**CMPS 350 Project Phase 2 – Report**

**Education Platform**

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

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

|  |  |
| --- | --- |
| **Group Members** | Hateim Elagha (202104575) (L01)  Ahmad Almashhadani (202203014) (L02)  Obada Alrefai (202110207) (L02)  **Emails:** [ha2104575@student.qu.edu.qa](mailto:ha2104575@student.qu.edu.qa); [aa2203014@student.qu.edu.qa](mailto:aa2203014@student.qu.edu.qa) ; [oa2110207@student.qu.edu.qa](file:///C:\Users\hatem\AppData\Local\Microsoft\Windows\INetCache\IE\TMSW796Z\oa2110207@student.qu.edu.qa); |
| **GitHub link** | https://github.com/Web350-Project/WebProject |

**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

The Qatar University Student Management Application is a secure web-based platform built with HTML/CSS/JavaScript and JSON data storage, featuring **dual authentication (username/password + third-party providers like Google/GitHub)** for all users. It supports **students** (course search/registration, learning path tracking), **instructors** (grade submission, access to **10+ statistics** like *Top 10 GPA Students* and *Average Grade per Instructor*), and **administrators** (course validation, registration management, and analytics such as *Dean’s List* and *Category-wise Student/Instructor Counts*). The system enforces prerequisite checks for course registration, provides dynamic filtering, and ensures real-time grade updates via instructor forms, all presented through a responsive UI. Backed by modular APIs, rigorous testing, and clean code practices, it delivers actionable insights—from *Lowest Grade per Student* to *Best Course Averages per Category*—while maintaining robust security and data integrity.

# Data Model

# A screenshot of a computer AI-generated content may be incorrect.Web API, Server Actions and repository

List all your implemented methods (functions) to query your data,

Show how you organized them in WebAPI and Server actions

Repo :

A computer screen shot of code

AI-generated content may be incorrect.

Get the classes with highest enrollment

A screenshot of a computer program

AI-generated content may be incorrect.

Get the average grade of each class

A screen shot of a computer code

AI-generated content may be incorrect.  
take top 10 students with gpa

A screen shot of a computer program

AI-generated content may be incorrect.  
category with average grade

A screen shot of a computer program

AI-generated content may be incorrect.  
number of instructors per category

   async getInstructorsPerCategory() {

        return await prisma.$queryRaw`

        SELECT

            c.Category,

            COUNT(DISTINCT cl.instructorUsername) as instructorCount,

            GROUP\_CONCAT(DISTINCT i.name) as instructorNames

        FROM courses c

        JOIN classes cl ON c.CNo = cl.CNo

        JOIN instructors i ON cl.instructorUsername = i.username

        GROUP BY c.Category

        ORDER BY instructorCount DESC

    `;

    }

    async getBestCourseInEachCategory() {

        return await prisma.$queryRaw`

            WITH CourseGrades AS (

                SELECT

                    c.Category,

                    c.CName,

                    c.CNo,

                    AVG(

                        CASE ec.Grade

                            WHEN 'A' THEN 4.0

                            WHEN 'B+' THEN 3.5

                            WHEN 'B' THEN 3.0

                            WHEN 'C+' THEN 2.5

                            WHEN 'C' THEN 2.0

                            WHEN 'D+' THEN 1.5

                            WHEN 'D' THEN 1.0

                            WHEN 'F' THEN 0.0

                            ELSE NULL

                        END

                    ) as avgGrade,

                    COUNT(ec.CRN) as enrollmentCount

                FROM courses c

                JOIN classes cl ON c.CNo = cl.CNo

                JOIN enrolledCourses ec ON cl.CRN = ec.CRN

                WHERE ec.Grade IS NOT NULL

                GROUP BY c.Category, c.CNo, c.CName

                HAVING COUNT(ec.CRN) >= 1

            ),

            RankedCourses AS (

                SELECT

                    \*,

                    ROW\_NUMBER() OVER (

                        PARTITION BY Category

                        ORDER BY avgGrade DESC

                    ) as rank

                FROM CourseGrades

            )

            SELECT

                Category,

                CName,

                CNo,

                avgGrade,

                enrollmentCount

            FROM RankedCourses

            WHERE rank = 1

            ORDER BY avgGrade DESC

        `;

    }

Get the highest course average grade in each category

A computer screen with text and images

AI-generated content may be incorrect.

Number of students in each category

A computer screen with text and images

AI-generated content may be incorrect.

Get students and their courses grades instructors

async getLowestGradeForEachStudent() {

    return await prisma.$queryRaw`

        WITH GradeMapping AS (

            SELECT

                s.name AS studentName,

                c.CName AS courseName,

                ec.Grade,

                CASE ec.Grade

                    WHEN 'A' THEN 4.0

                    WHEN 'B+' THEN 3.5

                    WHEN 'B' THEN 3.0

                    WHEN 'C+' THEN 2.5

                    WHEN 'C' THEN 2.0

                    WHEN 'D+' THEN 1.5

                    WHEN 'D' THEN 1.0

                    WHEN 'F' THEN 0.0

                    ELSE NULL

                END AS gradeValue

            FROM students s

            JOIN enrolledCourses ec ON s.username = ec.username

            JOIN classes cl ON ec.CRN = cl.CRN

            JOIN courses c ON cl.CNo = c.CNo

            WHERE ec.Grade IS NOT NULL

        ),

        RankedGrades AS (

            SELECT

                \*,

                ROW\_NUMBER() OVER (PARTITION BY studentName ORDER BY gradeValue ASC) AS rank

            FROM GradeMapping

        )

        SELECT

            studentName,

            courseName AS lowestCourseGrade,

            Grade AS letterGrade

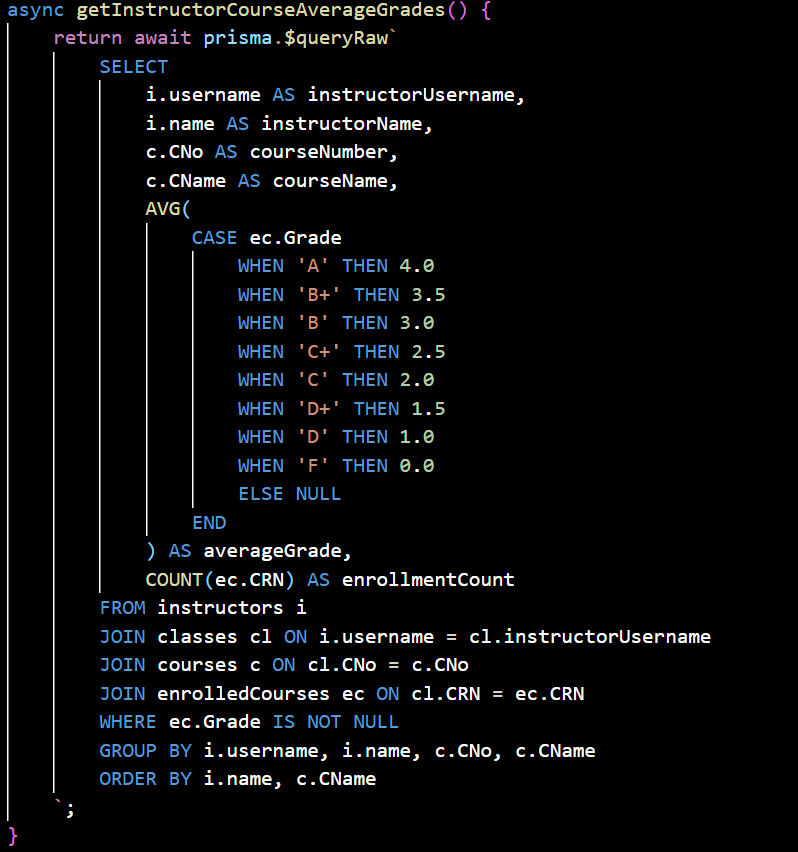
        FROM RankedGrades

        WHERE rank = 1

    `;

}

Lowest course grade for each student



Average grade for instructor in each course

A screen shot of a computer code

AI-generated content may be incorrect.

Dean’s list.

**Servers Actions**

**'use server'**

**import repo from '../repo/repo'**

**export async function fetchTopClassesByEnrollment() {**

**return await repo.getTopClassesByEnrollment()**

**}**

**export async function fetchTopCoursesByGrade() {**

**return await repo.getTopCoursesByGrade()**

**}**

**export async function fetchTopStudentsByGPA() {**

**return await repo.getTopStudentsByGPA()**

**}**

**export async function fetchTopCategoriesByGrade() {**

**return await repo.getTopCategoriesByGrade()**

**}**

**export async function fetchLowestCoursesByGrade() {**

**return await repo.getLowestCoursesByGrade()**

**}**

**export async function fetchHighestFailureRateCourses() {**

**return await repo.getHighestFailureRateCourses()**

**}**

**export async function fetchBestCourseInEachCategory() {**

**return await repo.getBestCourseInEachCategory()**

**}**

**export async function fetchEnrollmentTrends() {**

**return await repo.getEnrollmentTrends()**

**}**

**export async function fetchCreditDistribution() {**

**return await repo.getCreditDistribution()**

**}**

**export async function fetchStudentsInEachCategory() {**

**return await repo.getStudentsInEachCategory()**

**}**

**export async function fetchPrerequisiteImpact() {**

**return await repo.getPrerequisiteImpact()**

**}**

**export async function fetchInstructorsPerCategory() {**

**return await repo.getInstructorsPerCategory();**

**}**

**export async function fetchStudentsWithCoursesAndInstructors() {**

**return await repo.getStudentsWithCoursesAndInstructors();**

**}**

**export async function fetchLowestGradeForEachStudent() {**

**return await repo.getLowestGradeForEachStudent();**

**}**

**export async function fetchgetInstructorCourseAverageGrades() {**

**return await repo.getInstructorCourseAverageGrades();**

**}**

**export async function fetchgetDeansList() {**

**return await repo.getDeansList();**

**}**

**API:**

**nextAuth:**

// app/api/auth/[...nextauth]/route.js

import NextAuth from "next-auth";

import GitHubProvider from "next-auth/providers/github";

import CredentialsProvider from "next-auth/providers/credentials";

import { PrismaClient } from "@prisma/client";

import bcrypt from "bcrypt";

const prisma = new PrismaClient();

const authOptions = {

  providers: [

    CredentialsProvider.default({

      name: "Credentials",

      credentials: {

        username: { label: "Username", type: "text" },

        password: { label: "Password", type: "password" },

      },

      async authorize(credentials) {

        if (!credentials?.username || !credentials?.password) {

          console.log("Missing credentials");

          return null;

        }

        console.log(`Login attempt for username: ${credentials.username}`);

        try {

          const user = await prisma.user.findUnique({

            where: { username: credentials.username },

          });

          if (!user) {

            console.log(`User not found: ${credentials.username}`);

            return null;

          }

          if (user.password === "123" && credentials.password === "123") {

            console.log(`Authentication successful (plaintext) for: ${user.username}`);

            return {

              id: user.username,

              username: user.username,

              type: user.type,

              name: user.username

            };

          }

          try {

            const isValidPassword = await bcrypt.compare(credentials.password, user.password);

            if (isValidPassword) {

              console.log(`Authentication successful (bcrypt) for: ${user.username}`);

              return {

                id: user.username,

                username: user.username,

                type: user.type,

                name: user.username

              };

            }

          } catch (bcryptError) {

            console.log("Not a bcrypt hash, continuing with direct comparison");

          }

          if (credentials.password === user.password) {

            console.log(`Authentication successful (direct comparison) for: ${user.username}`);

            return {

              id: user.username,

              username: user.username,

              type: user.type,

              name: user.username

            };

          }

          console.log("Invalid password");

          return null;

        } catch (error) {

          console.error("Authentication error:", error);

          return null;

        }

      },

    }),

    GitHubProvider.default({

      clientId: process.env.GITHUB\_ID,

      clientSecret: process.env.GITHUB\_SECRET,

      async profile(profile) {

        console.log("GitHub profile:", profile);

        let user = await prisma.user.findUnique({

          where: { username: profile.login }

        }).catch(() => null);

        if (!user) {

          try {

            user = await prisma.user.create({

              data: {

                username: profile.login,

                password: "github-auth-" + Math.random().toString(36).substring(2),

                type: "github",

              }

            });

            console.log("Created new GitHub user:", profile.login);

          } catch (error) {

            console.error("Error creating GitHub user:", error);

            throw error;

          }

        }

        return {

          id: profile.login,

          username: profile.login,

          name: profile.name || profile.login,

          email: profile.email,

          type: 'github',

        };

      },

    }),

  ],

  callbacks: {

    async jwt({ token, user }) {

      if (user) {

        token.username = user.username;

        token.type = user.type;

        token.name = user.name || user.username;

      }

      return token;

    },

    async session({ session, token }) {

      if (token) {

        if (!session.user) {

          session.user = {};

        }

        session.user.username = token.username;

        session.user.type = token.type;

        session.user.name = token.name || token.username;

      }

      return session;

    },

  },

  session: {

    strategy: "jwt",

  },

  pages: {

    signIn: "/auth/signin",

    error: "/auth/signin",

  },

  secret: process.env.NEXTAUTH\_SECRET,

  debug: process.env.NODE\_ENV !== 'production',

};

const handler = NextAuth.default(authOptions);

export { handler as GET, handler as POST };

**login:**

import { PrismaClient } from "@prisma/client";

import bcrypt from "bcrypt";

import { signJwt } from "@/app/lib/jwt";

const prisma = new PrismaClient();

export async function POST(request) {

  try {

    const { username, password } = await request.json();

    if (!username || !password) {

      return Response.json(

        { error: "Missing username or password" },

        { status: 400 }

      );

    }

    const user = await prisma.user.findUnique({

      where: { username },

    });

    if (!user) {

      return Response.json(

        { error: "Invalid credentials" },

        { status: 401 }

      );

    }

    const isValid = await bcrypt.compare(password, user.password);

    if (!isValid) {

      return Response.json(

        { error: "Invalid credentials" },

        { status: 401 }

      );

    }

    const token = signJwt({

      username: user.username,

      type: user.type

    });

    return Response.json(

      {

        user: {

          username: user.username,

          type: user.type

        },

        token

      },

      { status: 200 }

    );

  } catch (error) {

    console.error(error);

    return Response.json(

      { error: "Internal server error" },

      { status: 500 }

    );

  }

}

**Users:**

// app/api/users/route.js

import { PrismaClient } from "@prisma/client";

import bcrypt from "bcrypt";

import { signJwt } from "@/app/lib/jwt";

const prisma = new PrismaClient();

export async function POST(request) {

  try {

    const { email, password, name, role = "user" } = await request.json();

    if (!email || !password || !name) {

      return Response.json(

        { error: "Missing required fields" },

        { status: 400 }

      );

    }

    const existingUser = await prisma.user.findUnique({

      where: { email },

    });

    if (existingUser) {

      return Response.json(

        { error: "User already exists" },

        { status: 409 }

      );

    }

    const hashedPassword = await bcrypt.hash(password, 10);

    const user = await prisma.user.create({

      data: {

        email,

        password: hashedPassword,

        name,

        role,

      },

    });

    const token = signJwt({ id: user.id, email: user.email, name: user.name, role: user.role });

    return Response.json(

      { user: { id: user.id, email: user.email, name: user.name, role: user.role }, token },

      { status: 201 }

    );

  } catch (error) {

    console.error(error);

    return Response.json(

      { error: "Internal server error" },

      { status: 500 }

    );

  }

}

# Implemented statistics use case

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

# Discussion of the project contribution of each team member

|  |  |
| --- | --- |
| **Student name** | **Student contributions** |
| Hateim Elagha (L01) | 30% |
| Ahmad Almashhadani (L02) | 40% |
| Obada Alrefai (L02) | 30% |
|  |  |
|  |  |