VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

CO2013 - DATABASE SYSTEMS

ASSIGNMENT 2

Instructor(s):	Nguyen Minh Tam	
Implementer(s):	Doan Anh Khoi	2352601
	Duong Cong Chi Dung	2352204
	Nguyen Tien Thanh	2053437
	Truong Quoc Thai	2353094

Ho Chi Minh City, 12-2025

Table of Contents

List of Figures

1 Introduction

1.1 Assignment Overview

The objective of this assignment is to transition from the conceptual database design established in the previous phase to a fully functional physical implementation. This phase focuses on using SQL to create tables, enforce complex integrity constraints, and develop database-side logic through triggers and stored procedures to ensure data consistency and automate academic workflows. Furthermore, the project demonstrates the integration between the data layer and the application layer by connecting the database to a custom-built software interface for real-time interaction.

- **Part 1: Database Creation**: Translation of the relational schema into physical tables with appropriate data types, primary/foreign keys, and row-based CHECK constraints.
- **Part 2: Database Programming**: Implementation of stored procedures for reporting, functions for academic calculations, and triggers for enforcing business rules such as assignment deadlines and quiz attempt limits.
- **Part 3: Application Development**: Development of a web interface to perform CRUD operations, retrieve filtered data, and execute database routines via the sManager account.

2 Database Implementation

2.1 Physical Database Design

The physical implementation of the `LMS_DB` database transforms the conceptual EER model into a relational schema. The design utilizes standard SQL Data Definition Language (DDL) to define tables, attributes, and relationships, ensuring data integrity through primary keys, foreign keys, and specific constraints.

2.1.1 User Management and Specialization

The system implements a "Table-per-Type" inheritance strategy to handle the specialization of users into Students, Instructors, and Administrators.

- **Users Table (Superclass):** This central table stores shared attributes such as `username`, `email`, `password_hash`, and `role`. The `user_id` serves as the primary key and is auto-incremented.
- **Subclass Tables:** `Students`, `Instructors`, and `Administrators` exist as separate tables. Their primary keys (`user_id`) simultaneously act as foreign keys referencing the `Users` table with `ON DELETE CASCADE`. This ensures referential integrity; if a user account is deleted, their specialized profile is also removed.
- **Student Codes:** Specific identity columns like `student_code` (e.g., 'STU001') are defined as primary keys for the subclass to support domain-specific identification.

```sql
-- Base User Table
CREATE TABLE Users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    username VARCHAR(50) NOT NULL UNIQUE,
    email VARCHAR(100) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    date_of_birth DATE,
    department VARCHAR(100),
    last_login DATETIME,
    status VARCHAR(20) DEFAULT 'Active',
    role ENUM('Student', 'Instructor', 'Administrator') NOT NULL
);

CREATE TABLE User_Phones (
    user_id INT,
    phone_number VARCHAR(20),
    PRIMARY KEY (user_id, phone_number),
    FOREIGN KEY (user_id) REFERENCES Users(user_id) ON DELETE CASCADE
);

-- Subclass: Students
CREATE TABLE Students (
    student_code VARCHAR(20) PRIMARY KEY,
    user_id INT NOT NULL UNIQUE,
    program VARCHAR(100),
    year_level INT,
    gpa DECIMAL(3, 2) DEFAULT 0.00,
    address_city VARCHAR(50),
    address_country VARCHAR(50),
    FOREIGN KEY (user_id) REFERENCES Users(user_id) ON DELETE CASCADE
);

-- Subclass: Instructors
CREATE TABLE Instructors (
    instructor_code VARCHAR(20) PRIMARY KEY,
    user_id INT NOT NULL UNIQUE,
    FOREIGN KEY (user_id) REFERENCES Users(user_id) ON DELETE CASCADE
);

CREATE TABLE Instructor_Specializations (
    instructor_code VARCHAR(20),
    specialization VARCHAR(100),
    PRIMARY KEY (instructor_code, specialization),
    FOREIGN KEY (instructor_code) REFERENCES
Instructors(instructor_code) ON DELETE CASCADE
);

-- Subclass: Administrators
CREATE TABLE Administrators (
    admin_code VARCHAR(20) PRIMARY KEY,
```

```
    user_id INT NOT NULL UNIQUE,
    position VARCHAR(100),
    FOREIGN KEY (user_id) REFERENCES Users(user_id) ON DELETE CASCADE
);

CREATE TABLE Admin_Privileges (
    admin_code VARCHAR(20),
    privilege VARCHAR(50),
    PRIMARY KEY (admin_code, privilege),
    FOREIGN KEY (admin_code) REFERENCES Administrators(admin_code) ON
DELETE CASCADE
);
```

2.1.2 Academic Management Tables

This module handles course offerings and registration.

- Courses: Includes `CHECK` constraints (e.g., `start_date < end_date`) to ensure temporal consistency.
- Prerequisites: Implements a recursive many-to-many relationship with a `min_grade` requirement.
- Teaching & Enrollments: Junction tables that resolve the many-to-many relationships between users and courses.

```
-- Course Entity
CREATE TABLE Courses (
    course_id INT AUTO_INCREMENT PRIMARY KEY,
    course_code VARCHAR(20) NOT NULL UNIQUE,
    title VARCHAR(255) NOT NULL,
    credits INT CHECK (credits > 0),
    department VARCHAR(100),
    academic_level VARCHAR(50),
    max_capacity INT DEFAULT 60,
    start_date DATE,
    end_date DATE,
    description TEXT,
    enrollment_start_date DATETIME,
    enrollment_end_date DATETIME,
    status VARCHAR(20) DEFAULT 'Planned',
    passing_score DECIMAL(5,2) DEFAULT 5.00,
    CONSTRAINT chk_course_dates CHECK (start_date < end_date)
);

-- Recursive Relationship: Prerequisites
CREATE TABLE Prerequisites (
    course_id INT,
    prerequisite_id INT,
    min_grade DECIMAL(3,2) DEFAULT 5.0,
    PRIMARY KEY (course_id, prerequisite_id),
```

```
    FOREIGN KEY (course_id) REFERENCES Courses(course_id) ON DELETE
CASCADE,
    FOREIGN KEY (prerequisite_id) REFERENCES Courses(course_id) ON
DELETE CASCADE
);

-- Teaching Assignments
CREATE TABLE Teaching (
    instructor_code VARCHAR(20),
    course_id INT,
    semester VARCHAR(20),
    role VARCHAR(50) DEFAULT 'Primary',
    PRIMARY KEY (instructor_code, course_id, semester),
    FOREIGN KEY (instructor_code) REFERENCES
Instructors(instructor_code) ON DELETE CASCADE,
    FOREIGN KEY (course_id) REFERENCES Courses(course_id) ON DELETE
CASCADE
);

-- Student Enrollments
CREATE TABLE Enrollments (
    enrollment_id INT AUTO_INCREMENT PRIMARY KEY,
    student_code VARCHAR(20),
    course_id INT,
    enrollment_date DATE DEFAULT (CURRENT_DATE),
    final_grade DECIMAL(4, 2) DEFAULT NULL,
    completion_status VARCHAR(20) DEFAULT 'Active',
    FOREIGN KEY (student_code) REFERENCES Students(student_code) ON
DELETE CASCADE,
    FOREIGN KEY (course_id) REFERENCES Courses(course_id) ON DELETE
CASCADE,
    UNIQUE (student_code, course_id)
);
```

2.1.3 Course Content Organization

Content is structured hierarchically. `Sections` act as containers within a course, and `Materials` (files, videos) are linked to specific sections.

```
-- Course Sections
CREATE TABLE Sections (
    section_id INT AUTO_INCREMENT PRIMARY KEY,
    course_id INT NOT NULL,
    title VARCHAR(100),
    description TEXT,
    order_num INT,
    visibility BOOLEAN DEFAULT TRUE,
    available_from DATETIME,
    available_until DATETIME,
```

```
    FOREIGN KEY (course_id) REFERENCES Courses(course_id) ON DELETE
CASCADE,
    UNIQUE (course_id, order_num)
);

-- Learning Materials
CREATE TABLE Materials (
    material_id INT AUTO_INCREMENT PRIMARY KEY,
    section_id INT NOT NULL,
    title VARCHAR(255),
    type VARCHAR(50),
    file_path TEXT,
    file_size VARCHAR(20),
    upload_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    visibility BOOLEAN DEFAULT TRUE,
    FOREIGN KEY (section_id) REFERENCES Sections(section_id) ON DELETE
CASCADE
);
```

2.1.4 Assessments (Assignments & Quizzes)

This module manages student evaluation.

- Assignments: Enforces deadlines via `CHECK (opening_date < due_date)` and tracks late submissions.
- Quizzes: A complex composite entity involving `Quizzes` (settings), `Questions`, `Choices`, and `Attempts`.

```
-- Assignments
CREATE TABLE Assignments (
    assignment_id INT AUTO_INCREMENT PRIMARY KEY,
    course_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    instruction TEXT,
    weight DECIMAL(5, 2),
    max_score INT DEFAULT 100,
    opening_date DATETIME,
    due_date DATETIME,
    submission_type VARCHAR(50),
    late_submission_allowed BOOLEAN DEFAULT FALSE,
    late_penalty DECIMAL(5,2) DEFAULT 0.00,
    FOREIGN KEY (course_id) REFERENCES Courses(course_id) ON DELETE
CASCADE,
    CONSTRAINT chk_assign_dates CHECK (opening_date < due_date)
);

-- Student Submissions
CREATE TABLE Submissions (
    submission_id INT AUTO_INCREMENT PRIMARY KEY,
    assignment_id INT NOT NULL,
```

```
    student_code VARCHAR(20) NOT NULL,
    date DATETIME DEFAULT CURRENT_TIMESTAMP,
    content TEXT,
    file_path TEXT,
    file_name VARCHAR(255),
    score DECIMAL(5, 2),
    status VARCHAR(20) DEFAULT 'Submitted',
    is_late BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (assignment_id) REFERENCES Assignments(assignment_id) ON
DELETE CASCADE,
    FOREIGN KEY (student_code) REFERENCES Students(student_code) ON
DELETE CASCADE,
    UNIQUE (assignment_id, student_code)
);

-- Quizzes
CREATE TABLE Quizzes (
    quiz_id INT AUTO_INCREMENT PRIMARY KEY,
    course_id INT,
    title VARCHAR(255),
    description TEXT,
    open_time DATETIME,
    close_time DATETIME,
    time_limit_minutes INT,
    attempts_allowed INT DEFAULT 1,
    grading_method ENUM('Highest', 'Average', 'Last') DEFAULT 'Highest',
    shuffle_questions BOOLEAN DEFAULT FALSE,
    shuffle_answers BOOLEAN DEFAULT FALSE,
    passing_score DECIMAL(5,2),
    total_points DECIMAL(6,2) DEFAULT 0.00,
    FOREIGN KEY (course_id) REFERENCES Courses(course_id) ON DELETE
CASCADE,
    CONSTRAINT chk_quiz_dates CHECK (open_time < close_time)
);

-- Quiz Questions
CREATE TABLE Questions (
    question_id INT AUTO_INCREMENT PRIMARY KEY,
    quiz_id INT NOT NULL,
    question_text TEXT NOT NULL,
    question_type ENUM('MultipleChoice', 'TrueFalse', 'ShortAnswer') NOT
NULL,
    points DECIMAL(3, 1) DEFAULT 1.0,
    correct_answer TEXT,
    FOREIGN KEY (quiz_id) REFERENCES Quizzes(quiz_id) ON DELETE CASCADE
);

CREATE TABLE Question_Choices (
    choice_id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    question_id INT NOT NULL,
    choice_text TEXT NOT NULL,
    is_correct BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (question_id) REFERENCES Questions(question_id) ON
DELETE CASCADE
);

-- Quiz Execution (Attempts & Answers)
CREATE TABLE Quiz_Attempts (
    attempt_id INT AUTO_INCREMENT PRIMARY KEY,
    quiz_id INT,
    student_code VARCHAR(20),
    attempt_number INT,
    start_time DATETIME DEFAULT CURRENT_TIMESTAMP,
    completion_time DATETIME,
    duration INT,
    total_score DECIMAL(5, 2),
    status VARCHAR(20) DEFAULT 'InProgress',
    FOREIGN KEY (quiz_id) REFERENCES Quizzes(quiz_id) ON DELETE CASCADE,
    FOREIGN KEY (student_code) REFERENCES Students(student_code) ON
DELETE CASCADE
);

CREATE TABLE Quiz_Answers (
    answer_id INT AUTO_INCREMENT PRIMARY KEY,
    attempt_id INT NOT NULL,
    question_id INT NOT NULL,
    selected_choice_id INT NULL,
    text_answer TEXT NULL,
    is_correct BOOLEAN DEFAULT FALSE,
    points_earned DECIMAL(4, 2) DEFAULT 0.00,
    time_taken INT,
    answered_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (attempt_id) REFERENCES Quiz_Attempts(attempt_id) ON
DELETE CASCADE,
    FOREIGN KEY (question_id) REFERENCES Questions(question_id) ON
DELETE CASCADE,
    FOREIGN KEY (selected_choice_id) REFERENCES
Question_Choices(choice_id) ON DELETE CASCADE,
    UNIQUE KEY unique_attempt_question (attempt_id, question_id)
);
```

2.2 Data Population

2.2.1 Insertion Strategy & Referential Integrity

To ensure data consistency and satisfy Foreign Key constraints, data population was executed in a specific hierarchical order. Parent entities were populated before child entities:

1. **User Accounts:** `Users` table populated first to generate `user_id`s.
2. **User Subclasses:** `Administrators`, `Instructors`, and `Students` linked to `Users`.
3. **Academic Catalog:** `Courses` created next.
4. **Course Relations:** `Prerequisites` and `Teaching` assignments.
5. **Course Content:** `Sections` followed by `Materials`.
6. **Assessments:** `Assignments` and `Quizzes` (including `Questions` and `Choices`).
7. **Student Activity:** `Enrollments`, `Submissions`, and `Quiz_Attempts`.

2.2.2 Complete Data Population Script

The following DML script was executed to populate the database with realistic sample data.

```
USE LMS_DB;

-- 1. BASE USERS
INSERT INTO Users (username, email, password_hash, first_name,
last_name, date_of_birth, department, last_login, status, role) VALUES
('admin1', 'admin1@bk.edu.vn', 'hash_admin1', 'System', 'Admin',
'1980-05-15', 'IT Department', '2025-12-07 08:30:00', 'Active',
'Administrator'),
('admin2', 'admin2@bk.edu.vn', 'hash_admin2', 'Huy', 'Vo', '1982-03-22',
'Academic Affairs', '2025-12-06 14:20:00', 'Active', 'Administrator'),
('admin3', 'admin3@bk.edu.vn', 'hash_admin3', 'Mai', 'Tran',
'1985-11-08', 'Student Services', '2025-12-05 09:15:00', 'Active',
'Administrator'),
('admin4', 'admin4@bk.edu.vn', 'hash_admin4', 'Khanh', 'Nguyen',
'1978-07-30', 'Finance', '2025-12-04 16:45:00', 'Active',
'Administrator'),
('tam_nguyen', 'tam@bk.edu.vn', 'hash_ins1', 'Tam', 'Nguyen',
'1975-09-12', 'CSE', '2025-12-07 10:00:00', 'Active', 'Instructor'),
('minh_le', 'minh@bk.edu.vn', 'hash_ins2', 'Minh', 'Le', '1980-02-28',
'CSE', '2025-12-07 09:30:00', 'Active', 'Instructor'),
('hoa_pham', 'hoa@bk.edu.vn', 'hash_ins3', 'Hoa', 'Pham', '1978-06-15',
'CSE', '2025-12-06 11:45:00', 'Active', 'Instructor'),
('duc_tran', 'duc@bk.edu.vn', 'hash_ins4', 'Duc', 'Tran', '1983-12-03',
'CSE', '2025-12-07 14:00:00', 'Active', 'Instructor'),
('long_tran', 'long@bk.edu.vn', 'hash_stu1', 'Long', 'Tran',
'2003-04-20', 'CSE', '2025-12-07 08:00:00', 'Active', 'Student'),
('thanh_nguyen', 'thanh@bk.edu.vn', 'hash_stu2', 'Thanh', 'Nguyen',
'2003-08-10', 'CSE', '2025-12-07 07:45:00', 'Active', 'Student'),
('dung_duong', 'dung@bk.edu.vn', 'hash_stu3', 'Dung', 'Duong',
'2002-01-25', 'CSE', '2025-12-06 20:30:00', 'Active', 'Student'),
('khoi_doan', 'khoi@bk.edu.vn', 'hash_stu4', 'Khoi', 'Doan',
'2004-11-18', 'CSE', '2025-12-06 19:00:00', 'Active', 'Student'),
('thai_truong', 'thai@bk.edu.vn', 'hash_stu5', 'Thai', 'Truong',
'2004-05-07', 'EE', '2025-12-05 22:15:00', 'Active', 'Student');

INSERT INTO User_Phones (user_id, phone_number) VALUES
(1, '0901111111'), (2, '0901222222'), (3, '0901333333'), (4,
```

```
'0901444444'),
(5, '0902222222'), (5, '0903333333'), (6, '0904444444'), (7,
'0905555555'),
(9, '0906666666'), (10, '0907777777');

-- 2. SPECIALIZED ROLES
INSERT INTO Administrators (admin_code, user_id, position) VALUES
('ADM001', 1, 'IT Manager'),
('ADM002', 2, 'Academic Director'),
('ADM003', 3, 'Student Affairs Coordinator'),
('ADM004', 4, 'Financial Controller');

INSERT INTO Admin_Privileges (admin_code, privilege) VALUES
('ADM001', 'Manage Users'), ('ADM001', 'Manage Courses'), ('ADM001',
'System Config'),
('ADM002', 'Manage Courses'), ('ADM002', 'View Reports'), ('ADM002',
'Manage Schedules'),
('ADM003', 'Manage Users'), ('ADM003', 'View Reports'), ('ADM003',
'Handle Complaints'),
('ADM004', 'View Reports'), ('ADM004', 'System Config'), ('ADM004',
'Manage Schedules');

INSERT INTO Instructors (instructor_code, user_id) VALUES
('INS001', 5), ('INS002', 6), ('INS003', 7), ('INS004', 8);

INSERT INTO Instructor_Specializations (instructor_code, specialization)
VALUES
('INS001', 'Database Systems'), ('INS001', 'Data Mining'),
('INS002', 'Software Engineering'), ('INS002', 'Agile Methodology'),
('INS003', 'Computer Networks'), ('INS003', 'Cybersecurity'),
('INS004', 'Operating Systems'), ('INS004', 'Cloud Computing');

-- Note: The before_student_insert trigger ensures IDs match the pattern
'STU00X'
INSERT INTO Students (student_code, user_id, program, year_level, gpa,
address_city, address_country) VALUES
('STU001', 9, 'Computer Science', 2, 3.45, 'Ho Chi Minh City',
'Vietnam'),
('STU002', 10, 'Computer Science', 2, 3.20, 'Ho Chi Minh City',
'Vietnam'),
('STU003', 11, 'Computer Engineering', 3, 3.65, 'Hanoi', 'Vietnam'),
('STU004', 12, 'Computer Science', 1, 3.10, 'Da Nang', 'Vietnam'),
('STU005', 13, 'Electrical Engineering', 1, 2.95, 'Ho Chi Minh City',
'Vietnam');

-- 3. ACADEMIC CATALOG
INSERT INTO Courses (course_code, title, credits, department,
academic_level, max_capacity, start_date, end_date, description,
enrollment_start_date, enrollment_end_date, status, passing_score)
```

```
VALUES
('CO2013', 'Database Systems', 3, 'CSE', 'Undergraduate', 60,
'2025-09-05', '2025-12-20', 'Introduction to database concepts,
relational model, SQL...', '2025-08-01 00:00:00', '2025-08-30 23:59:59',
'Active', 5.00),
('CO3001', 'Software Engineering', 3, 'CSE', 'Undergraduate', 50,
'2025-09-05', '2025-12-20', 'Covers SDLC, requirements engineering,
design patterns...', '2025-08-01 00:00:00', '2025-08-30 23:59:59',
'Active', 5.00),
('CO1005', 'Intro to Computing', 3, 'CSE', 'Undergraduate', 80,
'2024-09-05', '2024-12-20', 'Foundational course covering basic
programming...', '2024-08-01 00:00:00', '2024-08-30 23:59:59',
'Completed', 5.00),
('CO2001', 'Operating Systems', 4, 'CSE', 'Undergraduate', 55,
'2025-09-05', '2025-12-20', 'Study of OS concepts: process management,
memory...', '2025-08-01 00:00:00', '2025-08-30 23:59:59', 'Active',
5.00),
('CO3005', 'Computer Networks', 3, 'CSE', 'Undergraduate', 45,
'2025-02-15', '2025-06-20', 'Network architecture, protocols, TCP/IP
stack...', '2025-01-01 00:00:00', '2025-01-30 23:59:59', 'Planned',
5.00),
('CO4001', 'Machine Learning', 3, 'CSE', 'Graduate', 40, '2025-09-05',
'2025-12-20', 'Intro to ML algorithms: supervised, unsupervised...',
'2025-08-01 00:00:00', '2025-08-30 23:59:59', 'Active', 6.00);

INSERT INTO Prerequisites (course_id, prerequisite_id, min_grade) VALUES
(1, 3, 5.00), (2, 3, 5.00), (2, 1, 7.00), (4, 3, 5.00), (6, 1, 6.00);

INSERT INTO Teaching (instructor_code, course_id, semester, role) VALUES
('INS001', 1, 'HK251', 'Primary'), ('INS001', 3, 'HK241', 'Primary'),
('INS002', 2, 'HK251', 'Primary'), ('INS003', 5, 'HK252', 'Primary'),
('INS004', 4, 'HK251', 'Primary'), ('INS001', 6, 'HK251', 'Primary');

-- 4. COURSE CONTENT
INSERT INTO Sections (course_id, title, description, order_num,
visibility, available_from, available_until) VALUES
(1, 'Introduction to Databases', 'Overview of database concepts', 1,
TRUE, '2025-09-05 00:00:00', '2025-12-20 23:59:59'),
(1, 'Relational Model', 'Understanding relational data model', 2, TRUE,
'2025-09-12 00:00:00', '2025-12-20 23:59:59'),
(1, 'SQL Fundamentals', 'Basic SQL queries', 3, TRUE, '2025-09-19
00:00:00', '2025-12-20 23:59:59'),
(1, 'Database Normalization', 'Normal forms and optimization', 4, TRUE,
'2025-10-03 00:00:00', '2025-12-20 23:59:59'),
(2, 'Introduction to SE', 'SDLC models', 1, TRUE, '2025-09-05 00:00:00',
'2025-12-20 23:59:59'),
(2, 'Requirements Engineering', 'Gathering requirements', 2, TRUE,
'2025-09-19 00:00:00', '2025-12-20 23:59:59'),
(4, 'Process Management', 'Scheduling algorithms', 1, TRUE, '2025-09-05
```

```
00:00:00', '2025-12-20 23:59:59'),
(4, 'Memory Management', 'Virtual memory, paging', 2, TRUE, '2025-09-26
00:00:00', '2025-12-20 23:59:59');

INSERT INTO Materials (section_id, title, type, file_path, file_size,
upload_date, visibility) VALUES
(1, 'Course Syllabus', 'File', '/materials/co2013/syllabus.pdf', '245
KB', '2025-09-01 10:00:00', TRUE),
(1, 'Lecture 1 Video', 'Video', '/materials/co2013/lecture1.mp4', '520
MB', '2025-09-05 08:00:00', TRUE),
(2, 'Relational Algebra', 'File', '/materials/co2013/
relational_algebra.pdf', '180 KB', '2025-09-12 10:00:00', TRUE),
(3, 'SQL Cheat Sheet', 'File', '/materials/co2013/sql_cheatsheet.pdf',
'95 KB', '2025-09-19 10:00:00', TRUE),
(7, 'Process States', 'File', '/materials/co2001/process_states.pdf',
'450 KB', '2025-09-05 11:00:00', TRUE);

-- 5. ASSESSMENTS
INSERT INTO Assignments (course_id, title, instruction, weight,
max_score, opening_date, due_date, submission_type,
late_submission_allowed, late_penalty) VALUES
(1, 'ER Modeling', 'Design ER diagram for library system', 15.00, 100,
'2025-10-01 00:00:00', '2025-10-15 23:59:59', 'File Upload', TRUE,
10.00),
(1, 'SQL Queries', 'Solve SQL problems', 20.00, 100, '2025-10-16
00:00:00', '2025-11-01 23:59:59', 'File Upload', FALSE, 0.00),
(2, 'Requirements Doc', 'Create SRS document', 20.00, 100, '2025-10-05
00:00:00', '2025-10-20 23:59:59', 'File Upload', TRUE, 15.00),
(4, 'Process Scheduling', 'Implement simulator', 30.00, 100, '2025-10-20
00:00:00', '2025-11-15 23:59:59', 'File Upload', TRUE, 10.00);

INSERT INTO Quizzes (course_id, title, description, open_time,
close_time, time_limit_minutes, attempts_allowed, grading_method,
shuffle_questions, shuffle_answers, passing_score, total_points) VALUES
(1, 'Quiz 1: DB Intro', 'Basic concepts', '2025-09-10 08:00:00',
'2025-09-17 20:00:00', 30, 2, 'Highest', TRUE, TRUE, 5.00, 10.00),
(1, 'Quiz 2: SQL', 'SELECT, INSERT...', '2025-10-01 08:00:00',
'2025-10-08 20:00:00', 45, 1, 'Highest', TRUE, TRUE, 5.00, 10.00),
(4, 'Quiz 1: Process Mgmt', 'States, PCB', '2025-10-01 08:00:00',
'2025-10-08 20:00:00', 60, 1, 'Average', TRUE, FALSE, 5.00, 10.00);

-- Questions & Choices (Sample)
INSERT INTO Questions (quiz_id, question_text, question_type, points,
correct_answer) VALUES
(1, 'What does DBMS stand for?', 'MultipleChoice', 1.0, NULL),
(1, 'Define Primary Key', 'ShortAnswer', 2.0, 'Unique identifier...'),
(1, 'DB can have multiple PKs', 'TrueFalse', 1.0, 'False');

INSERT INTO Question_Choices (question_id, choice_text, is_correct)
```

```
VALUES
(1, 'Database Management System', TRUE), (1, 'Data Base Make System',
FALSE),
(3, 'True', FALSE), (3, 'False', TRUE);

-- 6. STUDENT ACTIVITY
INSERT INTO Enrollments (student_code, course_id, enrollment_date,
final_grade, completion_status) VALUES
('STU001', 1, '2025-08-15', NULL, 'Active'), ('STU002', 1, '2025-08-16',
NULL, 'Active'),
('STU003', 1, '2025-08-14', NULL, 'Active'), ('STU004', 1, '2025-08-20',
NULL, 'Active'),
('STU001', 3, '2024-08-15', 8.50, 'Completed'), ('STU002', 3,
'2024-08-16', 7.00, 'Completed');

INSERT INTO Submissions (assignment_id, student_code, date, content,
file_path, file_name, score, status, is_late) VALUES
(1, 'STU001', '2025-10-14 10:30:00', 'ER diagram...', '/sub/
stu001_er.pdf', 'er.pdf', 90.00, 'Graded', FALSE),
(1, 'STU004', '2025-10-16 08:00:00', 'Late submission', '/sub/
stu004_er.pdf', 'er.pdf', 72.00, 'Graded', TRUE);

INSERT INTO Quiz_Attempts (quiz_id, student_code, attempt_number,
start_time, completion_time, duration, total_score, status) VALUES
(1, 'STU001', 1, '2025-09-11 09:00:00', '2025-09-11 09:25:00', 25, 9.00,
'Finished'),
(1, 'STU002', 1, '2025-09-11 10:00:00', '2025-09-11 10:28:00', 28, 8.00,
'Finished');

INSERT INTO Quiz_Answers (attempt_id, question_id, selected_choice_id,
text_answer, is_correct, points_earned, time_taken, answered_at) VALUES
(1, 1, 1, NULL, TRUE, 1.00, 45, '2025-09-11 09:02:00'),
(1, 2, NULL, 'A primary key uniquely identifies...', TRUE, 2.00, 120,
'2025-09-11 09:05:00');
```

3 Database Programming

This section details the server-side logic implemented to automate processes, validate data, and enforce complex business rules that cannot be handled by standard constraints.

3.1 Stored Procedures

Two stored procedures were implemented to handle transactional logic (registration) and analytical reporting (course statistics).

3.1.1 Student Registration (`register_student_course`)

This procedure handles the logic for enrolling a student in a course. It performs necessary validations regarding course existence and capacity before allowing the insertion.

- **Input Parameters:**
  - `p_stu` (VARCHAR): The student code (e.g., 'STU001').
  - `p_course` (VARCHAR): The course code (e.g., 'CO2013').
- **Output Parameters:**
  - `p_msg` (VARCHAR): Returns status messages ('Success', 'Course Full', etc.).
- **Logic:** Uses `IF/ELSE` control flow to check if current enrollments count `<` max capacity.

```
CREATE PROCEDURE register_student_course(IN p_stu VARCHAR(20), IN
p_course VARCHAR(20), OUT p_msg VARCHAR(255))
BEGIN
    DECLARE v_cid INT;
    DECLARE v_cap INT;
    DECLARE v_curr INT;

    SELECT course_id, max_capacity INTO v_cid, v_cap FROM Courses WHERE
course_code = p_course;

    IF v_cid IS NULL THEN
        SET p_msg = 'Course not found';
    ELSE
        SELECT COUNT(*) INTO v_curr FROM Enrollments WHERE course_id =
v_cid;
        IF v_curr ⩾ v_cap THEN
            SET p_msg = 'Course Full';
        ELSE
            INSERT INTO Enrollments (student_code, course_id) VALUES
(p_stu, v_cid);
            SET p_msg = 'Success';
        END IF;
    END IF;
END //
```

3.1.2 Course Statistics Report (`get_course_stats`)

This procedure generates a performance report for a specific semester, demonstrating the use of aggregate functions and multi-table joins.

- **Input Parameters:** `p_semester` (VARCHAR).
- **Logic:** Joins `Courses`, `Enrollments`, and `Teaching` tables. Uses `GROUP BY` to calculate totals and `AVG` for grades, filtered by the `HAVING` clause.

```
CREATE PROCEDURE get_course_stats(IN p_semester VARCHAR(20))
BEGIN
    SELECT c.course_code, c.title, COUNT(e.student_code) as
total_enrolled, AVG(e.final_grade) as avg_grade
    FROM Courses c
    JOIN Enrollments e ON c.course_id = e.course_id
    JOIN Teaching t ON c.course_id = t.course_id
    WHERE t.semester = p_semester
    GROUP BY c.course_id
    HAVING total_enrolled > 0
    ORDER BY avg_grade DESC;
END //
```

3.2 Stored Functions

Functions were created to encapsulate reusable academic calculations.

3.2.1 Prerequisite Validation (`check_prereq_status`)

Determines if a student is eligible to take a course based on their academic history.

- **Input Parameters:** `p_student_code` (VARCHAR), `p_target_course_id` (INT).
- **Return Type:** `BOOLEAN` (True/False).
- **Logic:** Checks if there exists any prerequisite for the target course where the student has not achieved the required `min_grade`.

```
CREATE FUNCTION check_prereq_status(p_student_code VARCHAR(20),
p_target_course_id INT)
RETURNS BOOLEAN
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE v_missing INT;
    SELECT COUNT(*) INTO v_missing
    FROM Prerequisites p
    WHERE p.course_id = p_target_course_id
    AND NOT EXISTS (
        SELECT 1
        FROM Enrollments e
        WHERE e.student_code = p_student_code
        AND e.course_id = p.prerequisite_id
        AND e.completion_status = 'Completed'
        AND e.final_grade ⩾ p.min_grade
```

```
    );
    RETURN (v_missing = 0);
END //
```

3.2.2 Credit Calculation (`calculate_credits_earned`)

Calculates the total credits a student has successfully accumulated.

- **Input Parameters:** `p_student_code` (VARCHAR).
- **Return Type:** `INT` (Total credits earned).
- **Logic:** Sums credits only for courses where `final_grade` meets the `passing_score`.

```
CREATE FUNCTION calculate_credits_earned(p_student_code VARCHAR(20))
RETURNS INT
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE v_credits INT;
    SELECT SUM(c.credits) INTO v_credits
    FROM Enrollments e
    JOIN Courses c ON e.course_id = c.course_id
    WHERE e.student_code = p_student_code AND e.final_grade ⩾
c.passing_score;
    RETURN IFNULL(v_credits, 0);
END //
```

3.3 Triggers

Triggers were implemented to enforce strict business rules and automate derived data calculations.

3.3.1 Business Rule: Attempt Limits (`before_quiz_attempt`)

This `BEFORE INSERT` trigger enforces the rule that students cannot exceed the `attempts_allowed` set for a quiz. It also auto-calculates the `attempt_number`.

```
CREATE TRIGGER before_quiz_attempt
BEFORE INSERT ON Quiz_Attempts
FOR EACH ROW
BEGIN
    DECLARE v_max INT;
    DECLARE v_count INT;

    SELECT attempts_allowed INTO v_max FROM Quizzes WHERE quiz_id =
NEW.quiz_id;
    SELECT COUNT(*) INTO v_count FROM Quiz_Attempts WHERE quiz_id =
NEW.quiz_id AND student_code = NEW.student_code;

    IF v_count ⩾ v_max THEN
```

```
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error: Max attempts
reached';
    END IF;

    SET NEW.attempt_number = v_count + 1;
END //
```

3.3.2 Business Rule: Submission Deadlines (`validate_submission`)

This trigger enforces deadline policies. If a submission is late, it checks if the assignment allows late work. If allowed, it flags the submission as `is_late`; otherwise, it rejects the insert.

```
CREATE TRIGGER validate_submission
BEFORE INSERT ON Submissions
FOR EACH ROW
BEGIN
    DECLARE v_due DATETIME;
    DECLARE v_late_ok BOOLEAN;

    SELECT due_date, late_submission_allowed INTO v_due, v_late_ok FROM
Assignments WHERE assignment_id = NEW.assignment_id;

    IF NEW.date > v_due THEN
        IF v_late_ok = FALSE THEN
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error: Late
submission not allowed';
        ELSE
            SET NEW.is_late = TRUE;
        END IF;
    END IF;
END //
```

3.3.3 Derived Data: Auto-GPA (`update_enrollment_gpa`)

This `AFTER UPDATE` trigger automatically recalculates and updates a student's GPA in the `Students` table whenever a grade is finalized in the `Enrollments` table.

```
CREATE TRIGGER update_enrollment_gpa
AFTER UPDATE ON Enrollments
FOR EACH ROW
BEGIN
    DECLARE v_gpa DECIMAL(3,2);
    IF NEW.final_grade IS NOT NULL THEN
        SELECT AVG(final_grade) INTO v_gpa FROM Enrollments WHERE
student_code = NEW.student_code;
        UPDATE Students SET gpa = v_gpa WHERE student_code =
NEW.student_code;
```

```
    END IF;
END //
```

3.3.4 Identity Management: Student Code Generation (`before_student_insert`)

This trigger handles the auto-increment logic for alphanumeric keys.

```
CREATE TRIGGER before_student_insert
BEFORE INSERT ON Students
FOR EACH ROW
BEGIN
    DECLARE next_id INT;
    SELECT IFNULL(MAX(CAST(SUBSTRING(student_code, 4) AS UNSIGNED)), 0)
+ 1 INTO next_id FROM Students;
    SET NEW.student_code = CONCAT('STU', LPAD(next_id, 3, '0'));
END //
```

4 Application Building

4.1 Development Environment & Technology Stack

4.1.1 Overview

The application is built using modern web technologies optimized for performance, developer experience, and maintainability. The core technology stack consists of:

- **Frontend Framework:** Next.js 16.0.5 with React 19.2.0 for server-side rendering and modern UI development
- **Styling:** TailwindCSS 4 for utility-first responsive design
- **Database:** MySQL 8.0+ for robust relational data management
- **Type Safety:** TypeScript 5 for compile-time type checking and improved code quality
- **Form Management:** React Hook Form with Zod for efficient form handling and validation
- **Authentication:** Jose library for JWT-based secure session management

The technology choices prioritize developer productivity through strong typing and modern tooling, while ensuring production-grade performance through server-side rendering and optimized build processes.

4.1.2 Frameworks & Libraries

Core Framework: Next.js 16.0.5

- Provides server-side rendering (SSR) and static site generation (SSG)
- Built-in API routes for backend functionality
- Automatic code splitting and optimization
- File-based routing system

Database Driver:

- mysql2 (v3.15.3) for MySQL connectivity with Promise support

- Connection pooling for efficient database access

4.1.3 Database Management System

Database: MySQL 8.0+

MySQL was chosen for its:

- **Robust ACID compliance:** Ensuring data integrity for academic records
- **Advanced features:** Support for stored procedures, triggers, and functions required for complex business logic
- **Performance:** Efficient handling of complex joins and aggregate queries
- **Maturity:** Industry-standard RDBMS with extensive documentation and community support
- **Transaction support:** Critical for operations like student registration and grade updates

Database Access:

- Connection configuration centralized in `lib/db.ts`

```
import mysql from 'mysql2/promise';

export const dbConfig = {
    host: 'localhost',
    user: 'sManager',
    password: 'sManagerPass123!',
    database: 'LMS_DB',
};

export async function query({ query, values = [] }: { query: string;
values?: unknown[] }) {
    const db = await mysql.createConnection(dbConfig);
    try{
        // eslint-disable-next-line @typescript-eslint/no-explicit-any
        const [results] = await db.execute(query, values as any[]);
        // eslint-disable-next-line @typescript-eslint/no-explicit-any
        return results as any;
    } catch (error) {
        throw error;
    } finally{
        await db.end();
    }
}
```

- Parameterized queries to prevent SQL injection
- Promise-based async/await pattern for cleaner code
- Connection management with automatic cleanup

Database Account:

- `sManager` account with full privileges for application access

4.2 System Architecture

4.2.1 Architectural Pattern

The application implements a **three-tier web architecture** using Next.js 16 App Router, providing clear separation of concerns across presentation, business logic, and data access layers.

Tier 1: Presentation Layer (Client-Side)

- React 19 Server and Client Components for UI rendering
- Role-specific dashboard interfaces (`/dashboard/student`, `/dashboard/instructor`, `/dashboard/admin`)
- Reusable form components with React Hook Form and Zod validation
- TailwindCSS for responsive styling

Tier 2: Business Logic Layer (Server-Side)

- Next.js API Route Handlers (`/app/api/*`) implementing RESTful endpoints
- Authentication middleware using JWT (Jose library) for session management
- Role-based access control (RBAC) enforced at both middleware and API levels
- Request validation using Zod schemas
- Centralized error handling and response formatting

Tier 3: Data Access Layer

- MySQL 8.0+ database with stored procedures, triggers, and functions
- `mysql2` Promise-based driver for async database operations
- Connection pooling and parameterized queries for security
- Database access abstracted through utility functions in `lib/db.ts`

Architecture Benefits:

- **Separation of Concerns:** Clear boundaries between UI, business logic, and data
- **Security:** Middleware-enforced authentication, parameterized queries prevent SQL injection
- **Scalability:** Stateless API design with JWT-based sessions
- **Maintainability:** Modular structure with shared utilities and type definitions
- **Type Safety:** End-to-end TypeScript coverage from frontend to API routes

4.2.2 Request Flow Architecture

The typical request lifecycle follows this pattern:

1. **Client Request** → User interacts with UI component
2. **Middleware Layer** (`middleware.ts`) → Validates JWT token, enforces role-based routing
3. **API Route Handler** → Authenticates request, validates input, executes business logic
4. **Database Layer** → Executes queries, triggers, or stored procedures
5. **Response** → Formatted JSON response returned to client
6. **UI Update** → React component re-renders with new data

Authentication Flow:

- Login credentials submitted to `/api/login`
- Server validates against `Users` table, retrieves role-specific data
- JWT token created with user session data

- Token stored in HTTP-only cookie
- Middleware verifies token on subsequent requests
- Role-based dashboards rendered based on session data

4.2.3 Project Structure

The codebase follows Next.js App Router conventions with organized separation of frontend and backend concerns:

```
co2013/
├── app/                          # Next.js App Router directory
│   ├── api/                      # Backend API routes
│   │   ├── admin/                # Administrator-specific endpoints
│   │   │   └── users/            # User management CRUD
│   │   │       ├── [id]/route.ts     # GET/PUT/DELETE user by ID
│   │   │       ├── route.ts          # GET all users, POST new user
│   │   │       ├── roles/route.ts    # GET available roles
│   │   │       └── statuses/route.ts # GET available statuses
│   │   ├── courses/              # Course management endpoints
│   │   │   ├── [id]/route.ts         # GET/PUT/DELETE course by ID
│   │   │   ├── route.ts              # GET/POST courses
│   │   │   ├── list/route.ts         # GET filtered course list
│   │   │   ├── departments/route.ts  # GET unique departments
│   │   │   ├── academic-levels/route.ts
│   │   │   └── statuses/route.ts
│   │   ├── student/              # Student-specific endpoints
│   │   │   ├── courses/          # Enrolled courses
│   │   │   │   └── [courseId]/   # Course-specific operations
│   │   │   │       ├── assignments/
│   │   │   │       │   └── [assignmentId]/
│   │   │   │       │       ├── route.ts      # GET assignment details
│   │   │   │       │       └── submit/route.ts # POST submission
│   │   │   │       └── quizzes/
│   │   │   │           └── [quizId]/
│   │   │   │               ├── start/route.ts    # POST start attempt
│   │   │   │               ├── submit/route.ts   # POST submit quiz
│   │   │   │               ├── results/route.ts  # GET quiz results
│   │   │   │               └── questions/[questionId]/answer/route.ts
│   │   │   ├── available-courses/route.ts
│   │   │   └── register/route.ts # POST course registration
│   │   ├── instructor/           # Instructor-specific endpoints
│   │   │   └── courses/
│   │   │       ├── route.ts          # GET instructor's courses
│   │   │       └── [courseId]/roster/route.ts
│   │   ├── auth/                 # Authentication endpoints
│   │   │   └── me/route.ts           # GET current user session
│   │   ├── login/route.ts        # POST login
│   │   ├── logout/route.ts       # POST logout
│   │   ├── stats/route.ts        # GET system statistics
│   │   └── files/                # File handling
```

```
│   │       └── submissions/[submissionId]/route.ts
│   ├── dashboard/                # Frontend dashboard pages
│   │   ├── admin/                # Administrator dashboard
│   │   │   ├── courses/page.tsx      # Course management UI
│   │   │   ├── users/page.tsx        # User management UI
│   │   │   └── page.tsx              # Admin home
│   │   ├── instructor/           # Instructor dashboard
│   │   │   └── page.tsx
│   │   ├── student/              # Student dashboard
│   │   │   ├── courses/
│   │   │   │   └── [id]/page.tsx     # Course detail view
│   │   │   └── page.tsx
│   │   └── layout.tsx            # Shared dashboard layout
│   ├── page.tsx                  # Landing/login page
│   ├── layout.tsx                # Root layout
│   └── globals.css               # Global styles
├── components/                   # Reusable React components
│   └── admin/
│       ├── CourseForm/           # Course creation/editing form
│       │   ├── index.tsx
│       │   ├── CourseInfoFields.tsx
│       │   ├── PrerequisitesSection.tsx
│       │   └── validation.ts     # Zod schema
│       └── UserForm/             # User management form
│           ├── index.tsx
│           ├── BasicInfoFields.tsx
│           └── validation.ts
├── lib/                          # Shared utilities and configurations
│   ├── auth.ts                   # JWT auth functions
│   ├── db.ts                     # MySQL connection and query helper
│   ├── definitions.ts            # TypeScript type definitions
│   ├── utils.ts                  # Utility functions
│   └── validations/              # Zod validation schemas
│       ├── course.ts
│       └── user.ts
├── types/                        # Additional TypeScript types
│   ├── course.ts
│   └── user.ts
└── middleware.ts                 # Next.js middleware (auth + RBAC)
```

Key Architectural Components:

- **`middleware.ts`:** Intercepts all requests to enforce authentication and role-based access control before reaching pages or API routes
- **`lib/auth.ts`:** Centralized authentication logic with JWT creation, verification, and role-specific guards (`requireStudent()`, `requireInstructor()`, `requireAdministrator()`)
- **`lib/db.ts`:** Database connection abstraction with Promise-based query execution and automatic connection cleanup

- `lib/validations/`: Reusable Zod schemas for request validation shared between frontend forms and backend API routes
- **API Route Organization:** Endpoints grouped by user role (`/admin`, `/student`, `/instructor`) with nested resource routes following RESTful conventions
- **Component Structure:** Form components split into field groups with separate validation logic for maintainability

4.3 CRUD Operations for Course Object

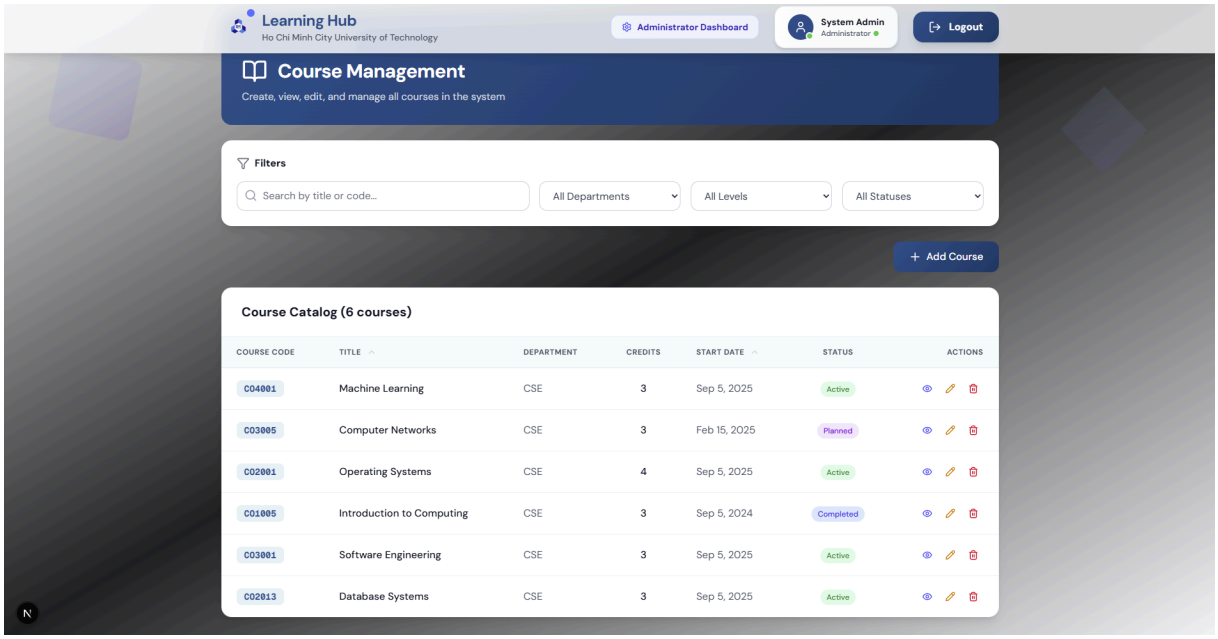

Figure 1: Course Management Interface for Admin

This section demonstrates the full implementation of Create, Read, Update, and Delete (CRUD) operations for the Course entity. The implementation follows a three-tier architecture with robust validation, error handling, and relational data management.

4.3.1 Logic Flow Overview

All Course CRUD operations follow a consistent request flow:

1. **Request Initiation:** Client (Admin Dashboard UI) sends HTTP request to API endpoint
2. **Authentication Layer:** `requireAdministrator()` middleware validates JWT token and verifies admin privileges
3. **Input Validation:** Request data is validated against Zod schemas (`courseCreateSchema` or `courseUpdateSchema`)
4. **Business Logic:** Additional validations (uniqueness checks, date constraints) are performed
5. **Database Transaction:** SQL queries are executed via parameterized statements using `mysql2`
6. **Related Data Handling:** Prerequisites relationships are managed (inserted/updated/deleted as needed)
7. **Response Formation:** Course data with related entities is fetched and returned as JSON

4.3.2 CREATE Operation

Endpoint: `POST /api/courses`

Logic Flow:

1. Authenticate administrator using JWT token verification
2. Parse and validate incoming JSON payload using `courseCreateSchema`
3. Check if `course_code` already exists in database (uniqueness constraint)
4. Build dynamic INSERT query including only provided fields
5. Execute course insertion and capture generated `course_id`
6. If prerequisites are provided, insert each into `Prerequisites` table with foreign key reference
7. Fetch the newly created course with its prerequisites for response
8. Return success response with created course data

4.3.3 READ Operation

Endpoints:

- `GET /api/courses` — List all courses with filters
- `GET /api/courses/[id]` — Get single course details

Logic Flow (Single Course):

1. Authenticate user (any authenticated role can view courses)
2. Extract course ID from URL parameters
3. Execute parallel queries to fetch:
    - Course basic data from `Courses` table
    - Associated sections from `Sections` table
    - Materials linked to sections via JOIN
    - Assignments for the course
    - Quizzes for the course
    - Prerequisites with course details via JOIN
4. Aggregate related data into hierarchical structure
5. Return comprehensive course object with all related entities

Logic Flow (List with Filtering):

1. Authenticate administrator
2. Parse query parameters for filtering: `search`, `department`, `academic_level`, `status`
3. Parse sorting parameters: `sort_by`, `sort_order`
4. Build dynamic SQL query with WHERE conditions for active filters
5. Add ORDER BY clause based on sort parameters (default: newest first)
6. Execute query and return results array

4.3.4 UPDATE Operation

Endpoint: `PUT /api/courses/[id]`

Logic Flow:

1. Authenticate administrator
2. Verify course exists by ID
3. Parse and validate update data using `courseUpdateSchema` (all fields optional for partial updates)

4. If `course_code` is being changed, ensure new code is unique
5. Build dynamic UPDATE query including only provided fields
6. Execute course update
7. Handle prerequisites update separately:
   - Delete all existing prerequisite relationships
   - Insert new relationships from provided data
8. Fetch and return updated course with current prerequisites

4.3.5 DELETE Operation

Endpoint: `DELETE /api/courses/[id]`

Logic Flow:

1. Authenticate administrator
2. Extract course ID from URL parameters
3. Execute DELETE query on `Courses` table
4. Database cascades deletion to related tables via `ON DELETE CASCADE`:
   - `Prerequisites` (both as course and as prerequisite)
   - `Teaching` assignments
   - `Enrollments`
   - `Sections` → `Materials`
   - `Assignments` → `Submissions`
   - `Quizzes` → `Questions` → `Question_Choices` → `Quiz_Attempts` → `Quiz_Answers`
5. Return success message

Database Cascade Behavior: The deletion of a course automatically removes all dependent records due to `ON DELETE CASCADE` constraints defined in the database schema:

```
-- Prerequisites cascade deletion
CREATE TABLE Prerequisites (
    course_id INT,
    prerequisite_id INT,
    FOREIGN KEY (course_id) REFERENCES Courses(course_id) ON DELETE
CASCADE,
    FOREIGN KEY (prerequisite_id) REFERENCES Courses(course_id) ON
DELETE CASCADE
);

-- Enrollments cascade deletion
CREATE TABLE Enrollments (
    student_code VARCHAR(20),
    course_id INT,
    FOREIGN KEY (course_id) REFERENCES Courses(course_id) ON DELETE
CASCADE
);

-- Sections cascade (which then cascade to Materials)
CREATE TABLE Sections (
    course_id INT NOT NULL,
```

```
    FOREIGN KEY (course_id) REFERENCES Courses(course_id) ON DELETE
CASCADE
);
```

4.3.6 Security and Error Handling

Authentication:

- All write operations (CREATE, UPDATE, DELETE) require Administrator role
- Read operations require any authenticated user
- JWT token validation performed via `requireAdministrator()` middleware

Input Validation:

- Zod schemas enforce type safety and business rules
- Custom refinements validate date ordering (`end_date > start_date`)
- Server-side validation prevents injection attacks via parameterized queries

Error Responses:

- `400 Bad Request` — Validation errors with detailed field-level messages
- `401 Unauthorized` — Missing or invalid JWT token
- `403 Forbidden` — Insufficient privileges (non-admin attempting write)
- `404 Not Found` — Course ID does not exist
- `409 Conflict` — Duplicate `course_code`
- `500 Internal Server Error` — Database or server errors

4.4 Data Retrieval

This section demonstrates the implementation of comprehensive data retrieval features for the Course object, showcasing list retrieval with advanced filtering, sorting, and full CRUD action integration from a single interface. The implementation is located in the Admin Dashboard at `/dashboard/admin/courses`.

4.4.1 Overview

The course catalog provides a centralized interface for administrators to:

- View a comprehensive list of all courses in the system
- Search courses by title or course code
- Filter courses by department, academic level, and status
- Sort courses by title or start date in ascending/descending order
- Perform CRUD operations: Create new courses, View details, Edit existing courses, and Delete courses
- Navigate between list view, detail view, and edit modes seamlessly

4.4.2 Data Retrieval Flow

The list retrieval process follows this architecture:

1. **Component Mount:** React `useEffect` hooks initialize filter options and fetch initial course list
2. **Dynamic Filter Options:** Department, academic level, and status options are fetched from database to reflect current data

3. **API Request Construction:** URL search parameters are built based on active filters and sort preferences
4. **Backend Processing:** API endpoint processes filters and returns matching courses
5. **State Management:** React state updates trigger UI re-render with filtered results
6. **User Interaction:** Actions (view, edit, delete) trigger API calls and state changes

4.4.3 Key Features Summary

Filtering Capabilities:

- Text search across course code and title (partial match)
- Department filter (dynamically populated from database)
- Academic level filter (Undergraduate, Graduate, Postgraduate, Certificate)
- Status filter (Planned, Active, Completed, Cancelled, Suspended)
- Multiple filters can be applied simultaneously
- "Clear All" button to reset all filters at once

Sorting Capabilities:

- Sort by Title (alphabetical)
- Sort by Start Date (chronological)
- Toggle between ascending and descending order
- Visual indicators show current sort field and direction

CRUD Integration:

- **Create:** "Add Course" button opens inline form
- **Read:** Table displays essential course information; "View" button shows full details
- **Update:** "Edit" button loads course data into editable form
- **Delete:** "Delete" button with confirmation dialog

User Experience Enhancements:

- Loading states for async operations
- Empty state messages guide users when no courses found
- Responsive design adapts to different screen sizes
- Smooth animations for form transitions
- Status badges with color coding for quick visual identification
- Formatted dates for readability

4.5 Call Function/Procedure

This section demonstrates how the application integrates with MySQL stored functions and procedures to execute complex business logic at the database level. We focus on the prerequisite validation feature used in the Student Dashboard's "Available Courses" section.

4.5.1 Overview

The application uses a hybrid approach for prerequisite checking:

- **Database Layer:** `check_prereq_status` function defined in MySQL for potential direct database-level enforcement
- **Application Layer:** TypeScript implementation in the API route for flexible integration with the web application
- **User Interface:** React components display prerequisite status with visual indicators

This demonstrates how database functions can inform application-level logic while maintaining separation of concerns.

4.5.2 Stored Function Definition

The `check_prereq_status` function validates whether a student has met all prerequisites for a target course:

```
CREATE FUNCTION check_prereq_status(
    p_student_code VARCHAR(20),
    p_target_course_id INT
)
RETURNS BOOLEAN
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE v_missing INT;

    -- Count how many prerequisites for this course the student is
MISSING
    SELECT COUNT(*) INTO v_missing
    FROM Prerequisites p
    WHERE p.course_id = p_target_course_id
    AND NOT EXISTS (
        SELECT 1
        FROM Enrollments e
        WHERE e.student_code = p_student_code
        AND e.course_id = p.prerequisite_id
        AND e.completion_status = 'Completed'
        AND e.final_grade ⩾ p.min_grade
    );

    -- If v_missing is 0, all requirements are met
    RETURN (v_missing = 0);
END
```

Function Logic:

1. **Input Parameters:**
   - `p_student_code`: The student's unique identifier (e.g., 'STU001')
   - `p_target_course_id`: The course ID the student wants to enroll in
2. **Return Value:**
   - `TRUE` if all prerequisites are satisfied (completed with sufficient grade)
   - `FALSE` if any prerequisites are missing or incomplete
3. **Validation Logic:**
   - Queries the `Prerequisites` table to find all required courses
   - For each prerequisite, checks if the student has:
     - Completed the prerequisite course (`completion_status = 'Completed'`)
     - Achieved the minimum required grade (`final_grade ⩾ min_grade`)
   - Returns TRUE only if all prerequisites are met

4.5.3 Application-Level Implementation

The student dashboard calls the stored function via the `/api/student/available-courses` endpoint. The implementation combines detailed prerequisite information for UI display with the database function for authoritative validation.

API Endpoint: `GET /api/student/available-courses`

Key Implementation Detail - Calling the Stored Function:

The critical line that calls the database function is:

```
const functionResult = await query({
  query: `SELECT check_prereq_status(?, ?) as prereqs_met`,
  values: [session.studentCode, course.course_id],
});
const dbPrereqsMet = functionResult[0]?.prereqs_met === 1;
```

This demonstrates:

1. **Function Invocation:** Using `SELECT function_name(param1, param2)` pattern
2. **Parameterized Call:** Passing student code and course ID as parameters
3. **Result Handling:** MySQL returns BOOLEAN as 1/0, so we compare with `=== 1`
4. **Aliasing:** Using `as prereqs_met` for readable result access

4.5.4 Frontend Integration

The Student Dashboard (`/dashboard/student/page.tsx`) consumes this API and displays prerequisite status.

4.5.5 Visual User Interface

The Available Courses section displays:

1. **Course Card Header:**
   - Course code, title, credits, department
   - Current enrollment count vs max capacity
2. **Prerequisites Section** (if applicable):
   - List of required prerequisite courses
   - Visual indicators:
     - Green checkmark (✓) for completed prerequisites
     - Red X (✗) for incomplete prerequisites
   - Minimum grade requirement for each prerequisite
3. **Warning Alerts:**
   - Red alert box when prerequisites are not met
   - Orange alert box when course is full
   - Prevents enrollment with disabled button
4. **Enrollment Action:**
   - "Enroll Now" button enabled only when:
     - All prerequisites are completed with sufficient grades
     - Course has available capacity

- Student is not already enrolled

4.6 Implementation Summary

At this stage, the core Application Building phase is complete. The backend successfully connects to the database via the `sManager` account, and the frontend allows for basic JWT authentication.

Demo video link:

https://drive.google.com/file/d1Amv3cIUpgFFzHbDil3Bw35xopsuBPM-b/view?usp=sharing

Github repo link: https://github.com/dungmilo123/251_co2013_asn_2