**CMPS 350 Project Phase 2 – Report**

**Education Platform**

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

**The report must be submitted in Word format only**

|  |  |
| --- | --- |
| **Group Members** | Mohammad Hassan (201807929)  Ahmed Mohammed Abu Arrah (202108841)  Omar amdadullah (202007896)  Mostafa Youssef - 202106932  **Emails:**  [**mh1807929@qu.edu.qa**](mailto:mh1807929@qu.edu.qa)  [**aa2108841@student.qu.edu.qa**](mailto:aa2108841@student.qu.edu.qa)  [**oa2007896@student.qu.edu.qa**](mailto:oa2007896@student.qu.edu.qa)  [**my2106932@student.qu.edu.qa**](mailto:my2106932@student.qu.edu.qa) |
| **GitHub link** | <https://github.com/mh1807929/Spring2025-Web-Development-Project.git> |

**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 |  | Working/Done |  |
| Init DB: populate the database with the data from the json files in seed.js | 5 |  | Working/Done |  |
| Server actions, APIs and Repository Implementation to read/write data from the database | 25 |  | Working/Done |  |
| Statistics use-case with NextJS | 40 |  | Working/Done |  |
| **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 |  | Working/Done |  |
| **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

The website for students, teachers, and administrative staff of Qatar University's Department of Computer Science and Engineering is called the QU Student Management System. The QU uses this website to manage all of its grade-related communications with instructors and students. Students can discover classes online and register to monitor their grades in a convenient way. Teachers use the website to submit grades, manage courses, and report on them. In the event that not enough students have signed up, the administrator can create, validate, and cancel courses. The webpage uses Prisma as the database to store several courses and students.

# Data Model

**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"

}

datasource db {

provider = "sqlite"

url = env("DATABASE\_URL")

}

model User {

id String @id @default(uuid())

username String @unique

password String

name String

role String // "student", "instructor", "admin"

// Student specific fields

studentId String? @unique

registeredClasses Class[] @relation("StudentClasses")

completedCourses Completion[]

// Instructor specific fields

expertise String?

assignedClasses Class[] @relation("InstructorClasses")

interests CourseInterest[]

}

model Course {

id String @id @default(uuid())

code String @unique

name String

category String

description String

prerequisites String

status String

classes Class[]

interests CourseInterest[]

completions Completion[]

createdAt DateTime @default(now())

updatedAt DateTime @updatedAt

}

model Class {

id String @id @default(uuid())

classId String @unique // e.g., "CMPS350-01"

schedule String

capacity Int

courseId String

course Course @relation(fields: [courseId], references: [id], onDelete: Cascade)

instructorId String?

instructor User? @relation("InstructorClasses", fields: [instructorId], references: [id])

registeredStudents User[] @relation("StudentClasses")

createdAt DateTime @default(now())

updatedAt DateTime @updatedAt

}

model Completion {

id String @id @default(uuid())

grade String

userId String

user User @relation(fields: [userId], references: [id], onDelete: Cascade)

courseId String

course Course @relation(fields: [courseId], references: [id], onDelete: Cascade)

completedAt DateTime @default(now())

@@unique([userId, courseId])

}

model CourseInterest {

id String @id @default(uuid())

userId String

user User @relation(fields: [userId], references: [id], onDelete: Cascade)

courseId String

course Course @relation(fields: [courseId], references: [id], onDelete: Cascade)

expressedAt DateTime @default(now())

@@unique([userId, courseId])

}

**ER Diagram**

A diagram of a course

AI-generated content may be incorrect.

# Web API, Server Actions and repository

**Repository Methods**

**User-related Methods**

* getUserByUsername(username): Retrieve a user with their completed courses
* getUserById(id): Retrieve a user with their completed courses
* getStudentById(studentId): Retrieve a student with completed and registered courses
* getAllStudents(): Retrieve all students with their completed and registered courses
* getAllInstructors(): Retrieve all instructors with their assigned classes and interests

**Course-related Methods**

* getAllCourses(): Retrieve all courses with their classes
* getCourseByCode(code): Retrieve a specific course by its code
* getCourseById(id): Retrieve a specific course by its ID
* getCoursesByCategory(category): Retrieve courses by category
* getCoursesByStatus(status): Retrieve courses by status
* createCourse(courseData): Create a new course
* updateCourse(id, courseData): Update an existing course
* deleteCourse(id): Delete a course

**Class-related Methods**

* getClassById(id): Retrieve a class with its course, instructor, and registered students
* getClassByClassId(classId): Retrieve a class by its unique class ID
* registerStudentForClass(classId, userId): Register a student for a class
* unregisterStudentFromClass(classId, userId): Unregister a student from a class
* assignInstructorToClass(classId, instructorId): Assign an instructor to a class

**Course Interest Methods**

* expressInterestInCourse(userId, courseId): Express interest in a course
* removeInterestInCourse(userId, courseId): Remove interest in a course
* getInstructorInterests(instructorId): Get courses an instructor is interested in
* getCourseInterests(courseId): Get instructors interested in a course

**Completion Methods**

* addCompletedCourse(userId, courseId, grade): Add a completed course for a student
* updateCompletedCourseGrade(userId, courseId, grade): Update a course completion grade

**Web APIs**

**Courses API**

* GET /api/courses: Retrieve all courses, Optional filters: category, status
* POST /api/courses: Create a new course
* GET /api/courses/[code]: Retrieve a specific course
* PUT /api/courses/[code]: Update a course
* DELETE /api/courses/[code]: Delete a course

**Classes API**

* GET /api/classes/[id]: Retrieve a specific class
* POST /api/classes/[id]: Register for a class
* DELETE /api/classes/[id]: Unregister from a class

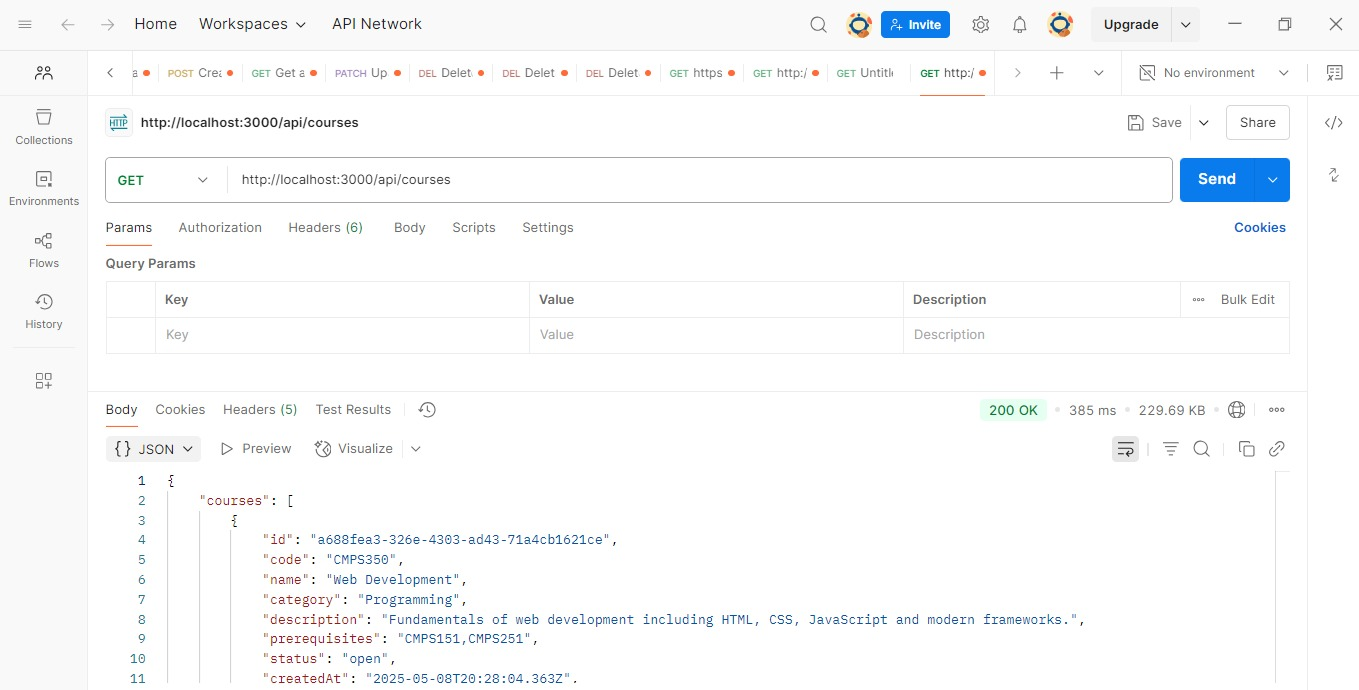
**Server Actions**

**Course-related Actions**

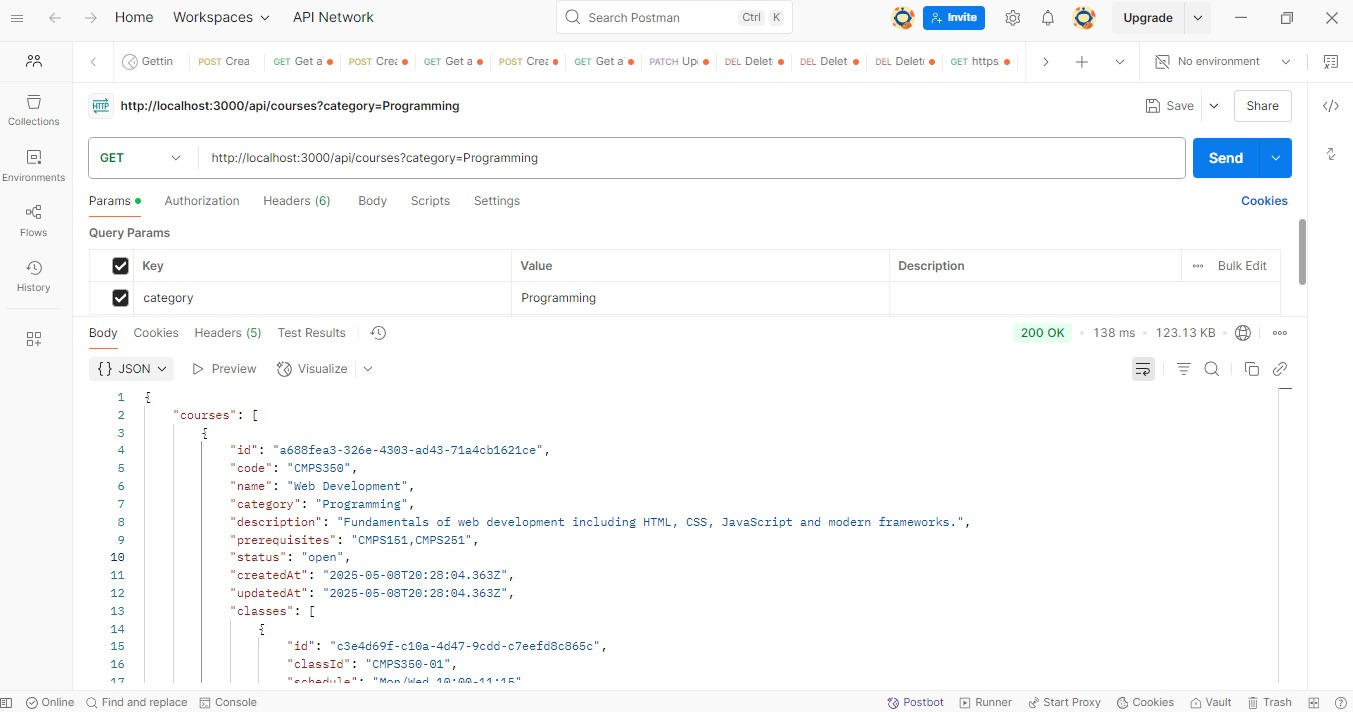
* getAllCourses(): Retrieve all courses
* getCourseByCode(code): Retrieve a course by code
* registerForClass(formData): Register a student for a class
* unregisterFromClass(formData): Unregister a student from a class

expressInterestInCourse(formData): Express interest in a course for instructors

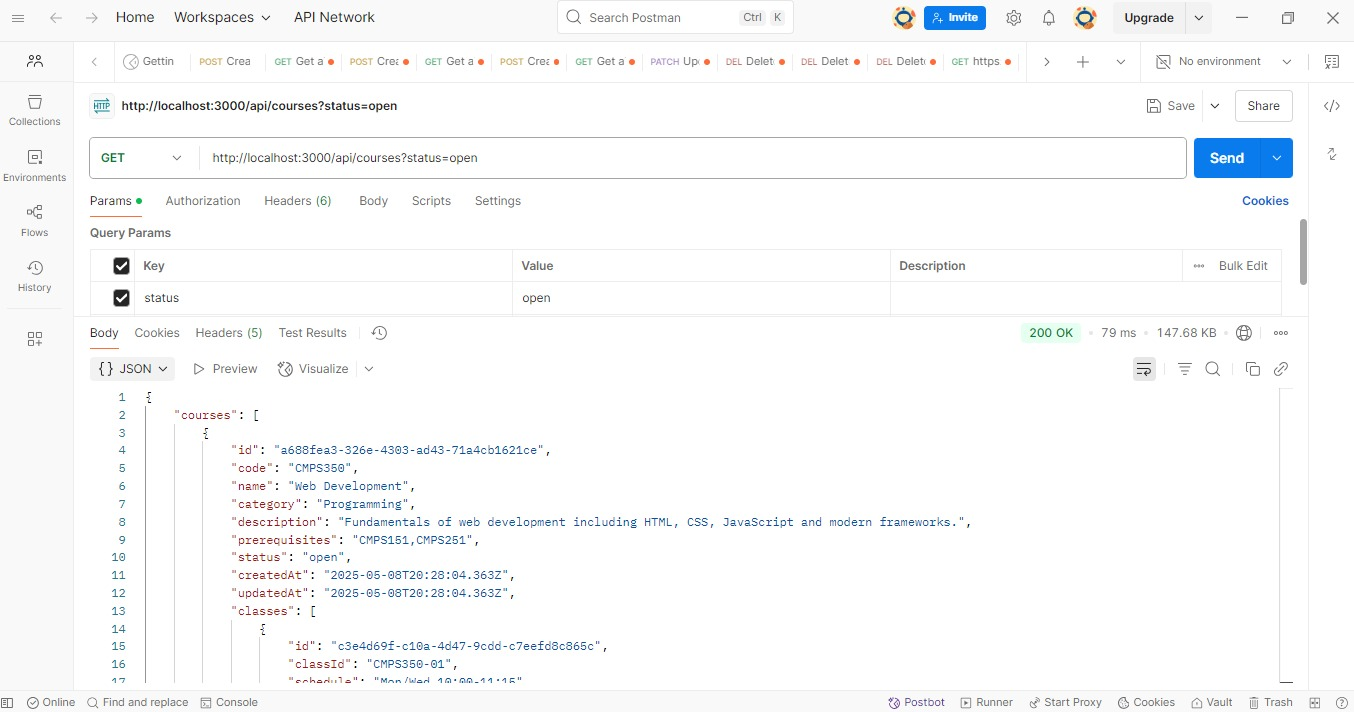
**GET all courses**

****

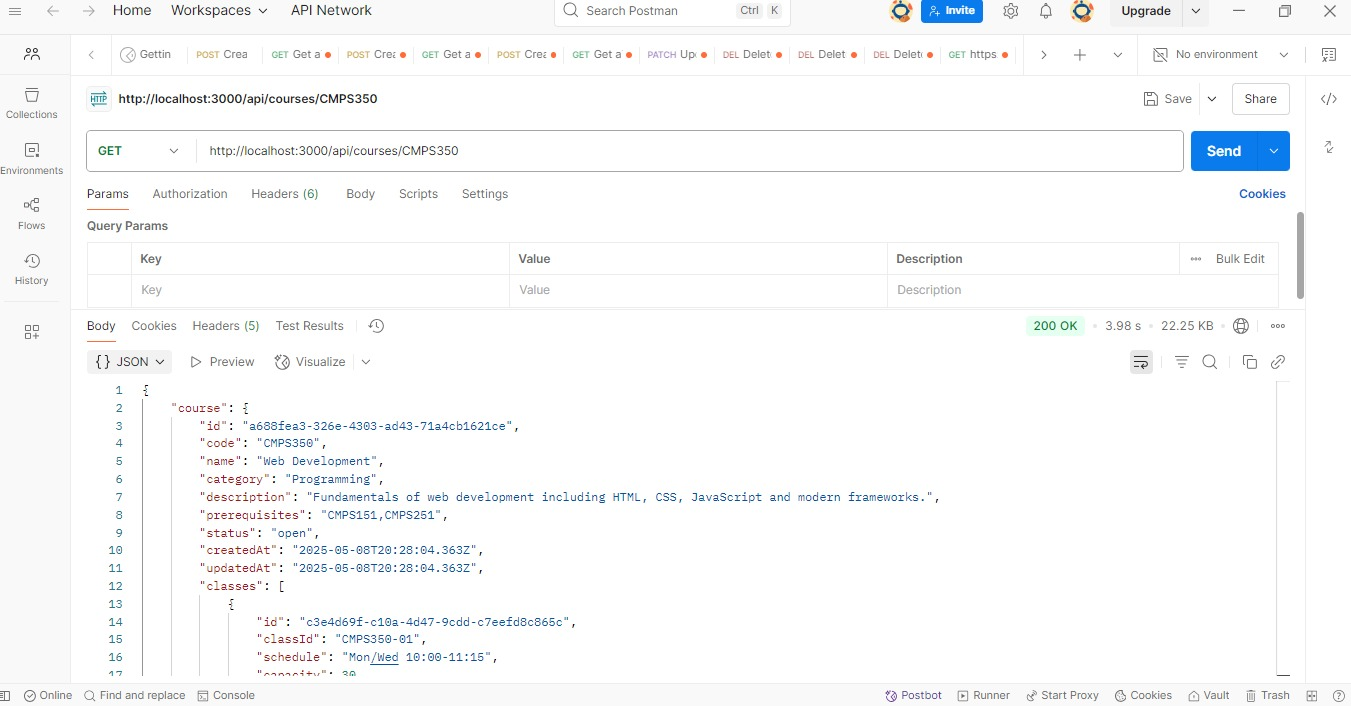
**GET courses by category**

****

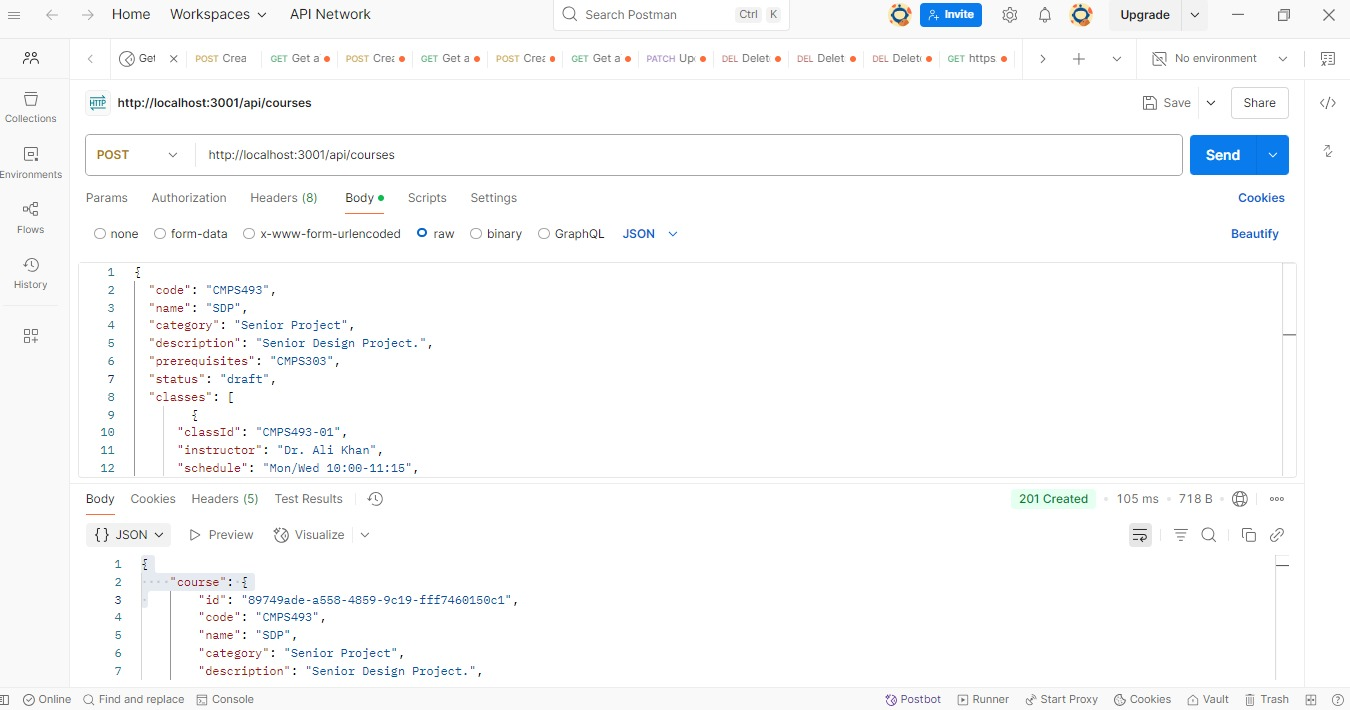
**GET courses by status**



**GET courses by course code**



**Create or POST a course**



# Implemented statistics use case

# User Interface

A screenshot of a dashboard

AI-generated content may be incorrect.

# Implemented queries

// 1. Total number of users by role

export async function getUserCountsByRole() {

  return prisma.user.groupBy({

    by: ['role'],

    \_count: true,

  });

}

// 2. Number of students registered per course

export async function getStudentCountPerCourse() {

  return prisma.course.findMany({

    select: {

      code: true,

      name: true,

      classes: {

        select: {

          registeredStudents: {

            select: { id: true }

          }

        }

      }

    }

  });

}

// 3. Instructor teaching load

export async function getInstructorClassCount() {

  return prisma.user.findMany({

    where: { role: 'instructor' },

    select: {

      username: true,

      name: true,

      \_count: {

        select: { assignedClasses: true }

      }

    }

  });

}

// 4. Course completion rates

export async function getCourseCompletionRates() {

  const courses = await prisma.course.findMany({

    include: {

      completions: true,

      classes: {

        select: {

          registeredStudents: true

        }

      }

    }

  });

  return courses.map(course => {

    const registered = course.classes.reduce((sum, cls) => sum + cls.registeredStudents.length, 0);

    return {

      code: course.code,

      name: course.name,

      completed: course.completions.length,

      registered,

      completionRate: registered ? (course.completions.length / registered) : 0

    };

  });

}

// 5. Most completed courses

export async function getMostCompletedCourses(limit = 5) {

  return prisma.course.findMany({

    orderBy: {

      completions: {

        \_count: 'desc'

      }

    },

    take: limit,

    select: {

      code: true,

      name: true,

      \_count: {

        select: { completions: true }

      }

    }

  });

}

// 6. Total active classes

export async function getTotalActiveClasses() {

  return prisma.class.count();

}

// 7. Courses by status

export async function getCourseCountByStatus() {

  return prisma.course.groupBy({

    by: ['status'],

    \_count: true

  });

}

// 8. Instructor interest count per course

export async function getInstructorInterestPerCourse() {

  return prisma.course.findMany({

    select: {

      code: true,

      name: true,

      \_count: {

        select: { interests: true }

      }

    }

  });

}

// 9. Average number of prerequisites per course

export async function getAveragePrerequisiteCount() {

  const courses = await prisma.course.findMany();

  const total = courses.reduce((sum, course) => {

    const count = course.prerequisites ? course.prerequisites.split(',').length : 0;

    return sum + count;

  }, 0);

  return courses.length ? (total / courses.length).toFixed(2) : 0;

}

// 10. Students who completed all core courses

const coreCourses = ['CMPS151', 'CMPS251', 'CMPS350'];

export async function getStudentsCompletedCoreCourses() {

  const students = await prisma.user.findMany({

    where: { role: 'student' },

    include: {

      completedCourses: {

        include: { course: true }

      }

    }

  });

  return students.filter(student => {

    const completedCodes = student.completedCourses.map(c => c.course.code);

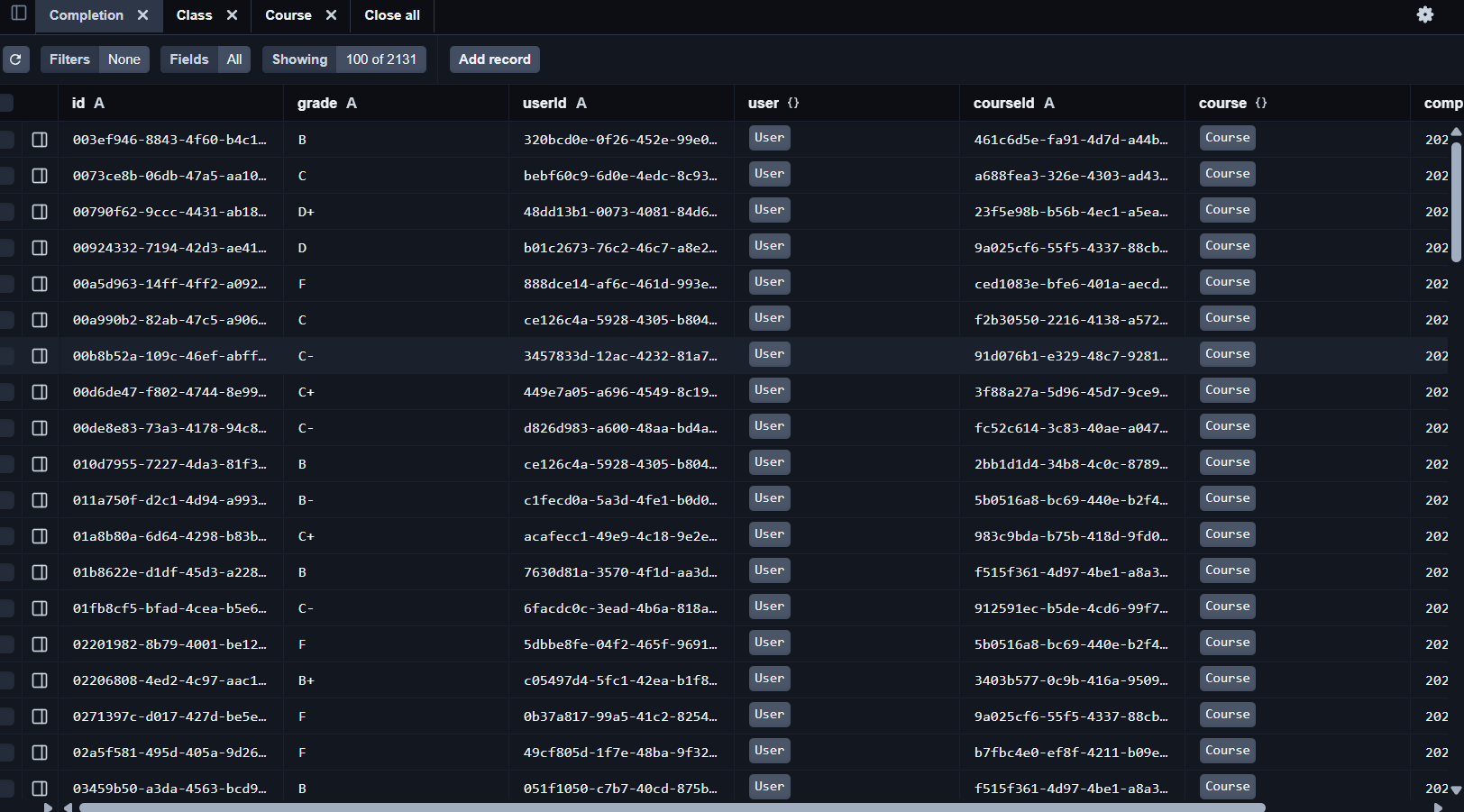
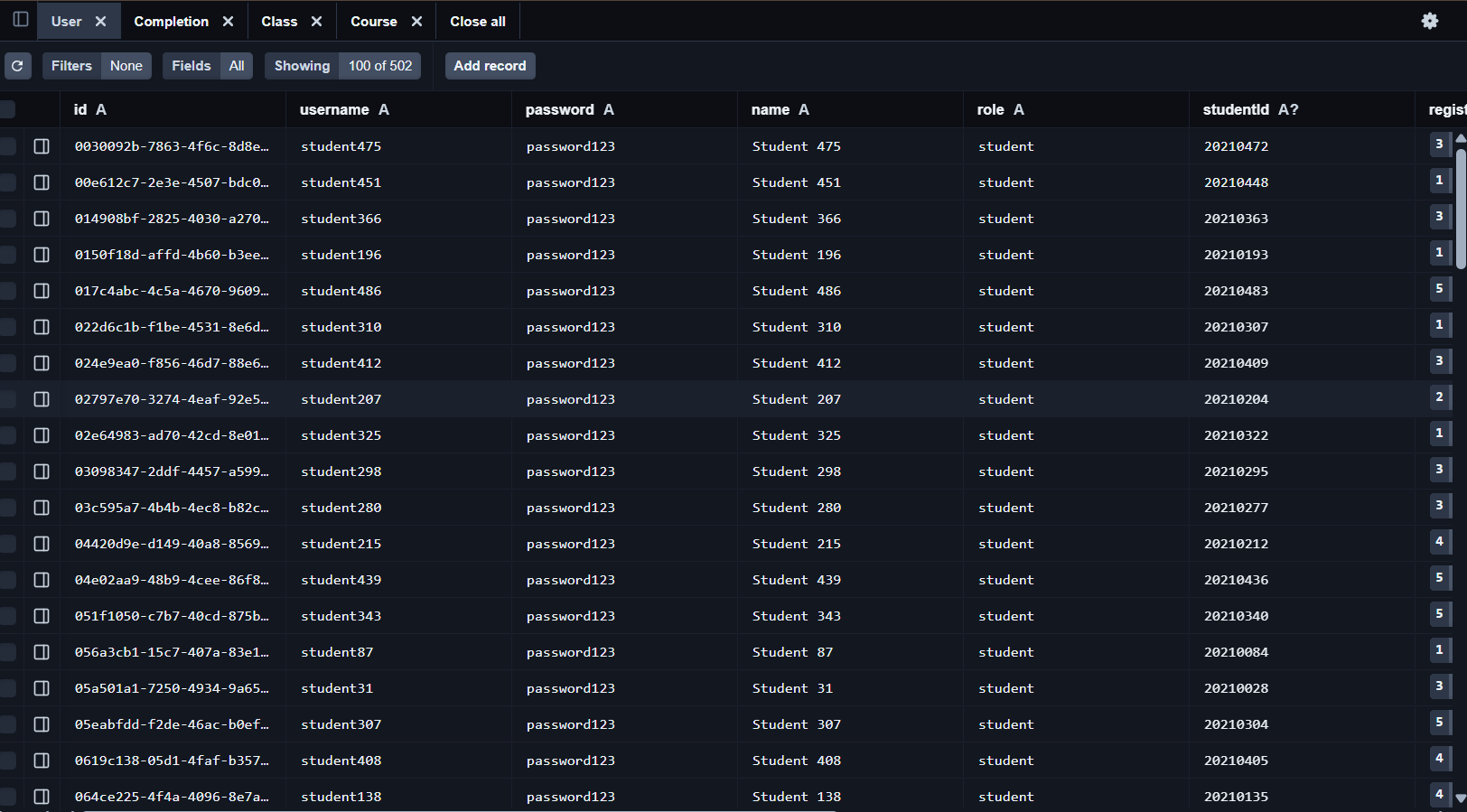
    return coreCourses.every(code => completedCodes.includes(code));

  }).map(s => ({ id: s.id, name: s.name, username: s.username }));

}

# Data used in the statics

The data are the data from the prisma database

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

# Conducted tests

A screenshot of a computer

AI-generated content may be incorrect.

# Authentication

**Local Sign-In with Credentials**

* **A custom sign-in page allows users to authenticate using a username and password.**

**Google OAuth Integration**

* + **Google Sign-In was configured using an OAuth client from the** [**Google Cloud Console**](https://console.cloud.google.com/) **.**
  + **Redirect URI:** [**http://localhost:3000/api/auth/callback/google**](http://localhost:3000/api/auth/callback/google)
  + **Credentials stored securely in .env.local:**

**. Session Management**

* Sessions are managed via NextAuth’s built-in functionality.
* Server-side session checks (**getServerSession**):
  + Unauthenticated users are redirected to **/signin**
  + Authenticated users gain access to dashboard content

A screenshot of a computer

AI-generated content may be incorrect.

# Discussion of the project contribution of each team member

|  |  |
| --- | --- |
| **Student name** | **Student contributions** |
| Mohammad Hassan | * Report * Screenshots * Testing + Postman * seed.js |
| Ahmed Mohammed Abu Arrah | * Report * implemented the Prisma schema and created the seed.js script to populate the database. I also developed the API routes, data repository, and server actions to support full CRUD operations. * Use case |
| Omar amdadullah | * Report * Authentication [Credential , Google] * Testing code * Use case |
| Mostafa Youssef | * Report * Styling [css + tailwind] * Authentication [Credential, Google] * Code Testing |
| Mohammad Hassan | * Report * Screenshots * Testing + Postman * seed.js |