DB Final Project

Database Management Systems

Dr. Babak Forouraghi

Nick Kraus,
John Rogers, and
Marc Ramsarran-Humphrey

12/4/25

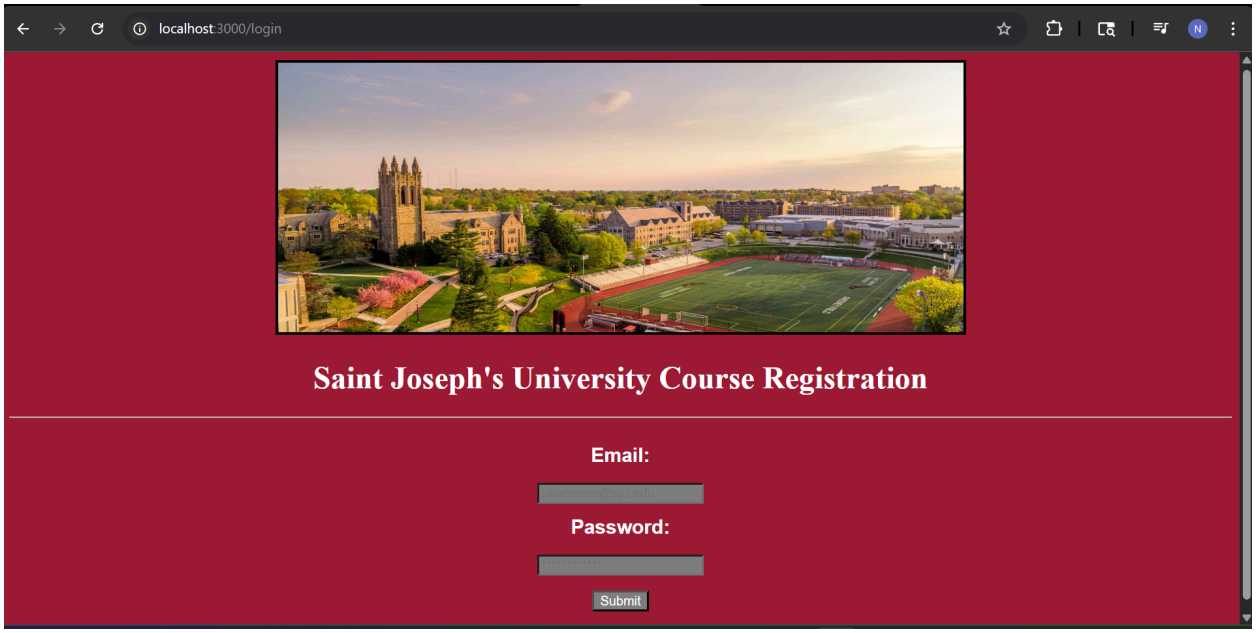
One Page Proposal

For our project, we built a course registration system using Node.js and MySQL that works as a college class registration website. The system lets students log in, search for classes, sign up for them, drop courses and update course information. Using express session the session will be tracked for the user and store their information while logged in. This also allows for our website to have a real login/out mechanic that doesn't allow any way of getting to other pages without logging in. The browser is what the student interacts with and the node.js server handles everything behind the scenes, like checking passwords, keeping track of who logged in and talking to the database our website uses three main tables from the database which are users sections and enrollments.

The user is first greeted with the login page which is required to fill out with a valid email and password from the database. Then the session tracking saves this users session ID and sends them to the menu. The menu gives the user the ability to traverse between pages, update, drop, register, and search and also logout. When students try to register or drop a class the server checks if the course exists whether the student is already enrolled and then adds or removes the row in the registrations table. If something is wrong like trying to register for something you already have or dropping a class you're not in the system will send a message letting the user know the error handling. The system is set up to make sure the user only registers valid section IDs, once per section, and a maximum of five classes.

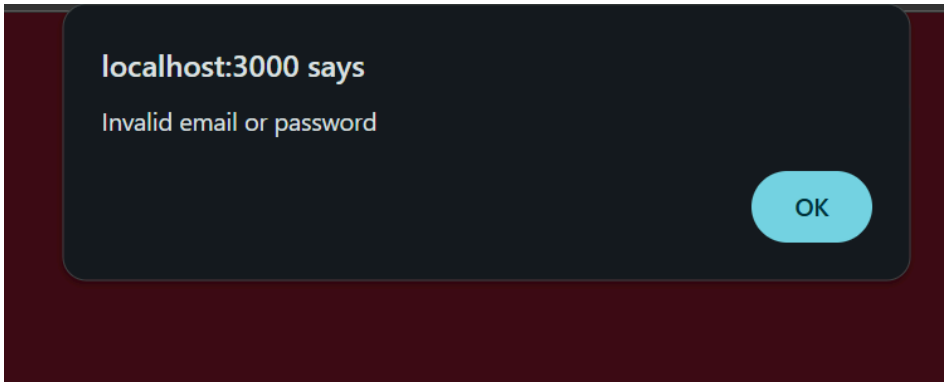
Functionalites

Log In and Out:



Any user who tries to access our website is first greeted with this page. Using session tracking, the requireLogin function guarantees users can't access other pages without signing in first. The HTML file requires that these forms be filled out; otherwise, it gives a warning.

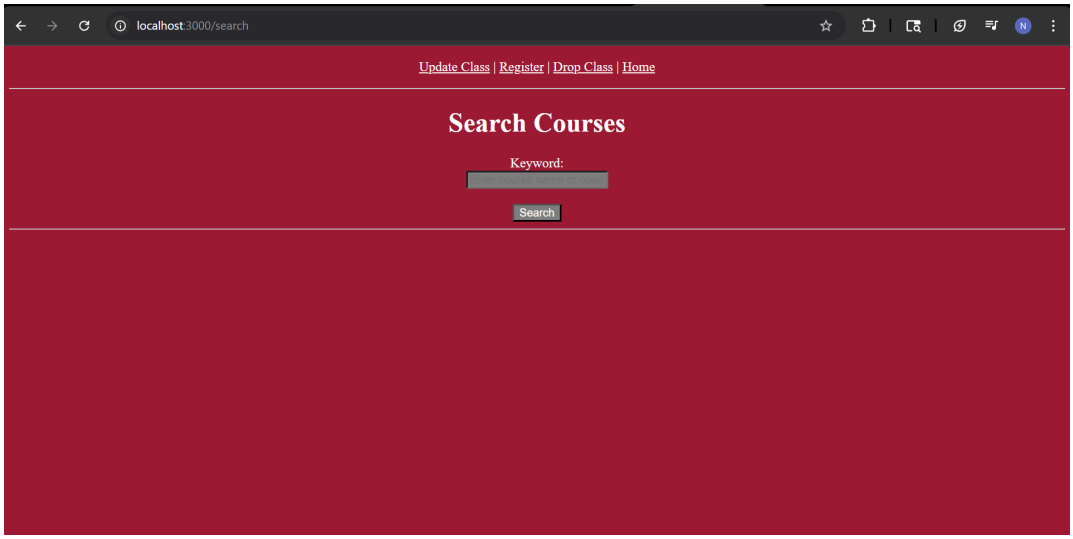
Emails and passwords are stored in the SQL database, and any user who doesn't input a verified email address and password receives an alert and is sent back to the login screen.



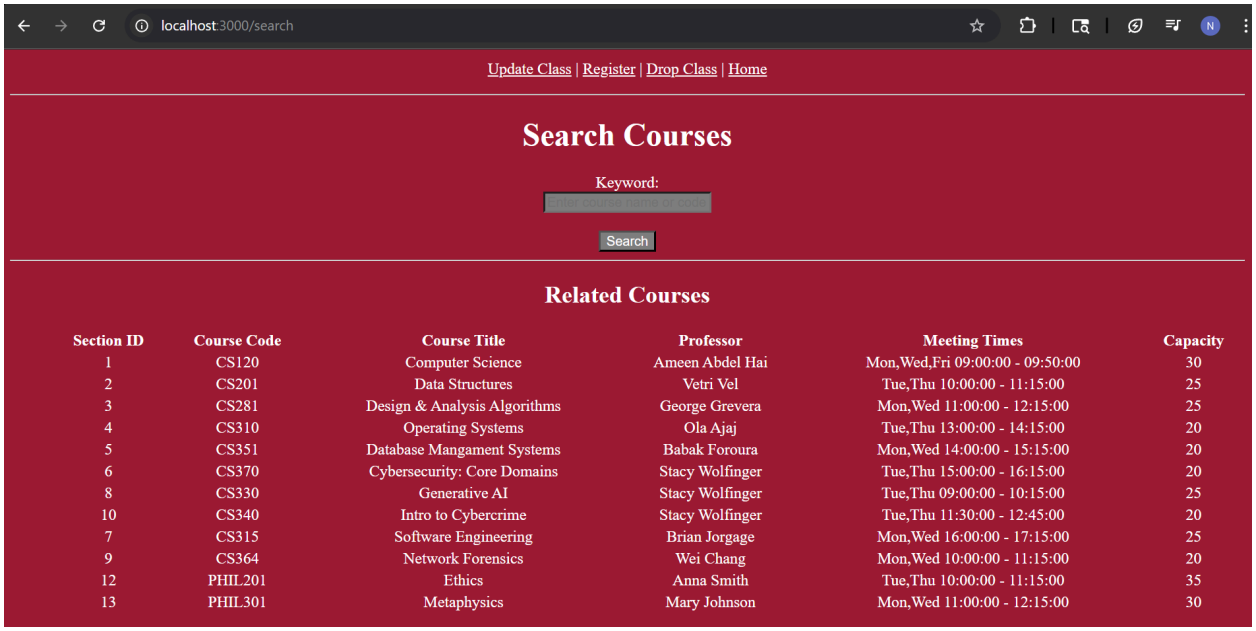
After logging in, the user is greeted with the Home Page, which holds the remaining functionalities and the ability to log out, stationed at the bottom of the page, which sends the user back to the login page and erases the session.

Search Courses:

After clicking the Search Courses link on the main menu, you are brought to the following page.



At the top of the page, there is a small navigation bar to have easy access to all other tabs on the site. The one search bar allows you to input text for a keyword search of available classes. This keyword searches for all classes with a title, code, or professor's name containing the keyword.

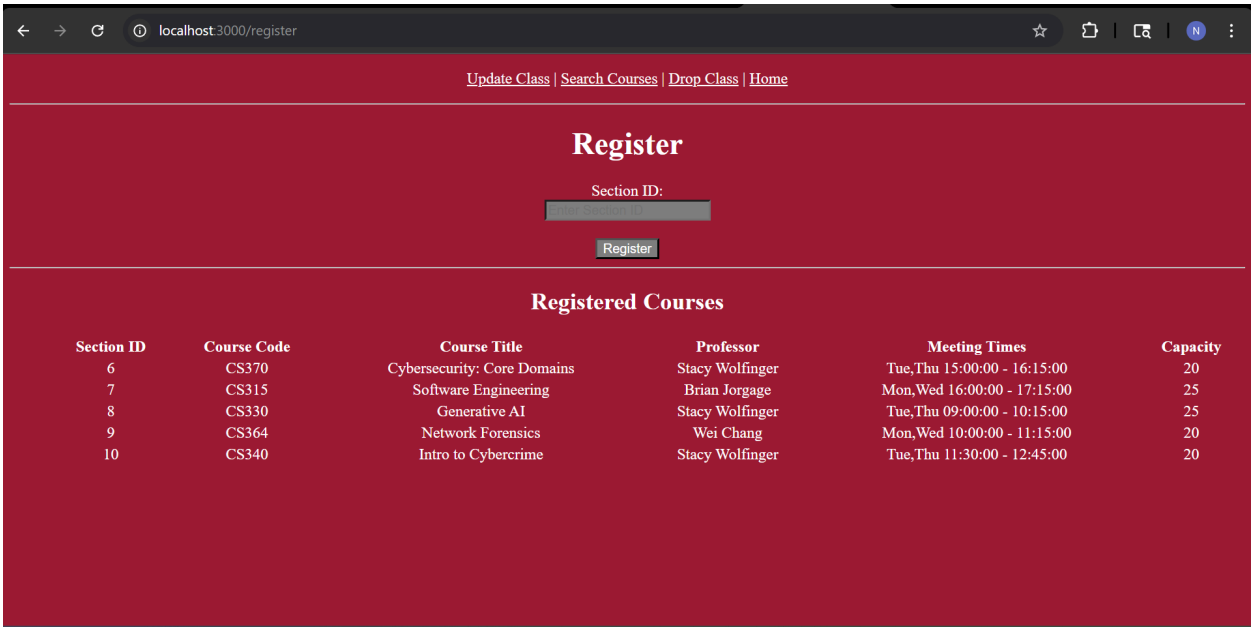


Section ID	Course Code	Course Title	Professor	Meeting Times	Capacity
1	CS120	Computer Science	Ameen Abdel Hai	Mon,Wed,Fri 09:00:00 - 09:50:00	30
2	CS201	Data Structures	Vetri Vel	Tue,Thu 10:00:00 - 11:15:00	25
3	CS281	Design & Analysis Algorithms	George Grevera	Mon,Wed 11:00:00 - 12:15:00	25
4	CS310	Operating Systems	Ola Ajaj	Tue,Thu 13:00:00 - 14:15:00	20
5	CS351	Database Mangament Systems	Babak Foroura	Mon,Wed 14:00:00 - 15:15:00	20
6	CS370	Cybersecurity: Core Domains	Stacy Wolfinger	Tue,Thu 15:00:00 - 16:15:00	20
8	CS330	Generative AI	Stacy Wolfinger	Tue,Thu 09:00:00 - 10:15:00	25
10	CS340	Intro to Cybercrime	Stacy Wolfinger	Tue,Thu 11:30:00 - 12:45:00	20
7	CS315	Software Engineering	Brian Jorgage	Mon,Wed 16:00:00 - 17:15:00	25
9	CS364	Network Forensics	Wei Chang	Mon,Wed 10:00:00 - 11:15:00	20
12	PHIL201	Ethics	Anna Smith	Tue,Thu 10:00:00 - 11:15:00	35
13	PHIL301	Metaphysics	Mary Johnson	Mon,Wed 11:00:00 - 12:15:00	30

The search returns an HTML table of the SQL query results underneath the search bar. The query shown above input "cs" and got all Computer Science classes along with Philosophy class titles that included "cs".

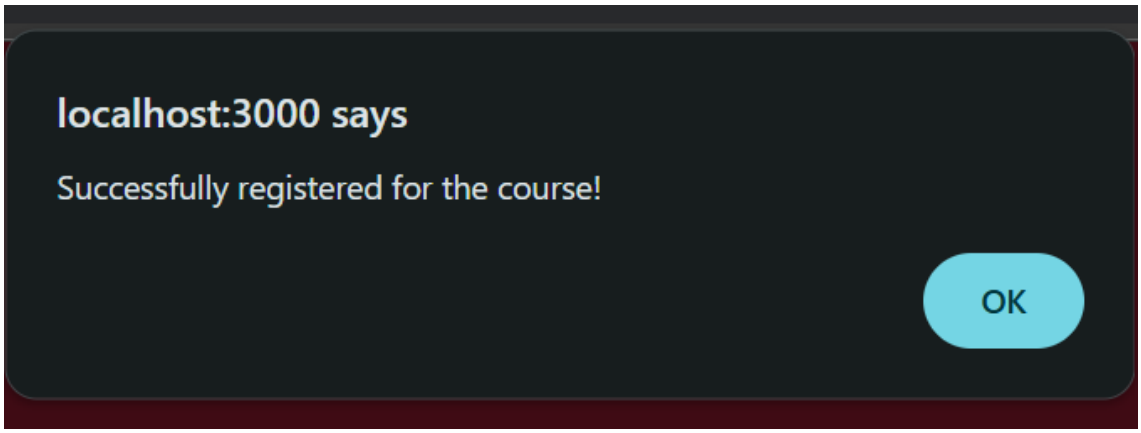
Register Courses:

Navigating to the Register page, you have a similar format along with a table at the bottom showing all registered classes. In this example, I am using a student who has 5 classes registered, which is the maximum. This page and all remaining unexplored pages show the currently registered classes for the signed-in user.

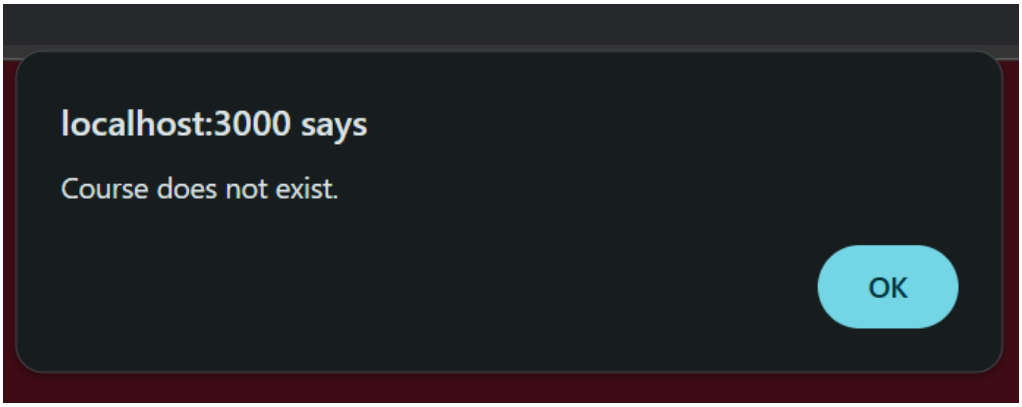


Section ID	Course Code	Course Title	Professor	Meeting Times	Capacity
6	CS370	Cybersecurity: Core Domains	Stacy Wolfinger	Tue,Thu 15:00:00 - 16:15:00	20
7	CS315	Software Engineering	Brian Jorgage	Mon,Wed 16:00:00 - 17:15:00	25
8	CS330	Generative AI	Stacy Wolfinger	Tue,Thu 09:00:00 - 10:15:00	25
9	CS364	Network Forensics	Wei Chang	Mon,Wed 10:00:00 - 11:15:00	20
10	CS340	Intro to Cybercrime	Stacy Wolfinger	Tue,Thu 11:30:00 - 12:45:00	20

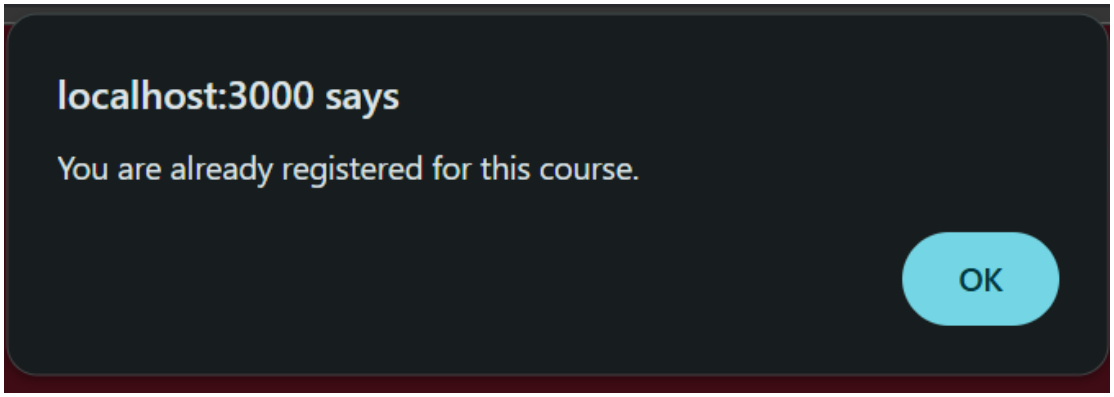
When a user inputs a valid section ID and presses register, they are given a confirmation message, and the database is updated.



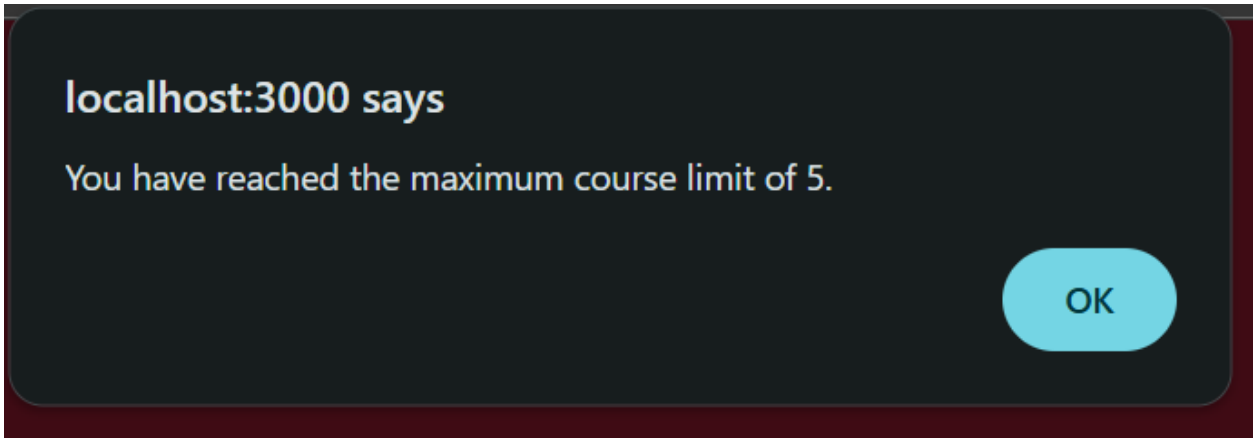
In the event that a non-existent section ID is input, the user is sent an alert.



In the event that a user inputs a section ID for a class that they are already registered for, they are sent an alert.

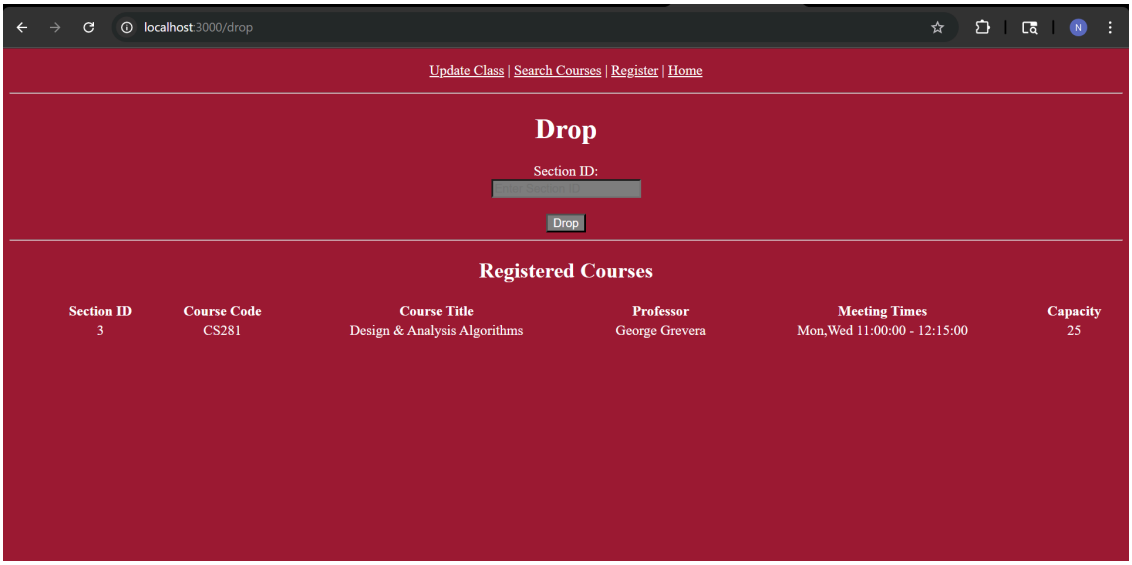


Also, in the event that a user attempts to register for a 6th class, the database does not change, and the user is sent an alert, as 5 is the maximum number of registered classes.

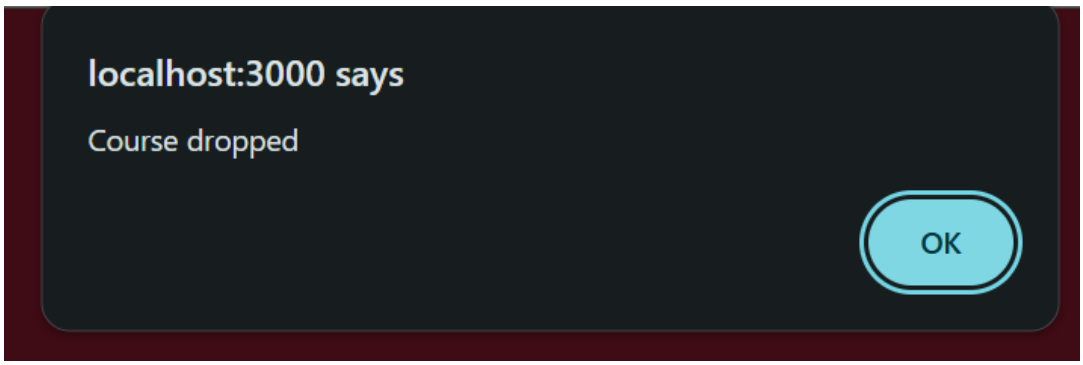


Drop Courses:

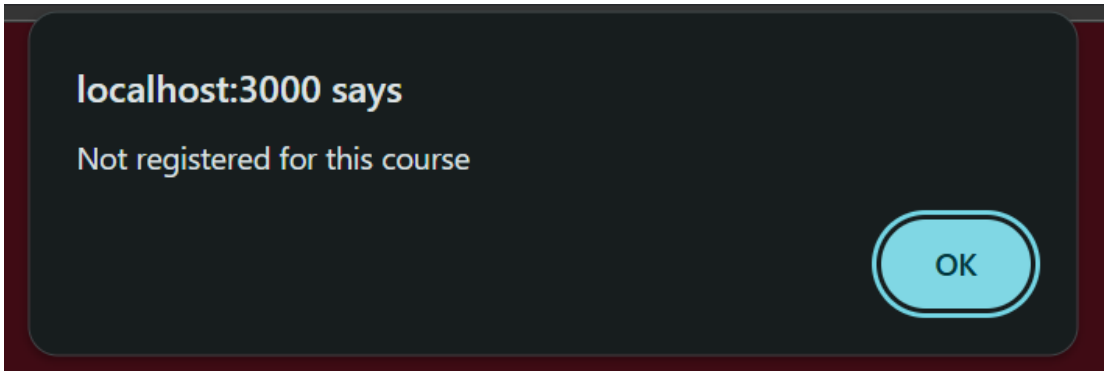
The next functionality is the drop course. This page, with a very similar format, allows the user to drop any registered course.



When the valid section ID is input and submitted, an alert is sent to the user confirming the drop, and the enrollment table in the database is changed using a delete function.

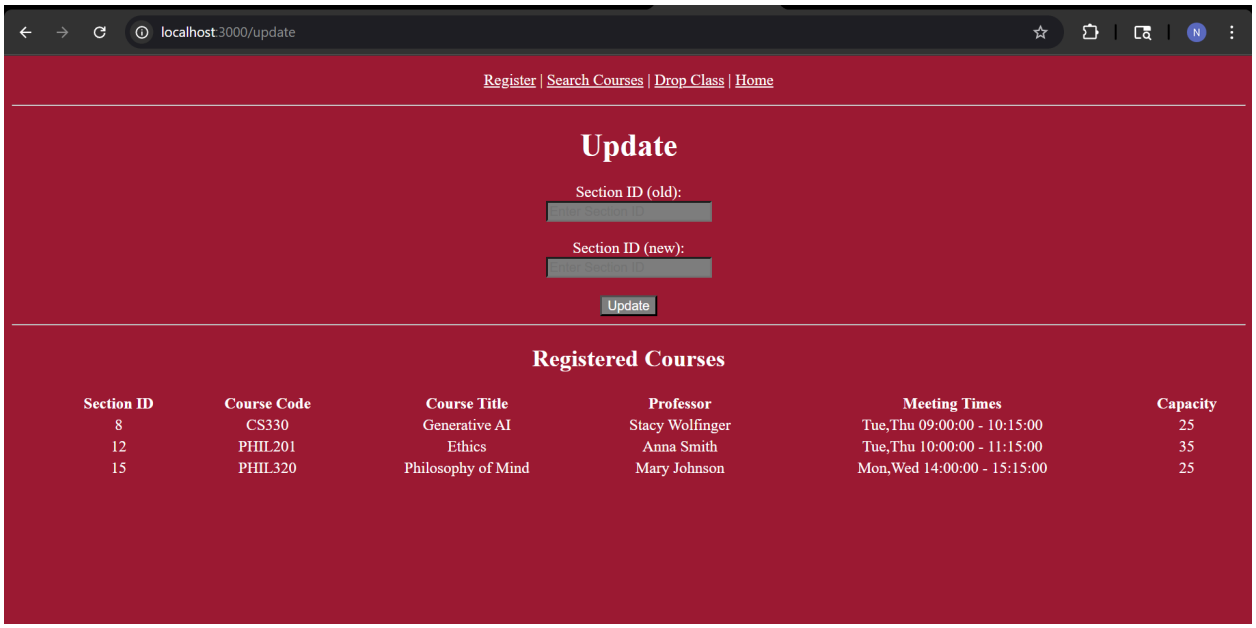


In the event of a user submitting an invalid Section ID or an ID for a course they aren't registered they receive an alert.

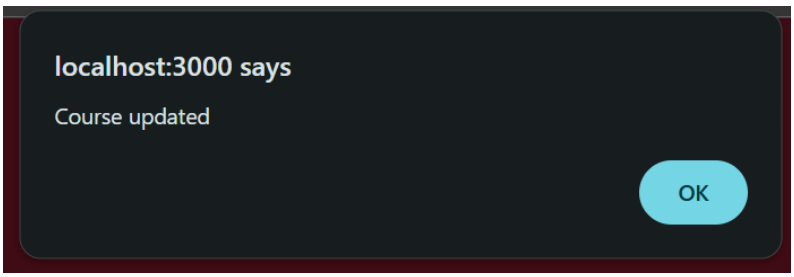


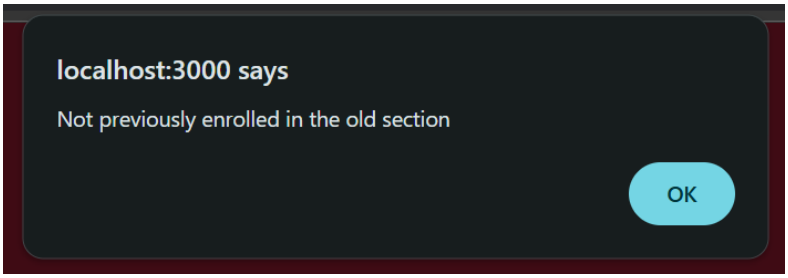
Update Courses:

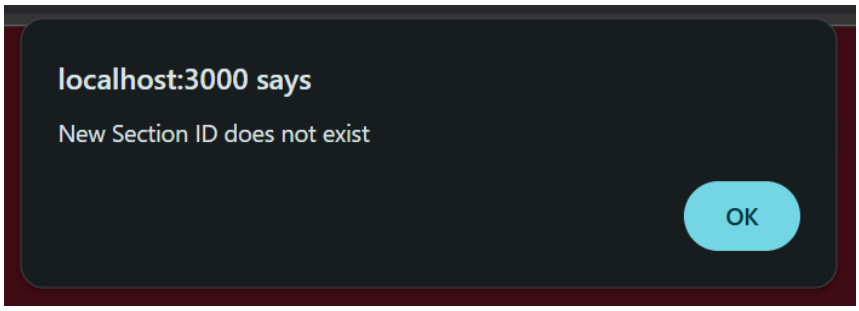
The final page and functionality of the website is the update course function.

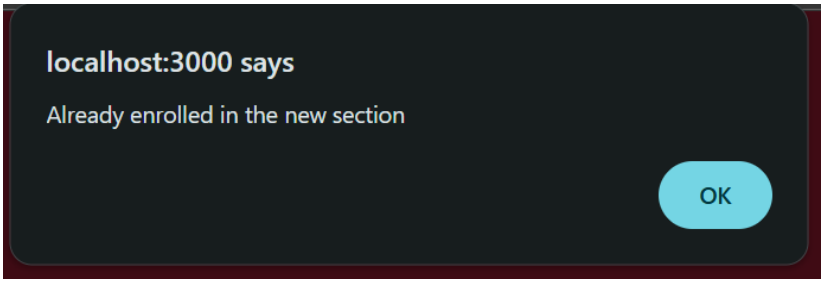


Just like the other pages, this one also has a dynamic view of registered courses and has two text inputs. The first input takes the ID of a course already registered that is to be updated to the other input, which is a new ID. This page also displays alerts when an update successfully occurs, a non-existent ID is input, the old ID is not currently registered or non-existent, and the new ID is already registered or non-existent.



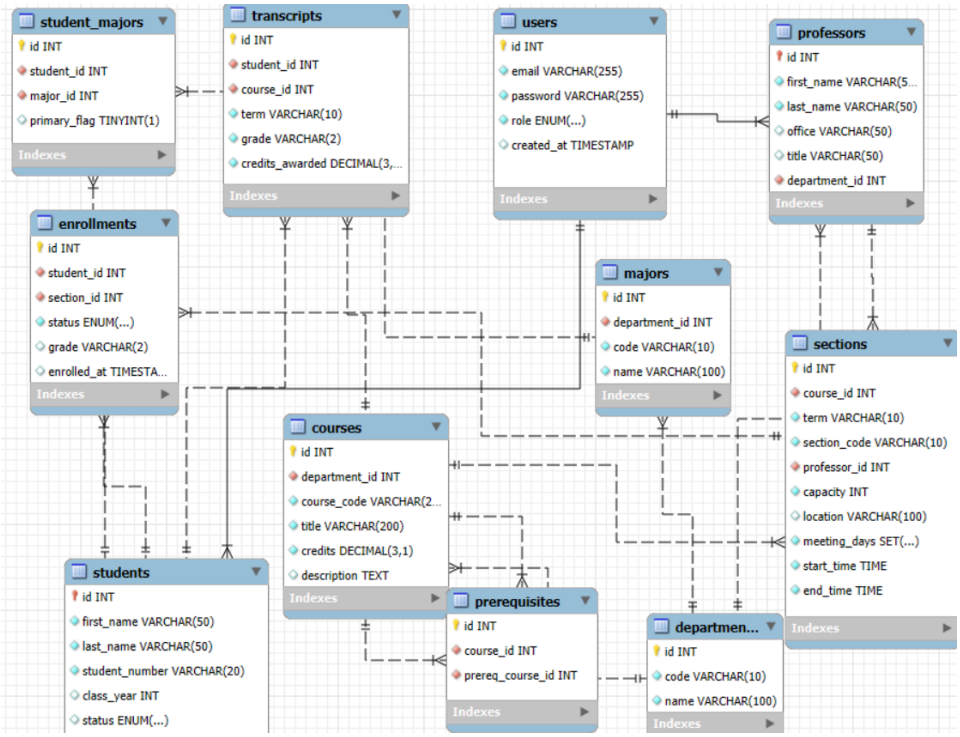






MySQL

ER Diagram:



SQL Code:

```sql
create database course_registration;
use course_registration;

-- Users table with plain text password
create table users (
  id int auto_increment primary key,
  email varchar(255) not null unique,
  password varchar(255) not null,   -- changed from password_hash
  role enum('student','professor','admin') not null,
  created_at timestamp default current_timestamp
);

create table departments (
  id int auto_increment primary key,
  code varchar(10) not null unique,
  name varchar(100) not null
);

create table professors (
  id int primary key,
  first_name varchar(50) not null,
  last_name varchar(50) not null,
  office varchar(50),
  title varchar(50),
  department_id int not null,
  foreign key (id) references users(id),
  foreign key (department_id) references departments(id)
);
```

```sql
create table students (
  id int primary key,
  first_name varchar(50) not null,
  last_name varchar(50) not null,
  student_number varchar(20) not null unique,
  class_year int,
  status enum('active','inactive') default 'active',
  foreign key (id) references users(id)
);

create table majors (
  id int auto_increment primary key,
  department_id int not null,
  code varchar(10) not null unique,
  name varchar(100) not null,
  foreign key (department_id) references departments(id)
);

create table student_majors (
  id int auto_increment primary key,
  student_id int not null,
  major_id int not null,
  primary_flag boolean default false,
  foreign key (student_id) references students(id),
  foreign key (major_id) references majors(id),
  unique key uniq_student_major (student_id, major_id)
);
```

```sql
create table courses (
    id int auto_increment primary key,
    department_id int not null,
    course_code varchar(20) not null,
    title varchar(200) not null,
    credits decimal(3,1) not null,
    description text,
    foreign key (department_id) references departments(id),
    unique key uniq_course (department_id, course_code)
);

create table prerequisites (
    id int auto_increment primary key,
    course_id int not null,
    prereq_course_id int not null,
    foreign key (course_id) references courses(id),
    foreign key (prereq_course_id) references courses(id),
    unique key uniq_prereq (course_id, prereq_course_id)
);
```

```sql
create table sections (
    id int auto_increment primary key,
    course_id int not null,
    term varchar(10) not null,
    section_code varchar(10) not null,
    professor_id int not null,
    capacity int not null,
    location varchar(100),
    meeting_days set('Mon','Tue','Wed','Thu','Fri') not null,
    start_time time not null,
    end_time time not null,
    foreign key (course_id) references courses(id),
    foreign key (professor_id) references professors(id),
    unique key uniq_section (course_id, term, section_code)
);

create table enrollments (
    id int auto_increment primary key,
    student_id int not null,
    section_id int not null,
    status enum('enrolled','waitlisted','dropped','completed') not null default 'enrolled',
    grade varchar(2),
    enrolled_at timestamp default current_timestamp,
    foreign key (student_id) references students(id),
    foreign key (section_id) references sections(id),
    unique key uniq_enrollment (student_id, section_id)
);
```

```sql
create table transcripts (
    id int auto_increment primary key,
    student_id int not null,
    course_id int not null,
    term varchar(10) not null,
    grade varchar(2) not null,
    credits_awarded decimal(3,1) not null,
    foreign key (student_id) references students(id),
    foreign key (course_id) references courses(id),
    unique key uniq_transcript (student_id, course_id, term)
);

```

This SQL script defines a course registration database schema that uses tables that are linked through foreign keys.

Tables included: users (students, professors, admins), academic departments, majors, courses, prerequisites, sections, student enrollments, and transcripts.

Server-Side

SQL Connection:

```javascript
const mysql = require('mysql2/promise');

// Database connection configuration
const dbConfig = {
    host: "localhost",
    user: "root",
    password: "toor",
    database: "course_registration",
    waitForConnections: true,
    connectionLimit: 10,
    queueLimit: 0
};

const pool = mysql.createPool(dbConfig);
pool.getConnection()
    .then(connection => {
        console.log("Database Pool: Successfully connected to MySQL");
        connection.release();
    })
    .catch(err => {
        console.error("Database Pool Error:", err.message);
    });

module.exports = pool;
```

This file accesses a connection to the MySQL database and exports it as pool to be used in the main index js file.

Get Functions:

```
// Route handlers
app.get("/login", async (req, res) => {
    // Await the file serving operation
    await readAndServe("./login.htm", res);
});

app.get("/", async (req, res) => {
    // Await the file serving operation
    await readAndServe("./login.htm", res);
});

app.get("/menu", requireLogin, async (req, res) => {
    // Await the file serving operation
    await readAndServe("./menu.html", res);
});

app.get("/search", requireLogin, async (req, res) => {
    // Await the file serving operation
    await readAndServe("./search.html", res);
});
```

These get functions serve as the handlers for url routing and manage all access for the login menu and search page. They don't need any extra HTML aside from what is already on the file so they just use the readAndServe function. All get functions besides login and log out have the requireLogin function as a parameter which confirms that user is logged in to access that url.

```
app.get("/register", requireLogin, async (req, res) => {
    // Await the file serving operation
    try {
        let html = await fsp.readFile("./register.html", "utf8");

        html += await registeredCourses(req);

        res.send(html);
    } catch (err) {
        console.error("Register Page Error:", err);
        res.status(500).send("Server error while loading register page");
    }
});

app.get("/drop", requireLogin, async (req, res) => {
    // Await the file serving operation
    try {
        let html = await fsp.readFile("./drop.html", "utf8");

        html += await registeredCourses(req);

        res.send(html);
    } catch (err) {
        console.error("Drop Page Error:", err);
        res.status(500).send("Server error while loading drop page");
    }
});
```

```
app.get("/update", requireLogin, async (req, res) => {
    // Await the file serving operation
    try {
        let html = await fsp.readFile("./update.html", "utf8");

        html += await registeredCourses(req);

        res.send(html);
    } catch (err) {
        console.error("Update Page Error:", err);
        res.status(500).send("Server error while loading update page");
    }
});

app.get("/logout", (req, res) => {
    req.session.destroy(() => {
        res.redirect("/login");
    });
});
```

Register drop and update have slightly different get methods because of the addition of a dynamic view of the users registered courses. These get methods don't use the readAndServe because there is additional html information to add to the file. They use the registeredCourses function which returns an html table of all registered classes.

```
// Function to get registered courses for the logged-in user
async function registeredCourses(req) {
    const query = "select section_id, course_code, courses.title, professors.first_name, professors.last_name, meeting_days, start_time, en
          + ", capacity from enrollments join sections on enrollments.section_id = sections.id join courses on sections.course_id = cours
          + " join professors on sections.professor_id = professors.id where enrollments.student_id = ?";

    const [rows] = await pool.query(query, [req.session.uid]);

    let html = "<html><body><h2>Registered Courses</h2><ul>";
    html += "<table style=\"width:100% ; text-align:center\"><tr><th>Section ID</th><th>Course Code</th><th>Course Title</th><th>Professor<
    rows.forEach(row => {
        html += `<tr><td>${row.section_id}</td><td>${row.course_code}</td><td>${row.title}</td><td>${row.first_name} ${row.last_name}</td><
    });
    html += "</table></body></html>";
    return html;
```

The registeredCourses function is used by multiple get statements to print all registered courses by the user. It sends an SQL query to the database and gets all necessary information.

```
const fsp = require("fs").promises;

async function readAndServe(path, res) {
    try {
        const data = await fsp.readFile(path);
        res.setHeader('Content-Type', 'text/html');
        res.end(data);
    } catch (err) {
        console.error("File Read Error:", err);
        res.status(404).send("<html><body><h1>404 Not Found</h1><p>The file " + path + " could not be served.</p></body></html>");
    }
}
```

The readAndServe function serves as a method to display the html files for the get functions.

```
// Function to track login status
function requireLogin(req, res, next) {
    if (!req.session.uid) {
        return res.redirect("/login");
    }
    next();
}
```

The requireLogin function if a session ID exists which happens after a successful login post function. If a session doesn't exist and the user isn't logged in they are redirected to the login page. If not the next line pushes this function on and allows the get method to be called.

Post Functions:

Drop

```
app.post("/drop", requireLogin, async (req, res) => {
    const studentId = req.session.uid;
    const section_id = req.body.course_id;

    try {
        //Deletes enrollment from enrollments table
        const [result] = await pool.query(
            "DELETE FROM enrollments WHERE student_id = ? AND section_id = ?",
            [studentId, section_id]
        );

        //Checks if user was registered for the course
        if (result.affectedRows === 0) {
            return res.send("<script>alert('Not registered for this course'); window.location.href='/drop';</script>");
        }

        res.send("<script>alert('Course dropped'); window.location.href='/menu';</script>");

    } catch (err) {
        console.error(err);
        res.status(500).send("Drop error");
    }
});
```

The drop function allows a logged-in student to remove a course they are currently enrolled in. When a student submits the drop form, the server will first pull their user ID from the session and the section ID they entered. It will then delete the matching enrollment record from the enrollments table. If the database shows that zero rows were deleted, the student was not registered for that course, so the system will alert them that they can't drop a course they aren't enrolled in. If a row is deleted, the drop is successful, and the user receives a confirmation message. The drop function checks whether the student is enrolled in the course before removing them from it, and displays the appropriate success or error message.

Update

```
app.post("/update", requireLogin, async (req, res) => {
    const oldSection_id = req.body.section_idold;
    const newSection_id = req.body.section_idnew;

    try {
        // Checks if old and new section IDs are the same
        if (oldSection_id === newSection_id) {
            return res.send("<script>alert('Old and new Section IDs cannot be the same'); window.location.href='/update';</script>");
        }

        // Checks if the student is enrolled in the new section
        const [enrollmentCheck] = await pool.query(
            "SELECT * FROM enrollments WHERE student_id = ? AND section_id = ?",
            [req.session.uid, newSection_id]
        );
        if (enrollmentCheck.length > 0) {
            return res.send("<script>alert('Already enrolled in the new section'); window.location.href='/update';</script>");
        }

        // Checks if the new section ID exists
        const [newSectionCheck] = await pool.query(
            "SELECT * FROM sections WHERE id = ?",
            [newSection_id]
        );

        if (newSectionCheck.length === 0) {
            return res.send("<script>alert('New Section ID does not exist'); window.location.href='/update';</script>");
        }

        // Updates enrollment in enrollments table
        const [result] = await pool.query(
            "UPDATE enrollments SET section_id = ? WHERE section_id = ? AND student_id = ?",
            [newSection_id, oldSection_id, req.session.uid]
        );

        // Checks if the update affected any rows
        if (result.affectedRows === 0) {
            return res.send("<script>alert('Not previously enrolled in the old section'); window.location.href='/update';</script>");
        } else
            res.send("<script>alert('Course updated'); window.location.href='/update';</script>");

    } catch (err) {
        console.error(err);
        res.status(500).send("Update error");
    }
});
```

The update function allows a student to switch between course sections in a single step. When the student submits the update form, the server retrieves both the old section and the new section and checks that the two section IDs are not the same. It then verifies that the student is not already enrolled in the new section and confirms that the new section actually exists in the database. Next, the function attempts to update the student's enrollment record by replacing the old section ID with the new one. If the update is successful, the student sees a confirmation message. The update function correctly moves a student from one section to another while preventing invalid updates or mistaken inputs.

Register

```
app.post("/register", requireLogin, async (req, res) => {
    const courseID = req.body.course_id;

    try {
        // Checks if course exists
        const [courseCheck] = await pool.query(
            "SELECT * FROM sections WHERE id = ?",
            [courseID]
        );

        if (courseCheck.length === 0) {
            return res.send(`
                <script>
                    alert("Course does not exist.");
                    window.location.href = "/register";
                </script>
            `);
        }

        // Checks if student is at course limit
        const [enrolledRows] = await pool.query(
            "select count(*) from enrollments where student_id = ?",
            [req.session.uid]
        );

        if (enrolledRows[0]['count(*)'] >= 5) {
            return res.send(`
                <script>
                    alert("You have reached the maximum course limit of 5.");
                    window.location.href = "/register";
                </script>
            `);
        }

        // Checks if already registered
        const [rows] = await pool.query(
            "select * from enrollments where student_id = ? AND section_id = ?",
            [req.session.uid, courseID]
        );

        if (rows.length > 0) {
            return res.send(`
                <script>
                    alert("You are already registered for this course.");
                    window.location.href = "/register";
                </script>
            `);
        }

        // Registers the user for the course
        await pool.query(
            "INSERT INTO enrollments (student_id, section_id) VALUES (?, ?)",
            [req.session.uid, courseID]
        );

        return res.send(`
            <script>
                alert("Successfully registered for the course!");
                window.location.href = "/register";
            </script>
        `);

    } catch (error) {
        console.error("Registration Error:", error);
        return res.status(500).send("Server error during registration");
    }
});
```

The register post function is intended to take an ID from the user and add to the enrollments table on sql with that ID and the student ID of the logged in user. It starts by getting a text input from html with the name course_id. This value is saved in a variable and then tested to make sure it is a valid. First the server calls an sql select statement to verify that the section ID exists. Then it uses another sql statement to determine if the student is enrolled in the max number of classes. Then the third and final check sends another sql query to verify that the student is already enrolled in that class. If any check fails the user is sent an alert and the registration does not got through. Then if everything verifies then the database is sent an insert into statement, the enrollments table is added to, and the user sent an alert that the registration worked.

Log In

```
app.post("/login", async (req, res) => {
  const { email, password } = req.body;

  try {
    const [rows] = await pool.query(
      "SELECT * FROM users WHERE email = ? AND password = ?",
      [email, password]
    );

    if (rows.length == 0) {
      return res.send(`
        <script>
            alert("Invalid email or password");
            window.location.href = "/login";
        </script>`);
    }

    // Save session
    req.session.uid = rows[0].id;

    // Redirect
    req.session.save(() => {
      res.redirect("/menu");
    });
  } catch (error) {
    console.error(error);
    return res.status(500).send("Server error");
  }
});
```

The login function function accepts two text inputs from the client in the form of email and password. Then a try catch method is used and an sql pool query searches the student table to see if the input email and password exists in the database. If the length of the returned query is empty then the the inputed information doesn't exist, the user is sent an alert, and returned to the login screen.

Search

```
app.post("/search", requireLogin, async (req, res) => {
    const keyword = req.body.keyword;

    try {
        let html = await fsp.readFile("./search.html", "utf8");

        // Sends sql query for courses table by name or description
        const query = "select sections.id, courses.course_code, courses.title, professors.first_name, professors.last_name, meeting_days, s
            + ", capacity from sections join courses on sections.course_id = courses.id"
            + " join professors on sections.professor_id = professors.id"
            + " where courses.title like ? or course_code like ? or professors.first_name like ? or professors.last_name like ?";

        const [rows] = await pool.query(
            query,
            ['%' + keyword + '%', '%' + keyword + '%', '%' + keyword + '%', '%' + keyword + '%']
        );


        if (rows.length === 0) {
            return res.send(`
                <script>
                    alert("No results found for '${keyword}'");
                    window.location.href = "/search";
                </script>`);
        }
```

```
        // Builds HTML response for search query results
        html += "<html><body><h2>Related Courses</h2><ul>";
        html += "<table style=\"width:100% ; text-align:center\"><tr><th>Section ID</th><th>Course Code</th><th>Course Title</th><th>Profes
        rows.forEach(row => {
            html += `<tr><td>${row.id}</td><td>${row.course_code}</td><td>${row.title}</td><td>${row.first_name} ${row.last_name}</td><td>$
        });
        html += "</table></body></html>";

        res.send(html);
    } catch (error) {
        console.error("Search Error:", error);
        res.status(500).send("Server error during search");
    }
});
```

The Search function is lets a logged in student look up courses by typing keywords so when the student submits the search form the server will take those keywords and run a query that looks for similarities in the database and if the matching course exist it shows its ID course code course title professor meeting time and capacity of the class if nothing matches it will show a message that no results were found.

Client

Login

```
<!--Nick Kraus-->
<!--Database Management Systems-->
<!--12/4/25-->
<!--Login Page - Final Project-->
<!DOCTYPE html>
<html>
    <head>
        <title>Log In</title>
        <link rel="icon" href="https://alumlc.org/img/uploads/31576_1695145979.jpg" type="image/x-icon">
        <style>
            body {
                background-color: #9e1b32; color: white;
            }
            label {
                font-family: arial;
                font-size: 20px;
            }
            input {
                background-color: grey;
                color: white;
            }
        </style>
    </head>
    <body style="text-align: center">

        <img src="https://www.sju.edu/sites/mergersjuedu/files/styles/2_1_1600x800/public/2025-07/sju-sunrise-aerial-2021-3200x1600-dev-opt
        height="275" width="700" style="border: 3px solid black">

        <h1>Saint Joseph's University Course Registration</h1>
```

```
        <hr>

        <form action="/login" method="post">
            <label>
                Email: <br>
                <input type="email" placeholder="username@sju.edu" name="email" required> <br>
            </label>


            <label>
                Password: <br>
                <input type="password" placeholder="**********" name="password" required>
            </label><br>

            <input type="submit" style="color: white; background-color: grey;" value="Submit">
        </form>
    </body>
</html>
```

This code serves as the clientside construction of the log in page. The page tab icon, page title, and style format are all located in the head. A link to an image is added to it for additional visuals. The login information is collected via the use of a form in which each input is required, so empty forms are not accepted. This form then posts to the server in order to access the next page.

Menu

```
<!--Nick Kraus-->
<!--Database Management Systems-->
<!--12/4/25-->
<!--Menu Page - Final Project-->
<!DOCTYPE html>
<html>
    <head>
        <title>Home Page</title>
        <link rel="icon" href="https://alumlc.org/img/uploads/31576_1695145979.jpg" type="image/x-icon">
        <style>
            body {
                background-color: #9e1b32; color: white; text-align: center;
            }
            a {
                color: white; text-decoration: none;
            }
            label {
                font-family: arial; font-size: 20px;
            }
            input {
                background-color: grey; color: white;
            }
            table {
                margin-left: auto;
                margin-right: auto;
                font-size: 24px;
                border: 2px solid grey;
                width: 60%;
                height: 100px;
            }
    </head>
    <body>
        <h1>Saint Joseph's University Course Registration</h1>

        <hr><br>

        <img src="https://www.sju.edu/sites/mergersjuedu/files/styles/16_9_960x540/public/2022-12/sju-campus-aerial-hero-2021-3200x1600.jpg
         height="275" width="500" style="border: 3px solid black">

        <br> <br>

        <table>
            <tr>
                <td><a  href="/search">Search Courses</a></td>
                <td><a href="/register">Register</a></td>
            </tr>
            <tr>
                <td><a href="/drop">Drop Courses</a></td>
                <td><a href="/update">Update Section</a></td>
            </tr>
        </table>

        <br>
        <hr>
        <a href="/logout" style="text-align: left">Logout</a>
    </body>
```

The menu page is the home page of the website which holds links to all other pages on it and the ability to log out. The head and style are identical to the other files with the addition of the table that stores the links to other pages. There is also another link to an image on this page for visuals. In the table there is links to all the other pages and at the bottom there is a link to logout.

Update

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Update</title>
        <link rel="icon" href="https://alumlc.org/img/uploads/31576_1695145979.jpg" type="image/x-icon">
    </head>
    <style>
        body {
            background-color: #9e1b32;
            color: white;
            text-align: center;
        }
        input {
            background-color: grey;
            color: white;
        }
    </style>
    <body>
        <p>
            <a href="/register" style="color: white;">Register</a> |
            <a href="/search" style="color: white;">Search Courses</a> |
            <a href="/drop" style="color: white;">Drop Class</a> |
            <a href="/menu" style="color: white;">Home</a>
        </p>

        <hr>

        <h1>Update</h1>
```

```html
        <form method="post" action="/update">
            <label>Section ID (old):</label> <br>
            <input type="text" name="section_idold" placeholder="Enter Section ID" required> <br> <br>

            <label>Section ID (new):</label> <br>
            <input type="text" name="section_idnew" placeholder="Enter Section ID" required> <br> <br>

            <input type="submit" value="Update"> <br>
        </form>

        <hr>
    </body>
</html>
```

This HTML file serves as the update page on the website. The Update page and the remaining pages have very similar set ups with the difference lying in strictly title or certain input. These page also share the same style and head setup as the others. The update page in particular two texts one being named, section_idold, and the other, section_idnew, which are used to update a certain id to another in enrollment. This file and the remaining ones also have a small line at the top with links to the other pages separated by an hr break that serves as a navigational bar.

Search

```
<!--Nick Kraus-->
<!--Database Management Systems-->
<!--12/4/25-->
<!--Search Page - Final Project-->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Search Courses</title>
        <link rel="icon" href="https://alumlc.org/img/uploads/31576_1695145979.jpg" type="image/x-icon">
    </head>
    <style>
        body {
            background-color: #9e1b32;
            color: white;
            text-align: center;
        }
        input {
            background-color: grey;
            color: white;
        }
    </style>
    <body>
        <p>
            <a href="/update" style="color: white;">Update Class</a> |
            <a href="/register" style="color: white;">Register</a> |
            <a href="/drop" style="color: white;">Drop Class</a> |
            <a href="/menu" style="color: white;">Home</a>
        </p>
```

```
        <hr>

        <h1>Search Courses</h1>

        <form method="POST" action="/search">
            <label>Keyword:</label> <br>
            <input type="text" name="keyword" placeholder="Enter course name or code" required> <br> <br>
            <input type="submit" value="Search"> <br>
        </form>

        <hr>

    </body>
</html>
```

The search feature is much like the update page in format and style. The difference is that it takes on input in its form that is named keyword. This form is posted to the server to search for classes using the keyword.

Drop

```
<!--Nick Kraus-->
<!--Database Management Systems-->
<!--12/4/25-->
<!--Drop Page - Final Project-->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Drop</title>
        <link rel="icon" href="https://alumlc.org/img/uploads/31576_1695145979.jpg" type="image/x-icon">
    </head>
    <style>
        body {
            background-color: #9e1b32;
            color: white;
            text-align: center;
        }
        input {
            background-color: grey;
            color: white;
        }
    </style>
    <body>
        <p>
            <a href="/update" style="color: white;">Update Class</a> |
            <a href="/search" style="color: white;">Search Courses</a> |
            <a href="/register" style="color: white;">Register</a> |
            <a href="/menu" style="color: white;">Home</a>
        </p>
```

```
        <h1>Drop</h1>

        <form method="post" action="/drop">
            <label>Section ID:</label> <br>
            <input type="text" name="course_id" placeholder="Enter Section ID" required> <br> <br>
            <input type="submit" value="Drop"> <br>
        </form>

        <hr>
    </body>
</html>
```

Drop page, just like the other pages is set up with the title at the top and a form with text and submit inputs. This form is posted to the server with a section ID that drops that course from the users enrollment.

Register

```
<!--Nick Kraus-->
<!--Database Management Systems-->
<!--12/4/25-->
<!--Register Page - Final Project-->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Register</title>
        <link rel="icon" href="https://alumlc.org/img/uploads/31576_1695145979.jpg" type="image/x-icon">
    </head>
    <style>
        body {
            background-color: #9e1b32;
            color: white;
            text-align: center;
        }
        input {
            background-color: grey;
            color: white;
        }
    </style>
    <body>
        <p>
            <a href="/update" style="color: white;">Update Class</a> |
            <a href="/search" style="color: white;">Search Courses</a> |
            <a href="/drop" style="color: white;">Drop Class</a> |
            <a href="/menu" style="color: white;">Home</a>
        </p>
```

```
        <hr>

        <h1>Register</h1>

        <form method="post" action="/register">
            <label>Section ID:</label> <br>
            <input type="text" name="course_id" placeholder="Enter Section ID" required> <br> <br>
            <input type="submit" value="Register"> <br>
        </form>

        <hr>
    </body>
</html>
```

Register the final page of the website is formatted very alike the last few with a title, navigational bar, and a form to submit an ID. This ID is posted to the Nodejs sever and is used to register the logged in user to that course. The input in this form and all others in the other pages are marked required and don't accept empty values.

Participation Statement

Member 1: Nick Kraus
My teammates and I agree that I handled 33% of the overall project. My specific tasks included:

Task 1: I designed and implemented the session express session tracking and requireLogin, registeredCourses function, app.. get route handling, and registered courses formatting on the add, drop, and update pages
Task 2: I designed all the HTML files and the other JS HTML injections
Task 3: I wrote the login post function and configured/debugged all other post functions
Task 4: I wrote the functionalities, client, get functions, and drop/register/login post function portions of the project in the final report

Member 2: John Rogers
My teammate and I agree that I handled 33% of the overall project. My specific tasks included:

Task 1: I designed and implemented the EER Diagram and relational schema
Task 2: I created the MySQL database, wrote, and tested all SQL statements
Task 3: I built the data access layer in NodeJS
Task 4: I wrote code that handles server-side validation and error checking

Member 3: Marc Ramsarran-Humphrey
My teammate and I agree that I handled 33% of the overall project. My specific tasks included:

Task 1: I built the Express.js server structure, including all routes, middleware, and session handling.
Task 2: I wrote the backend logic that connects the web forms to the MySQL database and implemented the POST endpoints for drop and update.

Task 3: Use Express sessions, routing structure, and SQL queries
Task 4: I documented what each backend route does for the final project handoff.