

# Study on Campus

CS 326 Group 11

Milestone #7 | May 7th, 2025

[Presentation link](#)

# Project Overview

Working with a study partner or group can provide increased understanding, accountability, routine, camaraderie, and productivity. However, finding an effective one can be difficult to students for a multitude of reasons: large classes, social anxiety, mismatched goals, unclear expectations, and lacking a method of contact. **As a solution, we propose a web application that offers a platform to help students more easily connect with each other, find study locations that promote productivity, and organize or find study groups.**

## Key Features

- **Study group posting:** Students can post open invites for study groups, along with their goals, preferences, and location. Users can browse and filter for groups that match their needs.
- **Communication between users:** Users can discuss meeting details and coordinate study group plans through public comments on invite posts.
- **User-reported location crowding score:** Users can rate how crowded popular study spots are to help others make more informed decisions.

# Team

## **Erika Elston** (Project Manager)

Primary Contributions: [SQLite/Sequelize Location and Report integration, frontend adjustments](#) for crowding score reports and location browsing.

## **Julia Farber** (Time Keeper)

Primary Contributions: [SQLite/Sequelize Post integration and frontend adjustments](#) for post browsing and post viewing.

## **Anastasia Isakov** (Project Organizer/Documentation Lead)

Primary Contributions: [SQLite/Sequelize User integration and registration logic](#). [Media queries for navbar](#).

## **Ashley Kang** (Quality Control)

Primary Contributions: [SQLite/Sequelize Settings integration, frontend and data structure adjustments](#).

M	T	W	Th	F
<h1>Historical Development Timeline</h1> <p>Previous timelines: <a href="#">Milestone 4</a>, <a href="#">Milestone 5</a>, <a href="#">Milestone 6</a></p>		4/23	4/24	4/25 <b>Deadline</b> Milestone #6 (Front/Back-End Design & Integration)  <b>Team meeting</b> Stand-up #5
4/28	4/29 <b>Development</b> - Began report SQLite/Sequelize model integration and necessary refactoring - Updated location reporting modal UI (DOM)	4/30	5/1 <b>Development</b> - Began location SQLite/Sequelize model integration	5/2 <b>Team meeting</b> Stand-up #6 <b>Development</b> - Began Post SQLite/Sequelize model integration - Fixed eventhub issues with post browsing and viewing
5/5 <b>Team meeting</b> Discussed progress and final goals	5/6 <b>Team meeting</b> Online progress check  <b>Development</b> - Completed report and location models	5/7 <b>Deadline</b> Milestone #7 (Back-End Persistence) <b>Development</b> - Finalized post related functions and completed post model - Completed frontend adjustments for updated location model - <b>Submitted PRs for:</b> report & location backend, settings, course catalog data structure, login/signup pages	5/8  <b>Project Demo Day in LGRC A112</b>	5/9 <b>Last day of classes</b>



Erika Elston

# Work Summary

## Key Issues:

- [Location Browsing and Crowding Score Reporting Screen](#) (UI)
- [Location and Crowding Information Data Structure](#) (Data)
- [Crowding Score Report Data Structure](#) (Data)
- [Crowding Score Reporting Integration](#) (Server)

## Key Commits:

- [b3cde14](#) (4/30): Initial SQLite/Sequelize implementation for crowding score reports.
- [630ccdb](#) (5/2): Initial SQLite/Sequelize implementation for locations. Several corresponding updates to controllers and frontend.
- [dfa557c](#) (5/4): Adjust report creation to update corresponding location.
- [ca1e47c](#) (5/6): UI adjustments for crowding score reporting modal and expanded card views.
- [ee341ae](#) (5/7): Improve location read() for improved default card sorting behavior.

## Pull Requests:

- [#94](#) (5/7): Implemented SQLite/Sequelize models for locations and crowding score reports. Updated multiple frontend and backend elements to maintain intended functionality.

# Feature Demo: Location Browsing & Crowding Score Reporting

**Branches:** [12-location-browsing-and-crowding-score-reporting-screen](#) (frontend) and [69-crowding-score-report-integration](#) (backend)

**Note:** most frontend adjustments were made on the backend branch for this milestone

For this milestone, I worked to complete the **crowding score report feature implementation** using **SQLite** and **Sequelize** for locations and reports. From the location browsing page, users can view and report the “crowding scores” of different study spots around campus.

## Endpoints

- **GET /report** endpoint – accepts an object with at least a **report id**, validates that the data exists, and returns the corresponding report object
- **POST /report** endpoint – accepts an object containing a **location name (string, required)**, **floor (string, optional)**, and **crowding score (number)**. Validates information, adds a unique **report id**, updates the corresponding **Location**, and stores the report in **reports SQLite database**
- **GET /locations** endpoint – attempts to access the **locations SQLite database** and returns the contents in order of recent updates, if there are no errors
- **PUT /location** endpoint – attempts to access and write to the the **locations SQLite database**, returns an **ok** status if there are no error

## Feature: View Study Locations

- **Description:** The user can view study locations and user-reported crowding scores on the location browsing page. The data for the **locations** and **crowding scores** are both pulled from the backend (**GET** /locations, **GET** /report).
- **Completion Level:** This feature is fully complete and functional, though there are a few very minor frontend adjustments that will be made in the future.

## Feature: Submit Crowding Report

- **Description:** The user can **submit a crowding score report** (**POST** /report), which is stored in a **reports SQLite database**. The **report id** is also stored with the corresponding location in the **locations SQLite database** (**GET** /locations, **PUT** /location).
- **Completion Level:** Fully complete and functional.

# Code Structure

## Focusing on Location Browsing & Crowding Score Report Integration

frontend/src/services/ReportRepositoryRemoteService.js

```
JS ReportRepositoryRemoteService.js
JS Service.js
```

### Methods, Organization, and Frontend/Backend Separation

I referenced the final tasks example from lecture for organizing my backend components. The controller, routes, and models for data handling were composed of three separate files and follow the design patterns seen in class.

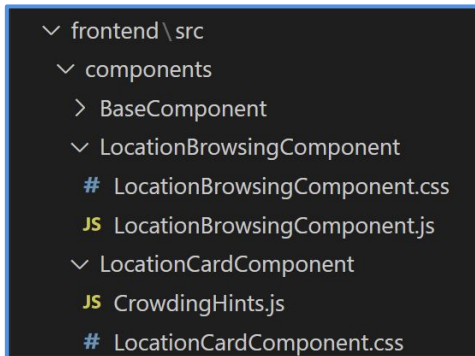
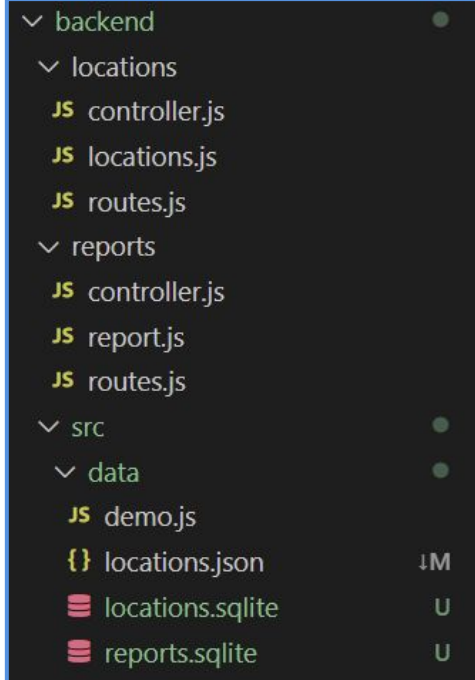
My frontend components are primarily located in frontend/src/components, in which major UI frontend components are separated by functionality, data requirements, and the pages they belong to.

frontend/src/eventhub/Events.js

```
// Location Browsing
ExpandLocationCard: 'ExpandLocationCard',
MinimizeLocationCard: 'MinimizeLocationCard',
OpenReportModal: 'OpenReportModal',
CloseReportModal: 'CloseReportModal',

// Crowding Score Reporting
AddReport: 'AddReport', // trigger html post method
AddReportSuccess: 'AddReportSuccess', // for UI updates
DeleteReport: 'DeleteReport', // trigger html delete method for deletion of singular report
DeleteReportSuccess: 'DeleteReportSuccess', // for UI updates
```

For both frontend and backend, files are labelled and organized by their role within the overall app and implementation, making components easy to locate and reference.





## Frontend Code: Crowding Score Reporting

```
attach event listeners for successful crowding score report
const reportSubmitButton = document.getElementById("report-submit");
reportSubmitButton.addEventListener("click", (event) => {
  event.stopPropagation(); // prevent any bubble up (should be find regardless, report

const reportSubject = document.getElementById("report-location-name").innerText // "L
  .split(" Floor ") // should be ["LocationName"] or ["LocationName", "FloorName"]

// check if no selection made
if (localStorage.getItem("crowding") === null) {
  alert("Please select a crowding score level.");
  return;
}

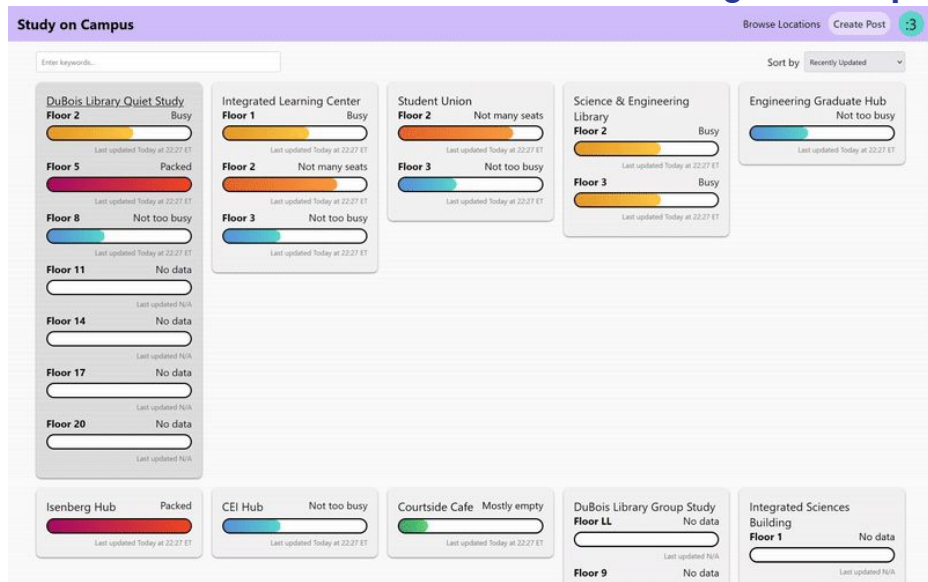
// construct data for add report
const data = {
  location: reportSubject[0],
  score: Number(localStorage.getItem("crowding")), // convert to number
}

if (reportSubject.length === 2) { // multi floor
  data.floor = reportSubject[1]; // add floor name
}

// one example connection to backend for LocationBrowsingComponent
hub.publish(Events.AddReport, data); // alert event hub -> trigger backend action
}

// attach event listeners for successful crowding score report
hub.subscribe(Events.AddReportSuccess, async () => {
  alert("Your report has been saved. Thank you.");

  this.#minimizeReportModal(); // close modal after successful report submission
  hub.publish(Events.MinimizeLocationCard); // close location card
  this.#locationsData = await this.#getLocations(); // update location data attribute
  await this.#renderCards(); // You, 1 second ago • Uncommitted changes
});
```



➤ **Challenges:** Even more so than with the last milestone, a major challenge was navigating the asynchronous functions. In the strictly frontend implementation of this page, only the location data aspect of the render function was asynchronous. With this milestone, most of the component rendering involved some asynchronous data retrieval, which required significant restructuring and debugging.

- **Description:** This segment of code corresponds with the crowding score reporting modal in LocationBrowsingComponent.js, specifically the **reporting feature frontend**. Clicking any of the choices in the modal stores the selection locally. Closing the modal erases this selection. The first event listener in the screenshot triggers when the user clicks the purple Submit button. If the user made a selection, the locally stored choice and location are packaged in an object and published with the AddReport event.
- **UI Impact:** When the backend components successfully add a user report (indicated by the **AddReportSuccess** event), a confirmation alert appears and the report modal is minimized. The page data is updated and the location cards are re-rendered.
- **Backend Integration:** The AddReport event is published when the Submit button is pressed, and calls the **POST /report**, **GET /locations**, and **PUT /location** endpoints which update the corresponding data sources.

## Backend Code: Crowding Score Reporting

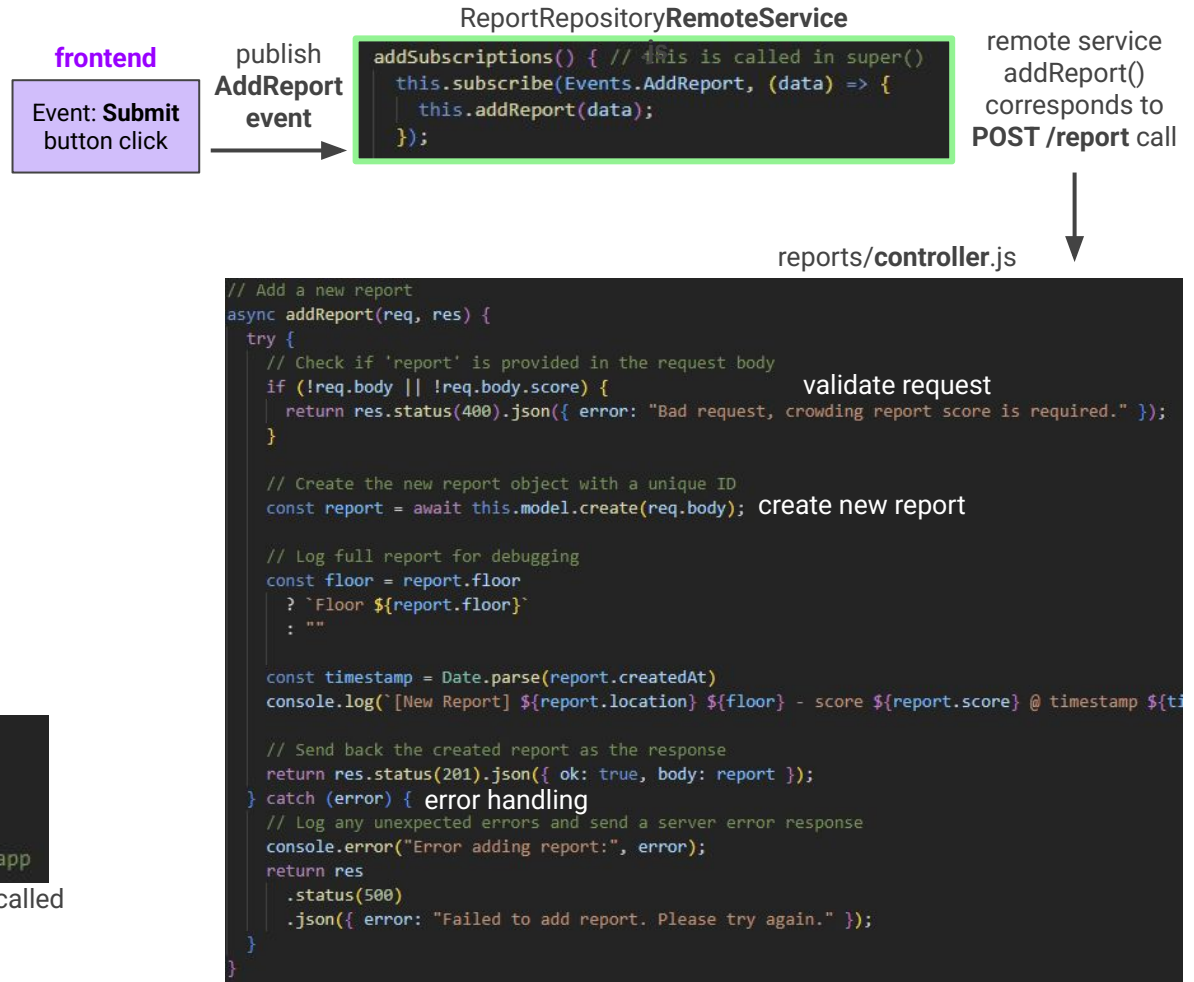
➤ **Description:** This segment of code highlights **what happens after the submit button is pressed on the report modal**. This segment **has not changed** since the last milestone. This is one aspect of the backend that connects with the frontend code in the previous slide (LocationBrowsingComponent.js).

➤ **UI Impact and Frontend Integration:** The **POST /report**, **GET /locations**, and **PUT /location** endpoints are called to provide the necessary data to update the Location Browsing page components. Upon successful storage, the **AddReportSuccess** event is published and the frontend content

```
// set up routes by using imported ReportRoutes
app.use("/", ReportRoutes); // mount on app

// set up routes for imported LocationRoutes
app.use("/locations", LocationRoutes); // mount on app
```

routes are set up when **node backend/src/server.js** is called



## Backend Code: Crowding Score Reporting (continued)

```
res.sendFile(this.jsonPath, (error) => {
  if (error) throw new Error("issue with sendFile for locations JSON: ${error}");
});
} catch (error) {
  console.log("Issue fetching locations JSON:", error)
  return res
    .status(500)
    .json({ error: "Failed to get locations JSON." });
}
```

error handling

try to get **location.json** data file, i.e., **GET /locations** HTTP method.  
this data is about the same as MockLocations.js from the solely  
UI implementation, but as a JSON file

```
// Update locations JSON
async updateLocations(req, res) {
  try {
    // check req body
    if (!req.body) {throw new Error("issue with body of request for updating locations JSON")}

    // write updated data to file
    await fs.writeFile(this.jsonPath, JSON.stringify(req.body), (error) => { // JSON stringify
      if (error) {throw new Error("issue writing to locations JSON: ${error}");
    });
    // ref: https://www.geeksforgeeks.org/node-js-fs-writefile-sync-method/
    // ref for callback cb error: https://stackoverflow.com/a/72432465

    return res.status(200).json({ok: true}); // successful update
  } catch (error) {
    console.log("Error updating locations JSON: ${error}")
    return res
      .status(500)
      .json({ error: "Failed to update locations JSON." });
  }
}
```

error handling

try to write update information to **location.json** data file, i.e., **PUT /locations/update** HTTP method

```
// Create the new report object with a unique ID
const report = await this.model.create(req.body.report);
```

reports/controller.js

```
async create(report) {
  report.id = _ReportModel.report_id++; // assign report id (and increment model id var)
  if (!this.reports.includes(report)) { // prevent duplicates
    this.reports.push(report); // add report to in memory "storage"

    // add report id to corresponding location 'reports' array (ref: https://www.geeksforgeeks.org/node-js-fetch-api-get-method-for-fetching-data-from-server/)
    const GETResponse = await fetch("http://localhost:3000/locations"); // GET method for locations
    if (!GETResponse.ok) {return new Error("Failed to fetch locations.json");}

    const jsonData = await GETResponse.json(); // location JSON data
    // doesn't need to be parsed again, ref: https://stackoverflow.com/a/77786334

    const locationInfo = report.location; // this should be an array: ["LocationName"] or ["LocationName", "Floor"]
    const location = jsonData.find(location => location.name === locationInfo[0]); // get location
    switch (locationInfo.length) {
      case 1: // single floor
        location.reports.push(report.id); // add report id to location's 'reports' array
        break;
      case 2: // multi floor
        const floor = location.floors.find(floor => floor.name === locationInfo[1]); // get floor
        floor.reports.push(report.id); // add report id to floor's 'reports' array
        break;
      default: // else, something wrong with location data array
        throw new Error("Issue with location data. Should be an array of length 1 or 2");
    }

    const PUTResponse = await fetch("http://localhost:3000/locations/update", {
      method: "PUT",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(jsonData),
    });
    if (!PUTResponse.ok) {throw new Error("Failed to update locations.json");}

    console.log("Report created:", report); // print to console for confirmation
    return report;
  }
}
```

add new report id to  
corresponding location's  
reports array in JSON data



## Backend Code: Crowding Score Reporting (continued)

locations/location.js

```
// define location model
export const Location = sequelize.define("Location", {
  id: { // unique universal identifier
    type: DataTypes.UUID,
    defaultValue: DataTypes.UUIDV4
  },

  name: {
    type: DataTypes.STRING,
    allowNull: false,
    primaryKey: true
  },

  address: {
    type: DataTypes.STRING,
    allowNull: false
  },

  type: {
    type: DataTypes.STRING,
    allowNull: false
  },

  reports: { // null if multi-floor, ref
    type: DataTypes.JSON,
    allowNull: true
  },

  floors: { // not null if multi-floor (
    type: DataTypes.JSON,
    allowNull: true
  }
});

// define report model
export const Report = sequelize.define("Report", {
  id: { // unique universal identifier
    type: DataTypes.UUID
  },

  location: { // name of corresponding location/floor
    type: DataTypes.STRING,
    allowNull: false
  },

  floor: { // floor name, if applicable
    type: DataTypes.STRING,
    // allowNull defaults to true
  },

  score: { // crowding report score
    type: DataTypes.NUMBER,
    allowNull: false
  },

  // timestamp/report submission is generated by
  // using Date.parse to convert createdAt value
});
```

defined using **Sequelize**, each **Location** is defined with one or more **reports** arrays, which hold the ids of reports corresponding to a location and/or floor; each **Report** is defined with a location name, optional floor name, and crowding score

**SQLite** doesn't support arrays, so any array used is stringified and stored as a JSON

reports/report.js

```
class _SQLiteReportModel {
  async create(report) {
    console.log("Received report to create:", report)
    // update corresponding location
    const location = await _SQLiteLocationModel.read(report.location); // get location
    // update corresponding location (from SQLite database) with report id
    if (location.type === "Single-Floor") {
      const reportsArr = JSON.parse(location.reports); // get reports array
      reportsArr.unshift(newReport.id); // add report id to BEGINNING of report
      location.update({ reports: JSON.stringify(reportsArr) }); // add report id
    } else if (location.type === "Multi-Floor") {
      const floors = JSON.parse(location.floors); // get floors array of objects
      const index = floors.findIndex(floor => floor.name === newReport.floor); //
      floors[index].reports.unshift(newReport.id); // add id to BEGINNING of floor
      location.update({ floors: JSON.stringify(floors) }); // update location floors
    }
    Reports, defined with score and location (and optional floor), are created and stored in SQLite database
    return newReport;
  }
}
```

```
// Create the new report object with a unique ID
const report = await this.model.create(req.body.report);
```

reports/report.js

id	location	floor	score
Filter...	Filter...	Filter...	Filter...
1	Courtside Cafe	NULL	1
2	CEI Hub	NULL	2
3	Isenberg Hub	NULL	5
4	Engineering Graduate Hub	NULL	2
5	Science & Engineering Library	2	3
6	Science & Engineering Library	3	2
7	Science & Engineering Library	3	4

## Backend Code: Crowding Score Reporting (continued)

```

async addReport(reportData) {
  const response = await fetch("/report", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(reportData),
  });

  if (!response.ok) {
    throw new Error(`Bad response (${response.status}): ${response.statusText}`);
  }

  const data = await response.json();
  EventHub.getInstance().publish(Events.AddReportSuccess, data);
  return data;
}

```

with **successful** report storage, the **AddReportSuccess** event is published with the new report

```

// attach event listeners for successful crowding score
hub.subscribe(Events.AddReportSuccess, async () => {
  alert("Your report has been saved. Thank you.");

  this.#hideElement(this.#reportModal); // close modal
  hub.publish(Events.MinimizeLocationCard); // close modal
  this.#locationsData = await this.#getLocations(); // get locations
  await this.#renderCards();
});

```

this connects back to the **frontend**, where the location cards are re-rendered with updated location data

```

// Create the new report object with a unique ID
const report = await this.model.create(req.body.report);

```

```

async create(report) {
  console.log("Received report to create:", report)
  // update corresponding location
  const location = await SQLiteLocationModel.read(report.locationId);

  const newReport = await Report.create(report); // build new report

  // // update reports array (depending on building type), .
  if (location.type === "Single-Floor") {
    const reportsArr = JSON.parse(location.reports); // get reports array
    reportsArr.unshift(newReport.id); // add report id to array
    location.update({ reports: JSON.stringify(reportsArr) });
  } else if (location.type === "Multi-Floor") {
    const floors = JSON.parse(location.floors); // get floors array
    const index = floors.findIndex(floor => floor.name === location.floorName);
    floors[index].reports.unshift(newReport.id); // add id to reports array
    location.update({ floors: JSON.stringify(floors) });
  }

  return newReport;
}

```

```

// Send back the created report as the response
return res.status(201).json({ ok: true, body: report });

```

# Challenges and Insights

## Obstacles:

- This milestone was an exercise in asynchronous programming for me. Much of my frontend components needed adjustments due to reliance on backend data; I often expected asynchronous operations to occur in a certain order and needed to make adjustments when they did not.
- I also realized that certain aspects of my frontend are a bit inefficient (e.g., rendering every location card on update, rather than only the updated card; filtering/sorting). I hope to do some additional work to reduce the number of unnecessary backend calls and improve the perceived render speed of the location browsing page.

**Insights:** Again, I truly appreciate the support that comes from working in a team environment, and being able to offer support to my teammates when needed! We've also briefly discussed continuing to work on the project and addressing some of our remaining future tasks after finals have wrapped up.



# Future Improvements and Next Steps

**Location Browsing and Crowding Score Reporting Screen (#12) and Crowding Score Report Integration (#69):** I'd like to further clean up my code on these branches and make sure everything is relatively intuitive. After finals have wrapped up, I'd still like to implement the Google Maps API embed for the expanded post view (#42).

**User Profile Screen (#6):** I'd eventually like to implement the Recent Post "embed" section.

**Navbar Dropdown Menu (#47):** Once user sessions are implemented, I'd like to complete the navigation elements for additional connectivity across the web app.

**Visual Preferences and Interface Customization (#36):** I still think that some of the visual preference functionality could be implemented with IndexedDB. Given time, I'd like to figure out some of the customization options.



Anastasia Isakov



# Work Summary

## Issues worked on in this Milestone:

- [Login and Signup Screen](#) (UI, Server)

## Commits:

- [55ef0bc](#) (5/5): Base for register component.
- [6833cf8](#) (5/6): Implemented register component.
- [1bc4cf2](#), [e5c4f4d](#) (5/6): Additional styling for media responsiveness for register component and slight structural changes.
- [46379f1](#) (5/6): Added username input to signup, starter files for backend (routes, controller).
- [b0d7d05](#) (5/6): Implemented register controller.
- [a2dd1a8](#) (5/6): Base for register remote service.
- [22ac202](#) (5/7): Added media responsiveness to navbar.
- [b435486](#) (5/7): Integration of register service with EventHub, implemented register controller and associated routes.
- [f1c7804](#) (5/7): Bug fixes in register controller, additional subscriptions and event handlers for backend message display.
- [ad49d11](#) (5/7): Removed leftover env configuration.
- [e569800](#) (5/7): Resolved merge conflicts.

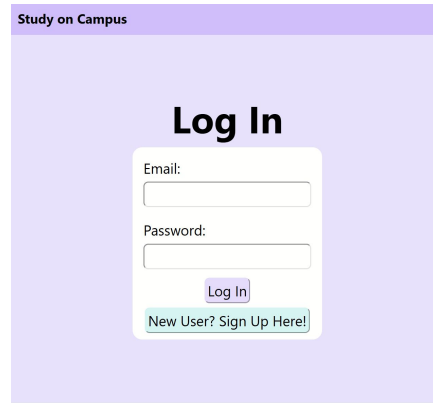
## Pull Requests:

- [#91](#) (5/7): Added additional media queries in navigation bar css for media responsiveness.
- [#93](#) (5/7): Frontend to backend implementation of login/signup feature, allowing new users to make accounts and old users to login with their credentials.

# Feature Demo: Login/Register

Branch: [22-log-in-and-sign-up-screens](#)  
Basic/Advanced Integration (2pts):  
POST /post

**Description:** The user can create a new account which will be stored in the database, or they can sign in with an existing account. The implementation ensures uniqueness among usernames and only one account per given email, while also having the logic that checks for existing accounts to allow users to log in.



Study on Campus

## Log In

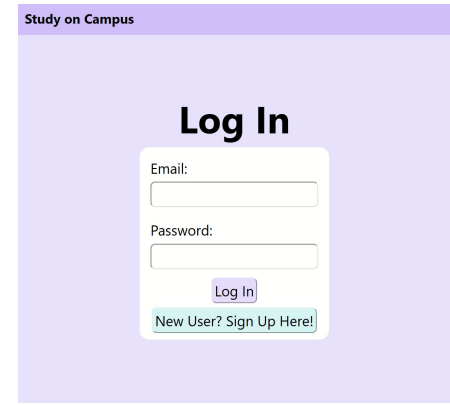
Email:

Password:

[Log In](#)

[New User? Sign Up Here!](#)

Sample Login (with valid and invalid credentials)



Study on Campus

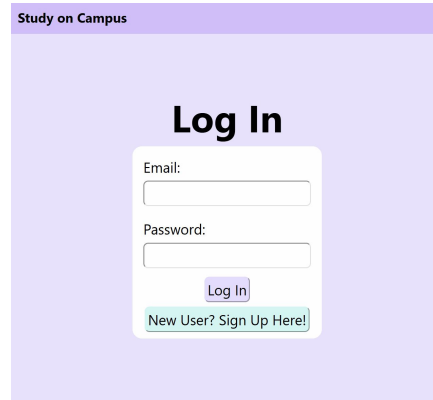
## Log In

Email:

Password:

[Log In](#)

[New User? Sign Up Here!](#)



Study on Campus

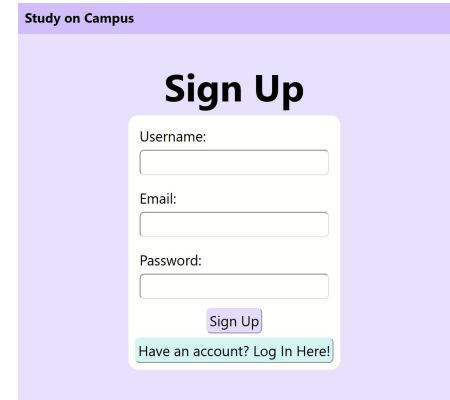
## Log In

Email:

Password:

[Log In](#)

[New User? Sign Up Here!](#)



Study on Campus

## Sign Up

Username:

Email:

Password:

[Sign Up](#)

[Have an account? Log In Here!](#)

Sample Sign Up (with valid and invalid inputs)

# Frontend Code

## Login/Signup Frontend

The frontend code has been updated to render the UI through RegisterComponent. The component handles various validations and feedback for the user as they input their information, such as specifying email and password format, as well as not allowing the user to submit until the inputs are all present with the correct format. The component also toggles between two displays - the login and signup. The CSS for the component has been adjusted to be media responsive.

The frontend integrates with the backend through RegisterRemoteService and RegisterRoutes. On submit, the page publishes either Login or Signup event instance to EventHub, which sends a POST request to the respective post endpoint. Depending on the validity of the credentials, the browser will either redirect to PostBrowsing on success, or report invalid credentials to the user on failure.

### Challenges:

- **Failure Feedback:** Figuring out how to bubble up the different reasons for invalid login/signup through EventHub events and the route responses, as well as outputting those messages to the user.

```
async login(data) {
  const response = await fetch("http://localhost:3000/users/login", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify(data)
  });
  const responseData = await response.json();
  console.log("Response Data:", responseData);
  console.log("Error Status:", responseData.status);
  console.log(responseData.message);
  if (!response.ok) {
    this.publish(Events.LoginFailure, { message: responseData.message || "Invalid credentials." });
  }
  else {
    this.publish(Events.LoginSuccess);
  }
  return responseData;
}

async signup(data) {
  const response = await fetch("http://localhost:3000/users/signup", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify(data)
  });
  const responseData = await response.json();
  if (!response.ok) {
    this.publish(Events.SignupFailure, { message: responseData.message || "Signup failed." });
  }
  else {
    this.publish(Events.SignupSuccess);
  }
  return responseData;
}
```

Top: RegisterRemoteService.js  
Top Right/Direct Right: RegisterController.js  
Bottom: UserRoutes.js (Integration with Backend)

```
this.router.post("/login", async (req, res) => {
  await RegisterController.login(req, res);
})

this.router.post("/signup", async (req, res) => {
  await RegisterController.signup(req, res);
})
```

```
#toggleFormState() {
  e.preventDefault();
  this.#isLoginMode = !this.#isLoginMode;
  this.#updateForm();
}

#showErrorMessage(inputElement, messageClass, box) {
  const messageElement = inputElement.parentNode.querySelector(`.${messageClass}`);
  if (box == 1) {
    messageElement.style.display = this.#validateEmail(inputElement.value) ? "none" : "block";
  } else if (box == 2) {
    messageElement.style.display = inputElement.value.length >= 8 ? "none" : "block";
  }
  else {
    messageElement.style.display = inputElement.value == "" ? "block" : "none";
  }
}

#validateEmail(email) {
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]{3,}$/;
  return emailRegex.test(email);
}
```

```
#attachEventListeners() {
  this.#attachToggleFormListener();
  this.#attachEmailListener();
  this.#attachPasswordListener();
  if (!this.#isLoginMode) {
    this.#attachUserIdListener();
  }
  this.#attachFormSubmitListener();

  this.#service.subscribe(Events.LoginSuccess, (responseData) => {
    this.#hideFailureMessage();
    window.location.href = "/pages/PostBrowsing/index.html"
  });

  this.#service.subscribe(Events.SignupSuccess, (responseData) => {
    this.#hideFailureMessage();
    window.location.href = "/pages/PostBrowsing/index.html"
  });

  this.#service.subscribe(Events.LoginFailure, (responseData) => {
    this.#handleFailure(responseData);
  });

  this.#service.subscribe(Events.SignupFailure, (responseData) => {
    this.#handleFailure(responseData);
  });
}

#handleFailure(responseData) {
  const errorMessageElement = this.#container.querySelector(".enter-error-message");
  if (errorMessageElement) {
    errorMessageElement.style.display = "block";
    errorMessageElement.textContent = responseData?.message || "Login or Signup failed.";
  }
}
```

# Code Structure

## Login/Signup Frontend and Backend Integration

### Frontend

frontend/src/components/RegisterComponent

- RegisterComponent.css
- RegisterComponent.js

frontend/src/pages/Register

- index.html
- script.js

frontend/src/services

- RegisterRemoteService.js

### Backend

backend/src/controller

- RegisterController.js

frontend/src/model

- ModelFactory.js
- SQLiteUserModel.js

frontend/src/routes

- UserRoutes.js

Current workflow clearly separates backend and frontend, with services and routes acting to connect the two together. The Register page with index.html and script.js calls on the RegisterComponent. The component renders the UI as well as instantiates the RegisterRemoteService. The service uses UserRoutes to interact with the backend on form submission, where the information is passed through the RegisterController and sent to the database. The database response then bubbles back up to the controller, which sends a response and message to the service through the route, and depending on the response in the service the component either sends the user to the PostBrowsing page or displays the error message.

# Backend Code

## Login/Signup Backend

The feature uses parts of the User Backend implemented as part of Milestone #6. The feature uses the login and signup endpoints in UserRoutes, which both get Post requests from RegisterRemoteService. The routes call RegisterController, which implements numerous status messages for both login and signup, depending on the validity of the information (is the username taken, does the account with the email exist, etc.). These messages are then sent to the service to display to user if necessary.

On sign up, the backend uses bcrypt to hash the password before storing the userId, email, and hashed password in the SQLiteUserModel from the previous Milestone. On login, the backend searches through the model for the matching email and password, returning accordingly.

### Challenges:

- **Failure Feedback:** Adjusting the code to bubble up the different reasons for invalid login/signup through route responses.
- **Database Interaction:** Figuring out the userId primary key and adjusting the frontend to account for its creation, comparing hashed passwords.

```
async create(user) {
  const userr = {
    userId: user.userId,
    email: user.email,
    password: user.password,
    name: user.name || user.userId,
  };
  const newUser = await User.create(userr);
  return await this.read({ userId: newUser.userId });
}

async read(user = null) {
  if (user) {
    if (user.userId) {
      const userr = await User.findOne({
        where: {userId: user.userId},
        include: {model: Fact, as: "facts"},
      });
      if (userr) return this.transformUser(userr);
    }
    if (user.email) {
      const userr = await User.findOne({
        where: {email: user.email},
        include: {model: Fact, as: "facts"},
      });
      if (userr) return this.transformUser(userr);
    }
    return null;
  }
  const users = await User.findAll({ include: { model: Fact, as: "facts" } });
  return users.map(this.transformUser);
}

async transformUser(userr) {
  userr.profileContent = userr.profileContent || {};
  userr.profileContent.about = {userr.facts || []}.map(fact => ({
    factName: fact.factName,
    factAnswer: fact.factAnswer,
  }));
  delete userr.facts;
  return userr;
}
```

```
this.router.post("/login", async (req, res) => {
  await RegisterController.login(req, res);
})

this.router.post("/signup", async (req, res) => {
  await RegisterController.signup(req, res);
})
```

```
async signup(req, res) {
  try {
    const { email, password } = req.body;

    if (!email || !password) {
      return res.status(400).json({ status: 400, message: "All fields are required." });
    }

    const existingUserId = await this.model.read({ userId });
    if (existingUserId) {
      return res.status(400).json({ status: 400, message: "Username already taken." });
    }

    const existingUserByEmail = await this.model.read({ email });
    if (existingUserByEmail) {
      return res.status(400).json({ status: 400, message: "Account with email already exists." });
    }

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

    const newUser = await this.model.create({
      userId: email,
      email: email,
      password: hashedPassword
    });

    return res.status(200).json({
      status: 200,
      message: "Signup successful",
      user: { userId: newUser.userId }
    });
  } catch (error) {
    console.error("Signup error:", error);
    return res.status(500).json({ status: 500, message: "Signup failed. Please try again." });
  }
}

async login(req, res) {
  try {
    const { email, password } = req.body;

    if (!email || !password) {
      return res.status(400).json({ status: 400, message: "Email and password are required." });
    }

    const user = await this.model.read({email});

    if (!user) {
      return res.status(401).json({ status: 401, message: "Invalid credentials." });
    }

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

    if (!hashedPassword) {
      return res.status(401).json({ status: 401, message: "Invalid credentials." });
    }


    return res.status(200).json({ status: 200, message: "Login successful" });
  } catch (error) {
    console.error("Login error:", error);
    return res.status(500).json({ status: 500, message: "Login failed. Please try again." });
  }
}
```

Top: RegisterController.js  
Top Left: SQLiteUserModel.js  
Left: UserRoutes.js (Integration with Frontend)

# Challenges and Insights

My work on this Milestone focused on entirely implementing the registration and login feature, from the frontend to the backend. One of the major challenges that came from this was packaging the payload in a request to be passed to the backend, to process that request with the necessary actions and respond with the appropriate status and message, and then to pass that status and message back to the frontend so that it could display feedback to the user if the attempts to register or login were unsuccessful. Figuring out how to implement this full-circle path was difficult and required a lot of debugging to figure out where exactly the flow was getting stuck and why.

My team was very helpful in helping me figure this out , since we were working off of the same models in the backend and were able to assist each other as a result. I really appreciated Julia's work on adjusting parts of the User Model from last Milestone, as it's functionality was a necessary part of my implementation for this Milestone.



# Future Improvements and Next Steps

[#32](#) **Forgot Password Screen(s)**: Alongside Google Authentication, implement a password reset screen that sends the user an email that gives them a link to reset their password.

[#77](#) **Post Drafts**: Try implementing a feature that allows the user to save and access drafts of posts - posts they have written some title, date, or location to, but have not submitted. These drafts would be displayed on the left, and the user would be able to click on them to pull up the draft, which they can edit, submit, or delete.





Julia Farber



# Work Summary

## Issues:

- [Time Keeper](#) (Team)
- [Post Browsing and Search Screen](#) (UI)
- [Post Viewing Screen](#) (UI)
- [Post Data Structure](#) (Data)
- [UI standardization](#) (Group assigned, UI).
- [Post Browsing Related Cleanup](#) (Data, UI, Frontend)
- [Backend organization](#) (Organization, backend).
- Also other future tasks that were either resolved or later mentioned in slides.

**Commits:** To keep pages to a minimum, please click on them to learn about what they specifically update. All work is done on the same branch, and put together on the same pull request below. All previous milestone commits are on the slideshows for the previous milestones.

- [6cdf9c](#), [5e416ad](#), [53c0ba5](#), [cfcd45](#), [8b65de6](#), [d0e9331](#), [674d7c4](#), [0d3f69b](#), [c77a29a](#), [0e70a3a](#), [205ff64](#).

## Pull Requests:

- [#92](#) – Made sure that post browsing and viewing now use the eventhub rather than the post remote service directly when events occur. Added proper event listeners for receiving post data. Also then later did the same for getting user data. Updated an endpoint for users for easy accessibility. Created posts DB and gently adjusted how users are accessed in user DB with sequelize/SQLite. Removed conflicting media CSS for navigation bar. Created User Remote Service for calling users. Added event listeners for the user data retrieval and properly fixed rendering issues with updated database access calls (user data from a user call, post data from a post call). Added tag color assignment function and added comment color functionality as well. Fixed all timeStamp and starting time issues as well, with proper formatting and storage in the DB. Posts created are shown on post browsing, and all posts on post browsing are viewable with their proper associated data and comments. See commit info for more details if necessary. Removed terminal logging of user and posts database initializations. Updated post creation to support properly post creation and fixing time issues.

# Work Summary Continued

**Commits:** All commits are those that are under these pull requests. Please, for all of them, go look at previous milestone slide submissions for more detail. Also, they are all under these pull requests, please look at them in detail.

## Pull Requests of Previous Milestones:

- [#27](#) – Initial HTML and CSS design of post-viewing-screen
- [#28](#) – Initial HTML and CSS design of post-browsing-screen
- [#57](#) – Created initial front-end of post browsing and post viewing.
- [#71](#) – Defining post data structure in typescript code.
- [#78](#) – Created post backend with in memory post model. Created backend routing, controller, calls etc for retrieving data. More details in description.
- [#84](#) – Described thoroughly on previous slide.

## Assigned tasks:

- Besides timekeeper, I had most responsibilities surrounding posts. Post browsing and search functionality, post viewing were all under my responsibility. I also am entirely responsible for the posts backend and database (SQLite). Anastasia worked on post creation, but in milestone 7 edited it to ensure it worked perfectly with the other post related pages. All non-closed issues are issues that were not achieved in this course, are potential stretch goals for personal work in the future (authentication related to post deletion/editing and also upgrading the UI for post viewing, etc).
- IN MILESTONE 7, I created the posts database in sequelize SQL and I properly made sure post viewing and browsing had proper eventhub subscriptions and publications. I also fixed any issues relating to fetching user data by creating a remote service for users and updating the get user endpoint data, and also touching up post creation to properly translate into the database and rendering of post browsing.
- <https://github.com/eelston/326Group11/issues/79> & <https://github.com/eelston/326Group11/issues/82> left open.

For milestone 7, I created a fully functional posts database through SQLite/sequelize.

**My Features (details on next slide):**

- A Post Browsing Screen that fetches from the database (view all posts as a feed).
- A search function that filters the backend data to show relevant posts.
- A feature to view a specific post (gets the post data from backend), and be able to comment on the post, delete the post, or delete a specific comment. Also fetches user data from the user database. All done through event listener.

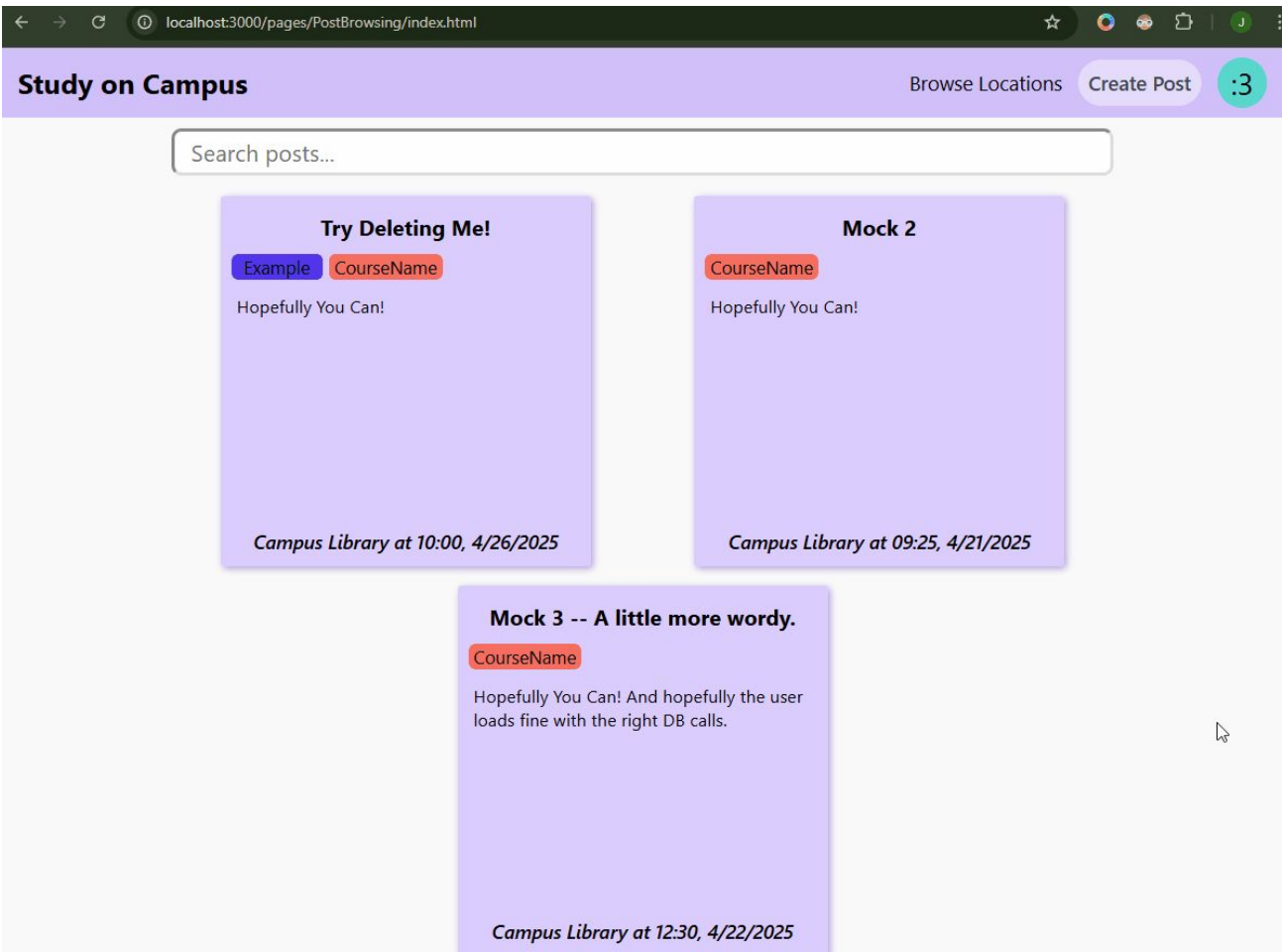
**All three features, were implemented on branches:**

posts-backend-and-post-browsing-viewing  
(front-end and back-end updates most recent)

**Also branches used for previous milestones (Prior to 6 and 7):**

indexedDB-load-all-posts,  
10-post-browsing-and-search-screen, and  
26-post-viewing-screen

# Features Demo



# Features Details

For this milestone, I worked on the SQLite and sequelize for Posts. I also collaborated with Anastasia to update an endpoint for getting a user's data and created a UserRepositoryRemoteService with the updated server fetch call in the user routing.

**Endpoints:** /posts is all posts, while /posts/:id is for a singular post.

- **GET /posts** endpoint , which accepts a search query, as this function returns all posts in the database or ones that fit the criteria of the search. Returns all posts (that meet the criteria of the search, if any search). Search query is optional.
- **GET /posts/:id** endpoint, which accepts a postId (integer) and returns a post object, which contains all the post information and the userId for the appropriate user call. postId is required.
- **GET /users/:userId** endpoint, which accepts a userId and returns a user object. userId is required.
- **POST /posts** endpoint, which creates a new post. Must pass in a post to add to the database. Creates all associated tags and comments. Created by me.
- **DELETE /posts/:id** endpoint, which will delete a post from the database, destroying all comments and tags associated with it.
- **PATCH /posts/:id** endpoint, which will update a post when a new comment is written. Requires an id and must be an existing post. Updates all comments (and tags, as in future an edit post functionality should exist.)
- **DELETE /posts** endpoint, which deletes all posts. This was not used in this application, but exists and functional. Doesn't take any parameters. Note that all other named endpoints except this one are used in the application.

**Feature: View Post Browsing Feed and Search/Filter the Posts**

**Description:** The user can view all non-expired posts and are able to search for posts by keyword, which will search the title, description (even if the preview is cut off), location, and tags. When an update occurs, the user will be updated with the newest posts on click/refresh of the feed. Uses GET /posts endpoint for loading all posts initially and GET /posts for filtering as well in a separate manner. FULLY COMPLETE AND FUNCTIONAL.

**“MEGA” Feature: View A Specific Post and Interact**

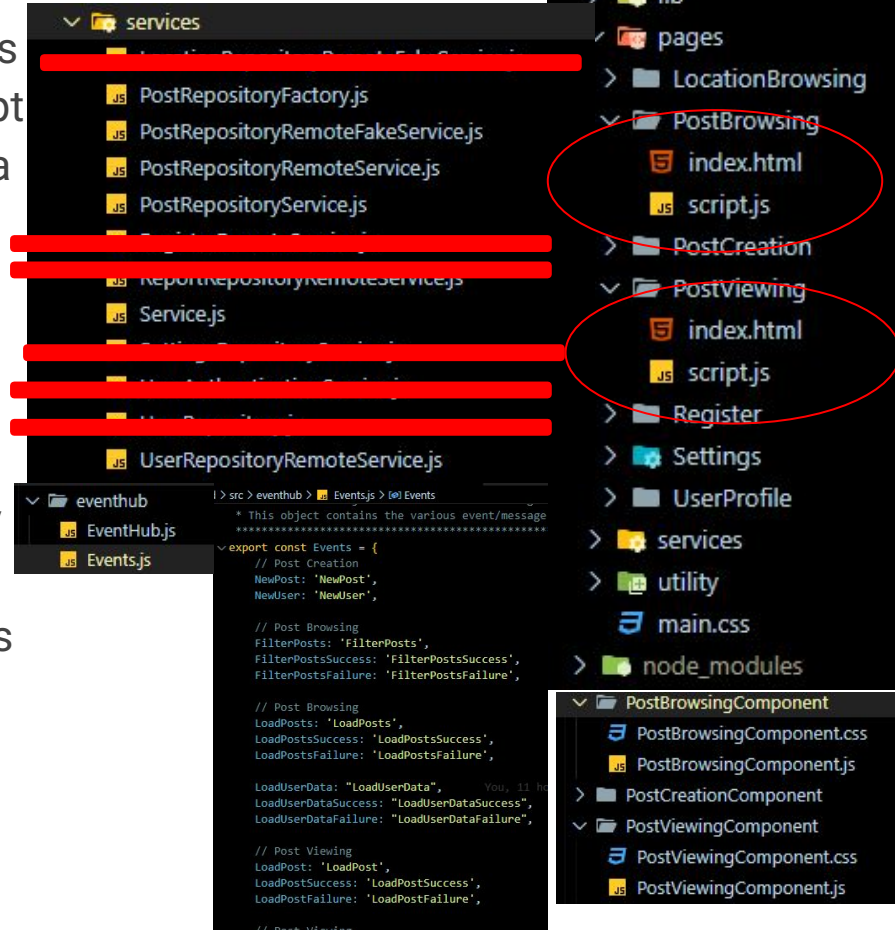
**Description:** The user can view a non-expired post and are able to interact with the post, and see post and commenter user details. User can delete (their) comments and posts, and leave a comment. This function uses GET /posts/:id for post data, GET /users/:userId for user Data, DELETE /posts/:id for post deletion, and PATCH /posts/:id for new comments and deletion of comments. FULLY COMPLETE AND FUNCTIONAL. This feature is many featured put together into one, for one component.

Assisted Anastasia in post creation by creating POST /posts endpoint.

# Code Structure & Organization (Front End)

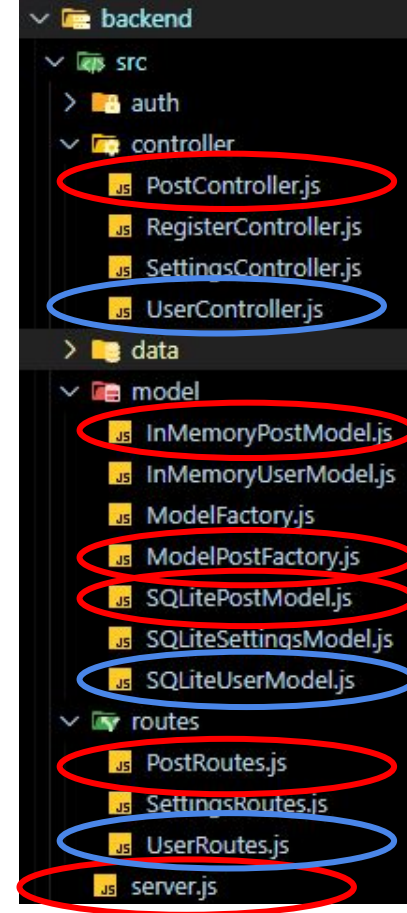
Not crossed out: my work (related to my functionalities)

The front end is organized very well. All components are in the components folder, with css and javascript for rendering and helper functions to properly load a page. Each page has its own corresponding component (mine are Post Viewing and Post Browsing). All events and event hub are in the eventhub folder. Each page has its own script and html and the script acts like a main.js for that page. The services I created are all the ones related to my functions, which are all the Post services and the User services. The structure of these heavily follows the examples and structure taught in class, as it uses the event hub subscriptions and Events, but also makes the proper fetch calls to the backend, connecting frontend and backend.



# Code Structure & Organization (Back End)

Blue is files I contributed to, but red are files that I did myself. The backend follows the course structure, although we have some folders specific to certain components, like locations and reports (not pictured here). However, looking at the folders labelled controller, model, routes, and the server.js itself, it adheres to the class structure. The other folders not pictured follow the same structure, but within those specific related folders themselves. However, that does not concern my work, as my work strictly adheres to what the course did. I created the In memory model for posts and the SQLite posts, and created a factory. The routes adhere to a similar structure as the course, except for having “posts” and “post” endpoints, I did posts/:id for ease in the database to have a singular endpoint. I changed one endpoint access for getUser to be similar for ease of access when fetching. All these routes are placed in server.js, which communicates the backend with the front end when it fetches from the server, which strictly uses the route to call the router functions and then finally access and do the operations in the model/database.





# Front-end Implementation

The `#getUserData` function from `PostViewingComponent.js` has been really useful, especially in this milestone. As this milestone clearly defined the data structures for Users and Posts, there was a need to reduce data duplication between the two databases. So, instead of passing in user data into the post, I decided to instead call the user data into the post viewing component to get the proper data on users. This distributed responsibility between the posts end point and the user endpoint, as the post viewing has a lot of info on the post and the original poster and commenters. With the eventhub events, and the remote service for user and remote service of posts, this demonstrates great understanding of databases and makes the application whole. With the proper eventhub calls to get the data, it shows that the separation of data was a good move and works because the proper data is called. It is because posts store the `userId`, and it makes it easier for the data to get called. So the data from user data goes to the user controller, with the modified endpoint `users/:userId`, to the routes and into the server, which is then retrieved by the `UserRemoteService` when the eventhub gets a call to retrieve user Data. Then, the eventhub allows for data to successfully be called and then the subscribed function is able to retrieve the data. In other words, the `postviewingcomponent` uses multiple remote services, which effectively uses the databases to prevent data duplication

I struggled for several milestones to understand fully how the eventhub worked. So, starting this milestone, I was confused with the reduction of data duplication – should I be storing services in the components? I realized I should not be. This milestone with the database truly nailed it down, as I understood I cannot have multiple services in the class stored. An eventhub truly helps reduce the amount of code and make it more de-coupled. However, this was challenging to figure out at first and took talking to my team about the eventhub until it clicked.

```
async loadPost(postId) {
  const response = await fetch(`http://localhost:3000/v1/posts/${encodeURIComponent(postId)}`);
  console.log("Response status: " + response.status);
  if (!response.ok) {
    this.publish(Events.LoadPostFailure);
    throw new Error("Failed to fetch post with postId ${postId}");
  }
  const data = await response.json();
  this.publish(Events.LoadPostSuccess, data);
  return data;
}
```

**PostRepositoryRemoteService.js**

```
#getUserData(userId) {
  const hub = EventHub.getInstance();
  return new Promise((resolve, reject) => {
    const returnData = (user) => {
      resolve(user);
    }
    const failure = (error) => {
      reject(new Error("Failed to fetch user data"));
    }
    hub.subscribe(Events.LoadUserDataSuccess, returnData);
    hub.subscribe(Events.LoadUserDataFailure, failure)
    hub.publish(Events.LoadUserData, userId);
  })
}

async render(postId) {
  console.log("loading a post with id: " + postId)
  const hub = EventHub.getInstance();
  hub.subscribe(Events.LoadPostSuccess, async (post) => {
    this.#post = post;
    document.title = `${this.#post.title} | Study on Campus`;
    console.log(this.#post);

    this.#container = document.createElement('div');
    this.#container.setAttribute('class', 'post-container');
    document.querySelector('main').appendChild(this.#container);
    await this.#renderPost();
    this.#comments = document.createElement('div');
    this.#comments.setAttribute('class', 'comment-container');
    document.querySelector('main').appendChild(this.#comments);
    this.#renderComments();
    this.#goBackListener();
    this.#postSettingsDropdownListener();
    this.#deleteButtonListener();
  });
  hub.publish(Events.LoadPost, postId);
}
```

```
async loadUser(userId) {
  const response = await fetch(`http://localhost:3000/users/${userId}`);
  console.log("Response status: " + response.status);
  if (!response.ok) {
    this.publish(Events.LoadUserDataFailure);
    throw new Error("Failed to fetch user with userId ${userId}");
  }
  const data = await response.json();
  this.publish(Events.LoadUserDataSuccess, data);
  return data;
}
```

**UserRepositoryRemoteService.js**

```
addSubscriptions() {
  this.subscribe(Events.LoadUserData, (userId) => {
    this.loadUser(userId);
  });
}
```

# Backend Implementation

The key piece of code, for this milestone is the SQLite relations set up and the update function in the SQLite model for posts. These two pieces of code enable the proper setup of the data for posts. Since posts can have many comments, we know that it is a many to one relationship. The same for tags and posts, with a many to one relationship as well. These relations shine through most in the update function in the sqlite for posts file, as I update the post data independently and destroy all the comments and tags, to then simply update them. This is done as in the future maybe one day, posts will be editable and tags will be as well, so it will be necessary to destroy and rebuild those related databases, to avoid outdated information from staying in the database. The SQLite database model connects through the post controller, where the controller calls the model functions by doing CRUD operations (update). The post router calls the controller functions and these set up the connection to the server. Either way, the relation setup with hasMany, BelongsTo have truly made everything possible for the database, with no need of an in memory model. This makes the application truly full-stack. The routes connect to the server using the /v1/posts route and data is accessible to the frontend in the remote services.

The challenge was to figure out how to have tags, which at the time seemed like an attribute, to work with the post, but upon further thinking and brainstorming amongst the team, we decided to have a database that has a foreign key that uses postId. It was better than creating a json with a bunch of tags, as the information is easier accessible. However, this might cause issues with speed, which might require tags to be JSON in the future. However, it is most certain that comments needed their own database as it would not be comfortable to access as a JSON. Updates should be easier to make, which this does. It was challenging to also figure out how to ensure to update the comments and posts "database" section, but I decided it's easier to delete all the tags and comments related to the post, and refresh with new instances, to prevent old data from cluttering.

```
// set up routes for imported PostRoutes
app.use("/v1", PostRoutes);
app.use("/users", UserRoutes);
server.js
```

```
// Updating a post (comments, and potentially editing a post)
this.router.patch("/posts/:id", async (req, res) => {
  await PostController.updatePost(req, res);
});
```

Post router

```
async updatePost(req, res) { // should be fine? but if any errors maybe i gotta debug...
  try {
    if (!req.body) {
      return res.status(400).json({ error: "Post description required " });
    }
    const post = await this.model.update(req.body);
    return res.status(201).json(post);
  } catch (error) {
    console.error("Error in updating post: ", error);
    return res.status(500).json({ error: "Failed to update post. Please try again." });
  }
}
```

postController.js

```
Post.hasMany(Comment, { foreignKey: "postId", as: "postComments" });
Comment.belongsTo(Post, { foreignKey: "postId", as: "post" }); // comment belongs to post
Post.hasMany(Tag, { foreignKey: "postId", as: "tags" }); // post can have many tags
Tag.belongsTo(Post, { foreignKey: "postId", as: "post" });

async update(post) {
  const { postComments = [], tags = [], ...otherPostData } = post;
  const postU = await Post.findPk(post.postId);
  if (!postU) {
    throw new Error(`Post with ID ${post.postId} not found.`);
  }
  await postU.update(otherPostData);
  // Destroy old comments in case any were deleted
  await Comment.destroy({ where: { postId: post.postId } });
  await Tag.destroy({ where: { postId: post.postId } });
  // Rewrite into database the new comments and tags
  await Promise.all(postComments.map(com => {
    Comment.create({ ...com, postId: post.postId })
  }));
  await Promise.all(tags.map(t => Tag.create({ ...t, postId: post.postId })));
  return await this.read(post.postId) // should return updated post.
}
```

SQLitePostModel.js



# Challenges and Insights

There were obstacles and lessons learned during this final milestone. I came to realize that my previous CS 345 coursework has been useful for the database, as I knew exactly what a primary key was, and how to set up the relations between each database. I knew how to debug it, and make it work the way I want to, like when I initially manually decided how comments would be assigned, but I ensured the unique primary keys did not conflict and properly incremented based not on the post, but rather on the number of existing comments. Since, as long the post ID is present, we know which post the comments belong to. I also learned this milestone that the teamwork really comes together like a puzzle piece at the end, when everything is due. In the first few milestones everything felt really separate at first, but this milestone made me realize as I use the users database in post viewing that all the work and data connect together in the backend, and the product of our work really comes through. I also further am reminded that debugging and communicating with the team is essential. Finally, I will say that I finally learned how the eventhub plays into everything, and made correct use of it through subscribing and publishing events, of users and post data! It was challenging for me to grasp until this milestone.

# Future Improvements and Next Steps

- [#82](#) – Adding authentication/permissions to deleting (and maybe editing) posts (users can only delete/edit their own posts/comments, with the exception of an admin/moderator).
- [#79](#) – Post Viewing: improvement of UI, adding edit post/comment functionality for users, and cleaning up code. Cleaning up the code will prevent confusion and increase readability to expand in future, which would benefit from helper methods. Adding edit/comment functionality gives users more control over their posts in case a typo or mistake is made. However, this depends on the restructuring of the UI by adding an edit post/button option and allowing for the interface to change when in editing mode, which was not originally planned for. So this is a future task to improve the post viewing page.

Unchanged since last milestone, as these look further ahead. Our team did not get to authentication, which permissions of post/comment ownership requires and when publishing posts. Additionally, team did not have enough time to restructure the post UI to enable editing, although the backend is entirely set up to enable its capabilities. It would also be ideal to make the code cleaner and more readable for post viewing component.



Ashley Kang

# Work Summary

## Issues:

- [Settings Screen Page](#) (UI)
- [Course Catalog Data Structure](#) (Data)
- [Quality Control](#) (Team)

## Commits:

- [cb04b39](#) (5/6): Removed in-memory storage and implemented SQLite and Sequelize for settings backend
- [5478802](#) (5/6): Edited user profile files to work with course catalog data structure
- [ec44bcd](#) (5/7): Switched settings to multiview layout and organized into components
- [0e60456](#) (5/7): Fixed component rendering and changed settings to accommodate Fact data structure
- [df95c62](#) (5/8): Resolved any SQLite issues with settings

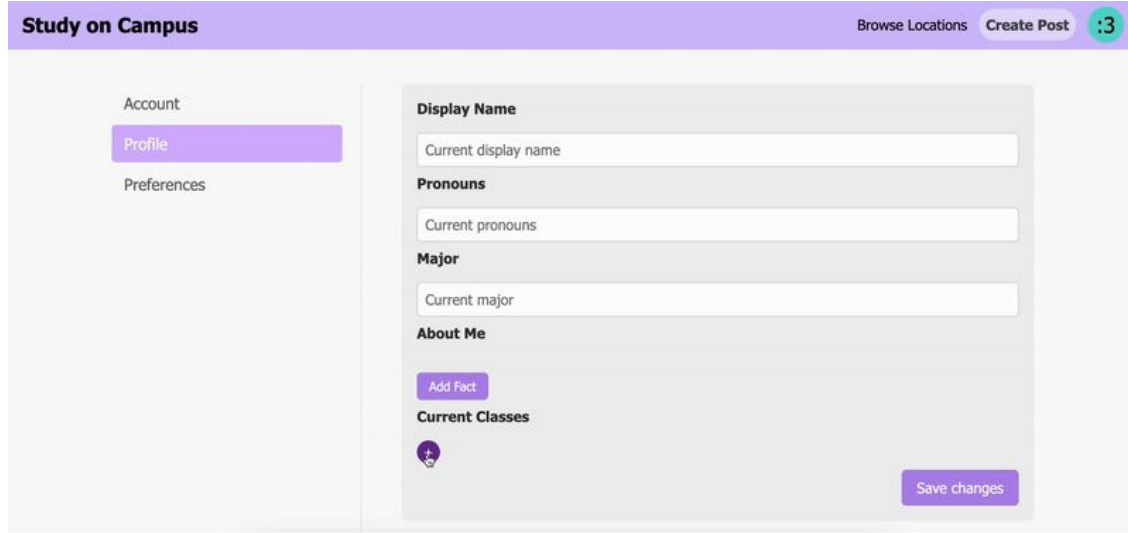
## Pull Requests:

- [#95](#) (5/7): Settings Screen SQLite and Sequelize Backend and Multiview
- [#90](#) (5/7): Edited code to accommodate Course Catalog data structure

# Feature Demo: Adding classes dropdown menu

Branch: [18-settings-screen](#)

In settings, users are able to add classes they are currently taking by using the two-tier dropdown menu. The first dropdown allows users to choose the subject, which then shows the corresponding courses. This feature is fully implemented on both frontend and backend. Once a class is added, users are able to click on the remove button to delete a class from their profile.



The screenshot displays the 'Study on Campus' settings interface. On the left, a sidebar contains 'Account' and 'Profile' (highlighted in purple) with a 'Preferences' link below it. The main content area is titled 'Study on Campus' and includes links for 'Browse Locations' and 'Create Post' (with a notification badge showing '3'). The 'Profile' section contains the following elements:

- Display Name:** A text input field with the placeholder 'Current display name'.
- Pronouns:** A text input field with the placeholder 'Current pronouns'.
- Major:** A text input field with the placeholder 'Current major'.
- About Me:** A large text area for a bio.
- Add Fact:** A purple button to add new facts.
- Current Classes:** A section for managing classes, featuring a small circular icon with a plus sign and a minus sign.
- Save changes:** A purple button at the bottom right of the profile section.

# Code Structure

The settings page follows a component-based architecture that maintains separation between frontend and backend. On the frontend, all major settings components, including account management, profile and class selection, and user preferences, are in directories under `src/components`, each with its own JS and CSS files to enforce modularity and separation of concerns. API communication is centralized in the `SettingsRepositoryService` within `src/services`, following a service-layer pattern and using a custom event system for component interaction. On the backend, a clear MVC-like structure is maintained, with `SettingsRoutes.js` handling routing, `SettingsController.js` managing business logic, and `SQLiteSettingsModel.js` defining data access using Sequelize. Routes, controllers, and models are separated into distinct folders under `src`, and middleware is used for validation and error handling. Critical components like settings management and class selection are clearly labeled and logically placed, such as the two-tier dropdown for courses in the `ProfileSettingsComponent`.

```
▼ frontend / src
  ▼ eventhub
    JS SettingsEvents.js

  JS SettingsRepositoryService.js
    frontend/src/services
```

## Frontend

```
▼ frontend / src
  ▼ components
    ▼ AccountSettingsComponent.js
      # AccountSettingsComponent.js
      JS AccountSettingsComponent.js
    > PreferencesComponent
    > ProfileSettingsComponent
    > SettingsComponent
```

## Backend

```
▼ backend
  ▼ src
    ▼ controller
      JS SettingsController.js
  ▼ backend
    ▼ src
      ▼ model
        JS SQLiteSettingsModel.js
  ▼ backend
    ▼ src
      ▼ routes
        JS SettingsRoutes.js
```

# Frontend Code

The class dropdown menu in the ProfileSettingsComponent implements a two-tier dynamic selection system that allows users to add courses by first selecting a subject, which then populates a filtered list of available course numbers. This functionality is built using vanilla JavaScript, with DOM manipulation to construct the UI and event listeners to update course options in real time. Integration into the broader architecture occurs through the component layer, where it operates under the parent SettingsComponent and communicates via a centralized event system defined in SettingsEvents.js. It also interfaces with the SettingsRepositoryService to persist class data and stay in sync with backend state. Key challenges included dynamically updating dropdowns without duplicating existing selections, which was solved using a filtered course list and responsive UI updates; maintaining consistent state between frontend and backend, achieved via event-driven updates and centralized state management; and handling errors gracefully, with user feedback and error events to alert users of validation or API issues.

frontend/src/components/ProfileSettingsComponent/ProfileSettingsComponent.js

```
export class ProfileSettingsComponent extends BaseComponent {
  #createClassGroup() {

    const dropdownMenu = document.createElement('div');
    dropdownMenu.classList.add('dropdown-menu');

    const subjectGroup = document.createElement('div');
    subjectGroup.classList.add('dropdown-group');

    const subjectLabel = document.createElement('label');
    subjectLabel.setAttribute('for', 'subject-select');
    subjectLabel.textContent = 'Subject: ';

    const subjectSelect = document.createElement('select');
    subjectSelect.id = 'subject-select';
    const defaultSubjectOption = document.createElement('option');
    defaultSubjectOption.value = '';
    defaultSubjectOption.textContent = '-- Select a subject --';
    subjectSelect.appendChild(defaultSubjectOption);
```

```
export class ProfileSettingsComponent extends BaseComponent {
  async updateCourseNumbers(subject) {
    if (!this.settingsService) return;

    const settings = await this.settingsService.getSettings();
    const currentClasses = settings.classes || [];

    const filteredCourses = MockCourses.filter(course => {
      return course.course_subject === subject &&
        !currentClasses.some(cls =>
          cls.subject === subject &&
            cls.number === course.course_number
        );
    });

    filteredCourses.forEach(course => {
      const option = document.createElement('option');
      option.value = course.course_number;
      option.textContent = `${course.course_number} - ${course.course_name}`;
      courseNumberSelect.appendChild(option);
    });
  }
}
```

# Backend Code

The backend implementation for the class dropdown menu uses a clear structure with Sequelize and SQLite to manage data. The Class model defines how class data is stored and includes methods for adding and validating entries. The controller (SettingsController.js) handles incoming requests, calls the model, and sends back responses. Routes in SettingsRoutes.js define the API endpoints and use middleware to check the request format and user ID. Sequelize is used to connect to the SQLite database, define relationships between models, and perform operations like creating and retrieving classes. Input validation and transaction management help keep the data accurate and prevent duplicates. Errors are caught and returned to the frontend in a consistent way, and middleware checks that requests are correctly formatted before they are processed. This backend connects with the frontend by providing endpoints that update the UI when users add or remove classes. The structure separates each part of the system (data handling, request processing, and route definition) making the system easier to maintain and update.

backend/src/model/SQLiteSettingsModel.js

```
const Class = sequelize.define("Class", {
  id: {
    type: DataTypes.STRING,
    primaryKey: true,
    allowNull: false
  },
  userId: {
    type: DataTypes.STRING,
    allowNull: false
  },
  subject: {
    type: DataTypes.STRING,
    allowNull: false
  },
  number: {
    type: DataTypes.STRING,
    allowNull: false
  },
  course_name: {
    type: DataTypes.STRING,
    allowNull: true
  }
});

class _SQLiteSettingsModel {
  async addClass(classData) {
    const newClass = await Class.create({
      id: Date.now().toString(),
      userId: classData.userId,
      subject: classData.subject,
      number: classData.number,
      course_name: classData.course_name || null
    });

    console.log('Model: Class created successfully, getting settings');
    return this.getSettings(classData.userId);
  } catch (error) {
    console.error('Model Error in addClass:', error);
    console.error('Stack:', error.stack);
    throw error;
  }
}
```

backend/src/controller/SettingsController.js

```
export class SettingsController {
  async addClass(req, res) {
    try {
      console.log('Received add class request with body:', req.body);
      const classData = req.body;
      if (!classData || Object.keys(classData).length === 0) {
        return res.status(400).json({ error: 'Class data is required' });
      }

      console.log('Calling model.addClass with:', classData);
      const result = await this.model.addClass(classData);
      console.log('Add class result:', result);
      res.status(201).json({
        message: 'Class added successfully',
        data: result
      });
    } catch (error) {
      console.error('Error adding class - Full error:', error);
      console.error('Error stack:', error.stack);

      if (error.message.includes('is required')) {
        res.status(400).json({ error: error.message });
      } else if (error.name === 'SequelizeValidationError') {
        res.status(400).json({ error: error.message });
      } else {
        res.status(500).json({
          error: 'Internal server error',
          details: error.message,
          stack: error.stack
        });
      }
    }
  }
}
```

backend/src/routes/SettingsRoutes.js

```
class SettingsRoutes {
  initializeRoutes() {
    this.router.post('/classes',
      [this.validateRequest.bind(this), this.validateUserId.bind(this)],
      this.controller.addClass.bind(this.controller)
    );

    this.router.delete('/classes/:userId/:classId',
      this.validateUserId.bind(this),
      this.controller.removeClass.bind(this.controller)
    );
  }
}
```



# Challenges and Insights

During this milestone, I ran into a lot of issues when switching the in-memory storage for settings to SQLite and Sequelize. One major aspect was going from synchronous to asynchronous operations to handle asynchronous database calls. New types of errors from database operations needed to be managed with more thorough error handling. Data relationships became more complex, requiring the use of Sequelize associations, and handling existing data during migration was addressed by initializing the database with test data. It was also challenging moving everything into components, as sometimes components wouldn't render correctly, or even render at all.

Especially for this milestone, working as a team was essential. As different parts of the website connected at the end, we needed to communicate and collaborate to make sure everything worked. We helped each other debug our code to ensure everything was functioning properly.



# Future Improvements and Next Steps

- Implement an undo button for changes made to user profile
- Add the option to customize user icon (change HEX code or upload an image)
- Give users the option to customize their display (light/dark/system)
- Add more class options by using UMass websites like SPIRE to view all available courses for the current semester

