**CMPS 350 Project Phase 2 – Report**

**Education Platform**

**(10% of the course grade)**

|  |  |
| --- | --- |
| **Group Members** | Fatima Ahmed (202008039)  Sadien Jamal Abu El-Rub (202203149)  Arwa Elaradi (202203637)  Saja Abdelmalik (201808179)  **Emails:**  [fa2008039@student.qu.edu.qa](mailto:fa2008039@student.qu.edu.qa)  [sa2203149@student.qu.edu.qa](mailto:sa2203149@student.qu.edu.qa)  [ae2203637@student.qu.edu.qa](mailto:ae2203637@student.qu.edu.qa)  [sa1808179@student.qu.edu.qa](mailto:sa1808179@student.qu.edu.qa) |
| **GitHub link** | Give a public link to you code (It is not acceptable to send codes by email) |

**Grades :**

**The student fills only the “Implementation Percentage”: Please specify either: *Working (completed x%)*, *Not Working (completed x%)* or *Not done*.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Criteria** | **%** | **Functionality**\* | **Quality of the implementation** | **Grade** |
| Design and implement the Data Model. | 10 |  |  |  |
| Init DB: populate the database with the data from the json files in seed.js | 5 |  |  |  |
| Server actions, APIs and Repository Implementation to read/write data from the database | 25 |  |  |  |
| Statistics use-case with NextJS | 40 |  |  |  |
| **Documentation**  - Data Model diagram.  - UI Design with screenshots and description.  - Database queries.  - Conducted tests and evidence.  - **Contribution** of each team member [-10pts if not done] | 20 |  |  |  |
| **Total** | 100 |  |  |  |
| Copying and/or plagiarism or not being able to explain or answer questions about the implementation. | -100 |  |  |  |

**Important remark: In case of copying and/or plagiarism or not being able to explain or answer questions about the implementation, you lose the whole grade.**

**\* Criteria for grading the functionality:**

- The functionality is working: you get 70% of the assigned grade.

- The functionality is not working: you lose 40% of assigned grade.

- The functionality is not implemented: you get 0.

- The remaining grade in all cases from above **is assigned to the quality of the implementation**,

- The grades are distributed on the various use cases, when the design/implementation is partial, you get only the grades of designed/implemented use cases.

Code quality criteria, include:

- Use of meaningful identifiers for variables and functions (e.g. using JavaScript naming conventions)

- Pages are responsive

- Clean code: simple and concise code, no redundancy

- Clean implementation without unnecessary files/code

- Use of comments where necessary

- Proper code formatting and indentation.

**You lose marks** for code duplication, poor/inefficient coding practices, poor naming of identifiers, unclean/untidy submission, and unnecessary complex/poor user interface design.

**Important Remark**:

**[Grades: 100-85]:** Will be given only to **fully functional application** with **all the quality criteria cited above met** and the project has excellent **design for the various functionalities**. **The report is professional**.

**[Grades: 85-80]:** Will be given only **to functional application** **with most of all the quality criteria cited above met** and the project has good design for the various functionalities. **The report is professional**.

**[Grades: 80-75]:** 80% of the application functionalities are functional. The project respects partially the quality criteria. **The report is professional** but misses some information.

The grades are not negotiable. We expect that only a small portion (around 15%) of the class will be able to meet the criteria for the grades **[100-85]. You should work hard to and demonstrate the merits of your application to earn those grades.+**

# Description of your proposed platform

This platform is a **Student Management System** developed with **Next.js, React, and Prisma**. The system allows the management of:

* Users (Admins, Instructors, Students)
* Courses and Classes
* Student Enrollments
* Academic Statistics

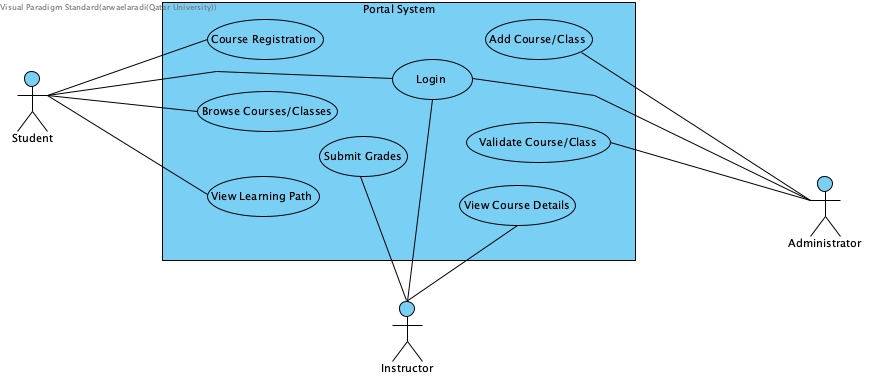
### **Core Features**:

* Admins can add/edit/delete users, courses, and classes.
* Instructors are assigned to specific classes and can view enrolled students.
* Students can view enrolled courses.
* Real-time statistics on top courses, failure rates, and enrollment distribution.

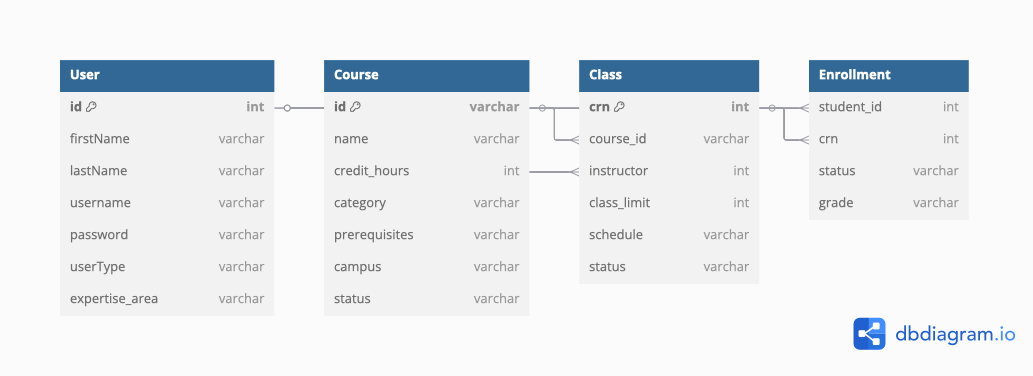
The data is stored in an **SQLite database**, accessed and modified using **Prisma ORM**. The app uses **server actions and REST APIs** to interact with the backend efficiently.

# Data Model

# *Use case diagram*



# *Entities class diagram*



#### **Entities and Relationships:**

* **User**
  + Represents both students and instructors.
  + Related to: Enrollment (as student) and Class (as instructor).
* **Course**
  + Represents a course offered at the institution.
  + Related to: Class.
* **Class**
  + Represents a specific section of a course taught by an instructor.
  + Related to: Course, User (instructor), and Enrollment.
* **Enrollment**
  + Represents the link between a student and a class.
  + Related to: User (student), Class.

### ***Prisma Schema***

// This is your Prisma schema file,

// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {

provider = "prisma-client-js"

output = "./client"

}

datasource db {

provider = "sqlite"

url = env("DATABASE\_URL")

}

model User {

id Int @id @default(autoincrement())

firstName String

lastName String

username String @unique

password String

userType String

expertise\_area String?

assigned\_classes String?

enrollments Enrollment[]

classes Class[] @relation("InstructorClasses")

}

model Course {

id String @id

name String

credit\_hours Int

category String

prerequisites String?

campus String

status String

enrollment Enrollment[]

classes Class[]

}

model Class {

crn String @id

course\_id String

instructor\_id Int

class\_limit Int

schedule String

status String

course Course @relation(fields: [course\_id], references: [id], onUpdate: Cascade, onDelete: Cascade)

instructor User @relation("InstructorClasses", fields: [instructor\_id], references: [id], onUpdate: Cascade, onDelete: Cascade)

enrollments Enrollment[]

}

model Enrollment {

id Int @id @default(autoincrement())

student\_id Int

crn String

status String

grade String

course\_id String

student User @relation(fields: [student\_id], references: [id], onUpdate: Cascade, onDelete: Cascade)

course Course @relation(fields: [course\_id], references: [id], onUpdate: Cascade, onDelete: Cascade)

class Class @relation(fields: [crn], references: [crn], onUpdate: Cascade, onDelete: Cascade)

}

# Web API, Server Actions and repository

### a. **Implemented Repository Functions**

Each repository file contains a standardized set of functions for data management. Here’s what’s implemented:

#### **repo/users.js**

* save(): Persist user data to storage.
* read(id): Read a specific user by ID.
* create(user): Create a new user.
* update(id, newData): Update a user’s data.
* remove(id): Delete a user.

#### **repo/classes.js**

* save(): Save classes to storage.
* read(id): Read a specific class.
* readCRN(crn): Read a class by CRN.
* create(classData): Create a new class.
* update(id, data): Update class information.
* remove(id): Delete a class.

#### **repo/courses.js**

* save(): Save course list to database.
* read(id): Get a course by ID.
* create(course): Create a new course.
* update(id, updatedData): Modify an existing course.
* remove(id): Delete a course.

#### **repo/data/seed.js**

* main(): Initializes and populates the database from JSON files.

### b. **Web API Organization (**/app/api/**)**

The application uses RESTful API route handlers in Next.js to interact with the backend repository layer. These API routes are implemented using the app/api folder structure and call functions from the repo directory to perform CRUD operations.

**Example: app/api/classes/[crn]/route.js**

This file provides GET, PUT, and DELETE operations for a class identified by its CRN (Course Reference Number).

import \* as repo from "@/repo/classes.js";

import { NextResponse } from "next/server";

// GET a class by CRN

export async function GET(reuqest, {params}){

try{

const {crn} = await params;

try{

const cls = await repo.read(crn);

return NextResponse.json(cls, {status: 200});

}

catch(e){

return NextResponse.json({ message: "Not found" }, { status: 404 });

}

}

catch(e){

console.error(e);

return NextResponse.json({ message: "Error" }, { status: 500 });

}

}

// UPDATE a class by CRN

export async function PUT(request, {params}){

try {

const { crn } = await params;

const body = await request.json();

const updated = await repo.update(crn, body);

console.log(updated);

return NextResponse.json(updated, { status: 200 });

} catch (e) {

console.error(e);

return NextResponse.json({ message: "Update failed" }, { status: 500 });

}

}

// DELETE a class by CRN

export async function DELETE(request, {params}){

try {

const { crn } = await params;

await repo.remove(crn);

return new NextResponse(null, { status: 204 });

} catch (e) {

console.error(e);

return NextResponse.json({ message: "Delete failed" }, { status: 500 });

}

}

# Implemented statistics use case

# User Interface

# Implemented queries

1] Number of users per user type (students, instructors, admins)

- Shows how many users exist for each user type in the system.

*// 1. Number of users per user type (students, instructors, admins)*

export async function **getUserTypeCounts**() {

return prisma.user.groupBy({

by: ['userType'],

\_count: { \_all: true },

});

}

2] Number of students per course

- Displays the enrollment count for each course.

*// 2. Number of students per course*

export async function **getStudentsPerCourse**() {

return prisma.enrollment.groupBy({

by: ['course\_id'],

\_count: { \_all: true },

});

}

3] Number of students per course category (e.g., electives, core)

- Groups courses by category and counts enrolled students in each.

*// 3. Number of students per course category*

export async function **getStudentsPerCourseCategory**() {

return prisma.course.findMany({

select: {

category: true,

enrollment: true,

},

});

}

4] Top 3 most enrolled courses

- Lists the three courses with the highest number of enrollments.

*// 4. Top 3 most enrolled courses*

export async function **getTopCourses**() {

return prisma.enrollment.groupBy({

by: ['course\_id'],

\_count: { \_all: true },

orderBy: { \_count: { \_all: 'desc' } },

take: 3,

});

}

5] Courses with the highest failure rate

- Shows which courses have the most failing grades (e.g., grade = 'F').

*// 5. Courses with the highest failure rate (grade = 'F')*

export async function **getCoursesWithHighestFailureRate**() {

return prisma.enrollment.groupBy({

by: ['course\_id'],

\_count: { \_all: true },

where: { grade: 'F' },

orderBy: { \_count: { \_all: 'desc' } },

take: 5,

});

}

6] Average grade per course

- Calculates the average grade for each course.

*// 6. Average grade per course*

export async function **getAverageGradePerCourse**() {

return prisma.enrollment.groupBy({

by: ['course\_id'],

\_avg: { grade: true }, *// Only works if grade is numeric*

});

}

7] Number of students per instructor

- Counts how many students are assigned to each instructor (via their classes).

*// 7. Number of students per instructor*

export async function **getStudentsPerInstructor**() {

return prisma.class.findMany({

select: {

instructor\_id: true,

enrollments: true,

},

});

}

8] Number of students failing more than 2 courses

- Identifies students who have failed more than two courses.

*// 8. Number of students failing more than 2 courses*

export async function **getStudentsFailingMoreThanTwoCourses**() {

const students = await prisma.enrollment.findMany({

where: { grade: 'F' },

select: { student\_id: true },

});

const failCounts = {};

students.forEach(*e* => {

failCounts[*e*.student\_id] = (failCounts[*e*.student\_id] || 0) + 1;

});

return Object.entries(failCounts).filter(([*\_*, *count*]) => *count* > 2);

}

9] Number of courses per category

- Shows how many courses exist in each category (e.g., Core. Elective)

*// 9. Number of courses per category*

export async function **getCoursesPerCategory**() {

return prisma.course.groupBy({

by: ['category'],

\_count: { \_all: true },

});

}

10] Most improved student (based on progress)

- Finds the student with the greatest improvement in grades over time (if grade history is available; otherwise, can be based on before/after averages).

*// 10. Most improved student (placeholder)*

export async function **getMostImprovedStudent**() {

return prisma.user.findMany({

where: { userType: 'student' },

});

}

# Data used in the statics

The statistics are calculated using data from the main entities in the student management system database, which is managed by Prisma and seeded with realistic values. The data includes:

* Users:
  + 500+ students
  + 10 instructors
  + 2 admins

Each user has fields such as id, firstName, lastName, username, password, userType, etc.

* Courses:
  + 50+ courses

Each course has fields such as id, name, credit\_hours, category (e.g., Core, Elective), prerequisites, campus, and status.

* Classes:
  + 100 classes (each course has 2 classes/sections)
  + Each class is linked to a course and an instructor.
* Enrollments:
  + Thousands of enrollments, linking students to classes and courses
  + Each enrollment has a student\_id, crn (class reference), course\_id, status (e.g., enrolled, completed), and grade (A, B, C, D, F).

How the data is generated:

* The database is seeded using the script prisma/seed.mjs, which creates realistic and diverse data for all entities.This ensures that all statistics are meaningful and reflect a real-world scenario.

How the data is used:

* Each statistic is computed by querying the relevant tables (User, Course, Class, Enrollment) using Prisma.
* For example, “Number of students per course” is calculated by counting enrollments for each course, and “Courses with the highest failure rate” is based on enrollments with a grade of 'F'.

# Conducted tests

# Implemented queries

# Discussion of the project contribution of each team member

|  |  |
| --- | --- |
| **Student name** | **Student contributions** |
| Fatima Ahmed (202008039) |  |
| Sadien Jamal Abu El-Rub (202203149) |  |
| Arwa Elaradi (202203637) |  |
| Saja Abdelmalik (201808179) | Designed the Entities class diagram, implemented it using Prisma with a SQLite database. |