**CMPS 350 Project Phase 2 – Report**

**Education Platform**

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

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

|  |  |
| --- | --- |
| **Group Members** | Alwaleed Sarieh Dsher (201807916)  Ali Adel Abouelkhir (202206551)  Yousef Elsherif (202206988)  **Emails:**  [as1807916@student.qu.edu.qa](mailto:as1807916@student.qu.edu.qa)  [aa2206551@student.qu.edu.qa](mailto:aa2206551@student.qu.edu.qa)  [ye2206988@student.qu.edu.qa](mailto:ye2206988@student.qu.edu.qa) |
| **GitHub link** | <https://github.com/oodyb/qu-app> |

**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(100%) |  |
| Init DB: populate the database with the data from the json files in seed.js | 5 |  | Working(100%) |  |
| Server actions, APIs and Repository Implementation to read/write data from the database | 25 |  | Working(100%) |  |
| Statistics use-case with NextJS | 40 |  | Working(100%) |  |
| **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(100%) |  |
| **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

Our Student Management Application is designed to streamline and manage information related to students, courses, instructors, and classes. The application allows administrators to add, update, and delete records of students and courses, as well as assign instructors to classes. It also supports generating statistics about student enrollments and course performance.

The main users of the system are university staff and administrators who need an efficient way to manage academic data.

The application is built using:

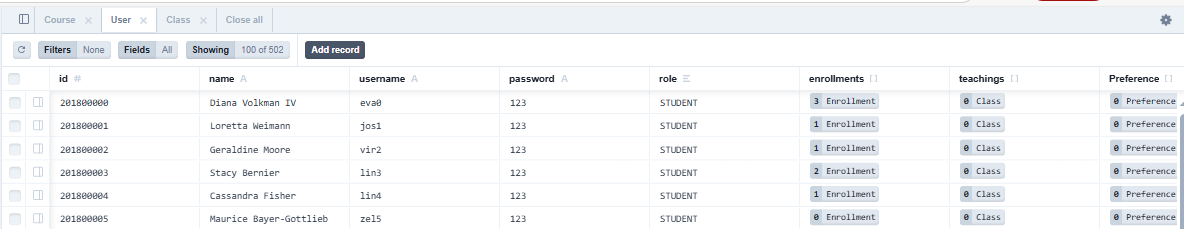
* **Next.js** for the frontend and server actions
* **Prisma** ORM for database operations
* **SQLite** as the relational database

# Data Model

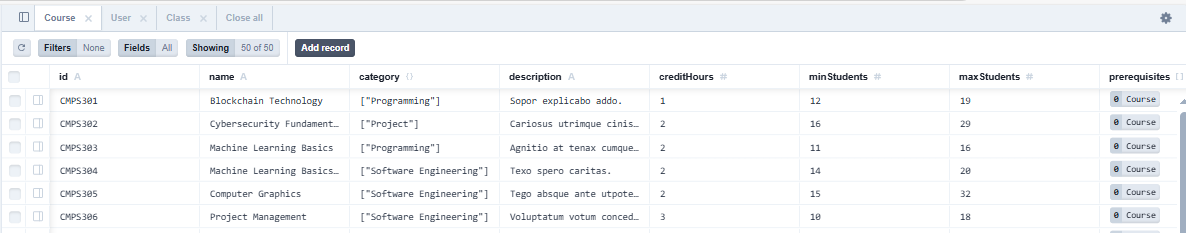
We designed our data model using **Prisma ORM**, which maps directly to our SQLite database. The core entities include User, Course, Class, Enrollment, and Preference. The relationships between these entities are as follows:

* A **User** can be a student, instructor, or admin, and can have multiple **Enrollments** and **Preferences**.
* A **Course** can have many **Classes**, and also define **prerequisite relationships** with other courses.
* A **Class** is a specific offering of a course, taught by an instructor (User), and can have multiple **Enrollments**.
* An **Enrollment** links a student to a class and stores an optional grade.
* A **Preference** represents an instructor’s preference toward a class/course.

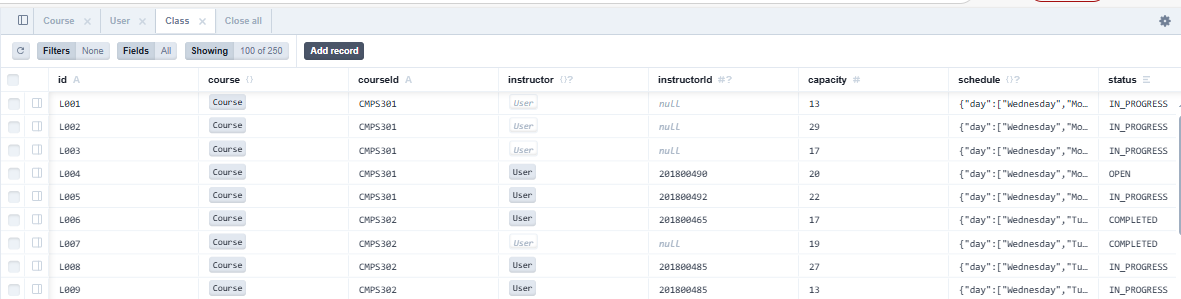
User Table:



Course Table:



Class Table:



# Web API, Server Actions and repository

We created a structured **Repository Layer** using Prisma Client to interact with the database. This layer is used by both **Server Actions** and **Web APIs** to provide clean and efficient data handling.

Below are examples of implemented repository functions:

* **getStudentLearningPath(studentId)**  
  Retrieves a student’s enrolled and completed classes, including instructor names and course categories. This is used in the statistics view to show a student's academic progress.
* **getAllInstructors()**  
  Returns a list of all users with the role INSTRUCTOR, providing their IDs and names for use in forms and assignment pages.

These functions are used in both:

* **Server Actions** – for secure server-side updates and reads
* **Web API Endpoints** – for use by frontend components needing fetchable data

We ensured that **queries are optimized** by filtering and joining data directly in the Prisma query, rather than in application logic.

|  |  |  |
| --- | --- | --- |
| User | instructor | Student |
| findUserByUsername(username) | submitPreference(instructorId, courseId, classId) | getStudentLearningPath (studentId) |
| findUserById(id) | getInstructorPreferences (instructorId) | getAvailableClasses() |
| createUser(userData) | submitGrades(courseId, classId, grades) | registerStudent(studentId, classId, courseId) |
|  | getStudentsInClass(classId, courseId) |  |
|  | getInstructorClasses(instructorId) |  |

|  |  |
| --- | --- |
| Admin | Statistics |
| getAllInstructors() | getGradeDistribution() |
| getAllPreferences() | getCoursePopularity() |
| getAllInProgress() | getClassStatusDistribution() |
| getAllClasses() | getInstructorLoad() |
| rejectPreference(preferenceId) | getClassCapacityVsEnrollment() |
| approvePreference(preferenceId, classId, courseId) | getCourseCategoryDistribution() |
| createClass(data) | getPreferencesPerCourse() |
| updateClassStatus(id, courseId, newStatus) | getStudentsPerInstructor() |
| deleteClassById(id, courseId) | getCoursesWithAndWithoutClasses() |
| createCourse(data) | getClassSizesPerCourse() |

# Implemented statistics use case

# User Interface

The Statistics Dashboard in our Student Management Application presents a comprehensive and interactive overview of various academic and operational metrics. It is implemented using **Next.js and React**, featuring clean visuals and responsive design principles.

### **Overview of the UI Design**

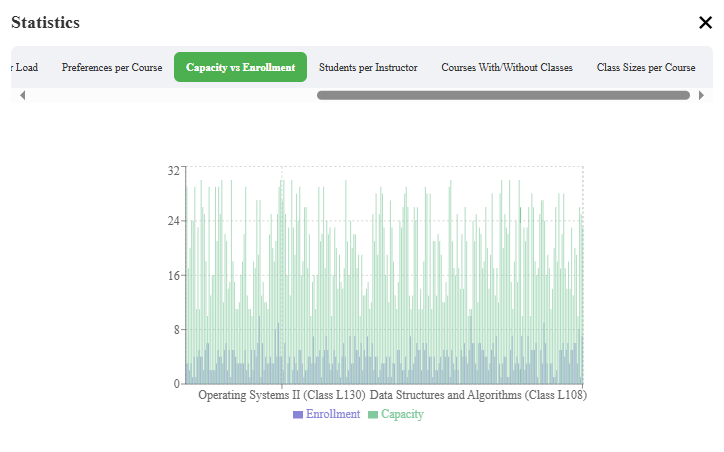
* The dashboard is accessible to admin users via the main navigation panel.
* It displays **ten interactive visualizations**, each tied to a backend query function.
* The layout groups related statistics (e.g., course-level, instructor-level) into distinct sections.

### **Displayed Statistics and Visuals**

|  |  |  |
| --- | --- | --- |
| Query Function | Visual Type | Description |
| getGradeDistribution() | Bar Chart | Shows distribution of grades (A/B/C/F) across all courses. |
| getCoursePopularity() | line chart | Displays the number of students enrolled per course. |
| getClassStatusDistribution() | Bar Chart | Illustrates the proportion of classes in each status. |
| getInstructorLoad() | Bar Chart | Indicates the number of classes assigned to each instructor. |
| getClassCapacityVsEnrollment() | Grouped Bar Chart | Compares capacity vs. actual enrollment for each class. |
| getCourseCategoryDistribution() | Pie Chart | Breaks down course offerings by category. |
| getPreferencesPerCourse() | line chart | Shows how many instructors submitted preferences for each course. |
| getStudentsPerInstructor() | Bar Chart | Displays how many students are currently enrolled under each instructor. |
| getCoursesWithAndWithoutClasses() | Pie Chart | Separates courses with active class instances vs. those without. |
| getClassSizesPerCourse() | line chart | Displays average or actual class sizes per course. |

**Interactivity**: Hover effects, legends, and tooltips provide contextual information for each data point.

### **Screenshots**

* “Grade Distribution” chart
* “Course Popularity” chart
* “Course Popularity” chart

### **Technologies Used**

* **Frontend**: Next.js, React
* **Charts**: recharts
* **Styling**: CSS

# Implemented queries

This section describes the database queries used to retrieve data for the Statistics Dashboard. Each query was implemented using **Prisma Client** and optimized to run directly on the database layer to avoid unnecessary data transfer.

Below is a description of each query:

### **1.** getGradeDistribution()

**Query Summary**:

* Groups enrollment records by grade
* Counts how many students received each grade

### **2.** getCoursePopularity()

**Query Summary**:

* Joins Enrollment, Class, and Course
* Counts enrollments grouped by course name

### **3.** getClassStatusDistribution()

**Query Summary**:

* Groups all class records by their status enum field
* Counts the number of classes in each category

### **4.** getInstructorLoad()

**Query Summary**:

* Joins Class and User
* Counts how many classes are assigned to each instructor

### **5.** getClassCapacityVsEnrollment()

**Query Summary**:

* Joins Class with Enrollment
* Groups by class and aggregates both capacity and actual enrollment

### **6.** getCourseCategoryDistribution()

**Query Summary**:

* Groups Course records by category (stored as JSON)
* Counts how many courses fall into each category

### **7.** getPreferencesPerCourse()

**Query Summary**:

* Joins Preference and Course
* Groups by course ID and counts total preferences

### **8.** getStudentsPerInstructor()

**Query Summary**:

* Traverses User → Class → Enrollment
* Groups enrollments by instructor and counts unique students

### **9.** getCoursesWithAndWithoutClasses()

**Query Summary**:

* Checks Course records for associated Class entries
* Separates into two lists: with classes and without

### **10.** getClassSizesPerCourse()

**Query Summary**:

* Joins Course → Class → Enrollment
* Groups by course and averages class size

# Data used in the statics

To ensure meaningful and realistic statistical insights, we populated our SQLite database with a **large and diverse dataset** during the seeding phase. The data was injected using a custom seed.js script, which transforms JSON data into database records using Prisma Client.

### **Summary of Seeded Data**

|  |  |  |
| --- | --- | --- |
| Entity | Count | Description |
| Students | 500+ | Each student has a unique profile and is enrolled in multiple classes. |
| Instructors | 50 | Instructors are assigned to classes and submit teaching preferences. |
| Courses | 50 | Courses span various categories and include credit hours, prerequisites, and capacity limits. |
| Classes | 250 | Each class is linked to a course and an instructor, with capacity and status fields. |
| Enrollments | 1000+ | Students are enrolled in different classes with randomly assigned grades. |
| Preferences | 1 | Instructors have submitted preferences for courses and/or specific class sections. |

### **Data Sources**

* All data originates from structured **JSON files** prepared during Phase 1.
* Categories were assigned to courses as structured **JSON fields** (e.g., "category": { "type": "Programming" }).
* Grades were randomly distributed to simulate real academic performance across multiple classes.
* Relationships were carefully seeded to reflect realistic academic patterns:
  + Students enrolled in both introductory and advanced courses
  + Some courses having no classes
  + Varying instructor loads and class capacities

### **Ensuring Meaningful Statistics**

The size and diversity of the dataset make the statistical outputs significant. For example:

* Grade distributions span A to F with a bell-shaped curve.
* Course popularity highlights top-performing or required courses.
* Instructor load differences are observable across departments.

# Conducted tests

To validate the correctness and performance of the statistics use-case, we conducted both **functional** and **data-driven tests**. These tests focused on confirming the accuracy of the queries, integrity of relationships, and correct rendering of statistics in the UI.

### ✅ **1. Functional Testing of Queries**

|  |  |  |
| --- | --- | --- |
| Query | Test Case Description | Result |
| getGradeDistribution() | Confirm correct count for each grade from enrollment data | Passed |
| getCoursePopularity() | Validate student counts match class enrollments | Passed |
| getInstructorLoad() | Check number of classes per instructor | Passed |
| getClassStatusDistribution() | Ensure enum counts match expected OPEN/INPROGRESS | Passed |

### ✅ **2. UI Rendering Verification**

We verified the rendering of charts using:

* Manual browser testing (Chrome, Edge)
* Responsive layout checks
* Real-time interaction testing (e.g., tooltips, hover)

**Test Checklist**:

* Charts load with real data
* Filters update the data dynamically
* Layout works on desktop and mobile
* No frontend or backend errors in console/network tab

### ✅ **3. Edge Case Testing**

We tested edge conditions such as:

* Courses with no enrollments or no classes
* Instructors with 0 preferences
* Grades with no occurrences (e.g., no “F”)

All queries returned safe defaults or empty arrays without causing crashes.

# . Implemented queries

**Admin.js:**

1. import prisma from "@/repos/prisma";
2. export async function getAllInstructors() {
3. return await prisma.user.findMany({
4. where: { role: "INSTRUCTOR" },
5. select: { id: true, name: true }
6. });
7. }
8. export async function getAllPreferences() {
9. return await prisma.preference.findMany({
10. include: {
11. instructor: true,
12. class: { include: { course: true } }
13. }
14. });
15. }
16. export async function getAllInProgress() {
17. const classes = await prisma.class.findMany({
18. where: { status: "IN\_PROGRESS" },
19. include: {
20. course: true, instructor: true
21. }
22. });
23. return classes.map((cls) => ({
24. id: cls.id,
25. courseId: cls.courseId,
26. courseName: cls.course.name,
27. schedule: cls.schedule,
28. instructorName: cls.instructor ? cls.instructor.name : "Unassigned",
29. status: cls.status,
30. }));
31. }
32. export async function getAllClasses() {
33. const classes = await prisma.class.findMany({
34. include: { course: true, enrollments: true, instructor: true }
35. });
36. return classes.map((cls) => ({
37. classId: cls.id,
38. courseId: cls.courseId,
39. courseName: cls.course.name,
40. category: cls.course.category,
41. status: cls.status,
42. currentStudents: cls.enrollments.length,
43. instructorName: cls.instructor ? cls.instructor.name : 'Unassigned',
44. instructorId: cls.instructorId,
45. capacity: cls.capacity,
46. }));
47. }
48. export async function rejectPreference(preferenceId) {
49. return await prisma.preference.delete({
50. where: { id: preferenceId },
51. });
52. }
53. export async function approvePreference(preferenceId, classId, courseId) {
54. const pref = await prisma.preference.findUnique({ where: { id: preferenceId } });
55. return await prisma.class.update({
56. where: {
57. id\_courseId: { id: classId, courseId: courseId }
58. }, data: { instructorId: pref.instructorId }
59. });
60. }
61. export async function createClass(data) {
62. const { id, courseId, instructorId, status, capacity, schedule } = data;
63. return await prisma.class.create({
64. data: {
65. id,
66. courseId,
67. instructorId: isNaN(instructorId) ? null : instructorId,
68. status,
69. capacity,
70. schedule,
71. }
72. });
73. }
74. export async function updateClassStatus(id, courseId, newStatus) {
75. return await prisma.class.update({
76. where: { id\_courseId: { id, courseId } },
77. data: { status: newStatus },
78. });
79. }
80. export async function deleteClassById(id, courseId) {
81. await prisma.enrollment.deleteMany({
82. where: { classId: id }
83. });
84. return await prisma.class.delete({
85. where: { id\_courseId: { id, courseId } }
86. });
87. }
88. export async function createCourse(data) {
89. const { id, name, category, description, creditHours, minStudents, maxStudents, prerequisites = [] } = data;
90. const flatPrereqs = Array.isArray(prerequisites)
91. ? prerequisites.flat().filter((id) => typeof id === "string" && id.trim() !== "")
92. : [];
93. const validPrereqCourses = await prisma.course.findMany({
94. where: { id: { in: flatPrereqs } },
95. select: { id: true },
96. });
97. const cleanedPrerequisites = validPrereqCourses.map((course) => ({ id: course.id }));
98. return await prisma.course.create({
99. data: {
100. id,
101. name,
102. category,
103. description,
104. creditHours,
105. minStudents,
106. maxStudents,
107. ...(cleanedPrerequisites.length > 0 && {
108. prerequisites: {
109. connect: cleanedPrerequisites,
110. },
111. }),
112. },
113. });
114. }

**Instructor.js:**

import prisma from "@/repos/prisma";

export async function submitPreference(instructorId, courseId, classId) {

  return await prisma.preference.create({

    data: {

      instructorId,

      courseId,

      classId,

    },

  });

}

export async function getInstructorPreferences(instructorId) {

  return await prisma.preference.findMany({

    where: { instructorId },

    select: {

      courseId: true,

      classId: true,

    },

  });

}

export async function submitGrades(courseId, classId, grades) {

  const gradeUpdates = Object.entries(grades).map(([studentId, grade]) => {

    return prisma.enrollment.updateMany({

      where: {

        studentId: parseInt(studentId),

        courseId,

        classId,

      },

      data: {

        grade,

      },

    });

  });

  await Promise.all(gradeUpdates);

  await prisma.class.update({

    where: {

      id\_courseId: {

        id: classId,

        courseId: courseId,

      },

    },

    data: {

      status: "COMPLETED",

    },

  });

  return { message: "Grades submitted and class completed" };

}

export async function getStudentsInClass(classId, courseId) {

  const enrollments = await prisma.enrollment.findMany({

    where: {

      classId,

      courseId,

    },

    include: {

      student: true,

    },

  });

  return enrollments.map(enrollment => ({

    studentId: enrollment.studentId,

    name: enrollment.student.name,

    classId: enrollment.classId,

  }));

}

export async function getInstructorClasses(instructorId) {

  const classes = await prisma.class.findMany({

    where: {

      instructorId,

      status: "IN\_PROGRESS",

    },

    include: {

      course: true,

    },

  });

  const grouped = {};

  classes.forEach(cls => {

    if (!grouped[cls.courseId]) {

      grouped[cls.courseId] = {

        courseId: cls.courseId,

        courseName: cls.course.name,

        classes: [],

      };

    }

    grouped[cls.courseId].classes.push({

      classId: cls.id,

    });

  });

  return Object.values(grouped);

}

**statistics.js:**

import prisma from "@/repos/prisma";

// 1. Grade distribution

export async function getGradeDistribution() {

    return await prisma.enrollment.groupBy({

        by: ['grade'],

        \_count: {

            grade: true,

        },

    });

}

// 2. Eenrollments per course

export async function getCoursePopularity() {

    return await prisma.enrollment.groupBy({

        by: ['courseId'],

        \_count: {

            courseId: true,

        },

    });

}

// 3. Class status distribution

export async function getClassStatusDistribution() {

    return await prisma.class.groupBy({

        by: ['status'],

        \_count: {

            status: true,

        },

    });

}

// 4. Nnumber of classes per instructor

export async function getInstructorLoad() {

    return await prisma.class.groupBy({

        by: ['instructorId'],

        \_count: {

            instructorId: true,

        },

        where: {

            instructorId: {

                not: null,

            },

        },

    });

}

// 5. Class capacity vs current enrollment

export async function getClassCapacityVsEnrollment() {

    const classes = await prisma.class.findMany({

        include: {

            enrollments: true,

        },

    });

    return classes.map(cls => ({

        classId: cls.id,

        courseId: cls.courseId,

        capacity: cls.capacity,

        enrolled: cls.enrollments.length,

    }));

}

// 6. Course categories distribution

export async function getCourseCategoryDistribution() {

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

    const categoryCount = {};

    for (const course of courses) {

        const categories = Array.isArray(course.category) ? course.category : [course.category];

        for (const cat of categories) {

            categoryCount[cat] = (categoryCount[cat] || 0) + 1;

        }

    }

    return Object.entries(categoryCount).map(([name, value]) => ({ name, value }));

}

// 7. Preferences per course

export async function getPreferencesPerCourse() {

    return await prisma.preference.groupBy({

        by: ['courseId'],

        \_count: {

            courseId: true,

        },

    });

}

// 8. Students per instructor

export async function getStudentsPerInstructor() {

    const classes = await prisma.class.findMany({

        include: {

            enrollments: true,

        },

        where: {

            instructorId: {

                not: null,

            },

        },

    });

    const instructorStudentCount = {};

    for (const cls of classes) {

        if (!cls.instructorId) continue;

        instructorStudentCount[cls.instructorId] = (instructorStudentCount[cls.instructorId] || 0) + cls.enrollments.length;

    }

    return Object.entries(instructorStudentCount).map(([instructorId, studentCount]) => ({

        instructorId: parseInt(instructorId),

        studentCount,

    }));

}

// 9. Courses With and Without Classes

export async function getCoursesWithAndWithoutClasses() {

    const courses = await prisma.course.findMany({

        include: {

            classes: true,

        },

    });

    let withClasses = 0;

    let withoutClasses = 0;

    for (const course of courses) {

        if (course.classes.length > 0) {

            withClasses++;

        } else {

            withoutClasses++;

        }

    }

    return [

        { name: "With Classes", value: withClasses },

        { name: "Without Classes", value: withoutClasses },

    ];

}

// 10. Class sizes per course

export async function getClassSizesPerCourse() {

    const classes = await prisma.class.findMany({

        include: {

            enrollments: true,

            course: true,

        },

    });

    return classes.map(cls => ({

        courseName: cls.course.name,

        classId: cls.id,

        enrolledStudents: cls.enrollments.length,

        capacity: cls.capacity,

    }));

}

**Students.js:**

import prisma from "@/repos/prisma";

export async function getStudentLearningPath(studentId) {

    const enrollments = await prisma.enrollment.findMany({

        where: { studentId },

        include: {

            class: {

                include: {

                    course: true,

                    instructor: true

                }

            }

        }

    });

    const enrolled = [];

    const completed = [];

    enrollments.forEach((enrollment) => {

        const cls = enrollment.class;

        const course = cls.course;

        const instructor = cls.instructor ? cls.instructor.name : "Unknown";

        const courseData = {

            name: course.name,

            category: course.category,

            instructor,

            classId: cls.id,

        };

        if (cls.status === "IN\_PROGRESS") {

            enrolled.push(courseData);

        } else if (cls.status === "COMPLETED") {

            completed.push({

                ...courseData,

                grade: enrollment.grade || "N/A",

            });

        }

    });

    return { enrolled, completed };

}

export async function getAvailableClasses() {

    const classes = await prisma.class.findMany({

        where: {

            status: "OPEN",

        },

        include: {

            course: true,

            instructor: true,

            enrollments: true,

        },

    });

    return classes.map(cls => ({

        classId: cls.id,

        courseId: cls.courseId,

        courseName: cls.course.name,

        category: cls.course.category,

        instructorName: cls.instructor ? cls.instructor.name : "Unassigned",

        schedule: cls.schedule,

        capacity: cls.capacity,

        currentStudents: cls.enrollments?.length || 0,

    }));

}

export async function registerStudent(studentId, classId, courseId) {

    const targetClass = await prisma.class.findUnique({

        where: {

            id\_courseId: { id: classId, courseId: courseId }

        },

        include: {

            course: {

                include: {

                    prerequisites: true

                }

            },

            enrollments: true

        }

    });

    if (!targetClass) {

        throw new Error("Class not found");

    }

    if (targetClass.enrollments.length >= targetClass.capacity) {

        throw new Error("Class is full");

    }

    const prerequisites = targetClass.course.prerequisites;

    const studentEnrollments = await prisma.enrollment.findMany({

        where: {

            studentId: studentId

        },

        include: {

            class: {

                include: {

                    course: true

                }

            }

        }

    });

    const completedCourses = new Set(

        studentEnrollments.map(e => e.class.course.id)

    );

    const missingPrereqs = prerequisites.filter(prereq => !completedCourses.has(prereq.id));

    if (missingPrereqs.length > 0) {

        throw new Error("Missing prerequisites: " + missingPrereqs.map(c => c.name).join(", "));

    }

    await prisma.enrollment.create({

        data: {

            studentId: studentId,

            classId: classId,

            courseId: courseId

        }

    });

    return { message: "Registration successful!" };

}

**User.js:**

import prisma from "@/repos/prisma";

export async function findUserByUsername(username) {

  return await prisma.user.findUnique({ where: { username } });

}

export async function findUserById(id) {

  return await prisma.user.findUnique({ where: { id } });

}

export async function createUser(userData) {

  return await prisma.user.create({

    data: {

      id: userData.id,

      username: userData.username,

      password: "123",

      name: userData.name,

      role: userData.role,

    },

  });

}

# Discussion of the project contribution of each team member

|  |  |
| --- | --- |
| **Student name** | **Student contributions** |
| Alwaleed Sarieh Dsher | 34% |
| Ali Adel Abouelkhir | 33% |
| Yousef Elsherif | 33% |