us to rapidly develop comprehensive RESTful API endpoints. The framework's modular design supported the implementation of 18 distinct route modules, each handling specific business domains such as user authentication, course management, real-time notifications, financial aid processing, analytics reporting, assessment grading, and content delivery operations. This modular approach was achieved through Flask's Blueprint system, which organizes the code utilizing modules and facilitates independent module testing and maintenance.

PostgreSQL 14 serves as our primary database management system, chosen for its advanced capabilities, including detailed trigger mechanisms, support for different data structures, and high performance with complex multi-table queries. To connect the backend to the Database, first, we got the necessary credentials from the environment file, such as the DB name, user, and password. Then, we utilized the `psycopg2`-binary driver with two connection functions in db.py: one for administrative operations on the default PostgreSQL database,

connect_postgres_db(), and the other for application data operations on the project database, connect_project_db(). The database initialization process was automated using Docker, where a schema.sql script was integrated as part of the Docker Compose service to populate the database on startup.

5.2. Database Access and SQL Query Injection

Our SQL statement creation and execution followed a hierarchical pattern across all route modules using psycopg2's cursor-based approach. When handling requests, each route establishes a database connection through the connect_project_db() function, creates a cursor object for query execution, and constructs parameterized SQL statements. For example, when retrieving user notifications in the notification.py module, we create SELECT statements with JOIN operations across the notification and receive tables using parameterized placeholders (%s) for user ID and status filtering values. Query execution utilizes the cursor.execute() method with parameter binding, where SQL statements are prepared with placeholders and actual values are passed as a separate tuple or dictionary. After execution, results are retrieved using cursor.fetchall() or cursor.fetchone(), depending on the expected result sets.

5.3. Docker Container Integration

Docker containerization was orchestrated through a docker-compose.yml configuration that manages three interconnected services for the database, Flask, and the frontend. The compose file establishes service dependencies, ensuring the database achieves a healthy status before the backend starts, and the backend becomes available before the frontend initializes, preventing startup conditions. Database persistence is maintained in containers as our schema.sql file is automatically mounted and executed during PostgreSQL initialization through the /docker-entrypoint-initdb.d/ directory. Environment variables are managed through shared .env files across all services, with port mappings exposing the database on 5433, backend on 5001, and frontend on 3000, providing complete environment isolation.

5.4. Frontend Architecture and UI implementation

Our React-based GUI was prepared through a component-based architecture organized around educational workflows and user roles. The interface preparation involved creating specialized pages for each user type: student interfaces for course browsing and enrollment, instructor dashboards for course creation and management, and administrative panels for system oversight and reporting. Component styling utilizes dedicated CSS files that implement responsive design patterns with custom animations, loading states, and interactive elements. The GUI preparation process involved establishing service layer communication through dedicated

API modules (course.js, notification.js, auth.js) that abstract backend communication and provide interfaces for React components to consume data reliably.

5.5. Constraint Enforcement

Our constraint enforcement was maintained through a three-stage validation system spanning the database, backend, and frontend layers. At the database level, PostgreSQL `CHECK` constraints validate data ranges (course prices ≥ 0, difficulty levels 1-5) and enumerated values (course status, user roles), while foreign key constraints with `CASCADE` operations maintain referential integrity across our over 25 interconnected tables. Backend validation occurs within Flask routes, where required field checking, data type validation, and logic rule enforcement happen before database operations, returning structured error responses for validation failures. Frontend constraint enforcement provides immediate user feedback through React form validation with real-time input checking, contextual error messages, and submission prevention until all validation criteria are satisfied, ensuring data quality before transmission to backend services.

5.6. Challenges and Limitations

The project encountered several challenges, some of which are related to the implementation stage related including complex join operations between two entities and maintaining triggers across several cases. Some of which are related to the connection of backend and database, especially Docker Compose service coordination and inter-container communication.

Progress Rate Accuracy: Maintaining accurate completion percentages when course content changes dynamically necessitated implementing dual database triggers that recalculate progress both when students complete activities and when instructors modify course structures.

Docker Development Workflow: Database state persistence during development iterations was problematic, requiring implementation of environment-controlled database reset functionality (`RESET_DB`) to provide up-to-date testing environments for us.

Docker Compose Service Dependencies: Coordinating startup sequences between PostgreSQL, Flask backend, and React frontend services caused connection timing problems, particularly when backend services attempted database connections before PostgreSQL completed initialization, resulting in connection refused errors and application startup failures. This was mitigated by implementing comprehensive health check configurations in the docker-compose.yml file.

6. Advance Database Components

6.1. Views

Our web app has several SQL views to simplify access to derived and aggregate data. These views encapsulate logic for computed fields, such as recommendations and progress tracking.

6.1.1. User with Age

Definition: This view dynamically calculates the age of each user based on their birth date.
Query:

```
CREATE VIEW user_with_age AS
SELECT
    id,
    first_name,
    last_name,
    birth_date,
    EXTRACT(YEAR FROM CURRENT_DATE) - EXTRACT(YEAR FROM
birth_date) AS age
FROM "user";
```

6.1.2. Instructor with Experience Year

Definition: This view computes the number of years each instructor has been active on the platform, based on their registration date.
Query:

```
CREATE VIEW instructor_with_experience_year AS
SELECT
    i.ID,
    u.first_name,
    u.last_name,
    EXTRACT(YEAR FROM CURRENT_DATE) - EXTRACT(YEAR FROM
u.registration_date) AS experience_year
FROM instructor i
JOIN "user" u ON i.ID = u.ID;
```

6.1.3. Course with is Free

Definition: Simplifies the logic for identifying whether a course is free or paid.
Query:

```
CREATE VIEW course_with_is_free AS
```

```
SELECT
    course_id,
    title,
    price,
    CASE
        WHEN price = 0 THEN TRUE
        ELSE FALSE
    END AS is_free
FROM course;
```

6.1.4. Course Content Count

Definition: Aggregates the total and completed content per student per course, supporting progress tracking features and visualizations in student dashboards.

Query:

```
CREATE VIEW course_content_count AS
SELECT
    e.student_id,
    c.course_id,
    SUM(CASE WHEN ct.content_id IS NULL THEN 0 ELSE 1 END) AS
total_content_count,
    SUM(CASE WHEN cmp.is_completed = TRUE THEN 1 ELSE 0 END) AS
completed_content_count
FROM enroll e
LEFT JOIN course c ON e.course_id = c.course_id
LEFT JOIN section s ON s.course_id = c.course_id
LEFT JOIN content ct ON ct.course_id = s.course_id AND ct.sec_id
= s.sec_id
LEFT JOIN complete cmp
    ON cmp.course_id = ct.course_id
   AND cmp.sec_id = ct.sec_id
   AND cmp.content_id = ct.content_id
   AND cmp.student_id = e.student_id
   AND cmp.is_completed = TRUE
GROUP BY e.student_id, c.course_id;
```

6.1.5. Enrolled Course Categories

Definition: Provides a mapping between students and the categories of courses they are enrolled in.

Query:

```
CREATE VIEW enrolled_course_categories AS
SELECT e.student_id, c.category
FROM enroll e
JOIN course c ON e.course_id = c.course_id;
```

6.1.6. Recommended Course Base

Definition: Filters out only the accepted courses to form a base dataset for recommendation algorithms. This avoids recommending draft or rejected content.

Query:

```
CREATE VIEW recommended_course_base AS
SELECT c.course_id, c.title, c.category, c.difficulty_level,
c.enrollment_count
FROM course c
WHERE c.status = 'accepted';
```

6.1.7. Recommended Category Base

Definition: Counts the number of accepted courses per category. This supports recommending popular categories to students during course discovery.

Query:

```
CREATE VIEW recommended_category_base AS
SELECT category, COUNT(*) AS course_count
FROM course
WHERE status = 'accepted'
GROUP BY category;
```

6.2. Triggers

Our web app has several SQL triggers to change other table contents after a change in another table. These triggers are automatically launched when they are needed, and reduces work-load of programmers.

6.2.1. Enrollment Count Adjustment on Delete

Definition: When a student unenrolls from a course, this trigger automatically decreases the enrollment_count of the corresponding course in the course table.

Query:

```
CREATE OR REPLACE FUNCTION decrement_enrollment_count()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE course
    SET enrollment_count = enrollment_count - 1
    WHERE course_id = OLD.course_id;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_decrement_enrollment_count
AFTER DELETE ON enroll
FOR EACH ROW
EXECUTE FUNCTION decrement_enrollment_count();
```

6.2.2. Admin Report Count Tracker

Definition: Keeps track of the number of reports created or deleted by each admin by incrementing or decrementing the report_count in the admin table accordingly.

Query:

```
CREATE OR REPLACE FUNCTION update_admin_report_count()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        UPDATE admin
        SET report_count = report_count + 1
        WHERE id = NEW.admin_id;

    ELSIF TG_OP = 'DELETE' THEN
        UPDATE admin
        SET report_count = report_count - 1
        WHERE id = OLD.admin_id;
    END IF;

    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_admin_report_count
AFTER INSERT OR DELETE ON admin_report
```

```
FOR EACH ROW
EXECUTE FUNCTION update_admin_report_count();

ALTER TABLE admin_report
ADD CONSTRAINT uq_admin_report UNIQUE (admin_id, report_id);
```

6.2.3. Instructor Rating Updater

Definition: Ensures that the i_rating field in the instructor table reflects the average rating from feedback submitted for the instructor's courses.

Query:

```
CREATE OR REPLACE FUNCTION update_instructor_rating()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE instructor
    SET i_rating = (
        SELECT AVG(f.rating)
        FROM feedback f
        JOIN course c ON f.course_id = c.course_id
        WHERE c.creator_id = instructor.id
    )
    WHERE instructor.id = (
        SELECT c.creator_id
        FROM course c
        WHERE c.course_id = NEW.course_id
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_i_rating
AFTER INSERT ON feedback
FOR EACH ROW
EXECUTE FUNCTION update_instructor_rating();
```

6.2.4. Enrollment Count Increment on Insert

Definition: Each time a student enrolls in a course, this trigger increases the enrollment_count in the course table.

Query:

```
CREATE OR REPLACE FUNCTION update_enrollment_count()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE course
    SET enrollment_count = enrollment_count + 1
    WHERE course_id = NEW.course_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER enrollment_count_updater
AFTER INSERT ON enroll
FOR EACH ROW
EXECUTE FUNCTION update_enrollment_count();
```

6.2.5. Section Allocated Time Tracker

Definition: Automatically recalculates and updates the total allocated_time of a section whenever a new piece of content is added to it.

Query:

```
CREATE OR REPLACE FUNCTION update_section_allocated_time()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE section
    SET allocated_time = (
        SELECT SUM(c.allocated_time)
        FROM content c
        WHERE c.course_id = NEW.course_id AND c.sec_id =
NEW.sec_id
    )
    WHERE section.course_id = NEW.course_id
    AND section.sec_id = NEW.sec_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_section_allocated_time
AFTER INSERT ON content
```

```
FOR EACH ROW
EXECUTE FUNCTION update_section_allocated_time();
```

6.2.6. Progress Rate Updater on Content Completion

Definition: Automatically recalculates and updates the progress_rate in the enroll table whenever a student completes a content item. It ensures that the progress percentage reflects the current ratio of completed content to the total content in the course.

Query:

```
CREATE OR REPLACE FUNCTION update_progress_rate()
RETURNS TRIGGER AS $$
DECLARE
    total_count INTEGER;
    completed_count INTEGER;
BEGIN
    SELECT COUNT(*) INTO total_count
    FROM content
    WHERE course_id = NEW.course_id;

    SELECT COUNT(*) INTO completed_count
    FROM complete
    WHERE course_id = NEW.course_id AND student_id =
NEW.student_id AND is_completed = TRUE;

    UPDATE enroll
    SET progress_rate = CASE
        WHEN total_count = 0 THEN 0
        ELSE ROUND(100.0 * completed_count / total_count)
    END
    WHERE course_id = NEW.course_id AND student_id =
NEW.student_id;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_progress_rate
AFTER INSERT OR UPDATE ON complete
FOR EACH ROW
EXECUTE FUNCTION update_progress_rate();
```

6.2.7. Progress Rate Updater on Content Addition

Definition: When a new content item is added to a course, this trigger updates the progress_rate for all enrolled students in that course. It ensures the percentage reflects the new total content count, adjusting progress rates accordingly even if no new completions occur.

Query:

```
CREATE OR REPLACE FUNCTION update_progress_rate_on_content()
RETURNS TRIGGER AS $$
BEGIN
    -- Update progress_rate for all students enrolled in the
course
    UPDATE enroll
    SET progress_rate = CASE
        WHEN total.total_count = 0 THEN 0
        ELSE ROUND(100.0 * COALESCE(completed.completed_count,
0) / total.total_count)
    END
    FROM (
        SELECT course_id, COUNT(*) AS total_count
        FROM content
        WHERE course_id = NEW.course_id
        GROUP BY course_id
    ) AS total,
    (
        SELECT course_id, student_id, COUNT(*) AS
completed_count
        FROM complete
        WHERE course_id = NEW.course_id AND is_completed = TRUE
        GROUP BY course_id, student_id
    ) AS completed
    WHERE enroll.course_id = NEW.course_id
    AND enroll.course_id = total.course_id
    AND enroll.student_id = completed.student_id;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_progress_rate_on_content
```

```
AFTER INSERT ON content
FOR EACH ROW
EXECUTE FUNCTION update_progress_rate_on_content();
```

6.2.8. Instructor Course Count Updater

Definition: Increments the course_count field of an instructor whenever they create a new course. This ensures that the total number of authored courses by each instructor is kept accurate automatically.

Query:

```
CREATE OR REPLACE FUNCTION update_instructor_course_count()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE instructor
    SET course_count = course_count + 1
    WHERE id = NEW.creator_id;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_instructor_course_count
AFTER INSERT ON course
FOR EACH ROW
EXECUTE FUNCTION update_instructor_course_count();
```

6.2.9. Instructor Course Count Decrement on Course Deletion

Definition: Automatically decrements the `course_count` field of an instructor whenever one of their courses is deleted. This keeps the instructor's authored course tally accurate without requiring manual updates.

Query:

```
CREATE OR REPLACE FUNCTION decrement_instructor_course_count()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE instructor
    SET course_count = course_count - 1
    WHERE id = OLD.creator_id;
```

```
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_decrement_course_count
AFTER DELETE ON course
FOR EACH ROW
EXECUTE FUNCTION decrement_instructor_course_count();
```

6.2.10. Content Order Number Shifter

Definition: Ensures consistent ordering of content items within a section by shifting existing order_number values downward when a new content item is inserted at a specific position. This prevents order conflicts and maintains proper sequencing.

Query:

```
CREATE OR REPLACE FUNCTION shift_order_numbers()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE content
    SET order_number = order_number + 1
    WHERE course_id = NEW.course_id
      AND sec_id = NEW.sec_id
      AND order_number >= NEW.order_number;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_shift_order_numbers
BEFORE INSERT ON content
FOR EACH ROW
EXECUTE FUNCTION shift_order_numbers();
```

6.2.11. Section Order Number Shifter

Definition: Maintains proper sequencing of sections within a course by automatically shifting existing order_number values downward when a new section is inserted at a specific order position. This avoids overlaps.

Query:

```
CREATE OR REPLACE FUNCTION shift_section_order_numbers()
RETURNS TRIGGER AS $$
BEGIN
  UPDATE section
  SET order_number = order_number + 1
  WHERE course_id = NEW.course_id
    AND order_number >= NEW.order_number;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_shift_section_order
BEFORE INSERT ON section
FOR EACH ROW
EXECUTE FUNCTION shift_section_order_numbers();
```

6.2.12. Automatic Completion After Grading

Definition: Automatically marks a content item as completed for a student when a grade is assigned by the instructor.. This applies to tasks like assignments or assessments.

Query:

```
CREATE OR REPLACE FUNCTION mark_completion_on_grade()
RETURNS TRIGGER AS $$
BEGIN
  -- Only run logic if grade is newly set and is NOT NULL
  IF NEW.grade IS NOT NULL AND (OLD.grade IS NULL OR OLD.grade
IS DISTINCT FROM NEW.grade) THEN

    -- Try to update existing row
    UPDATE complete
    SET is_completed = TRUE
    WHERE course_id = NEW.course_id
      AND sec_id = NEW.sec_id
      AND content_id = NEW.content_id
      AND student_id = NEW.student_id;

    -- If no row was updated, insert a new one
    IF NOT FOUND THEN
```

```
      INSERT INTO complete (course_id, sec_id, content_id,
student_id, is_completed)
      VALUES (NEW.course_id, NEW.sec_id, NEW.content_id,
NEW.student_id, TRUE)
      ON CONFLICT (course_id, sec_id, content_id, student_id)
      DO UPDATE SET is_completed = TRUE;
    END IF;

  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_mark_completion_on_grade
AFTER UPDATE OF grade ON submit
FOR EACH ROW
EXECUTE FUNCTION mark_completion_on_grade();
```

6.2.13. Automatic Enrollment After Financial Aid Approval

Definition: Automatically enrolls a student in a course when their financial aid application is approved by the course instructor. Ensures that approved students are granted access to the course without requiring manual enrollment.

Query:

```
CREATE OR REPLACE FUNCTION enroll_on_financial_aid_approval()
RETURNS TRIGGER AS $$
BEGIN
    -- Only proceed if status is approved
    IF NEW.status = 'approved' THEN
        -- Insert into enroll if not already present
        INSERT INTO enroll (course_id, student_id, enroll_date,
progress_rate)
        VALUES (NEW.course_id, NEW.student_id, CURRENT_DATE, 0)
        ON CONFLICT (course_id, student_id) DO NOTHING;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trg_enroll_after_financial_aid_approval
AFTER INSERT OR UPDATE OF status
ON apply_financial_aid
FOR EACH ROW
EXECUTE FUNCTION enroll_on_financial_aid_approval();
```

6.2.14. Certificate Count Update On Certificate Creation

Definition: Increments the certificate count of a student whenever a new certificate is issued. This ensures the student's **certificate_count** field remains accurate and automatically reflects completed certifications.

Query:

```
CREATE OR REPLACE FUNCTION increment_certificate_count()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE student
    SET certificate_count = certificate_count + 1
    WHERE ID = NEW.student_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_certificate_count
AFTER INSERT ON earn_certificate
FOR EACH ROW
EXECUTE FUNCTION increment_certificate_count();
```

6.2.15. Certificate Count Update On Certificate Deletion

Definition: Decrements the **certificate_count** of students if a certificate is deleted.

Query:

```
CREATE OR REPLACE FUNCTION
decrement_certificate_count_on_certificate_delete()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE student
    SET certificate_count = certificate_count - 1
    WHERE ID IN (
        SELECT student_id
```

```
        FROM earn_certificate
        WHERE certificate_id = OLD.certificate_id
    );
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER handle_certificate_delete
BEFORE DELETE ON certificate
FOR EACH ROW
EXECUTE FUNCTION
decrement_certificate_count_on_certificate_delete();
```

6.2.16. Course Status Change Notifications

Definition: Generates notifications when a course status changes (draft→pending, pending→accepted/rejected). Notifies instructors about approval/rejection and admins about courses needing review.

Query:

```
CREATE OR REPLACE FUNCTION generate_financial_aid_notification()
RETURNS TRIGGER AS $$
DECLARE
    notify_id VARCHAR(8);
    course_title VARCHAR(150);
    student_name VARCHAR(150);
BEGIN
    -- Only trigger if status has changed
    IF OLD.status = NEW.status THEN
        RETURN NEW;
    END IF;

    -- Get course title and student name
    SELECT title INTO course_title FROM course WHERE course_id =
NEW.course_id;
    SELECT first_name || ' ' || last_name INTO student_name FROM
"user" WHERE id = NEW.student_id;

    notify_id := 'N' || SUBSTRING(MD5(RANDOM()::TEXT), 1, 7);

    IF NEW.status = 'approved' THEN
```

```
        INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
        VALUES (notify_id, 'financial_aid_approved', 'course',
NEW.course_id,
                'Your financial aid application for "' ||
course_title || '" has been approved!');
        INSERT INTO receive (notification_id, id) VALUES
(notify_id, NEW.student_id);

    ELSIF NEW.status = 'rejected' THEN
        INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
        VALUES (notify_id, 'financial_aid_rejected', 'course',
NEW.course_id,
                'Your financial aid application for "' ||
course_title || '" has been rejected.');
        INSERT INTO receive (notification_id, id) VALUES
(notify_id, NEW.student_id);

    ELSIF NEW.status = 'pending' THEN
        notify_id := 'N' || SUBSTRING(MD5(RANDOM()::TEXT), 1,
7);
        INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
        VALUES (notify_id, 'financial_aid_pending', 'course',
NEW.course_id,
                student_name || ' has applied for financial aid
for your course "' || course_title || '".');
        INSERT INTO receive (notification_id, id)
        SELECT notify_id, c.creator_id FROM course c WHERE
c.course_id = NEW.course_id;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_financial_aid_notification
AFTER INSERT OR UPDATE OF status ON apply_financial_aid
FOR EACH ROW
EXECUTE FUNCTION generate_financial_aid_notification();
```

6.2.17. Financial Aid Application Notifications

Definition: Notifies students about their financial aid application status (approved/rejected) and instructors about new applications requiring evaluation.

Query:

```
CREATE OR REPLACE FUNCTION generate_financial_aid_notification()
RETURNS TRIGGER AS $$
DECLARE
    notify_id VARCHAR(8);
    course_title VARCHAR(150);
    student_name VARCHAR(150);
BEGIN
    -- Only trigger if status has changed
    IF OLD.status = NEW.status THEN
        RETURN NEW;
    END IF;

    -- Get course title and student name
    SELECT title INTO course_title FROM course WHERE course_id =
NEW.course_id;
    SELECT first_name || ' ' || last_name INTO student_name FROM
"user" WHERE id = NEW.student_id;

    notify_id := 'N' || SUBSTRING(MD5(RANDOM()::TEXT), 1, 7);

    IF NEW.status = 'approved' THEN
        INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
        VALUES (notify_id, 'financial_aid_approved', 'course',
NEW.course_id,
                'Your financial aid application for "' ||
course_title || '" has been approved!');
        INSERT INTO receive (notification_id, id) VALUES
(notify_id, NEW.student_id);

    ELSIF NEW.status = 'rejected' THEN
        INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
```

```
        VALUES (notify_id, 'financial_aid_rejected', 'course',
NEW.course_id,
                'Your financial aid application for "' ||
course_title || '" has been rejected.');
        INSERT INTO receive (notification_id, id) VALUES
(notify_id, NEW.student_id);

    ELSIF NEW.status = 'pending' THEN
        notify_id := 'N' || SUBSTRING(MD5(RANDOM()::TEXT), 1,
7);
        INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
        VALUES (notify_id, 'financial_aid_pending', 'course',
NEW.course_id,
                student_name || ' has applied for financial aid
for your course "' || course_title || '".');
        INSERT INTO receive (notification_id, id)
        SELECT notify_id, c.creator_id FROM course c WHERE
c.course_id = NEW.course_id;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_financial_aid_notification
AFTER INSERT OR UPDATE OF status ON apply_financial_aid
FOR EACH ROW
EXECUTE FUNCTION generate_financial_aid_notification();
```

6.2.18. Student Enrollment Notifications

Definition: Notifies both students and instructors when a new enrollment occurs. Students receive confirmation and instructors are informed about new students in their courses.

Query:

```
CREATE OR REPLACE FUNCTION generate_enrollment_notification()
RETURNS TRIGGER AS $$
DECLARE
    notify_id VARCHAR(8);
    course_title VARCHAR(150);
```

```
    student_name VARCHAR(150);
    instructor_id VARCHAR(8);
BEGIN
    -- Get course details
    SELECT title, creator_id INTO course_title, instructor_id
    FROM course WHERE course_id = NEW.course_id;

    -- Get student name
    SELECT first_name || ' ' || last_name INTO student_name
    FROM "user" WHERE id = NEW.student_id;

    -- Notification for student
    notify_id := 'N' || SUBSTRING(MD5(RANDOM()::TEXT), 1, 7);
    INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
    VALUES (notify_id, 'enrollment_success', 'course',
NEW.course_id,
            'You have successfully enrolled in "' ||
course_title || '". You can start learning now!');
    INSERT INTO receive (notification_id, id) VALUES (notify_id,
NEW.student_id);

    -- Notification for instructor
    notify_id := 'N' || SUBSTRING(MD5(RANDOM()::TEXT), 1, 7);
    INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
    VALUES (notify_id, 'new_student', 'course', NEW.course_id,
            student_name || ' has enrolled in your course "' ||
course_title || '".');
    INSERT INTO receive (notification_id, id) VALUES (notify_id,
instructor_id);

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_enrollment_notification
AFTER INSERT ON enroll
FOR EACH ROW
EXECUTE FUNCTION generate_enrollment_notification();
```

6.2.19. Course Completion Notifications

Definition: Notifies both students and instructors when a student completes a course (progress rate reaches 100%). Encourages feedback and celebrates achievement.

Query:

```
CREATE OR REPLACE FUNCTION
generate_course_completion_notification()
RETURNS TRIGGER AS $$
DECLARE
    notify_id VARCHAR(8);
    course_title VARCHAR(150);
    student_name VARCHAR(150);
    instructor_id VARCHAR(8);
BEGIN
    -- Only trigger if progress_rate updated to 100
    IF OLD.progress_rate >= 100 OR NEW.progress_rate < 100 THEN
        RETURN NEW;
    END IF;

    -- Get course details
    SELECT title, creator_id INTO course_title, instructor_id
    FROM course WHERE course_id = NEW.course_id;

    -- Get student name
    SELECT first_name || ' ' || last_name INTO student_name
    FROM "user" WHERE id = NEW.student_id;

    -- Notification for student
    notify_id := 'N' || SUBSTRING(MD5(RANDOM()::TEXT), 1, 7);
    INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
    VALUES (notify_id, 'course_completed', 'course',
NEW.course_id,
            'Congratulations! You have completed "' ||
course_title || '". Please share your feedback!');
    INSERT INTO receive (notification_id, id) VALUES (notify_id,
NEW.student_id);

    -- Notification for instructor
    notify_id := 'N' || SUBSTRING(MD5(RANDOM()::TEXT), 1, 7);
```

```
    INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
    VALUES (notify_id, 'student_completed_course', 'course',
NEW.course_id,
            student_name || ' has completed your course "' ||
course_title || '".');
    INSERT INTO receive (notification_id, id) VALUES (notify_id,
instructor_id);

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_course_completion_notification
AFTER UPDATE OF progress_rate ON enroll
FOR EACH ROW
WHEN (NEW.progress_rate = 100)
EXECUTE FUNCTION generate_course_completion_notification();
```

6.2.20. Feedback Submission Notifications

Definition: Notifies instructors when students submit feedback/ratings for their courses, including the rating value for assessment.

Query:

```
CREATE OR REPLACE FUNCTION generate_feedback_notification()
RETURNS TRIGGER AS $$
DECLARE
    notify_id VARCHAR(8);
    course_title VARCHAR(150);
    student_name VARCHAR(150);
    instructor_id VARCHAR(8);
BEGIN
    -- Get course details
    SELECT title, creator_id INTO course_title, instructor_id
    FROM course WHERE course_id = NEW.course_id;

    -- Get student name
    SELECT first_name || ' ' || last_name INTO student_name
    FROM "user" WHERE id = NEW.student_id;
```

```
    -- Notification for instructor
    notify_id := 'N' || SUBSTRING(MD5(RANDOM()::TEXT), 1, 7);
    INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
    VALUES (notify_id, 'new_feedback', 'course', NEW.course_id,
            student_name || ' has left a ' || NEW.rating ||
'-star feedback for your course "' || course_title || '".');
    INSERT INTO receive (notification_id, id) VALUES (notify_id,
instructor_id);

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_feedback_notification
AFTER INSERT ON feedback
FOR EACH ROW
EXECUTE FUNCTION generate_feedback_notification();
```

6.2.21. Assignment Grading Notifications

Definition: Notifies students when their assignments/assessments are graded, indicating whether they passed or failed based on the passing grade threshold.

Query:

```
CREATE OR REPLACE FUNCTION generate_grade_notification()
RETURNS TRIGGER AS $$
DECLARE
    notify_id VARCHAR(8);
    content_title VARCHAR(150);
    course_title VARCHAR(150);
    passing_grade INTEGER;
BEGIN
    -- Only trigger if grade is being added/updated (not NULL)
    IF NEW.grade IS NULL THEN
        RETURN NEW;
    END IF;

    -- Get content and course details
    SELECT c.title, course.title, t.passing_grade
    INTO content_title, course_title, passing_grade
```

```
    FROM content c
    JOIN course ON c.course_id = course.course_id
    JOIN task t ON c.content_id = t.content_id AND c.course_id =
t.course_id AND c.sec_id = t.sec_id
    WHERE c.content_id = NEW.content_id AND c.course_id =
NEW.course_id AND c.sec_id = NEW.sec_id;

    notify_id := 'N' || SUBSTRING(MD5(RANDOM()::TEXT), 1, 7);

    -- Create notification based on grade
    IF NEW.grade >= passing_grade THEN
        INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
        VALUES (notify_id, 'assignment_passed', 'content',
NEW.content_id,
                'You passed "' || content_title || '" in the
course "' || course_title || '" with a grade of ' || NEW.grade
|| '.');
    ELSE
        INSERT INTO notification (notification_id, type,
entity_type, entity_id, message)
        VALUES (notify_id, 'assignment_failed', 'content',
NEW.content_id,
                'You did not pass "' || content_title || '" in
the course "' || course_title || '". Your grade: ' || NEW.grade
|| '.');
    END IF;

    INSERT INTO receive (notification_id, id) VALUES (notify_id,
NEW.student_id);

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_grade_notification
AFTER INSERT OR UPDATE OF grade ON submit
FOR EACH ROW
EXECUTE FUNCTION generate_grade_notification();
```

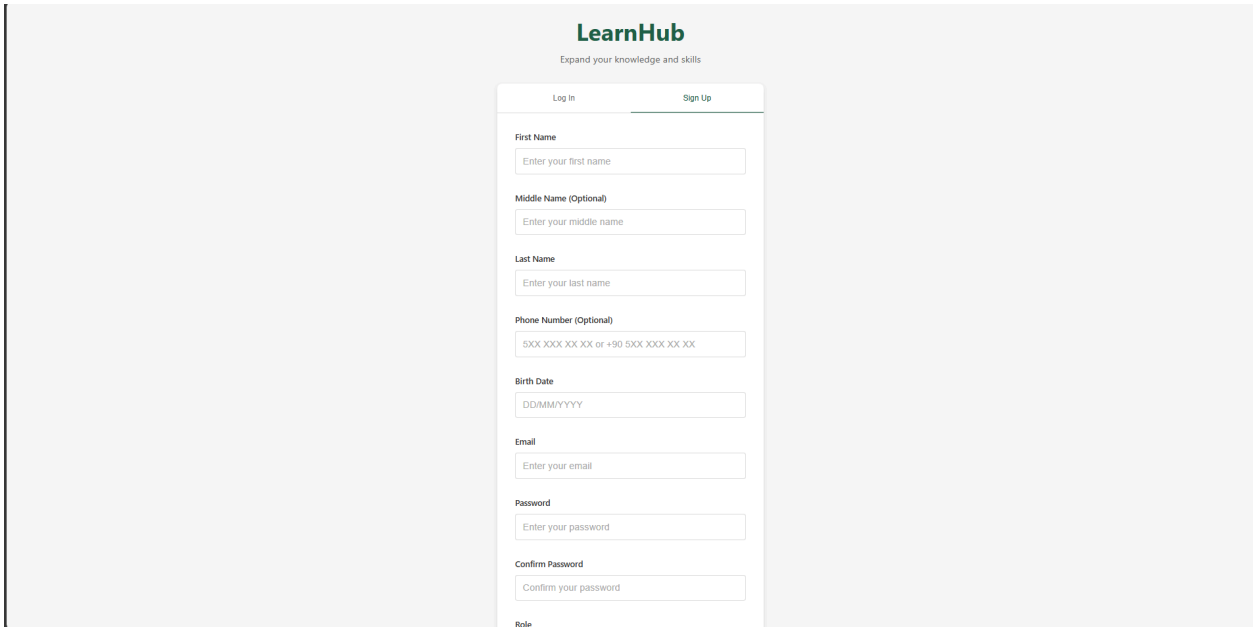
6.3. Constraints

To ensure data integrity, consistency, and validity across the database, various constraints are incorporated into the database schema. **UNIQUE** constraint is used to guarantee the uniqueness of attributes. For example, a unique constraint on a user's email prevented duplicated emails. Similarly, the **NOT NULL** constraint is used to enforce the presence of essential data such as a user's first name or email. **DEFAULT** is used to avoid null values in non-essential fields. **CHECK** constraints are used extensively throughout the schema to ensure attribute values specify necessary conditions. Status attributes across entities like the course, and apply_financial_aid can be given as an example of this. These attributes and also attributes like role need to have one of the pre-determined, predefined values; thus **IN** is used along **CHECK** to ensure that. And **BETWEEN** or comparison operators are used with **CHECK** to validate attribute values. In addition, **referential integrity** is enforced using **FOREIGN KEY** constraints, often combined with **ON DELETE CASCADE** actions. This ensures that relationships between tables remain valid even as data changes. For instance, in the earn_certificate table, the foreign key on certificate_id uses **ON DELETE CASCADE**, meaning that when a certificate is deleted from the certificate table, all associated records in earn_certificate are automatically removed. **ON DELETE SET NULL** is also used for top students, courses, instructors etc. in the report entities to make sure the whole report record is not deleted when a mentioned student, instructor or course gets deleted.

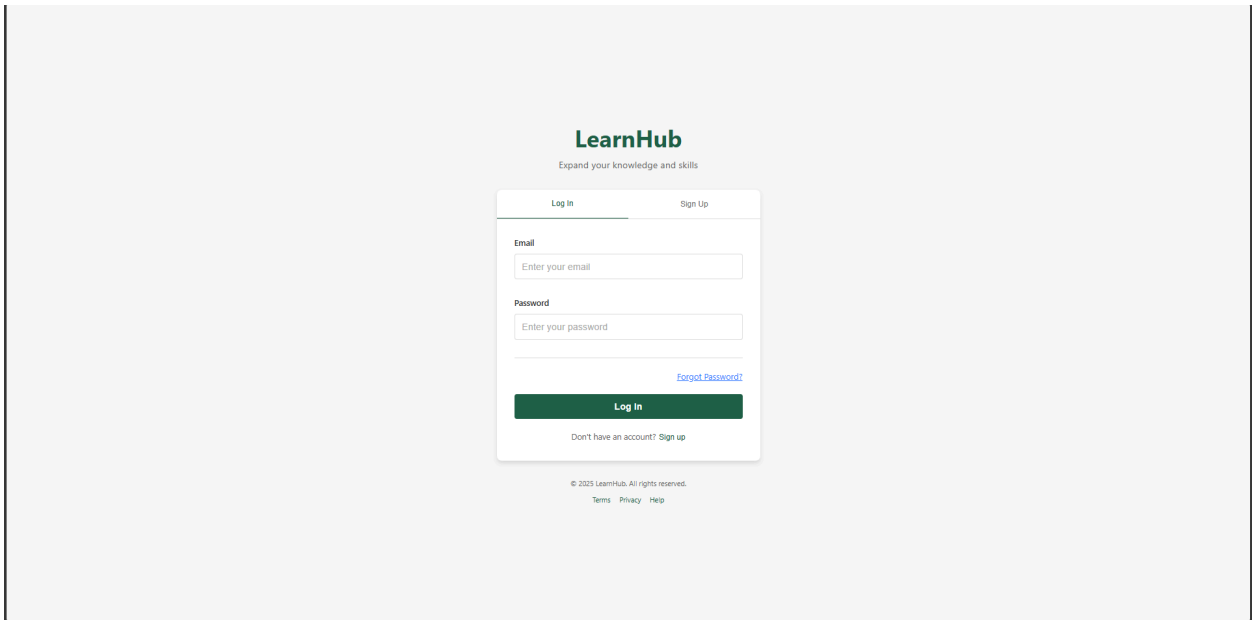
7. User's Manual

7.1. User Manual

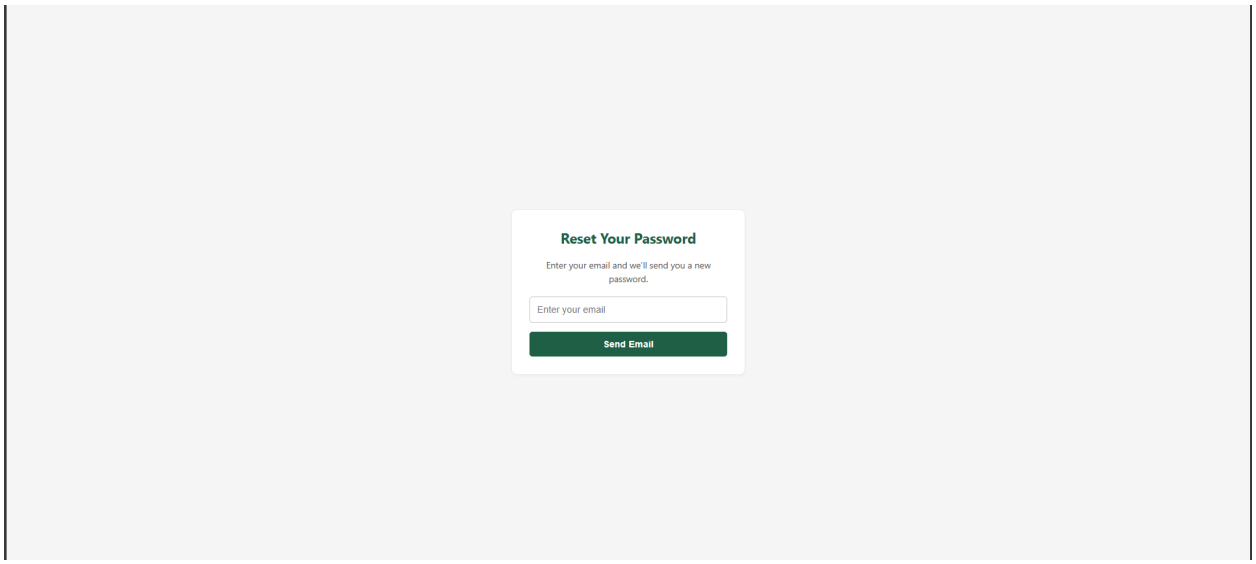
7.1.1. Register

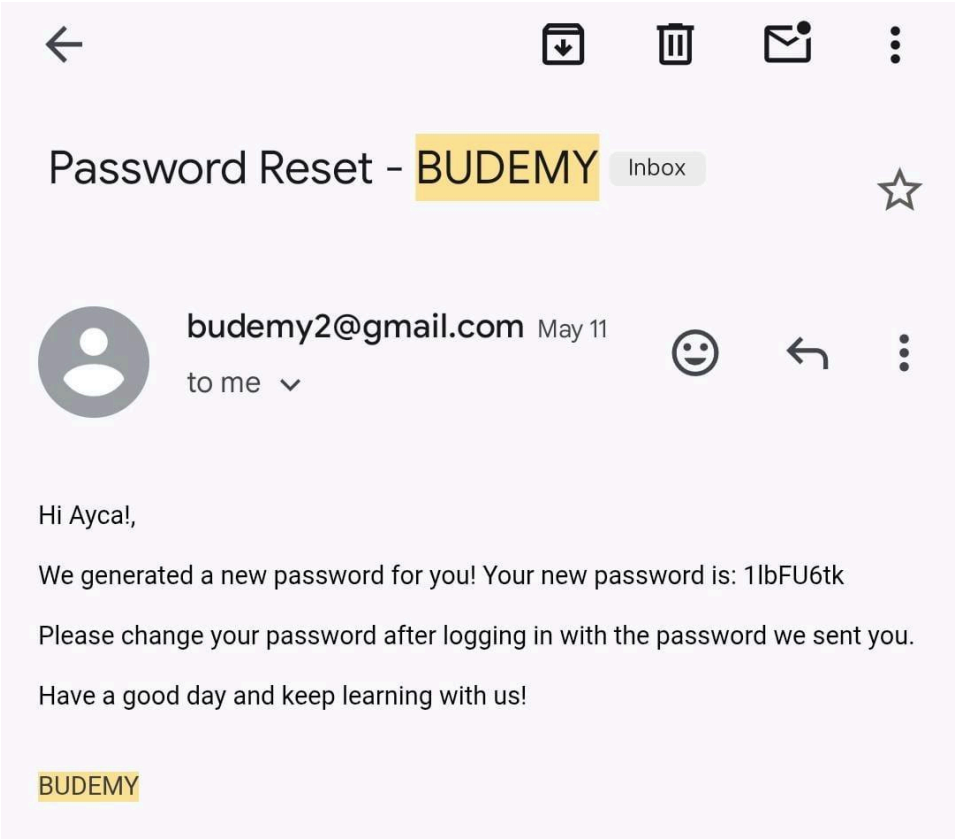


7.1.2. Login



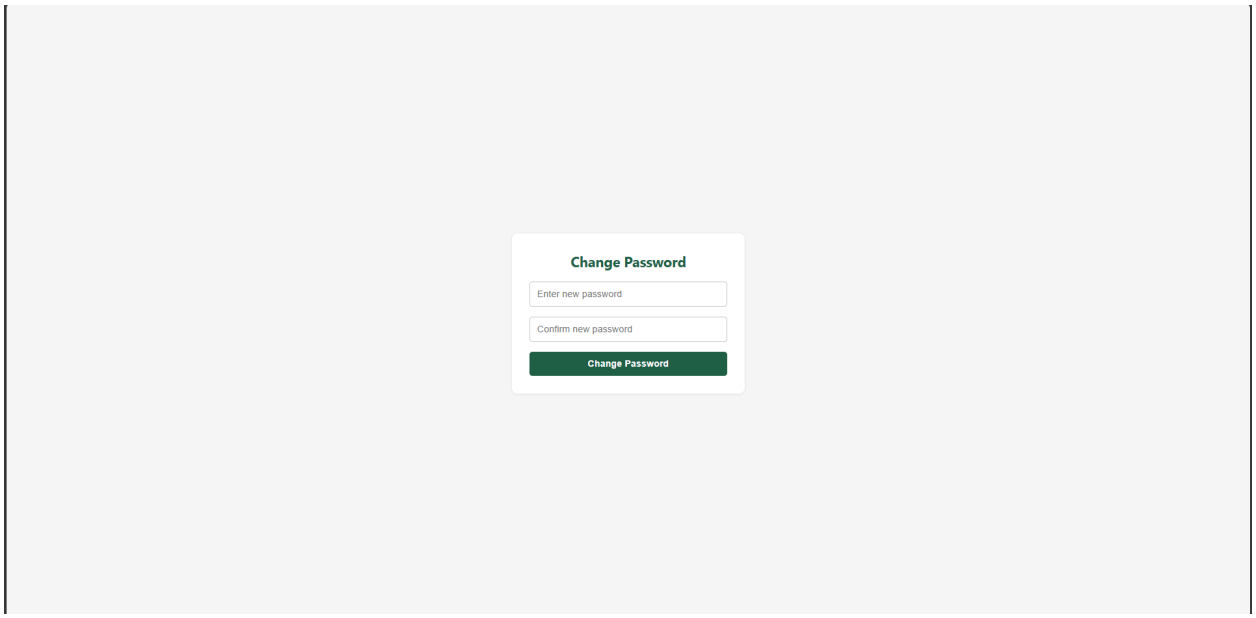
7.1.3. Forgot Password



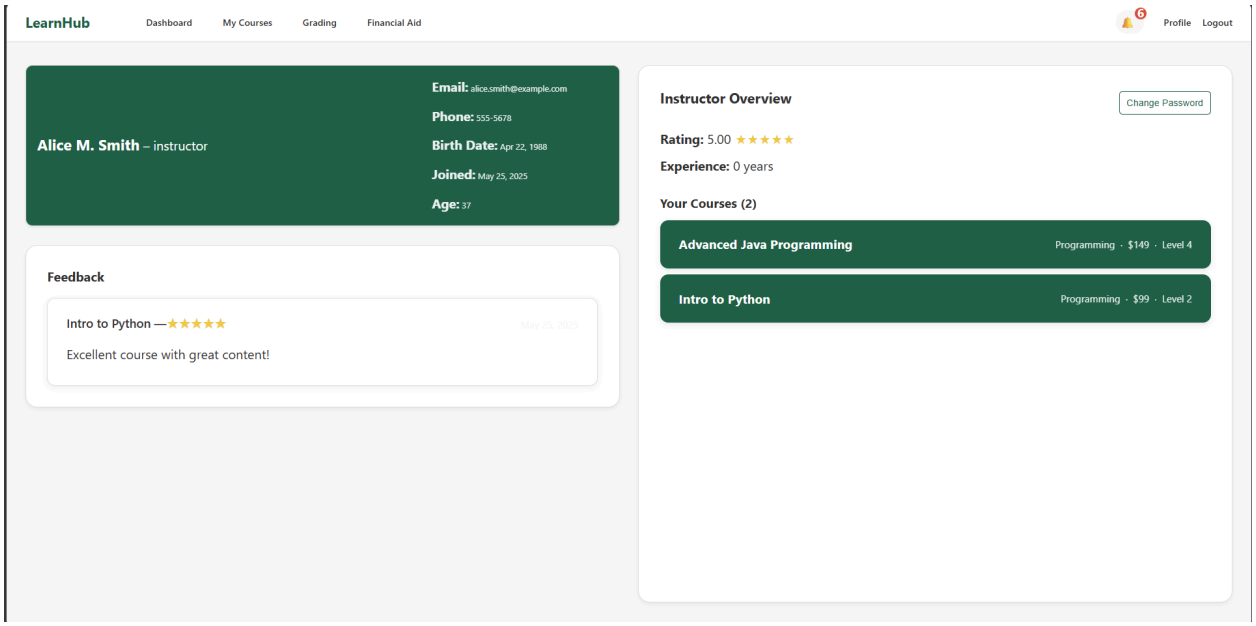


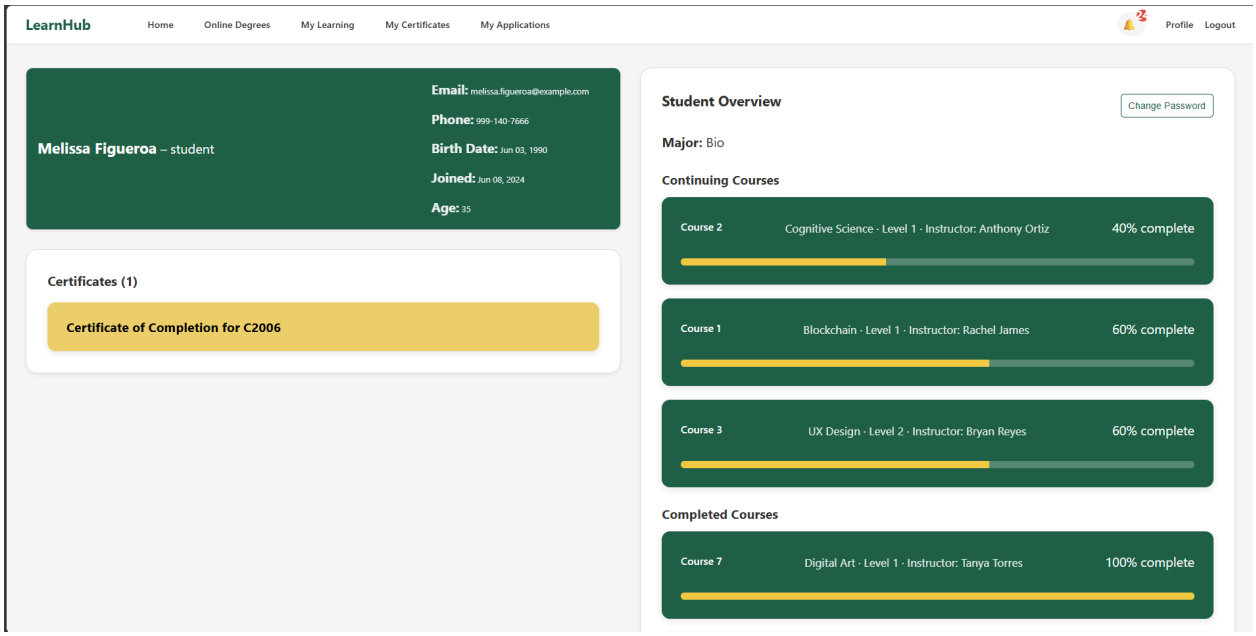
All users can reset their passwords via email. Whether or not the email entered exists in the database is checked; it should be the email user registered with.

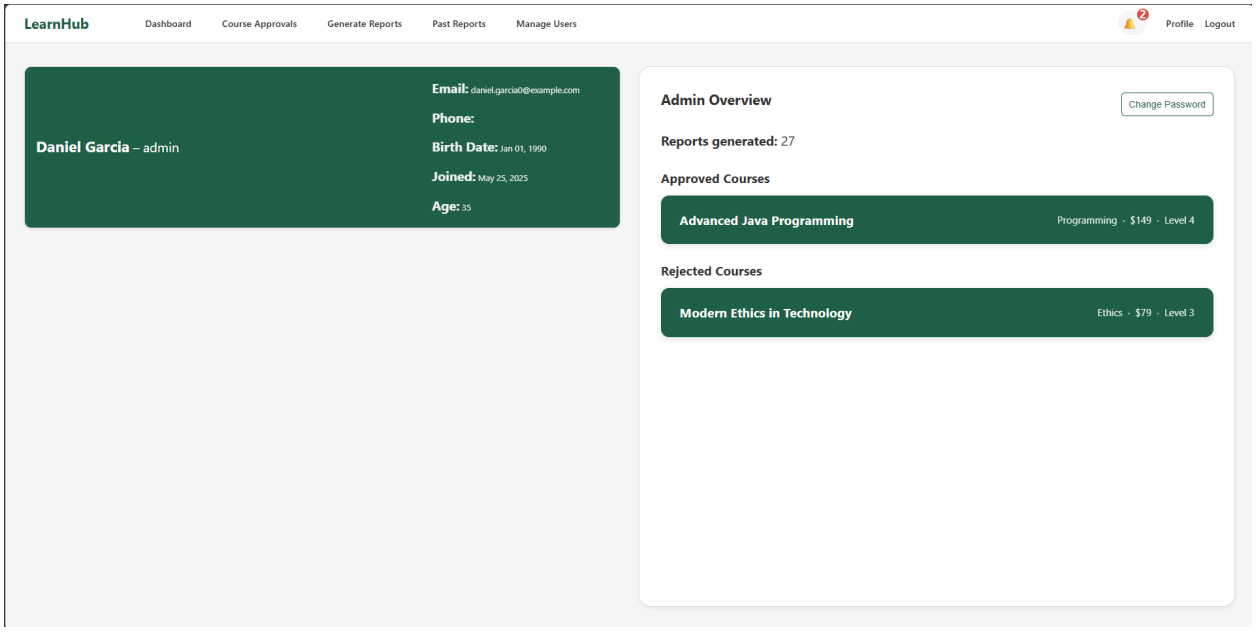
7.1.4. Change Password



7.1.5. Profile





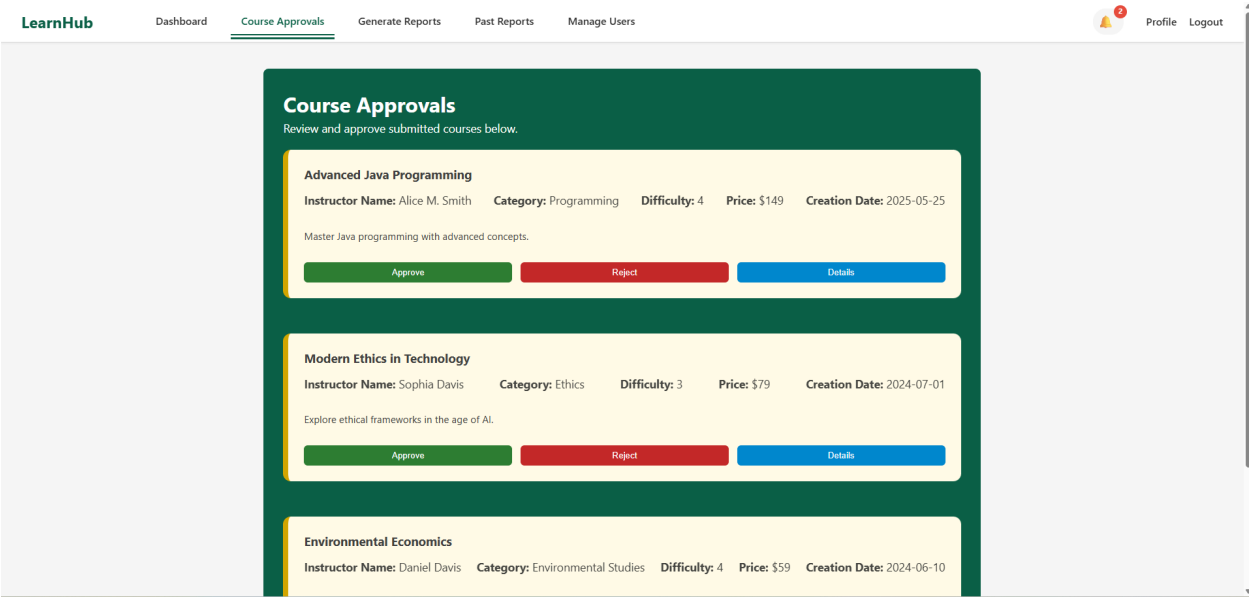


All users can view their profiles, displaying different information for each role.

7.2. Admin Manual

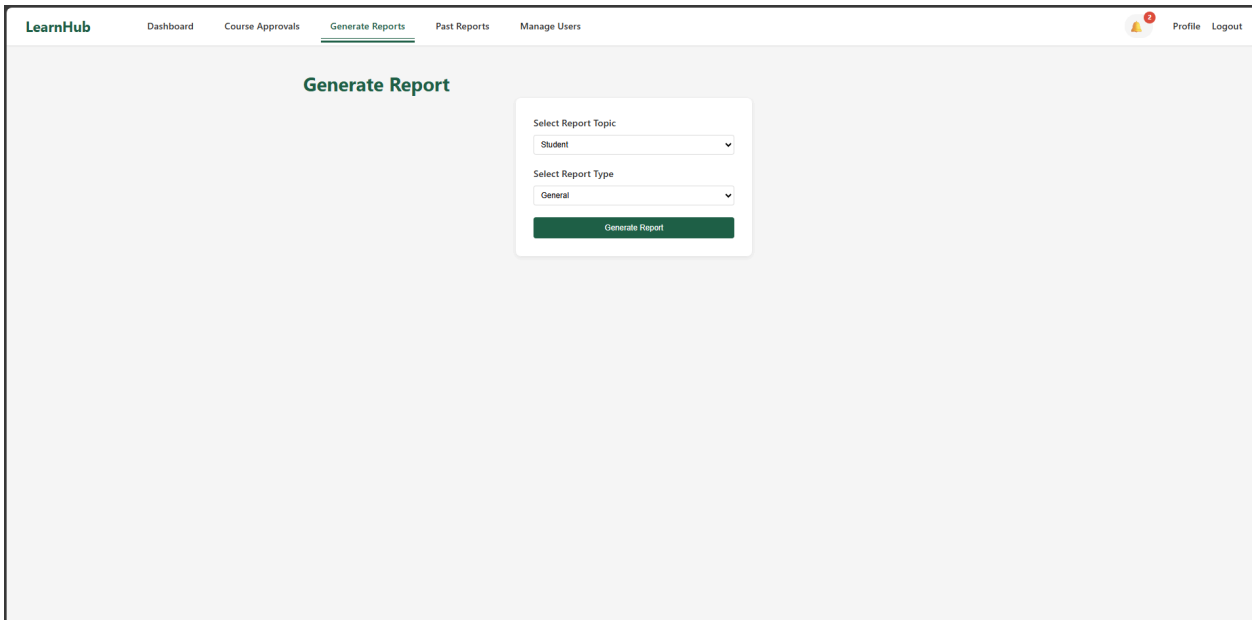
7.2.1. Admin Dashboard

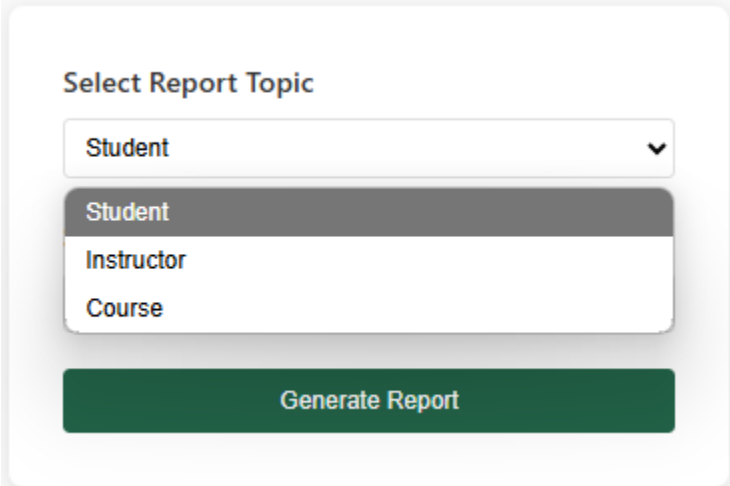
7.2.2. Admin Course Approvals



Admin users can approve or reject submitted courses by reviewing each course's information and clicking the corresponding **Approve** or **Reject** button.

7.2.3. Admin Generate Report





Select Report Topic

Student

Select Report Type

General

General

Range Specific

Select Report Topic

Student

Select Report Type

Range Specific

Start

End

--

--

te Report

--

2025-04

2025-03

2025-02

2025-01

2024-12

2024-11

2024-10

2024-09

2024-08

2024-07

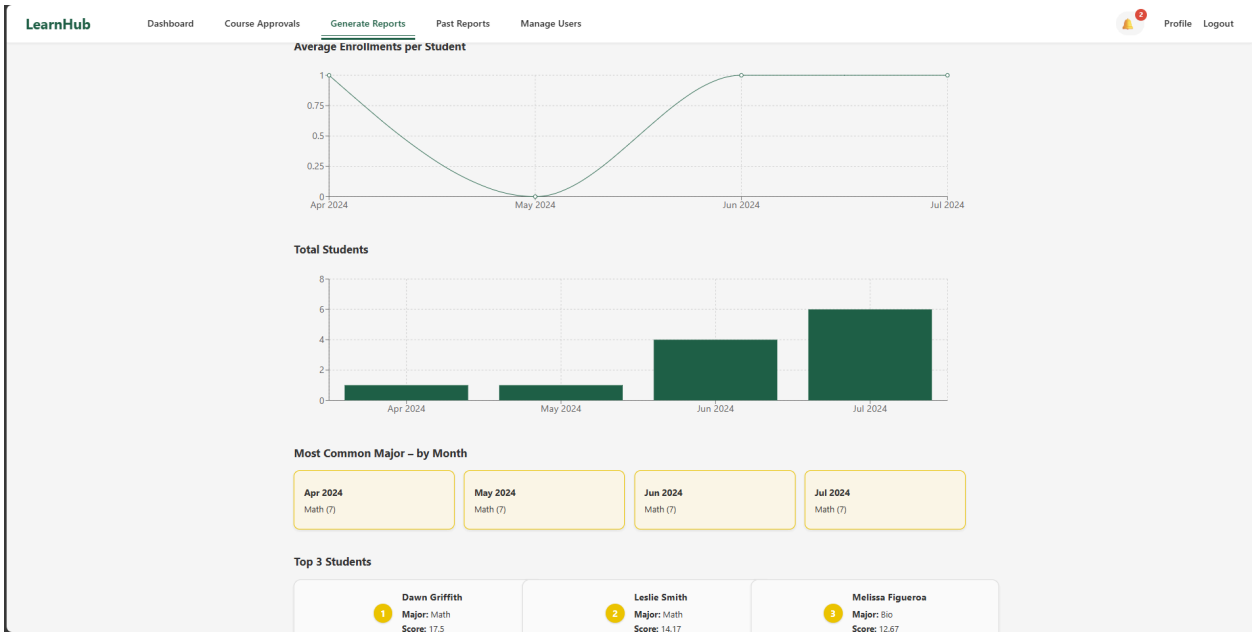
2024-06

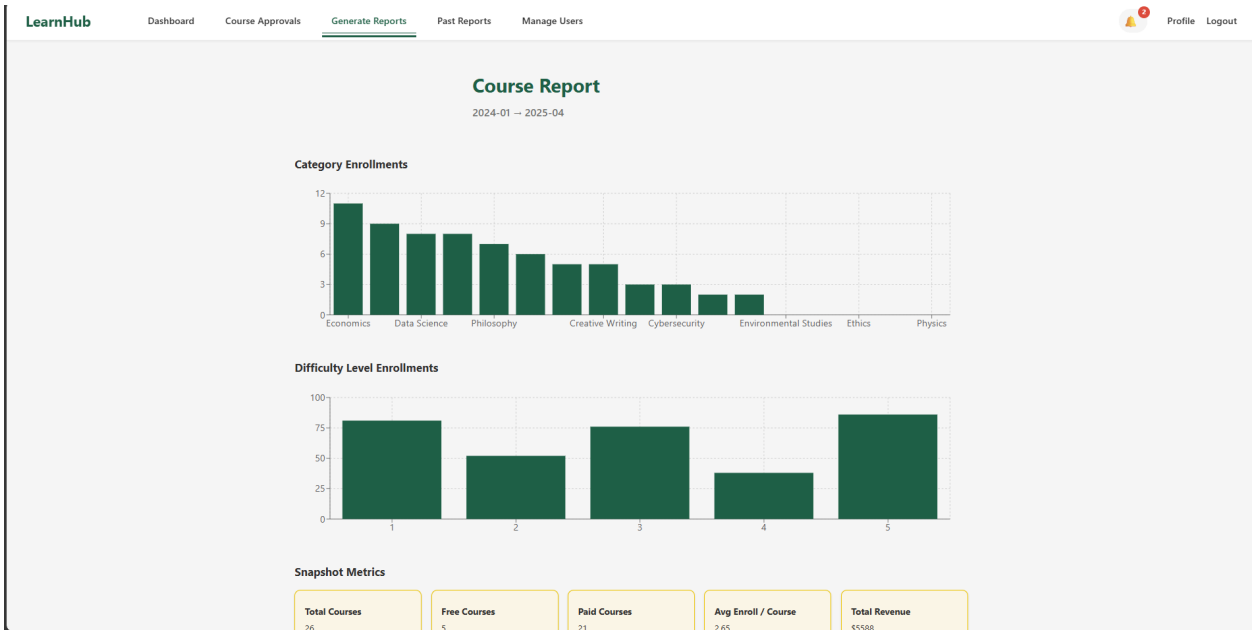
2024-05

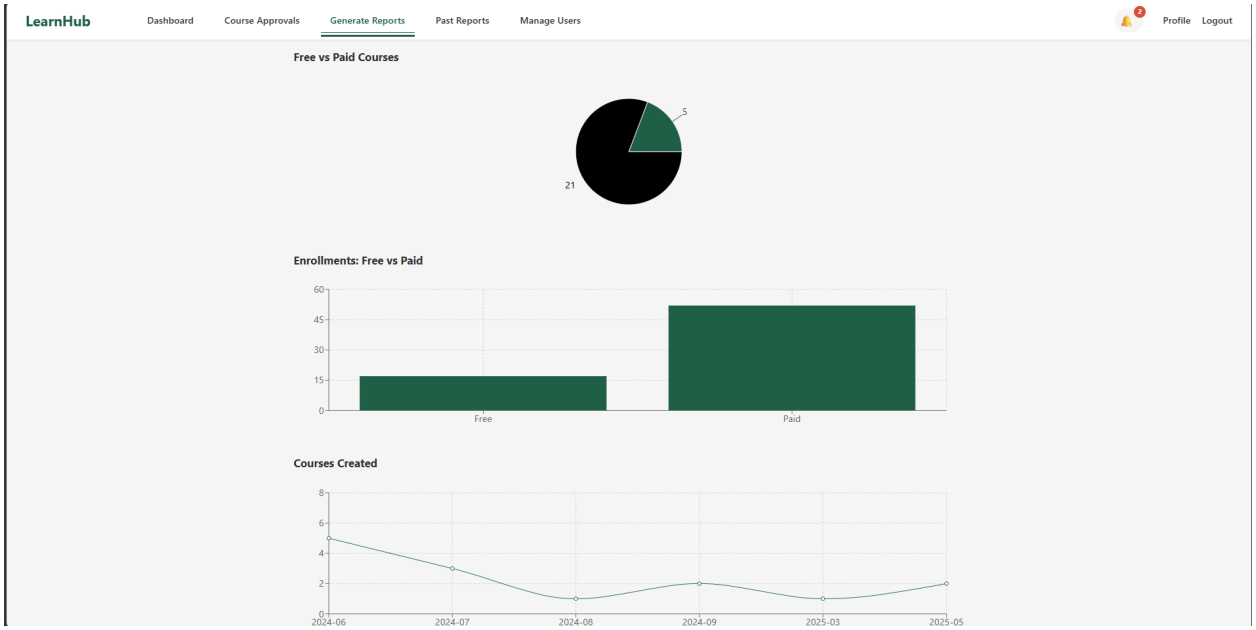
2024-04

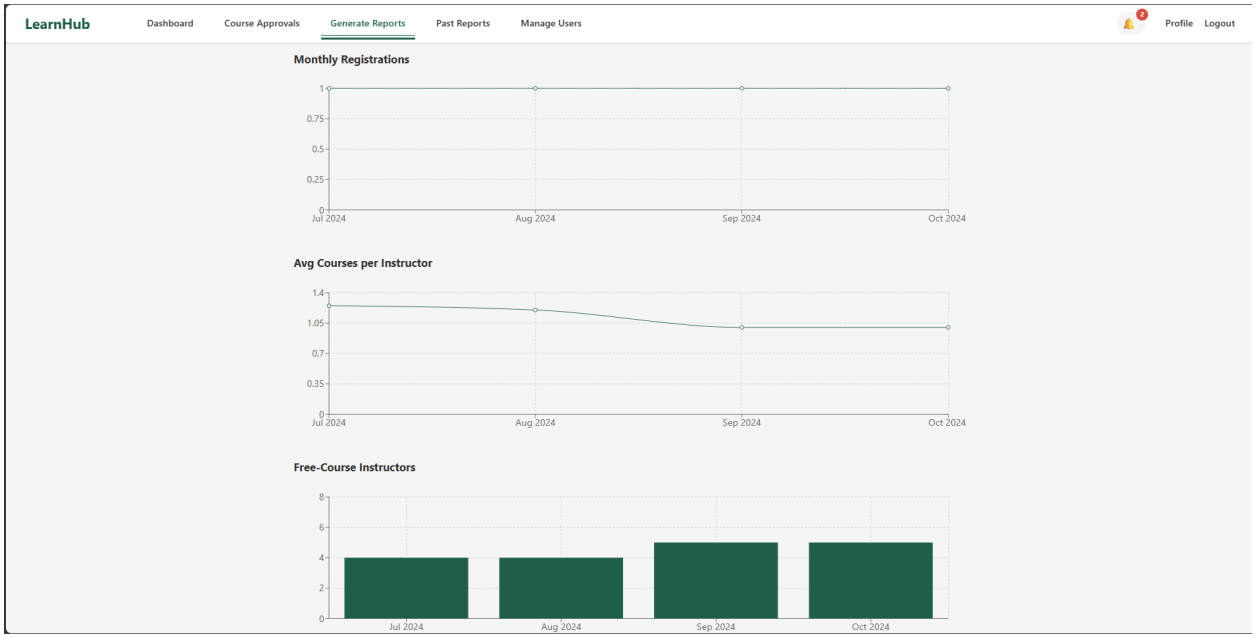
Admins can generate 6 types of reports to view statistics. The dates for start and end dates start from the last completed month (today is May 26th so the last finished month is April) and go one year prior. When they click generate report, they are directed to the Report Results page to view the resulting report.

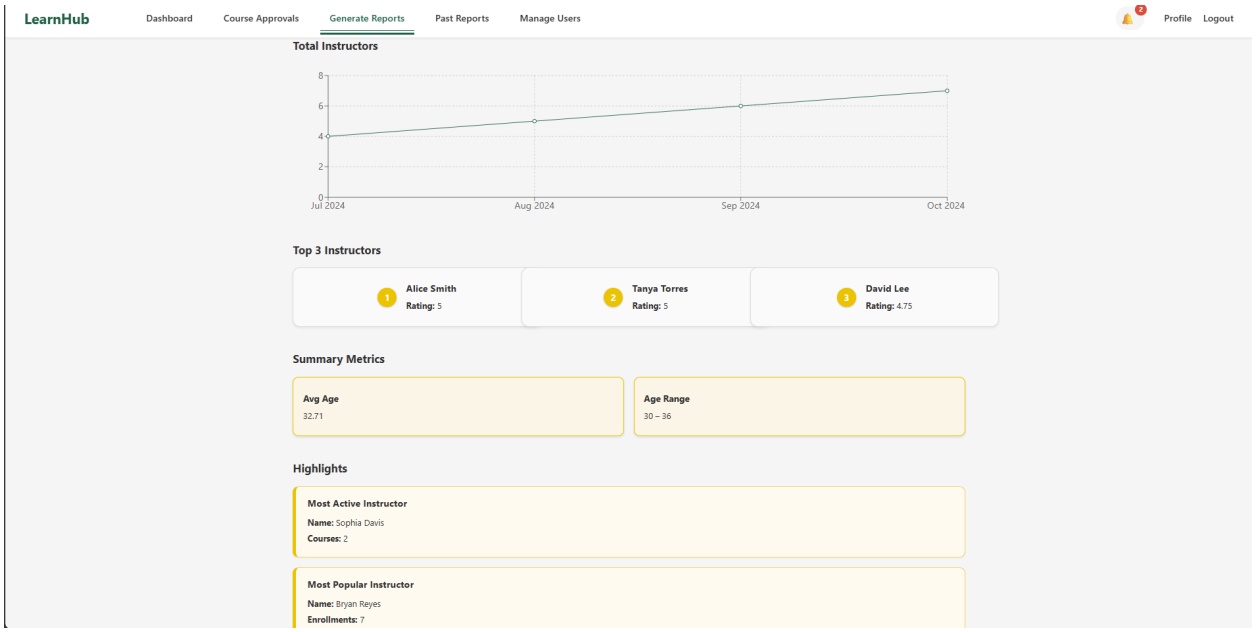
7.2.4. Admin Report Results





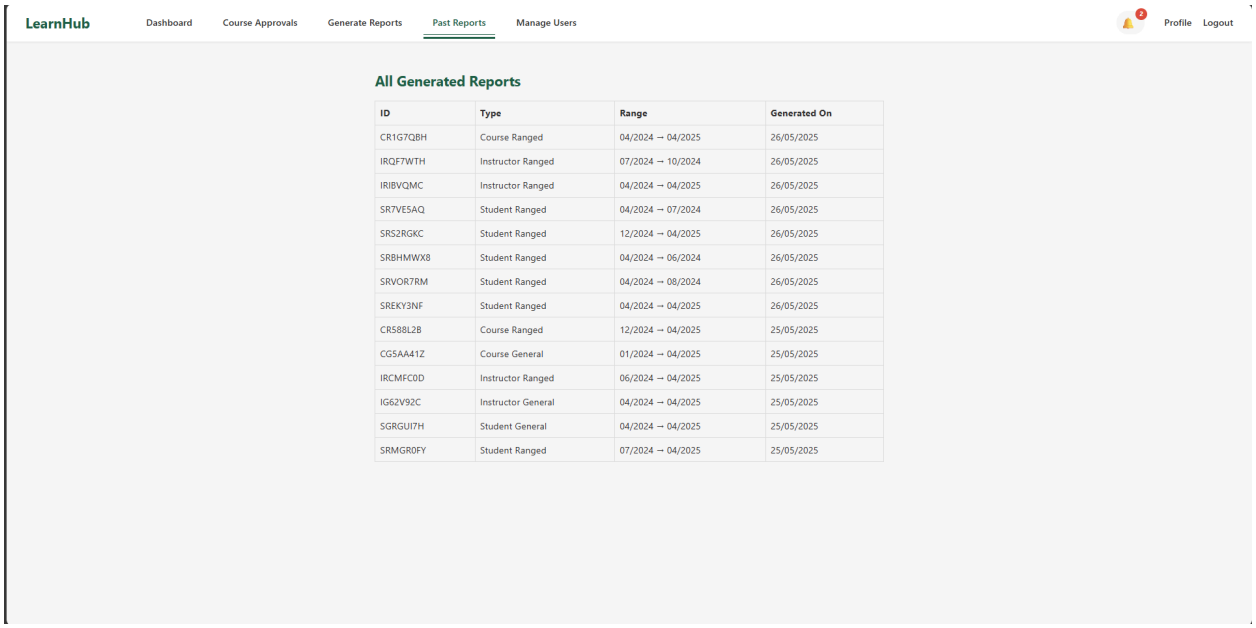

LearnHub
Dashboard
Course Approvals
Generate Reports
Past Reports
Manage Users
2
Profile
Logout
Free vs Paid Courses
5
21
Enrollments: Free vs Paid
60
45
30
15
0
Free
Paid
Courses Created
8
6
4
2
0
2024-06
2024-07
2024-08
2024-09
2025-03
2025-05





The admin can view the report they just generated on this page. Above are some examples of the report results. These are just a few examples of the report outputs—far more data can be generated beyond what's shown here.

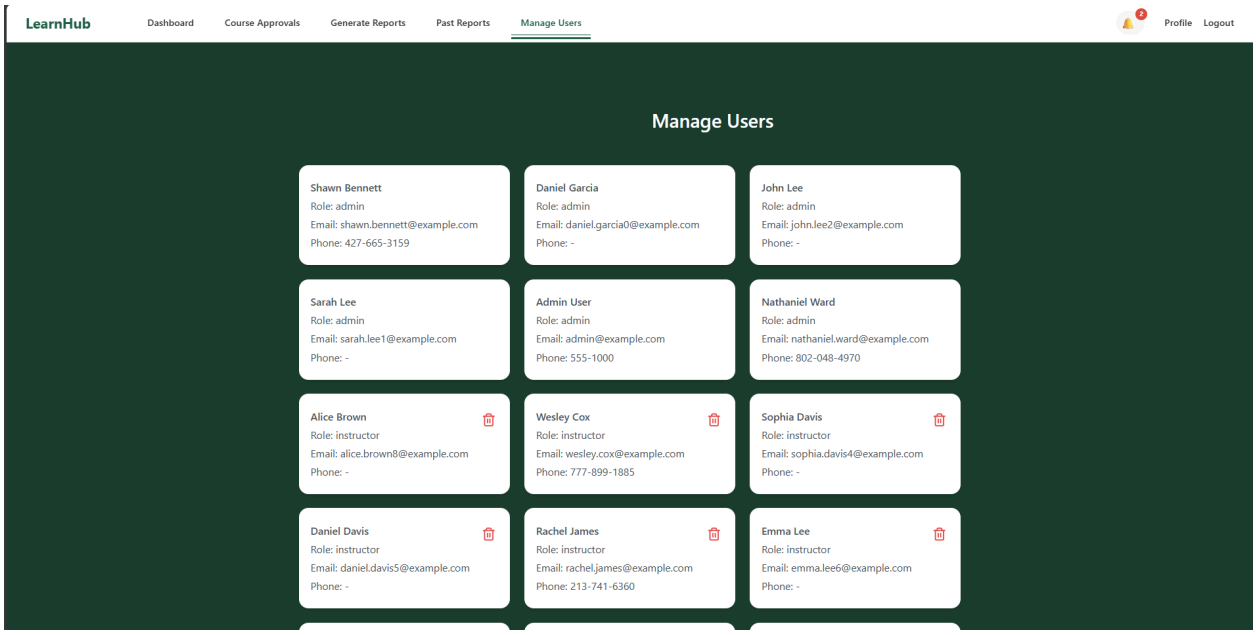
7.2.5. Admin Past Reports



ID	Type	Range	Generated On
CR1G7QBH	Course Ranged	04/2024 → 04/2025	26/05/2025
IRQF7WTH	Instructor Ranged	07/2024 → 10/2024	26/05/2025
IRIBVQMC	Instructor Ranged	04/2024 → 04/2025	26/05/2025
SR7VE5AQ	Student Ranged	04/2024 → 07/2024	26/05/2025
SRS2RGKC	Student Ranged	12/2024 → 04/2025	26/05/2025
SRBHMWX8	Student Ranged	04/2024 → 06/2024	26/05/2025
SRVOR7RM	Student Ranged	04/2024 → 08/2024	26/05/2025
SREKY3NF	Student Ranged	04/2024 → 04/2025	26/05/2025
CR588L2B	Course Ranged	12/2024 → 04/2025	25/05/2025
CG5AA41Z	Course General	01/2024 → 04/2025	25/05/2025
IRCMFC0D	Instructor Ranged	06/2024 → 04/2025	25/05/2025
IG62V92C	Instructor General	04/2024 → 04/2025	25/05/2025
SGRGUI7H	Student General	04/2024 → 04/2025	25/05/2025
SRMGR0FY	Student Ranged	07/2024 → 04/2025	25/05/2025

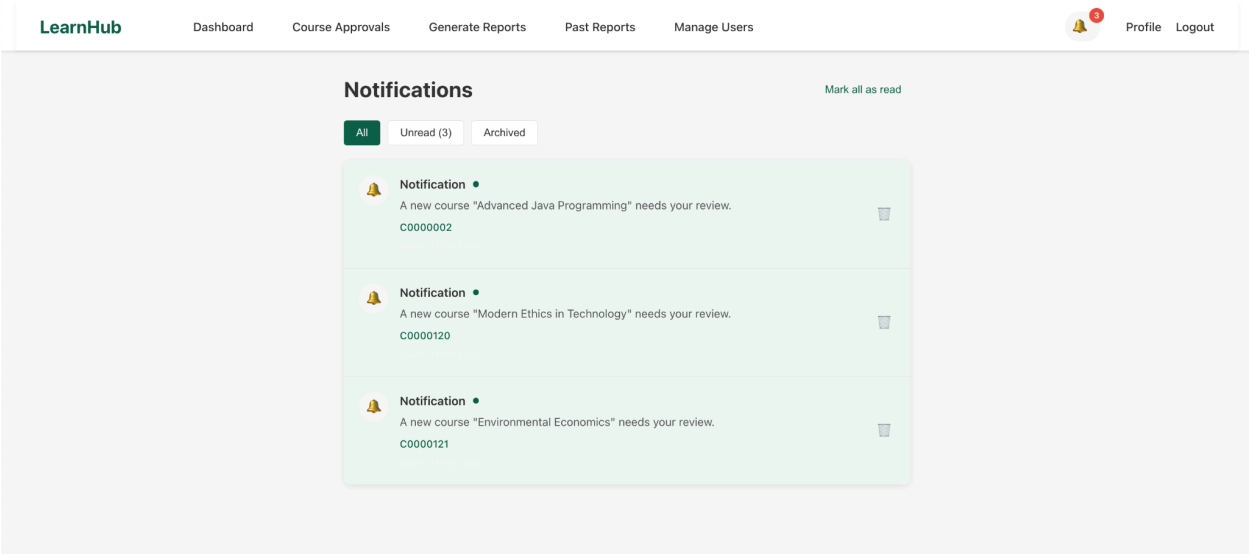
Admin users can view all reports they generated before and will be directed to the Report Results page if they click on one to view a past report.

7.2.6. Admin Manage Users



Admin users are allowed to delete instructor and student accounts. The first six cards do not have delete buttons because they are admin users.

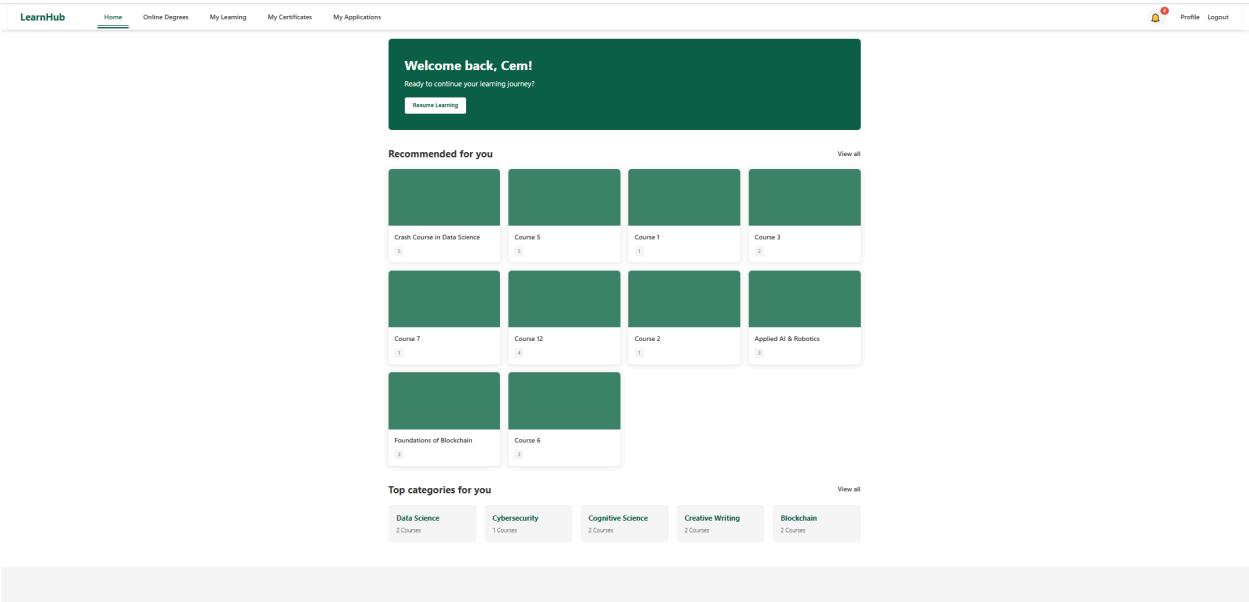
7.2.7. Admin Notifications



Admin users have a notification page. They receive notification when new course material is inserted by the instructor, declaring that the admin should review it.

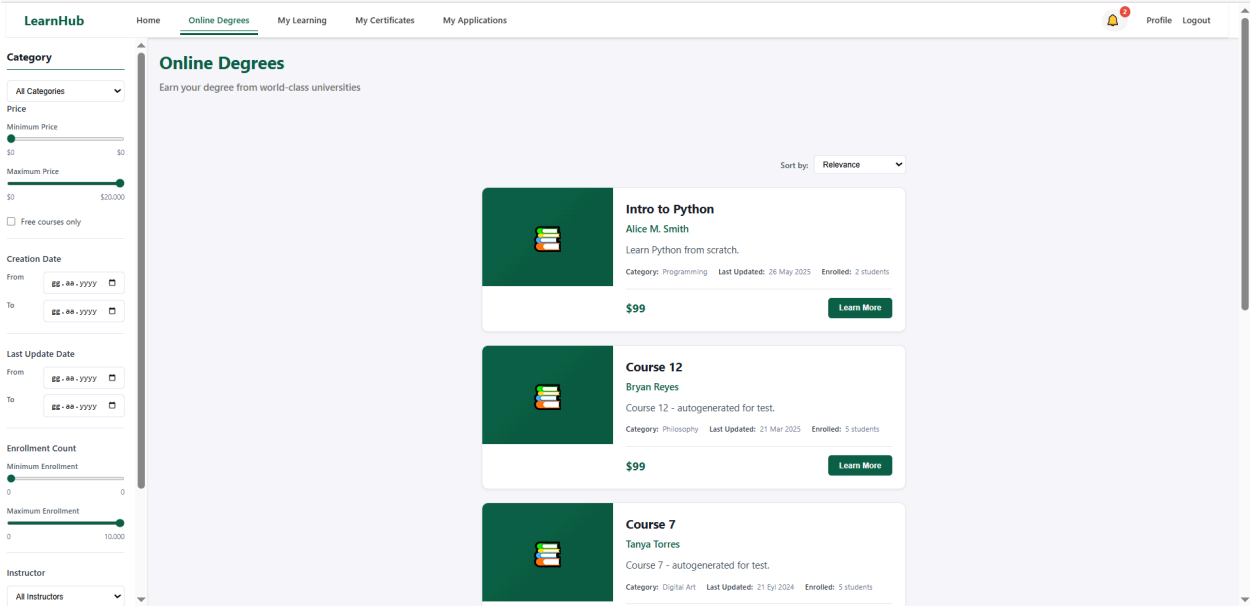
7.3. Student Manual

7.3.1. Student Dashboard



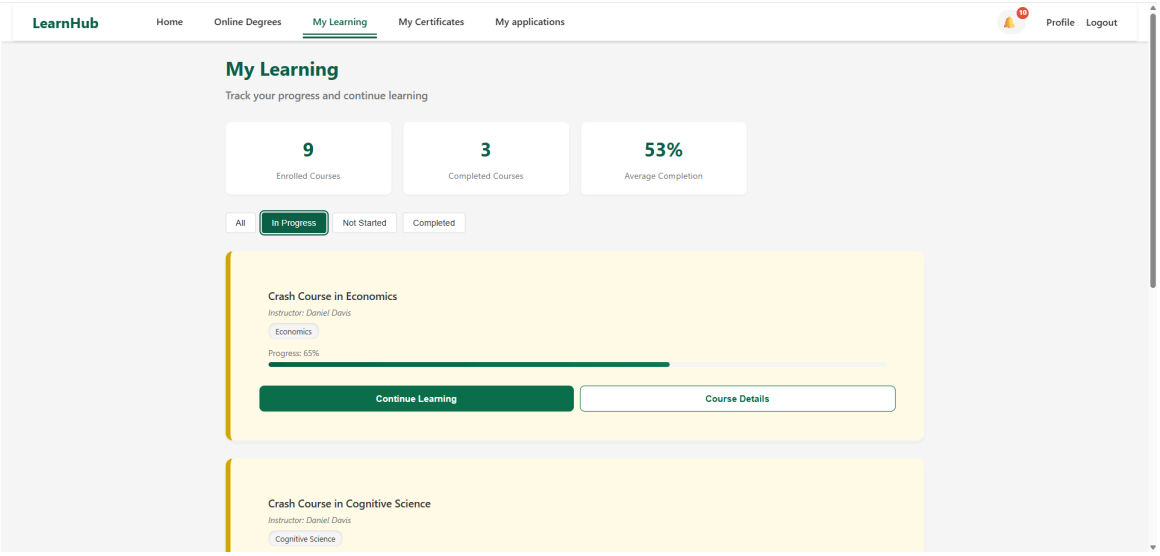
Students can view their recent learnings. Recommended courses and top categories for the student user.

7.3.2. Student Online Degrees



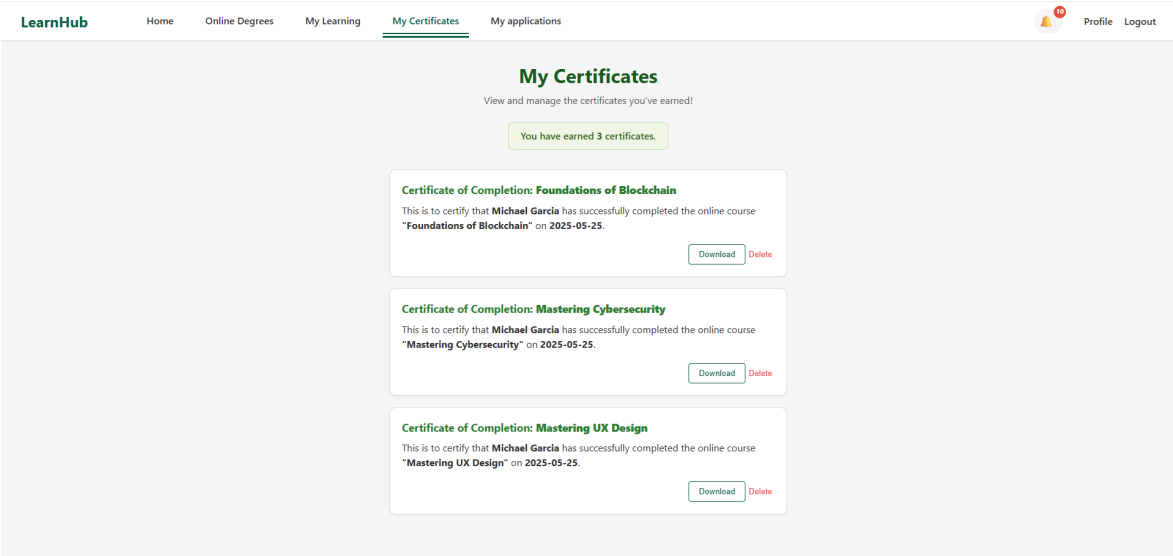
Students can view courses in the system and filter them as their looking criterias.

7.3.3. Student My Learning



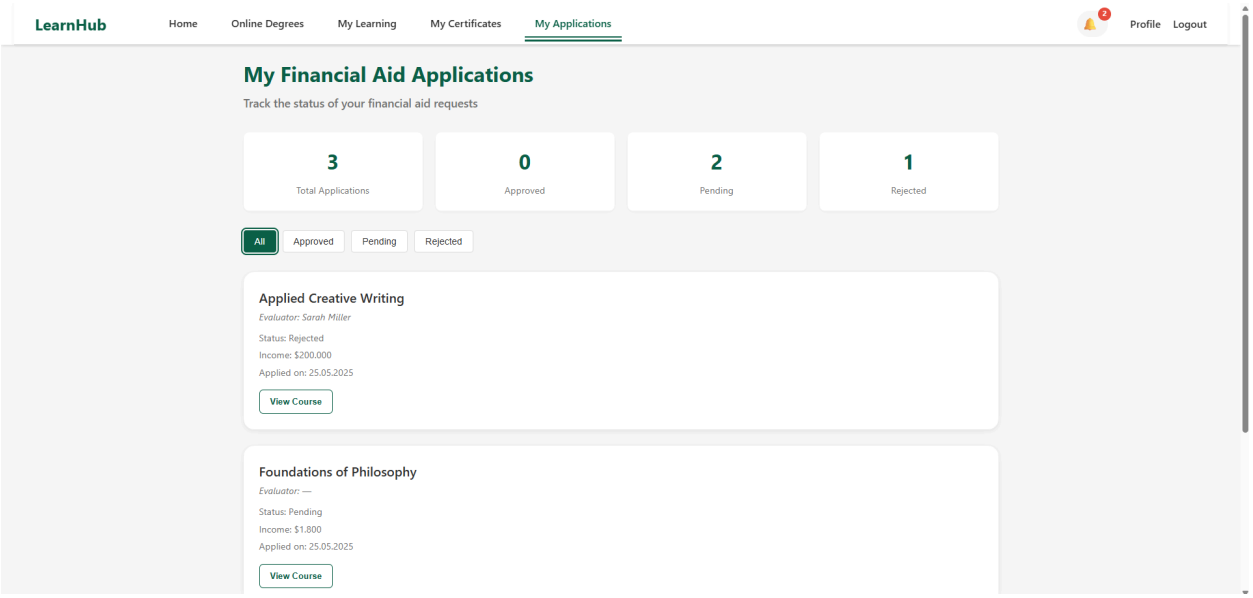
Students can monitor their learning progress, filter enrolled courses by progress status, view their average completion rate and resume learning, or view course details directly from the **My Learning** page.

7.3.4. Student My Certificates



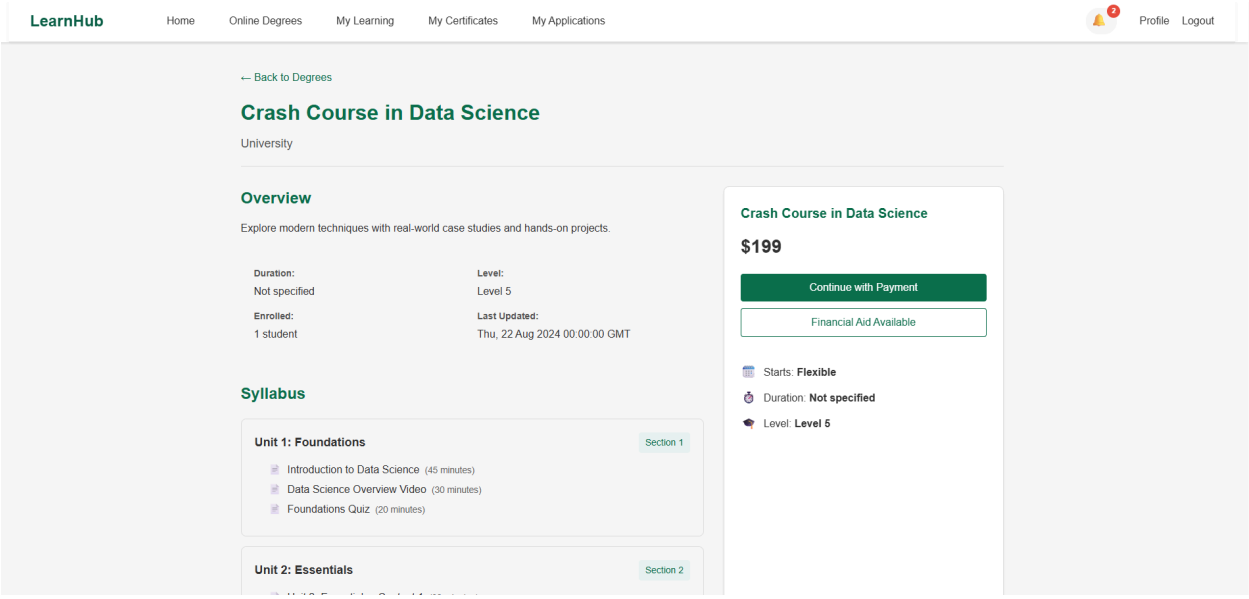
Students can view, download, or delete the certificates they have earned from completed courses on the **My Certificates** page.

7.3.5. Student My Applications



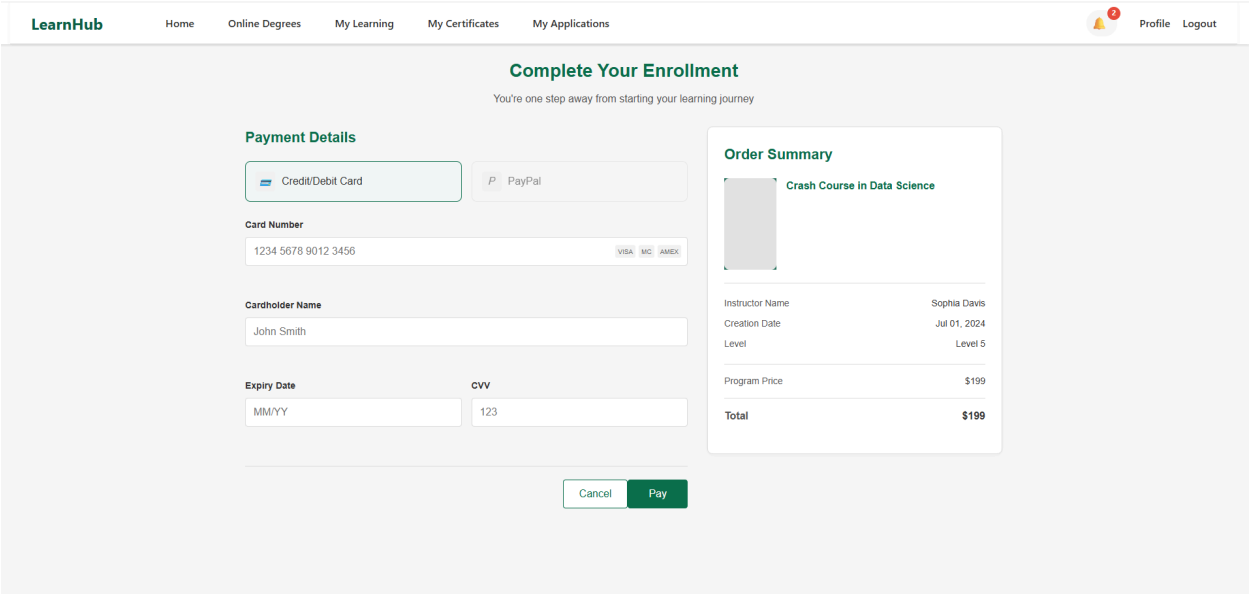
Students can track the status of their financial aid applications on the **My Applications** page, where they can view total, approved, pending, and rejected applications along with evaluator and course information. They can view detailed course information by clicking to the View Course button.

7.3.6. Student Course Overview



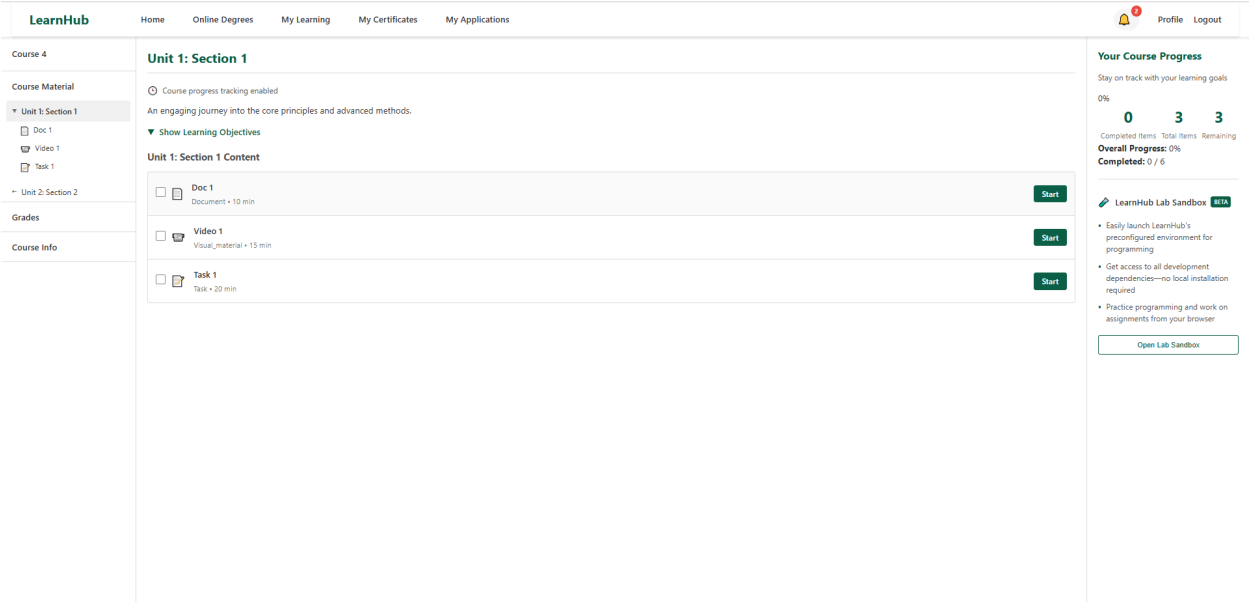
Student users can view course details and either apply for financial aid or proceed with payment to enroll in the course directly from the course page.

7.3.7. Student Payment Page



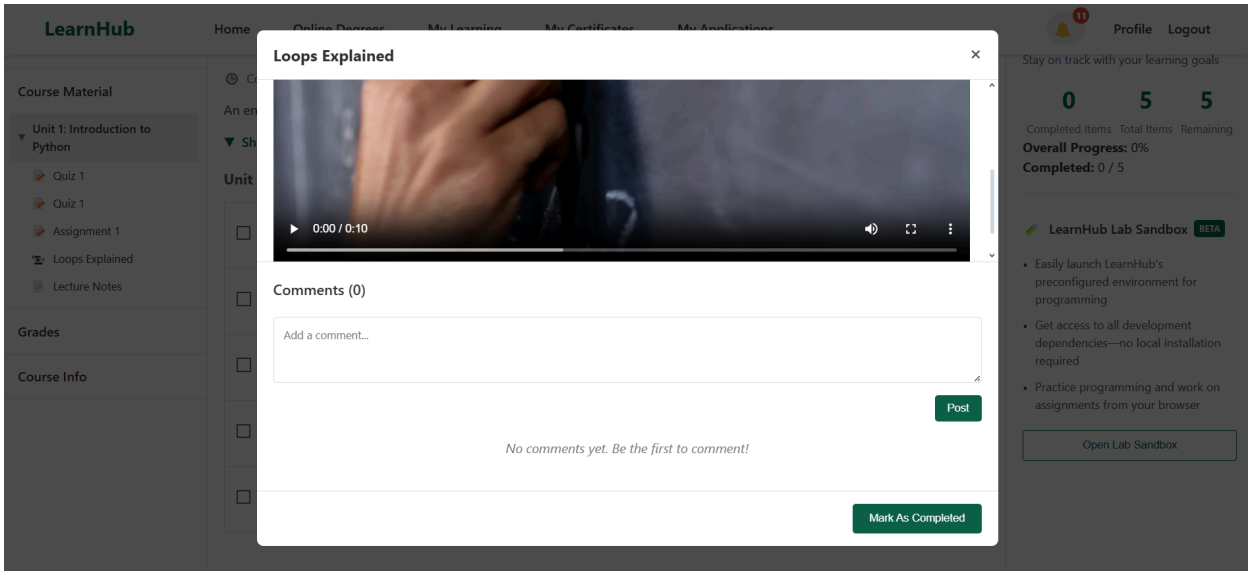
Student users can securely complete their enrollment in paid courses by entering their credit/debit card or PayPal details and confirming the payment from the checkout page.

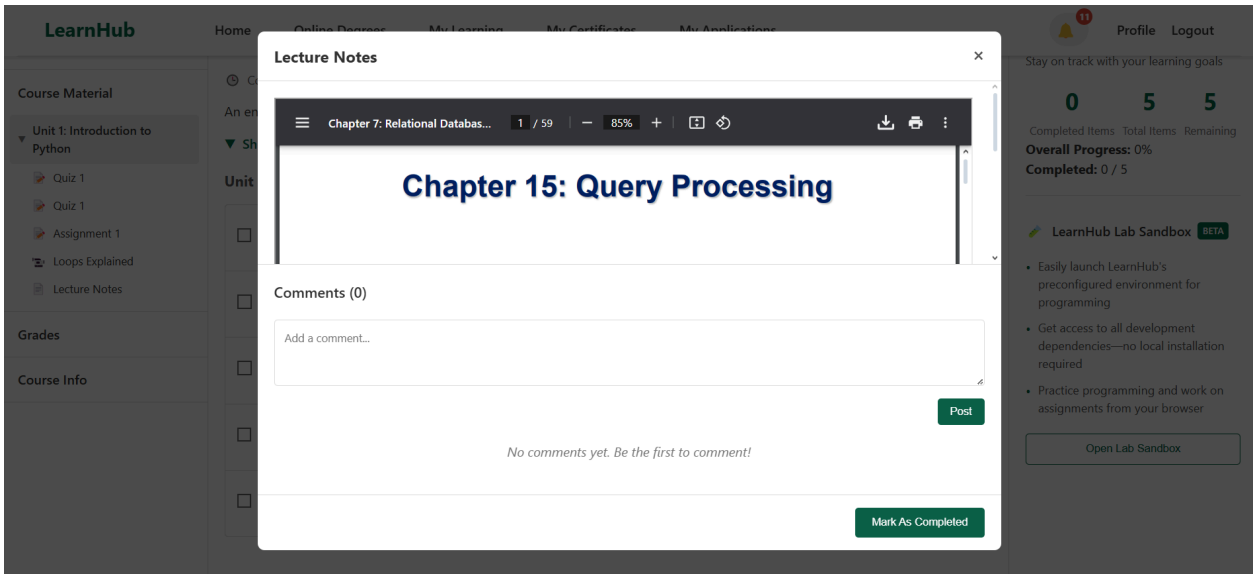
7.3.8. Student Course Details

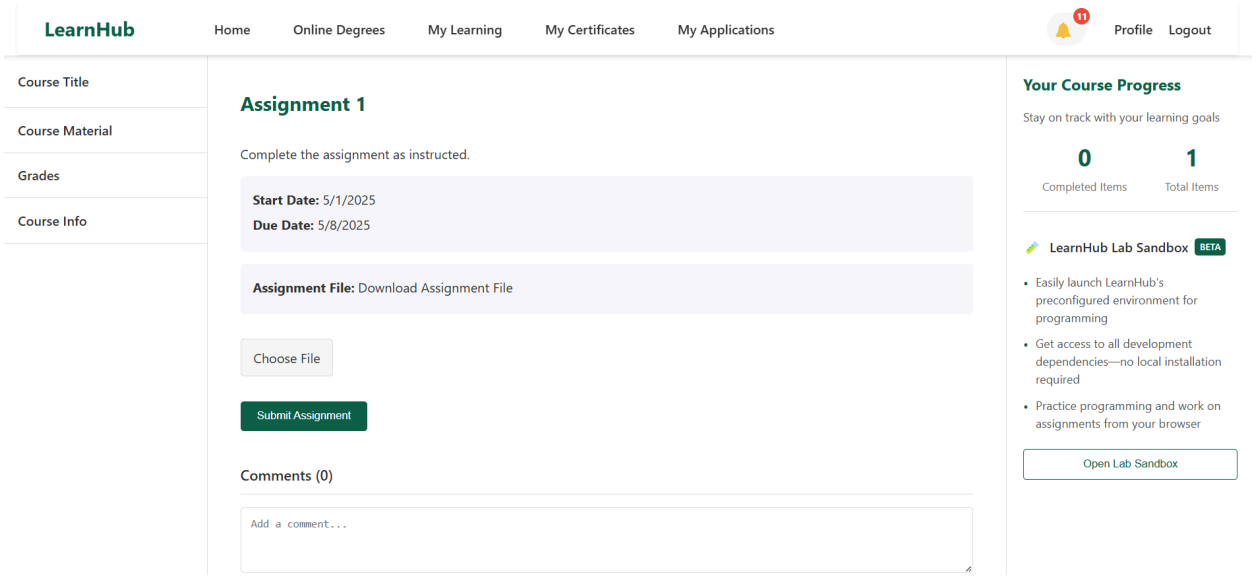


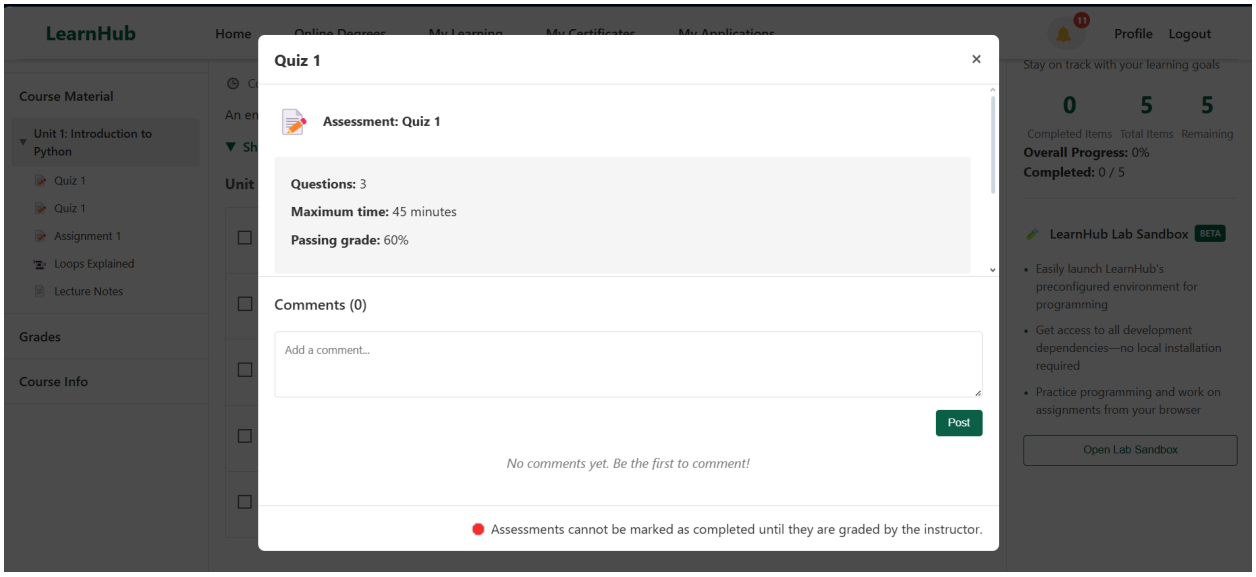
Student users can see the content of the courses, content of the visual materials, assessment on assignments.

7.3.9. Student Content View



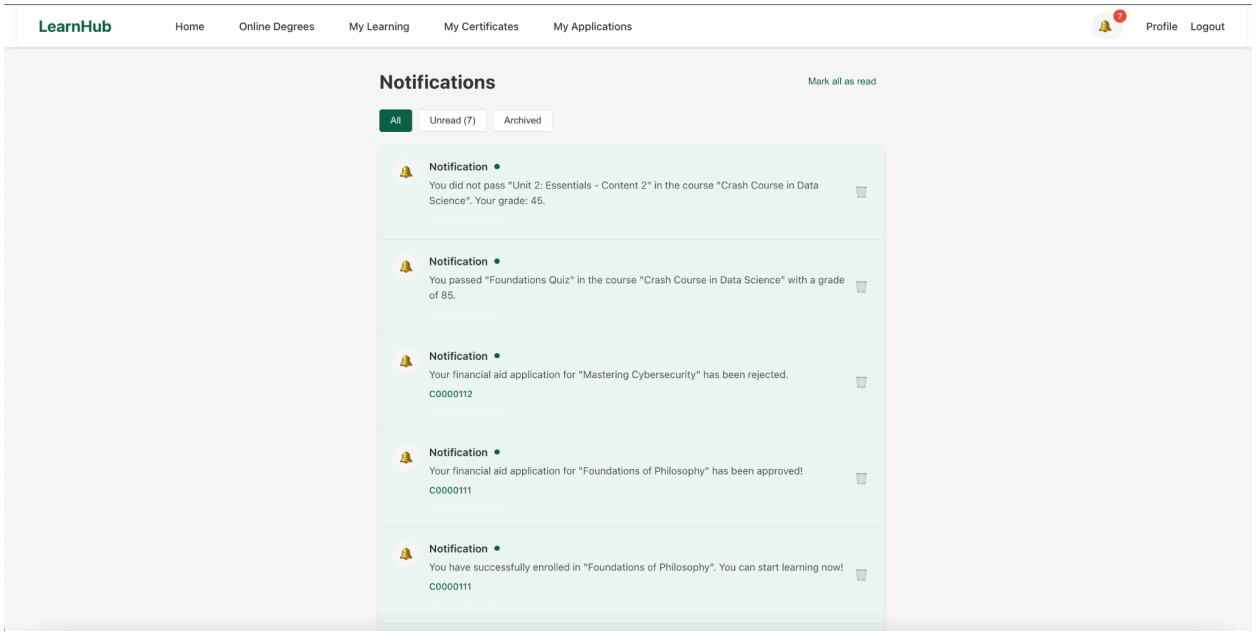

LearnHub
Home
Profile
Logout
Course Material
Unit 1: Introduction to Python
Quiz 1
Quiz 1
Assignment 1
Loops Explained
Lecture Notes
Grades
Course Info
Lecture Notes
Chapter 7: Relational Databas...
1 / 59
85%
Chapter 15: Query Processing
Comments (0)
Add a comment...
Post
No comments yet. Be the first to comment!
Mark As Completed
Stay on track with your learning goals
0
5
5
Completed Items
Total Items
Remaining
Overall Progress: 0%
Completed: 0 / 5
LearnHub Lab Sandbox
BETA
Easily launch LearnHub's preconfigured environment for programming
Get access to all development dependencies—no local installation required
Practice programming and work on assignments from your browser
Open Lab Sandbox


LearnHub
Home
Online Degrees
My Learning
My Certificates
My Applications
Profile
Logout
Course Title
Course Material
Grades
Course Info
Assignment 1
Complete the assignment as instructed.
Start Date: 5/1/2025
Due Date: 5/8/2025
Assignment File: Download Assignment File
Choose File
Submit Assignment
Comments (0)
Add a comment...
Your Course Progress
Stay on track with your learning goals
0
1
Completed Items
Total Items
LearnHub Lab Sandbox
BETA
Easily launch LearnHub's preconfigured environment for programming
Get access to all development dependencies—no local installation required
Practice programming and work on assignments from your browser
Open Lab Sandbox



Students can see documents, visual materials, assessments and assignments. They can mark documents and visual materials as complete, but assessments and assignments will be automatically completed when assessment or assignment is graded by the instructor. Documents can be viewed in the web app, and also can be downloaded. Visual materials can be displayed, played and paused. The assignment file, which is uploaded by the instructor, can be downloaded by the student; students can upload a file as a submission to this task. Assessments are basically quizzes that can consist of questions, students can start assessment and answer questions.
In addition, comments can be uploaded to content and other comments sent by other users can be seen.

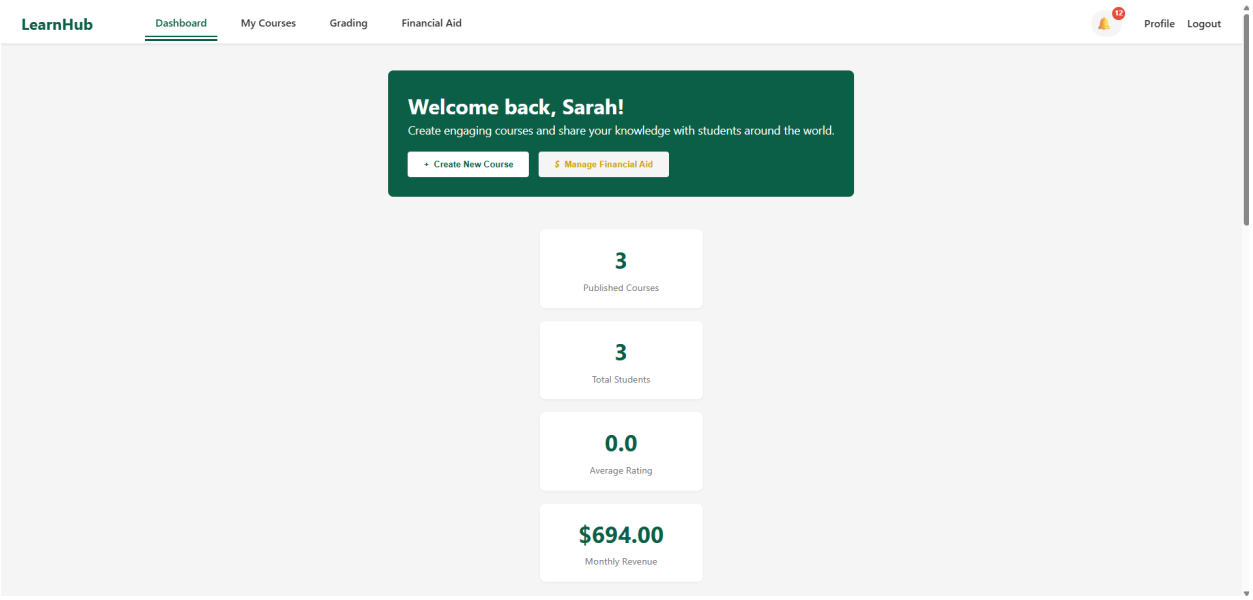
7.3.10. Student Notifications



Students receive automated notifications for key events related to their learning journey and course activities. They are alerted when they successfully enroll in courses, when their assignment grades are posted (both passing and failing), when their financial aid applications are approved or rejected, and when they complete entire courses. They can click on the notification button appearing in the top right corner to access all notifications and mark them read or archived.

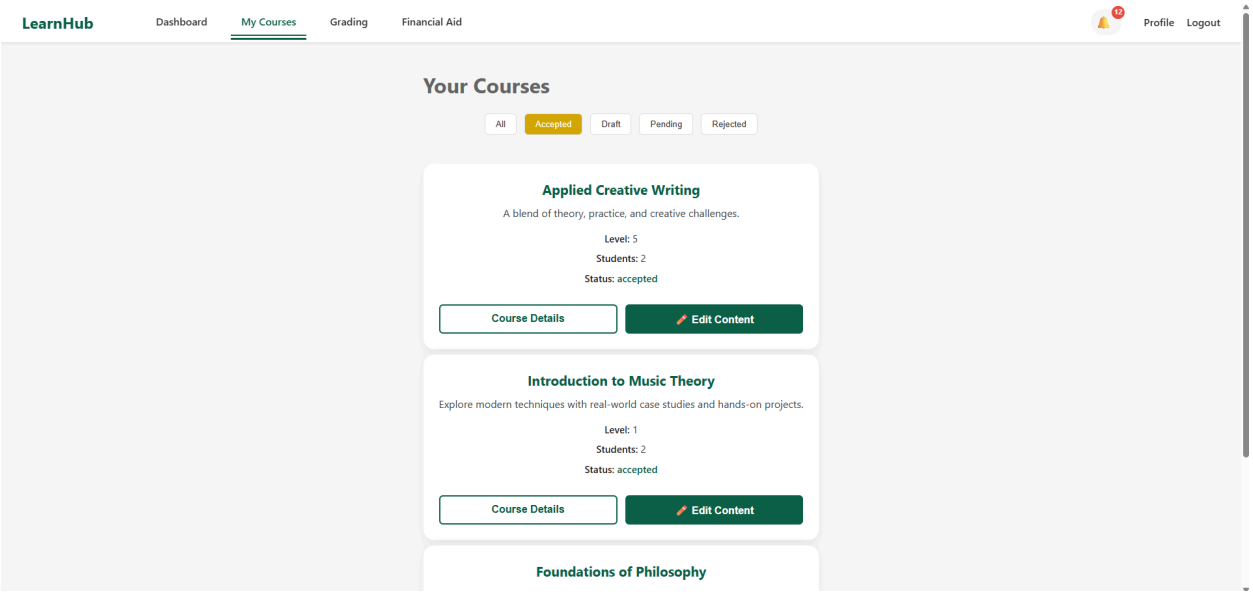
7.4. Instructor Manual

7.4.1. Instructor Dashboard



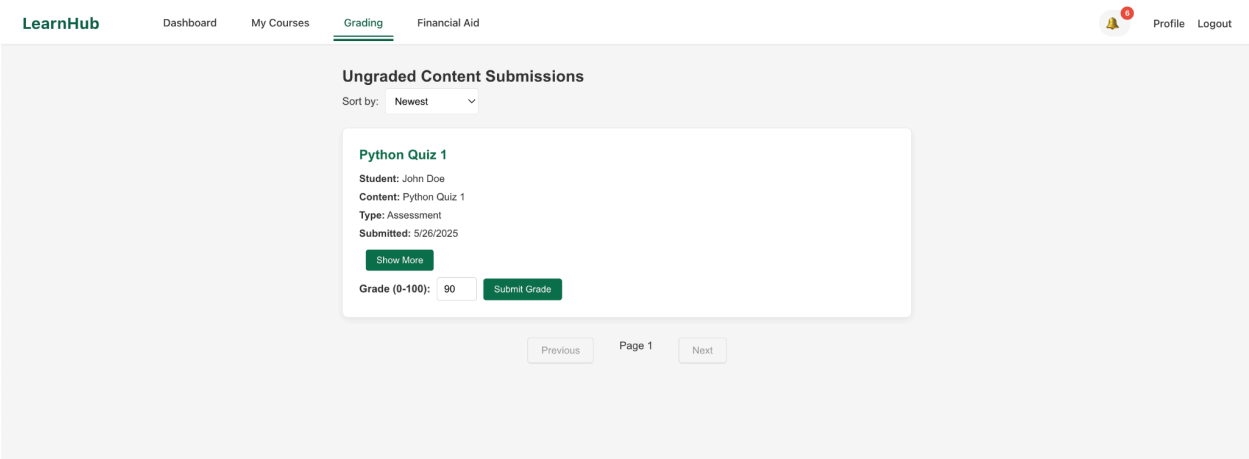
Instructor users can briefly view their statistics including published courses, total students, average ratings, and revenue, from the dashboard, and quickly access course creation or financial aid management features.

7.4.2. Instructor My Courses



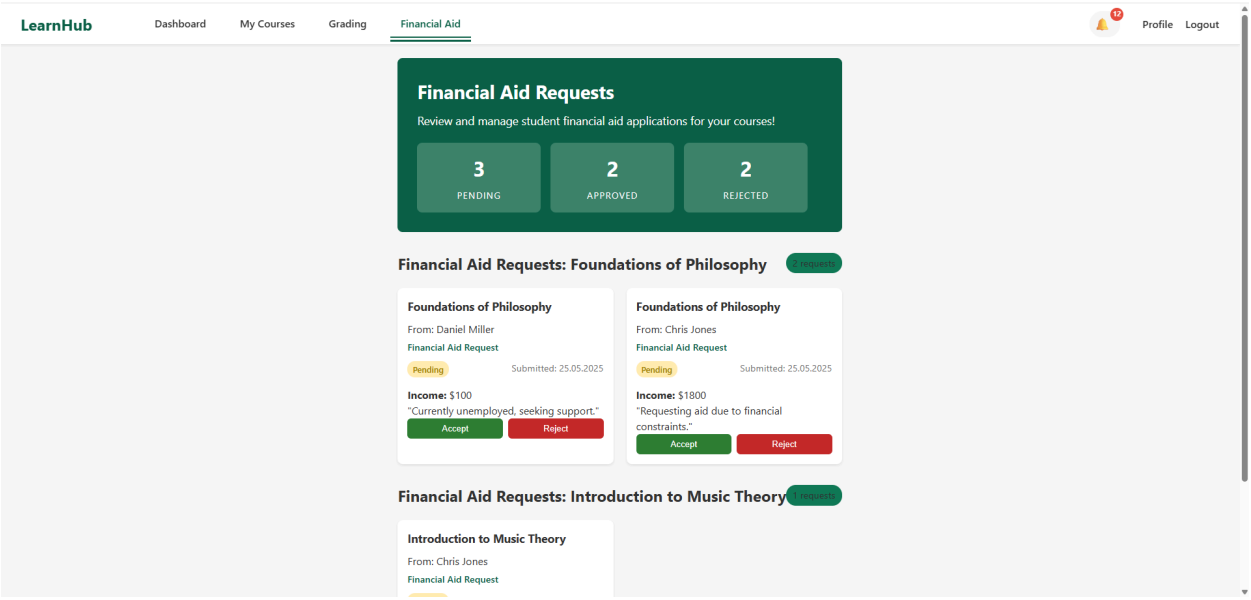
Instructor users can view all their courses categorized by status (Accepted, Draft, Pending, Rejected), access course details, and use the **Edit Content** button to update the course structure and materials. They can also view Course Details using the corresponding button.

7.4.3. Instructor Grading



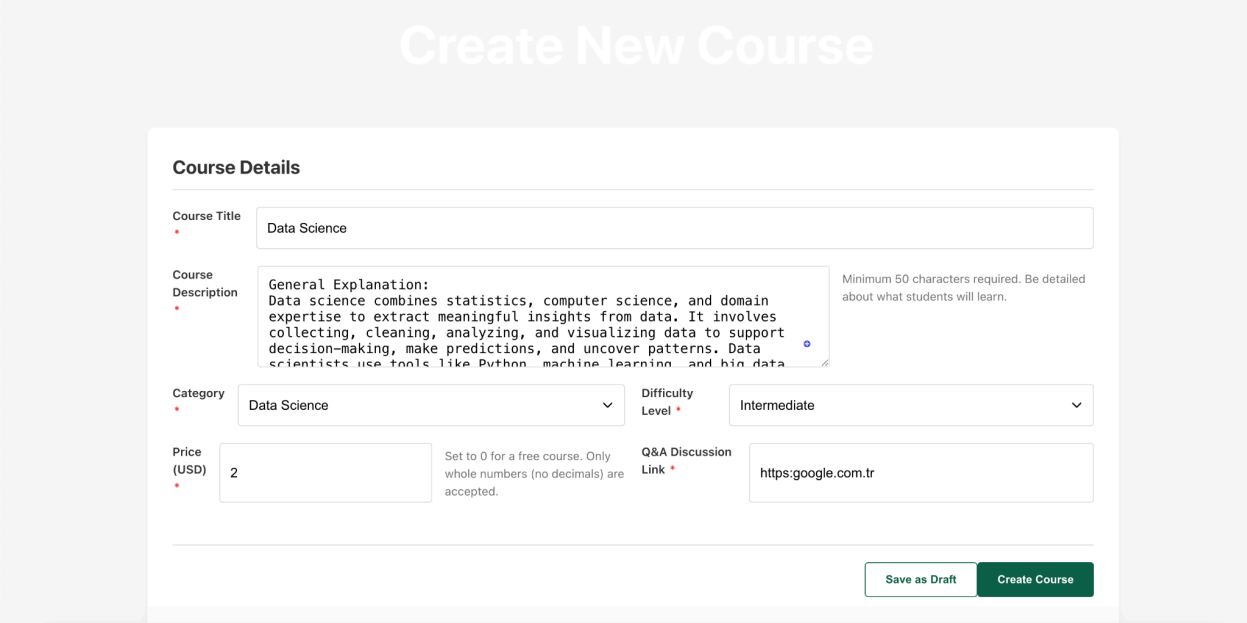
Instructors can access ungraded student submissions, such as assignment and assessment, through the grading interface. They can review answers of submissions and grade them by submitting appropriate scores based on their grading criteria.

7.4.4. Instructor Financial Aid

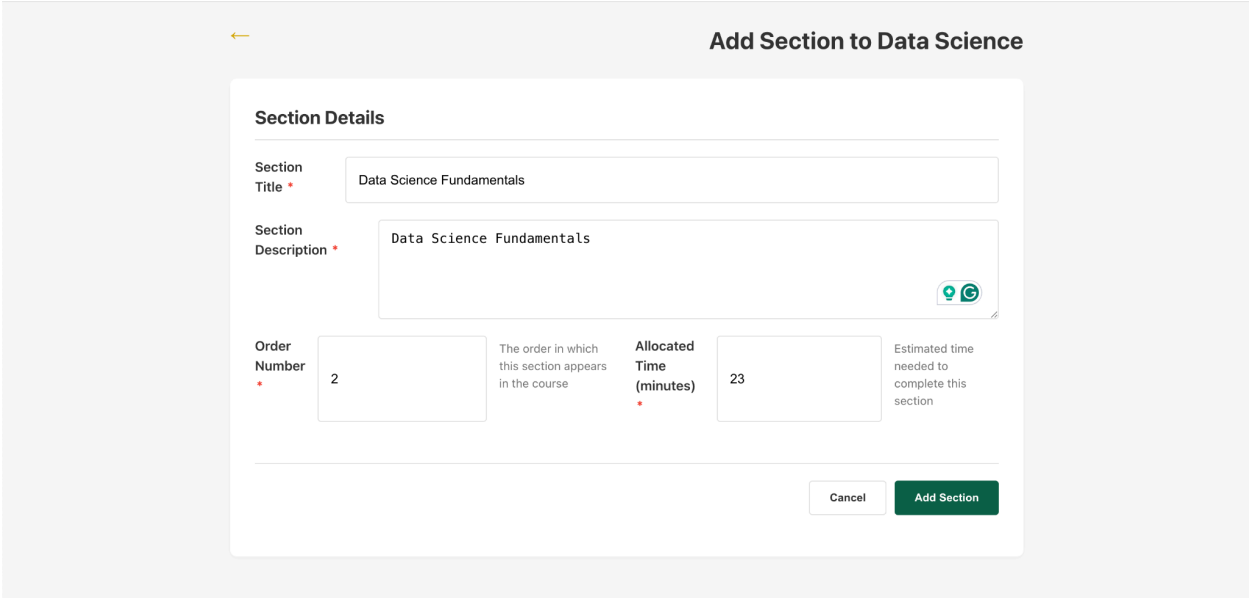


Instructor users can review financial aid requests submitted for their courses, see applicant details and income statements, and approve or reject each request directly from this interface. They can also view previously approved or rejected details.

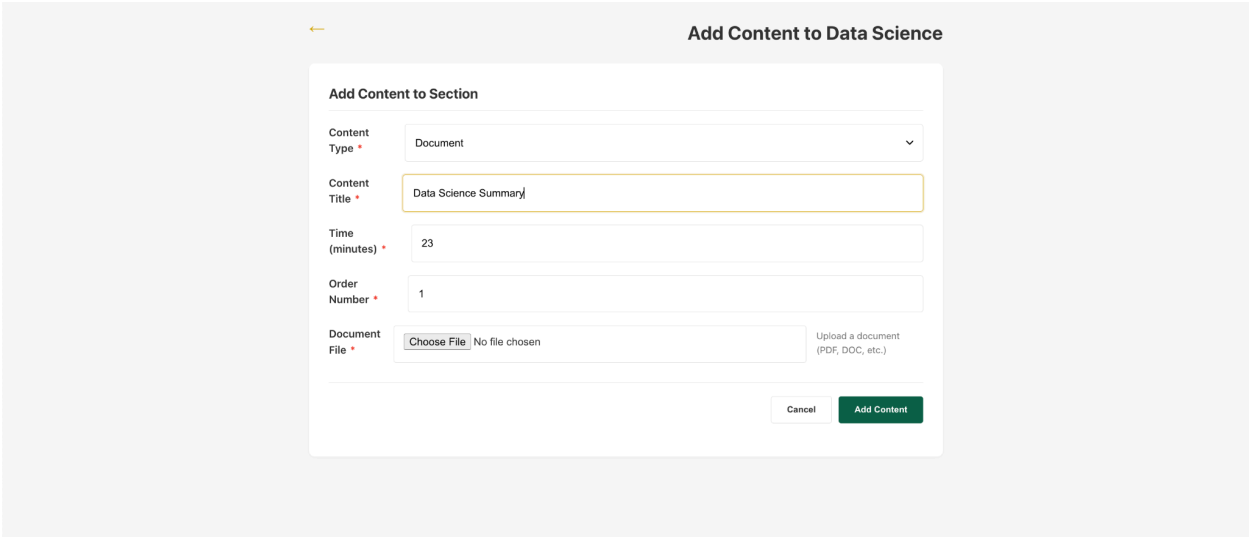
7.4.5. Instructor Course Creation



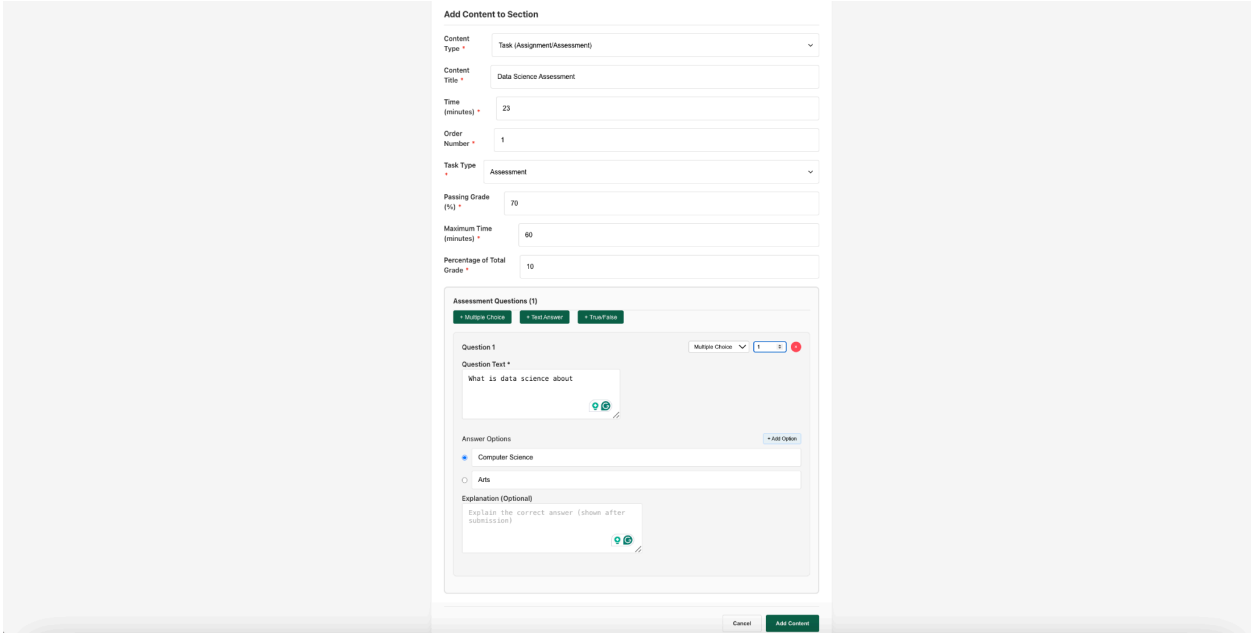
Course Details: Instructors can create new courses by providing essential details, including title, description, category, difficulty level, pricing information, and Q&A link. Once created with the save as a draft button, the course becomes a draft. Clicking on the **create course button** ensures that its state becomes pending and requires admin approval before becoming accessible to students. After the Course Details page, instructors navigated to the Add Section Page.

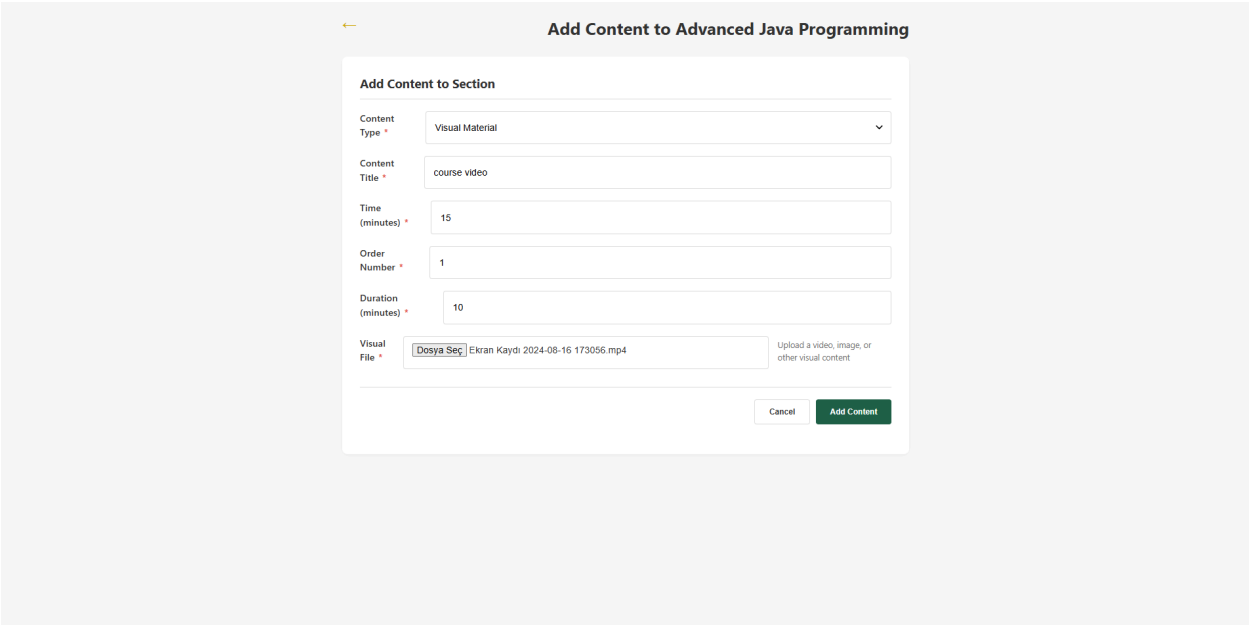


Add Section to Course Page: Each course is organized into logical sections that represent different units or modules of learning material. Instructors define section titles, descriptions, ordering, and allocated time to create a structured learning path that guides students through the course progressively. After adding a section, the instructor is redirected to the content page.



Document Type Content Addition Example



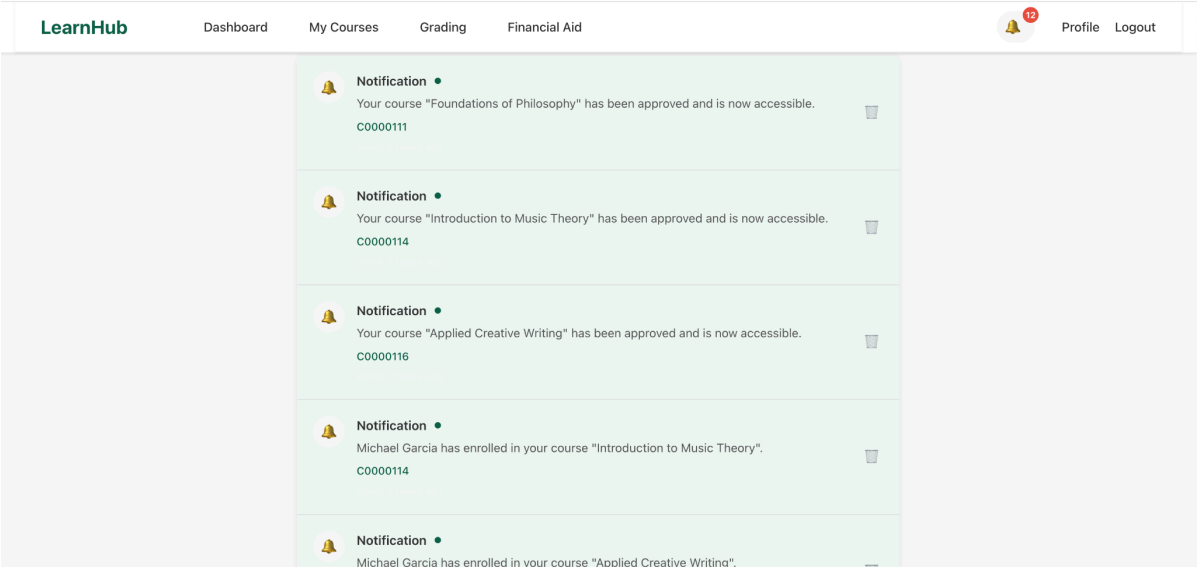


Assessment Type Content Addition Example

Add Content Page: Within each section, instructors can add three types of content: documents for reading materials, visual materials for videos and multimedia, and tasks for assessments and

assignments. Each content item includes allocated time estimates and specific ordering to ensure students experience a well-paced and comprehensive learning journey.

7.4.6. Instructor Notifications



Instructors receive automated notifications for key events related to their courses and students. They are alerted when new students enroll in their courses, when students submit feedback or complete their courses, and when financial aid applications are submitted for their courses requiring evaluation. They can click on the notification button appearing in the top right corner to access all notifications and mark them read or archived.