

# Study on Campus

CS 326 Group 11

Milestone #6 | April 25th, 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: Location browsing screen [frontend](#) and [backend](#) feature, data, and routing implementation. [Miscellaneous frontend work](#).

## Julia Farber (Time Keeper)

Primary Contributions: [Post browsing screen](#), [post viewing screen](#) frontend and backend feature implementation. Posts data backend implementation.

## Anastasia Isakov (Project Organizer/Documentation Lead)

Primary Contributions: [Post creation screen](#) frontend/backend feature implementation. [User Integration](#) data backend and routing implementation.

## Ashley Kang (Quality Control)

Primary Contributions: [Settings Screen](#) frontend and backend feature implementation.

Su	M	T	W	Th	F	Sa
<h1>Historical Development Timeline</h1> <p><a href="#">Milestone 4 timeline</a>, <a href="#">Milestone 5 timeline</a></p>					4/11 <b>Deadline</b> Milestone #5 (Front-End Design & Implementation) <b>Development</b> Initial frontend merged to main	4/12
4/13	4/14	4/15	4/16	4/17	4/18 <b>Monday schedule</b> <b>Team meeting</b> Discuss Milestone #6	4/19
4/20 <b>Development</b> Began navbar component development and main.css adjustments	4/21 <b>No class</b>	4/22 <b>Development</b> Began post browsing and post viewing creation of backend for posts data PR for navbar component Began report backend model	4/23 <b>Development</b> Began post creation conversion to component, additional fixes, and report modal UI/frontend integration	4/24 <b>Team meeting</b> Working on Milestone #6 together in class	4/25 <b>Deadline</b> Milestone #6 (Front/Back-End Design & Integration) <b>Team meeting</b> Stand-up #5 <b>Development</b> PR for post creation changes	4/26 <b>Development</b> Began backend for user integration and adding post deletion and updating for Post Viewing.
4/27 <b>Development</b> PR for user backend, post browsing and post viewing and post backend and settings backend.	4/28 <b>Development</b> PR for integrating new posts with backend, bugs for post viewing/browsing, location browsing and crowding score reporting					



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)
- [User Profile Screen](#) (UI)
- [User Profile Data Structure](#) (Data)
- [Crowding Score Reporting Integration](#) (Server)
- [Navbar Dropdown Menu](#) (UI)

## Key Commits:

- [4130718](#) (4/23): Implemented UI for crowding score report modal.
- [d2b7fa5](#) (4/25): Implementation for storing crowding score selection locally, frontend aspect of crowding score report submission.
- [3e0d9ad](#) (4/27): Backend implementation for location data.
- [4b76a99](#) (4/28): Partial backend implementation for report data and location browsing page UI updates.
- [3f0e08a](#) (4/28): Report model, controller, and routing. Integrates with location JSON data and impacts location browsing page UI.

## Pull Requests:

- [#72](#) (4/22): Implemented JS component and UI for navbar and dropdown menu for site navigation, with event listeners for opening & closing via mouse click.
- [#74](#) (4/25): Implemented UI for crowding score report modal, with event listeners for: opening & closing modal via click and/or Escape key press, storing crowding level selection to Local Storage, and publishing event for report submission functionality.
- [#85](#) (4/28): Implemented report model, controller, and routing. Implemented location JSON, controller, and routing. Reworked and restructured location browsing page and card rendering components to function with asynchronous backend functions.

# Feature Demo: Location Browsing & Crowding Score Report

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

**Advanced Integration (4pts):** GET (report), POST (report), GET (locations), PUT (location)

For this milestone, I worked to implement the **crowding score report feature**, a function which affects the the location browsing page. From this 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 **string[]** with **location information, crowding score**, and **timestamp**. Validates information, adds a unique **report id**, updates the corresponding **location.report[]** in **locations.json**, and stores the report in memory
- **GET /locations** endpoint – attempts to access the **locations.json** file and returns the JSON file contents if there are no errors; uses the Node.js **fs module** and takes no arguments
- **PUT /location** endpoint – attempts to access and write to the the **locations.json** file, 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).
- **Point Value:** 1-2 points. Uses two HTTP methods for data retrieval, but required significant adjustments to previous UI rendering.
- **Completion Level:** This feature is mostly complete, though there are a few bugs that still need to be addressed, which mainly affect the UI.

## Feature: Submit Crowding Report

- **Description:** The user can **submit a crowding score report** (**POST** /report), which is stored on an in-memory array of reports. The **report id** is also stored in a JSON file containing each location and corresponding information (**GET** /locations, **PUT** /location).
- **Point Value:** 3 points. Uses three HTTP methods in connection with each other. Required routing logic and server-side validation.
- **Completion Level:** Functional, but may require some edits for the sake of the UI and remaining bugs.

# Code Structure

## Focusing on Location Browsing & Crowding Score Report Integration

### 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.

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/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
```

frontend/src/services/ReportRepositoryRemoteService.js

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

326GROUP11

backend

locations

JS controller.js

JS routes.js

reports

JS controller.js

JS report.js

JS routes.js

> src

frontend \src

components

> BaseComponent

LocationBrowsingComponent

# LocationBrowsingComponent.css

JS LocationBrowsingComponent.js

LocationCardComponent

JS CrowdingHints.js

# LocationCardComponent.css



## Frontend Code: Crowding Score Reporting

```
// attach event listener for crowding score modal submit button click
const reportSubmitButton = document.getElementById("report-submit");
reportSubmitButton.addEventListener("click", (event) => {
  event.stopPropagation(); // prevent any bubble up (should be find regardless, report modal itself has no event listener)

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

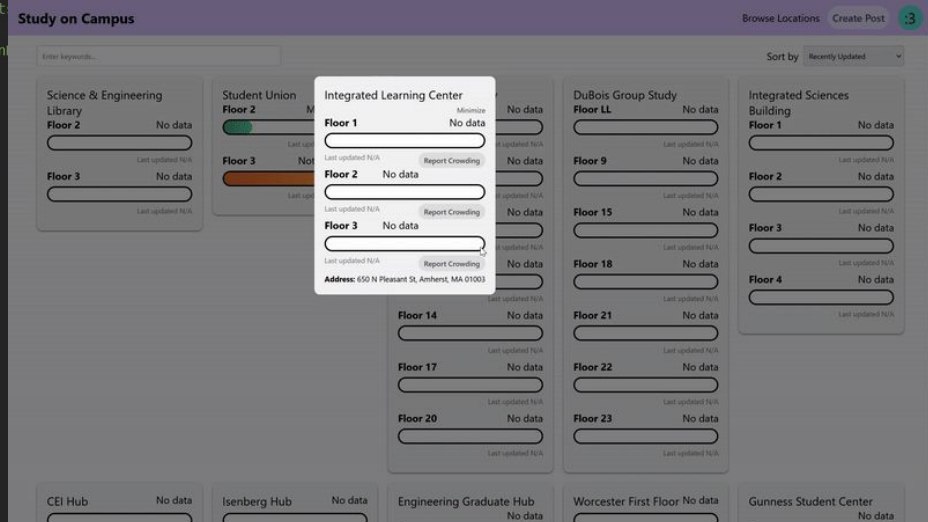
  // check if no selection made
  if (localStorage.getItem("crowding") === null) {
    alert("Please select a crowding score level.");
    return;
  }

  // construct data for add report
  const data = {
    report: {
      location: reportSubject,
      score: Number(localStorage.getItem("crowding")), // convert to number
      timestamp: Date.now()
    }
  };

  // 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.#hideElement(this.#reportModal); // close modal after successful report submission
  hub.publish(Events.MinimizeLocationCard); // close modal
  this.#locationsData = await this.#getLocations(); // update location data attribute
  await this.#renderCards();
});
```



➤ **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 has made a selection, the locally stored choice, location, and timestamp 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.

➤ **Challenges:** A major challenge for this milestone 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 a lot of restructuring and debugging. A few other elements required additional work, like figuring out how to pull location data from component text, rather than require passing additional data between multiple files.

## Backend Code: Crowding Score Reporting

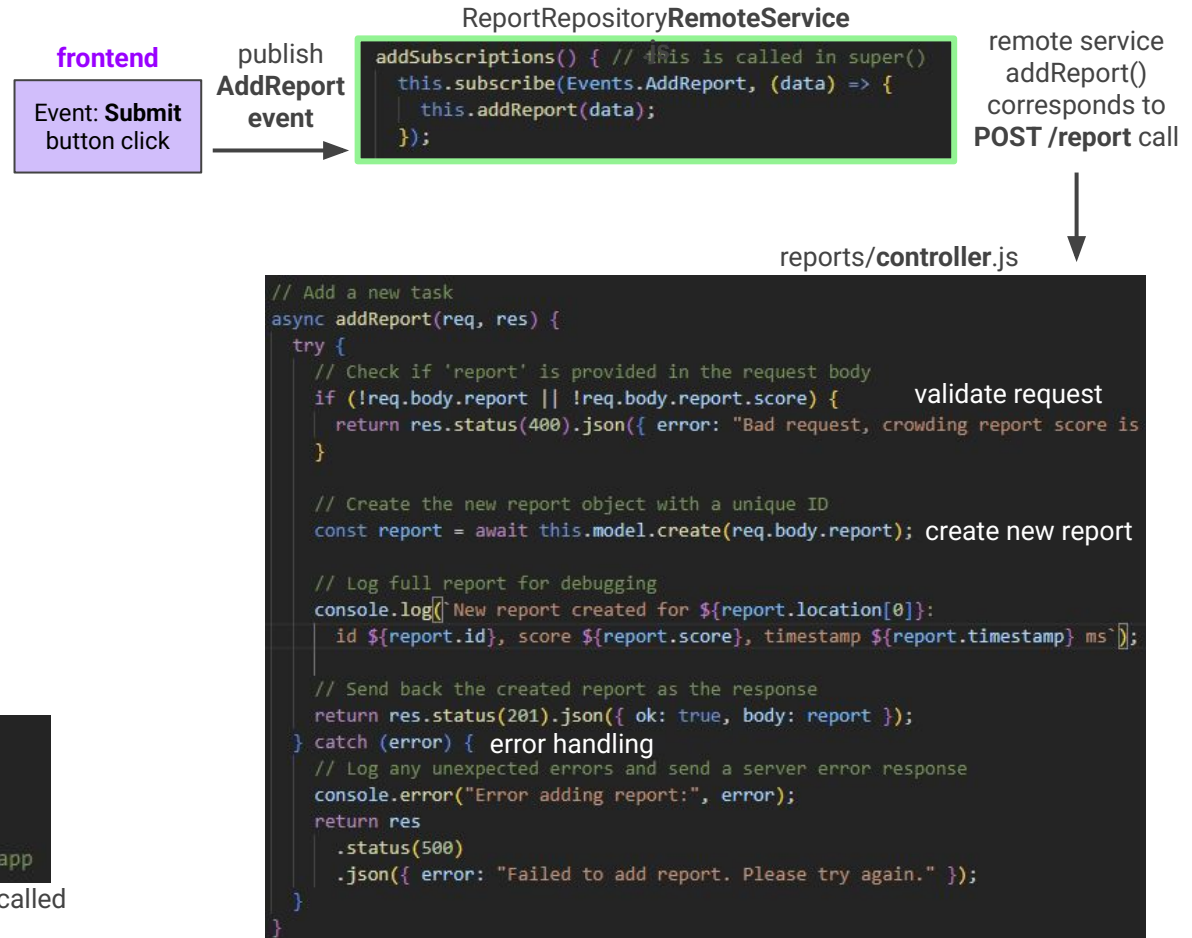
➤ **Description:** This segment of code highlights **what happens after the submit button is pressed on the report modal**. This is one aspect of the backend that connects with the frontend code in the previous slide (LocationBrowsingComponent.js).

➤ **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 is updated.

```
// 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 location JSON data
    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 object
    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 object
        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)

```
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);
```

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.geeksforgee
    const GETresponse = await fetch("http://localhost:3000/locations"); // GET method for lo
    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 ["
    const location = jsonData.find(location => location.name === locationInfo[0]); // get ma
    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.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

# Challenges and Insights

## Obstacles:

- I found that separating the backend components into separate files (controller, model, and routes) a bit confusing at times. While the separation makes sense, I found it somewhat tedious to debug and make changes to argument structures for different functions.
- I had some difficulties with the interactions between the report and location data structures, as I initially wasn't sure how to handle JSON file editing and server-side fetch calls. I was able to debug most of the issues through print statements and research, but there are still some issues with certain function calls that need addressing.

**Insights:** I appreciate the support that comes from working in a team environment, as well as being able to offer support to my teammates when needed! I also appreciate how much great documentation there is for various JS modules & functions—I needed to reference a lot of function documentation for this milestone.



# 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. Given time, I'd still like to implement the Google Maps API embed for the expanded post view (#42).

**User Profile Screen (#6):** I'd like to add a navigation option that leads to the settings page, once user sessions are implemented. I'd also like to add the Recent Post "embed" section next milestone.

**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. This comes secondary to the baseline functionality, through.



Anastasia Isakov

# Work Summary

## Issues worked on in this Milestone:

- [Post Creation Screen](#) (UI)
- [User Integration](#) (Server)

## Commits:

- [e1fc82e](#) (4/24): Converted Post Creation page to a component.
- [96da5a4](#) (4/25): Added date and time user input to post creation.
- [00956b9](#) (4/25): Added extra stylings for different screen sizes (width for 768px and width for 480px).
- [2fcaa77](#), [3e9bc62](#) (4/25): Fixed header overflow, added the isExpired value to post creation.
- [2b81c3e](#), [d3bd82f](#), [d183510](#) (4/27): Added user controller, added user routes, added added in-memory user model.
- [33ddf43](#), [c4c7773](#), [4ce6c38](#) (4/27): Added SQLite user model, without full user definition.
- [5543e83](#), [cb1ce5a](#) (4/27): Added user and fact definitions.
- [b81ec6c](#), [1b01bd8](#) (4/27): Fixes to user backend logic.
- [51c9cbb](#) (4/27): Changed default model to in-memory and adjusted in-memory model logic.
- [1da1b0c](#), [e9c94a3](#), [aa9b6a6](#) (4/28): Fixed bugs related to navigation bar and button redirects, integrated post creation with remote service.

## Pull Requests:

- [#75](#) (4/25): Conversion of Post Creation to component, with added date/time user input, stylings for different screens, and fixes to display and logic.
- [#80](#) (4/27): Added in User Integration backend, with a SQLite user model handling users and associated facts.
- [#83](#) (4/28): Fixed bugs related to navigation bar and button redirects, integrated post creation with remote service.



# Feature Demo: Post Creation

Branch: [19-post-creation-screen](#)

Basic/Advanced Integration (1pts): POST /post

```
localhost:3000/v1/post
Pretty-print
{
  "timestamp": 1745776663767,
  "isExpired": false,
  "postComments": [],
  "userId": "00000000",
  "name": "displayName",
  "pronouns": "she/he/they"
},
{
  "postId": 3,
  "title": "Try To Delete Me! ",
  "tags": [
    {
      "color": "#f76e50",
      "tag": "Tags Go Here"
    }
  ],
  "description": "Some text goes here. A lot o",
  "location": "Location here",
  "startTime": {
    "time": "Time",
    "date": "Date"
  },
  "timestamp": 1745776663767,
  "isExpired": false,
  "postComments": [],
  "userId": "00000000",
  "name": "displayName",
  "pronouns": "she/he/they"
},
{
  "postId": 4,
  "title": "Title Goes Here TestExpired... ",
  "tags": [
    {
      "color": "#f76e50",
      "tag": "Tags Go Here"
    }
  ],
  "description": "Some text goes here. A lot o",
  "location": "Location here",
  "startTime": {
    "time": "Time",
    "date": "Date"
  },
  "timestamp": 1745776663767,
  "isExpired": true,
  "postComments": [],
  "userId": "00000000",
  "name": "displayName",
  "pronouns": "she/he/they"
}
```

```
localhost:3000/v1/post
Pretty-print
{
  "location": "Location here",
  "startTime": {
    "time": "Time",
    "date": "Date"
  },
  "timestamp": 1745776663767,
  "isExpired": false,
  "postComments": [],
  "userId": "00000000",
  "name": "displayName",
  "pronouns": "she/he/they"
},
{
  "postId": 4,
  "title": "Title Goes Here TestExpired... ",
  "tags": [
    {
      "color": "#f76e50",
      "tag": "Tags Go Here"
    }
  ],
  "description": "Some text goes here. A lot o",
  "location": "Location here",
  "startTime": {
    "time": "Time",
    "date": "Date"
  },
  "timestamp": 1745776663767,
  "isExpired": true,
  "postComments": [],
  "userId": "00000000",
  "name": "displayName",
  "pronouns": "she/he/they"
},
{
  "post": {
    "title": "Test Post",
    "location": "Test Location",
    "description": "Test text",
    "tags": [
      "tag 1",
      "tag 2",
      "tag 3"
    ],
    "startTime": "2028-02-12T17:00:00.000Z",
    "timestamp": "2025-04-28T04:14:03.829Z",
    "isExpired": false,
    "comments": null
  },
  "postId": 5
}
```

**Description:** .Interacts with the backend to allow users to write and submit posts.

**Reason:** .Relatively simple, not persistent. Utilizes backend implemented by Julia as a component of Post Browsing/Post Viewing.

× 2/12/2028, 12:00:00 PM

× tag 1 × tag 2 × tag 3

UI on Post Submission

# Frontend Code

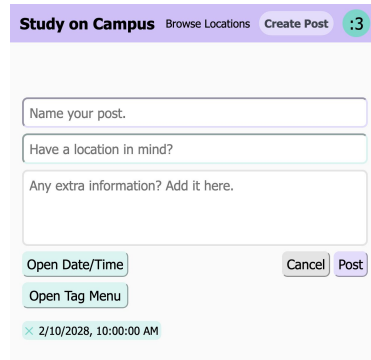
## Post Creation Frontend

The frontend code renders the UI through `PostCreationComponent`, which has been updated to include an added date/time input as well as media queries for different devices. The component handles various validations for the post, such as ensuring that a title is submitted, restricting the date and time to future date and times, and handling duplicate tags.

The frontend integrates with the backend through `PostRepositoryRemoteService` and `PostRoutes`. On submit, the page publishes a `StorePost` event instance to `EventHub`, which sends a POST request to the post endpoint,

### Challenges:

- **Date/Time Menu:** Implementing how the date/time menu popped up as well as how the date and time rendered took some rearranging of the css, as well as factoring it into the additional Media Queries.
- **Publish Post:** I largely adapted the `IndexedDb` logic from a previous milestone but then linked it to the post service. Currently though, while the post does get added, the Post Browsing page doesn't render it because it seems to be pulling from different endpoints. This will be restructured in the next Milestone.



```
if (input === 'Invalid Date') {
  errorMessage.innerHTML = "Must be a valid date."
  errorMessage.style.display = "block";
  e.target.value = '';
  return;
}

if (input < now) {
  errorMessage.innerHTML = "Must be a future date and time."
  errorMessage.style.display = "block";
  e.target.value = '';
  return;
}

errorMessage.style.display = "none";

const dateTimeList = this.#container.querySelector('#datetime-list');
const formatted = input.toLocaleString();

if (this.#datetime) {
  const existing = dateTimeList.querySelector('.datetime');
  const label = existing.querySelector('span:not(.remove)');
  label.textContent = formatted;
} else {
  const dateTimeItem = document.createElement('div');
  dateTimeItem.classList.add('datetime');

  const dateTimeLabel = document.createElement('span');
  dateTimeLabel.textContent = formatted;

  const removeBtn = document.createElement('span');
  removeBtn.textContent = 'x';
  removeBtn.classList.add('remove');

  dateTimeItem.appendChild(removeBtn);
  dateTimeItem.appendChild(dateTimeLabel);
  dateTimeList.appendChild(dateTimeItem);
}
```

Date/Time tag, Media Query for screen smaller than 768px

```
#add-tags-btn {
  margin-top: 0.5rem;
}

#tag-input, #datetime-input,
label[for="datetime-input"] {
  font-size: 1rem;
  padding: 0.25rem;
}

.tag, .datetime {
  font-size: 1rem;
  padding: 0.25rem;
}

.extra-input {
  display: flex;
  flex-direction: column;
}
```

```
@media (max-width: 768px) {
  form {
    padding: 10px;
  }

  .post-container {
    padding: 2vw;
  }

  .post-title input,
  .post-location input,
  .post-body textarea {
    font-size: 1.2rem;
    padding: 1.5vw;
  }

  .post-body textarea {
    min-height: 20dvh;
  }

  #submit-post,
  #submit-cancel,
  #add-tags-btn,
  #add-datetime-btn {
    font-size: 1.2rem;
    padding: 1vw;
  }
}
```

```
const post = {
  title: titleInput.value.trim(),
  location: locationInput.value.trim(),
  description: bodyInput.value.trim(),
  tags: this.#tags,
  startTime: new Date(this.#datetime),
  timeStamp: new Date(),
  isExpired: false,
  comments: null
};

this.#publishPost(post);
this.#clearInputs(titleInput, locationInput, bodyInput, tagList, dateTimeList);
window.location.href = "/pages/PostBrowsing/index.html"
```

### Post submitting logic

```
#publishPost(post) {
  const hub = EventHub.getInstance();
  hub.publish(Events.StorePost, { post });
}

#clearInputs(titleInput, locationInput, bodyInput, tagList, dateTimeList) {
  titleInput.value = '';
  locationInput.value = '';
  bodyInput.value = '';
  tagList.innerHTML = '';
  dateTimeList.innerHTML = '';
  this.#tags = [];
  this.#datetime = null;
}
```

# Code Structure

## Post Creation Frontend, User Integration Backend

### Frontend

frontend/src/components/PostCreationComponent

- PostCreationComponent.css
- PostCreationComponent.js

frontend/src/pages/PostCreation

- index.html
- script.js

frontend/src/services

- PostRepositoryFactory.js
- PostRemoteService.js

### Backend

backend/src/controller

- UserController.js

frontend/src/model

- InMemoryUserModel.js
- 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. Not listed here since Julia implemented it, but Post Creation links to PostRoutes in the routes folder through the PostRemoteService. The UserRoutes will also do the same with a UserRemoteService, which will be worked on further as part of the Milestone #7 Login/Registration and authentication task.

# Backend Code

## To be used for User Integration

The backend currently has 3 routes that interact with a User model. The default is a non-persistent In Memory model, but a SQLite model has also been implemented.

Establishes the /login endpoint (to fetch the users information based on email), /users endpoints (to fetch all users for testing), and the /user endpoint, through which new users can be made using POST (will be integrated with registration), user information can be retrieved using GET, and user information can be updates using PUT (to be integrated with the User Profile/Settings screens).

### Challenges:

- **SQLiteUserModel:** Figuring out how to link Facts and Users together, as well as how to encapsulate the model for Users in the lib folder. Created two separate Tables for Users and Facts and used userId as a foreign key in facts,.

```
async getUser(req, res) {
  try {
    if (!req.body || !req.body.userId) {
      return res.status(400).json({ error: "User ID required to get account." });
    }
    const user = await this.model.read(req.body);
    if (!user) {
      return res.status(400).json({ error: "User not found." });
    }
    res.json({ user });
  } catch (error) {
    console.error("Error getting user:", error);
    return res
      .status(500)
      .json({ error: "Failed to get user. Please try again." });
  }
}

async addUser(req, res) {
  try {
    if (!req.body || !req.body.email || !req.body.password) {
      return res.status(400).json({ error: "Email and password required to make account." });
    }
    const user = await this.model.create(req.body);
    return res.status(201).json({ user });
  } catch (error) {
    console.error("Error adding user:", error);
    return res
      .status(500)
      .json({ error: "Failed to add user. Please try again." });
  }
}

async updateUser(req, res) {
  try {
    if (!req.body || !req.body.userId) {
      return res.status(400).json({ error: "User ID required to update account." });
    }
    const user = await this.model.update(req.body);
    if (!user) {
      return res.status(400).json({ error: "User not found." });
    }
    return res.status(200).json({ user });
  } catch (error) {
    console.error("Error updating user:", error);
    return res
      .status(500)
      .json({ error: "Failed to update user. Please try again." });
  }
}
```

```
this.router.get("/user", async (req, res) => {
  await UserController.getUser(req, res);
})

this.router.post("/user", async (req, res) => {
  await UserController.addUser(req, res);
})

this.router.put("/user", async (req, res) => {
  await UserController.updateUser(req, res);
})
```

```
async create (user) {
  this.users.push(user);
  return user;
}

async read (user = null) {
  if (user === null) {
    return this.users;
  }
  if (user.userId) {
    return this.users.find((u) => u.userId === user.userId);
  }
  return this.users.find((u) => u.email === user.email);
}

async update (user) {
  const index = this.users.findIndex((u) => u.userId === user.userId);
  if (index !== -1) {
    this.users[index] = user;
    return user;
  }
  return null;
}
```

# Challenges and Insights

The major challenges for this milestone related to implementing parts of the backend. The coding of the backend itself required thinking through the different pages/routes we would have and what would be relevant for each of them - for example, while I was implementing the read function for the `SQLiteUserModel`, I realized that there would need to be separate login and getUser routes to which values were available (login only provides email and password, not userId). The model also uses “subtypes” as part of it, which took some time to figure out.

The other challenging part of this implementation was that I’m currently not working on the Settings page nor the User Profile page, which are the major pages that would use User Integration as part of them. As a result, this required a lot more communication with my team members who were working on these pages.




# Future Improvements and Next Steps

**#22 Login and Signup Screens:** Implement working authentication for the login and signup screens, where users data is saved as part of their account. This should also work with the implementation of log out, upon the user would be sent back to the signup screen and their account is inaccessible.

**#32 Forgot Password Screen(s):** Get a password reset screen implemented that allows a user to receive a code with their email and use it to reset their password. Handle alongside working authentication for Milestone #7.

**#77 Post Drafts:** If time permits, 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)
- Future Task [#79](#), [#82](#) (UI) (Data)
- [UI Standardization](#) (UI)
- And other closed issues...

## Commits:

- [036f731](#), [32e1bf9](#) (4/22) – Created initial backend model, router, and controller draft. Created post repo factory.
- [9932c89](#), [4fbb7be](#) (4/22) – Fixed and added more event listeners to switch pages, created a dropdown for post settings when viewing a post.
- [19d4a8f](#), [9fde286](#) (4/27) – Implemented posts functionality, connected to frontend (server routing used in remote service).
- [2d480ec](#), [039eda6](#) (4/27) – Implemented comment deletion, and updated html/css and javascript to dropdown settings for post/comment respectively.
- [d9a3515](#), [5aac260](#) (4/27) – Pulled from main to update code, and fixed switching pages linking URL code.
- [f439448](#), [c002ff0](#), [d6ca1c6](#) (4/27) – Updated css for a button for post viewing, pulled from main and resolved merge conflicts, and updated css for better visuals for post browsing.
- [bb13f01](#), [be37711](#) (4/28) – Pulled from main and fixed bugs with event Listeners that became redundant with new navigation bar component. Removed Unused backend function as well.

## Pull Requests:

- [#78](#) – Integrating all of post browsing and post viewing screens updates and connecting to the backend. Also wrote all of the posts backend that is being used. Further details written in the pull request description.
- [#84](#) – With navigation bar component, removed event listeners that are no longer needed that the navigation bar component takes care of. Also removed one unused function in backend. Fixed bugs overall.

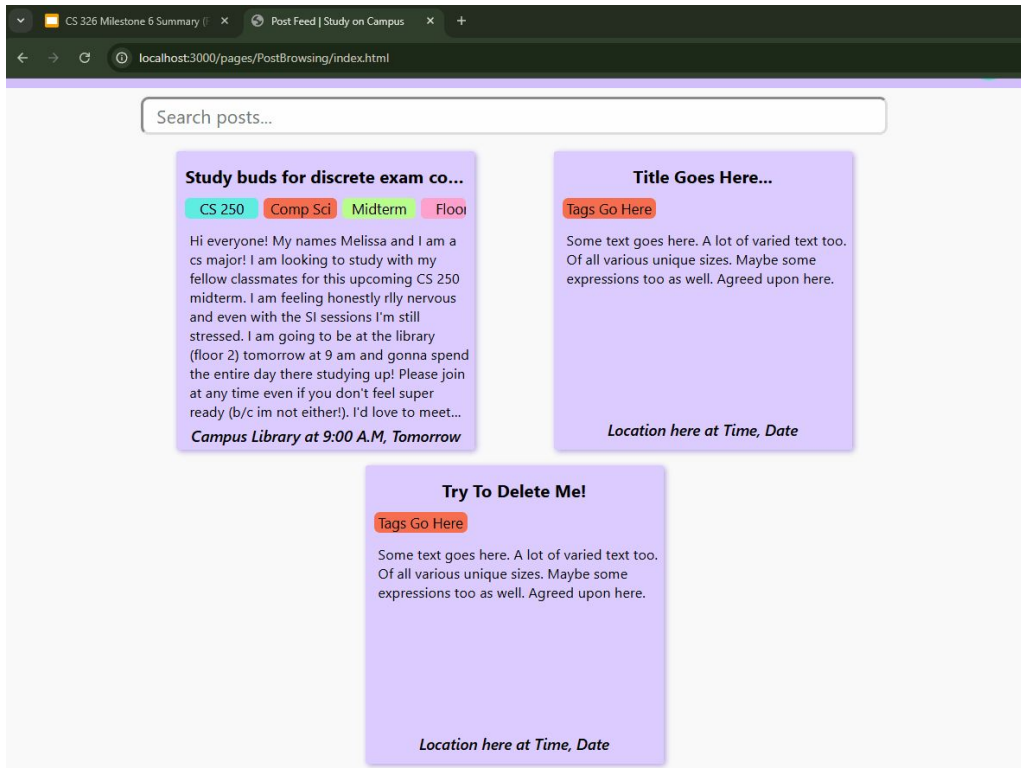


# Features Demo:

All implemented on branch:

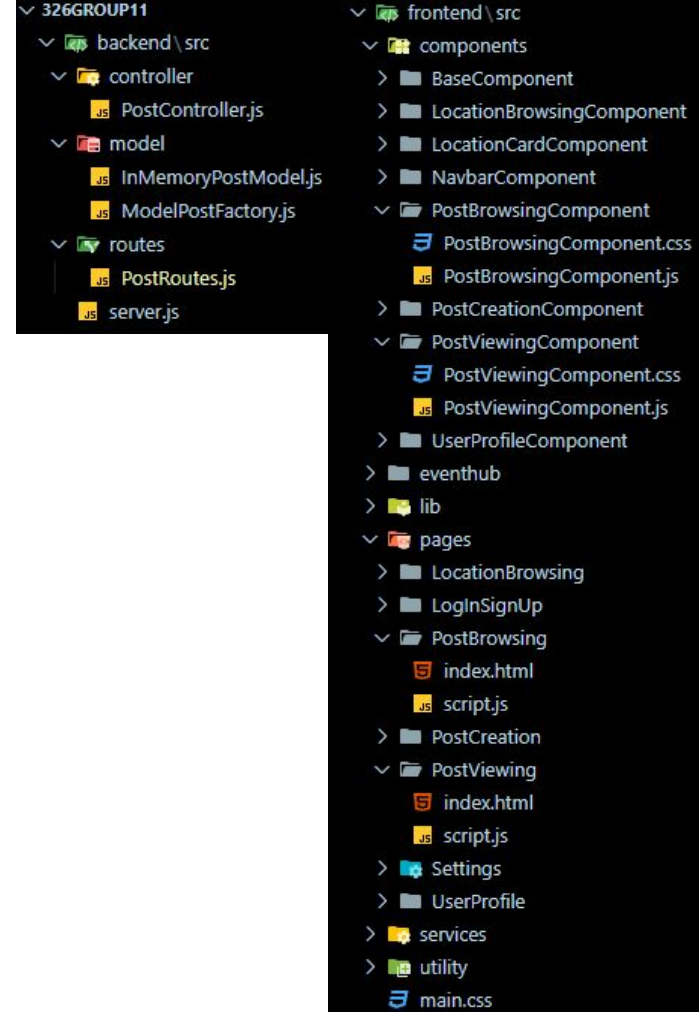
- **posts-backend-and-post-browsing-viewing (backend and frontend of all features!)**

- 1) **Loading All Non-Expired Posts on Post Browsing Page (COMPLETE/FUNCTIONAL) (Basic: 1 pt).** (A) Uses GET (read) /posts endpoint. Loads all posts for user to browse that aren't expired.
- 2) **Delete, Comment on, and Load a Post on Post Viewing Page (COMPLETE/FUNCTIONAL) (Advanced: 3 pt).** (A,C,D) Uses GET (read), PUT (update), and DELETE (delete) on /post endpoint. Loads a post a user clicks, getting it from database. Allows users to delete their post, add comments on all posts, and to delete their comments.
- 3) **Search Bar Functional with Filtering on All Posts (COMPLETE/FUNCTIONAL) (Basic: 1 pt).** (A) Uses GET (read) /posts endpoint, like loading, but more specific and all occurs in backend logic. Similar to (1) but allows for specific searches to only go through to view those specific posts related.
- 4) **Loading a Post URL Linking (Unrelated: 0 pt).** Uses urls and window href to allow (2) to work from post browsing to post viewing.



# Code Structure & Organization

This application and my work adheres to the class presented structure. There are two parent folders, backend and frontend. They both have their own respective src (source) folders. The frontend folder has the components, utility, services, and event hub. We also have a pages folder, where each respective screen has their own html file and script that runs in the html file to initialize those pages when they run. The frontend ensures that controllers have their component's code, while the services connect to the database/backend that the data is being accessed/updated/used from. There is a factory to control the versions being used, and the one that connects to the backend in this milestone (which is also being used) is in services, as PostRepositoryRemoteService, which accesses the backend for the posts and post endpoints (through server routing). The backend has the database, which is currently a version of memory (no SQLite). This model that connects to the "database," is then used by the controller to have requests processed, like update the post in the model for instance. Then, the router uses these controller methods to hook it up to the server, to allow these server fetch calls to be used by the front end in the service PostRepositoryRemoteService. Loose coupling is held through the separation of database, and different parts of the backend with different responsibilities for maintaining connections to the database and front end. The separation of backend and front end allows for clear communication and proper separation of responsibilities for files to ensure readability and proper organization. My critical components are: Post Browsing Component and Post Viewing Component with their respective pages and component files. Also all the Post service files: (Factory, RemoteFakeService, RemoteService, and RepositoryService (IndexedDB)). I have the post events in the event file, and the service files must properly update the event bus for clear tracking of history of events. The backend files I created are the Post Controller files, ModelPostFactory and the InMemoryPostModel which creates a memory data structure database for posts, and I creates the Post routing for the server, which is hooked up with v1/.



# Front End Implementation

These files adhere to the components, services, utility, component, eventhub folders. Component file is in the component folder (js file with css fille). There is also a pages folder that holds the scripts for the html pages that load to ensure proper behavior without using multiview on the entire application.

In Post Viewing Component (PostViewingComponent.js in the PostViewingComponent Folder in Components), I delete the comments using a private method, which separates and isolates from the render() function to ensure it is readable and not cluttered. This calls a method to the PostRepositoryRemoteService, which has the updatePost function that makes the fetch calls that call upon the backend server. This uses the server routes to access backend functions and database to make changes and update information. This ensures loose coupling between the frontend and backend to separate functionality of the controller and server and connection to database to ensure similar acting functions are in the same file. This properly distributes responsibility across files without overwhelming and clutter. Adds structure. Also, the PostViewingComponent.js uses the PostViewingComponent.css by the loadCSS() function in the component. There is more. The challenge faced was that I didn't know whenever to use put or patch, as they are so similar. Also, another challenge faced was ensuring the backend properly was functioning and not causing errors. A lot of testing was needed and ensuring the remote service repository file was making async calls to backend so it has enough time. That had caused an issue for me as well when things weren't behaving correctly as I forgot to wait on these function calls. Tasks v4 from class was heavily referenced and learnt from.

```
#deleteCommentListener(commentId) {  
  const deleteButton = document.getElementById(`commentId${commentId}`)  
  console.log(commentId);  
  console.log(deleteButton);  
  deleteButton.addEventListener('click', async () => {  
    if (confirm("Are you sure you want to delete your comment? This action CANNOT be undone.")) {  
      const index = this.#post.postComments.findIndex(c => c.commentId === commentId);  
      this.#post.postComments.splice(index, 1);  
      await this.#service.updatePost(this.#post);  
      const comment = document.getElementById(`commentId${commentId}`);  
      if (comment) {  
        comment.remove();  
        this.#renderComments();  
      }  
    }  
  })  
}
```

/components/PostViewingComponent/PostViewingComponent.js

```
async updatePost(postData) {  
  const response = await fetch(`http://localhost:3000/v1/post?id=${encodeURIComponent(postData.postId)}`, {  
    method: "PATCH",  
    headers: {"Content-Type": "application/json"},  
    body: JSON.stringify(postData)  
  });  
  if (!response.ok) {  
    throw new Error(`Failed to update post with postId ${postData.postId}`)  
  }  
  const updatedPost = await response.json();  
  return updatedPost;  
}
```

/services/TaskRepositoryRemoteService.js

# Backend Implementation

Models, routes, controllers, and middleware are on slides labeled “Code Structure & Organization.” Server.js code for post routing is on this slide.

The model creates the post data structure and in memory “database” for the server. The controller uses this model to use CRUD methods on the model and properly manipulate the data to do things like getting all posts from the posts endpoint, or all info on a post on the post endpoint. Then the router connects to the controller by creating fetching routes for the server, so fetch can be called on the server for data. Code integrates with front end by the fetch calls made in frontend, services, specifically in PostRepositoryRemoteService.js where it makes fetch calls to the server for posts as requests.

I am showcasing the code for getting all posts, as this was the first functionality I focused on and it was the hardest to figure out. I read from the in memory memory model and I then perform crud on the memory model in the controller, where I also created the filtering functionality that is used, to ensure routing does not have more than one GET /posts method, as filtering and getting all posts rely on that, and it would cause issues in the router setup. That was a challenge to notice and figure out through working on it. It was hard to also fix the issue of ensuring the backend (preview in browser) was supposed to be on port 3000 not 5500 through live server, which my teammates helped me figure out. This code connects to the front end through PostRepositoryRemoteService.js through loadAllPosts() and filterPosts(), which the difference is one passes in a search query and one does not. It was the first thing I tackled and it was the hardest with the new concepts of this milestone. Tasks\_v4 from class was heavily referenced in order to learn from imitating and experimenting.

```
// Get all Posts:
async getAllPosts(req, res) {
  const allPosts = await this.model.read();
  const nonExpiredPosts = allPosts.filter(post => (post.isExpired === false));
  const search = req.query.s?.toLowerCase() || "";
  const filtered = nonExpiredPosts
    .filter(post => (search === "") || (post.title.toLowerCase().includes(search) ||
      post.description.toLowerCase().includes(search) ||
      post.location.toLowerCase().includes(search) ||
      post.tags.some(tag => tag.tag.toLowerCase().includes(search))));
  res.json({posts: filtered});
  // Response is obj with a 'posts' property containing an array of posts.
  // Could be anything but define it as an obj with a 'posts' property to keep
  // responses consistent across different endpoints.
}
/controller/PostController.js
```

```
async read(id = null) {
  if (id) {
    return this.posts.find((post) => post.postId === Number(id));
  }
  return this.posts;
}
/model/InMemoryPostModel.js
```

```
// Getting all the posts
this.router.get("/posts", async (req, res) => {
  await PostController.getAllPosts(req, res);
});
/routes/postRoutes.js
```

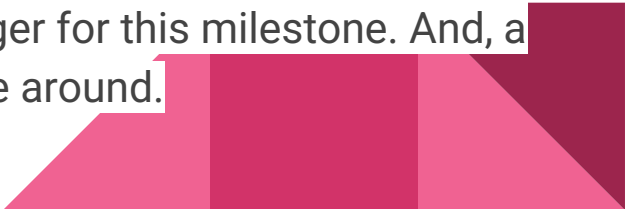
```
// setup middleware to parse incoming JSON data & add new data
app.use(express.json()); // erika, 2 weeks ago * init server.js

app.use("/v1", PostRoutes);

app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
server.js
```

# Challenges and Insights

There were many obstacles faced and lessons learned during this milestone. One lesson I learned was when I was struggling to get the pages and page switching to work correctly on my teammates' machines, which appeared to be working locally just fine. However, we learned that I was putting an absolute path on the wrong port for liveserver and not for the backend server we have. I realized not only my mistake, but was reinforced in the belief that my teammates are excellent for active/insightful debugging and learning. They were able to ensure I was on the right port and server and helped me fix this error before the last milestone. Another lesson that I learned was that taking steps back can help progress you further. I took time to go back to milestone 5 slides and examples and then proceeded to milestone 6 to fill in any gaps that remained and ensured I fully understood how the frontend and backend connect and how the server plays a role. I also learned that the backend is where the database connects to, to provide data for the entire application. My work was more clear and stronger for this milestone. And, a lot was learned and challenges faced with less confusion this time around.



# 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). Milestone 7 ideas and brainstorming.
- [#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.





Ashley Kang

# Work Summary

## Issues:

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

## Commits:

- [c3eb8e0](#) (4/27): Integrated backend into settings page
- [4486eb9](#) (4/27): Updated styling for class dropdown menu and prevent adding class duplicates
- [B542c15](#) (4/27): Fixed class dropdown menu and biography text box positioning in mobile view

## Pull Requests:

- [#81](#) (4/27): Added settings screen backend integration



# Feature Demo: Adding classes dropdown menu

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

1. **Get and Update User Settings (COMPLETE/FUNCTIONAL) (Basic: 2 pt).** (G)  
Uses GET (read) /settings/:userId endpoint.  
Loads user settings and preferences from storage for display on settings page.
2. **Full Settings Management with Profile and Account (COMPLETE/FUNCTIONAL) (Advanced: 3 pt).** (G,P,U) Uses multiple HTTP methods:
  - GET /settings/:userId to retrieve settings
  - POST /settings/preferences for creating new settings
  - PUT /settings/account and /settings/profile for updating existing settings Implements comprehensive settings management with validation and error handling.
3. **Course Management in Settings (COMPLETE/FUNCTIONAL) (Basic: 2 pt).** (P,D)  
Uses two HTTP methods:
  - POST /settings/classes for adding new classes to user profile
  - DELETE /settings/classes/:userId/:classId for removing classes from profile Allows users to manage their course list by adding or removing classes with server-side validation.

The screenshot displays the 'Study on Campus' settings interface. At the top, a purple header bar contains the title 'Study on Campus' on the left, and 'Browse Locations', 'Create Post', and a green circular icon with ':3' on the right. The main content area is divided into three columns. The left column, titled 'Account', contains fields for 'User ID' (value: 123), 'Email' (placeholder: email address), and 'Password' (placeholder: \*\*\*\*\*). Each field has an 'Edit' button to its right. Below the password field is a 'Show Password' checkbox and a 'Save changes' button. The middle column, titled 'User Profile', contains fields for 'Display name' (placeholder: Current display name), 'Pronouns' (placeholder: Current pronouns), 'Major' (placeholder: Current major), and 'Bio' (placeholder: Current biography). Each field has an 'Edit' button to its right. Below the bio field is a 'Save changes' button. The right column, titled 'Preferences', contains three checkboxes: 'Display major on posts', 'Display pronouns on posts', and 'Receive email notifications'. Each checkbox has a 'Save changes' button to its right. At the bottom of the right column, there is a 'Current classes' section with a purple plus icon and a 'Save changes' button.

# Code Structure

I organized the files for settings following the class methods for structure. On the backend (/backend/src/), I used a clear MVC pattern: models in /models/, controllers in /controllers/, routes in /routes/, and server setup in server.js. This keeps data, logic, and API endpoints cleanly separated. On the frontend (/frontend/src/), I followed a pages pattern, grouping each page's HTML, JS, and CSS together (e.g., /pages/Settings/). I separated API communication into /services/SettingsRepositoryService.js and managed events in /eventhub/SettingsEvents.js. I maintained a strict frontend-backend separation: the backend exposes REST APIs under /api, and the frontend accesses them through service classes. EventHub manages UI state independently of backend logic. For component organization, I kept related files together, used consistent naming (Service, Events), and located files intuitively under /pages/, /services/, and /eventhub/.

## Backend

- ✓ backend / src
  - ✓ controllers
    - JS settings.js
  - ✓ models
    - JS settings.js
  - ✓ routes
    - JS settings.js
  - JS server.js

## Frontend

```
JS Service.js
JS SettingsRepositoryService.js

/**
 * An object containing event types used in settings.
 */
export const Events = {
  SAVE_PREFERENCES: 'SAVE_PREFERENCES',
  SAVE_PROFILE: 'SAVE_PROFILE',
  SAVE_ACCOUNT: 'SAVE_ACCOUNT',
  ADD_CLASS: 'ADD_CLASS',
  SETTINGS_ERROR: 'SETTINGS_ERROR',
  SETTINGS_SUCCESS: 'SETTINGS_SUCCESS'
};
```

- ✓ frontend / src
  - > components
  - > eventhub
  - > lib
  - ✓ pages
    - > LocationBrowsing
    - > LoginSignUp
    - > PostBrowsing
    - > PostCreation
    - > PostViewing
  - ✓ Settings
    - <> index.html
    - JS script.js
    - # style.css

# Frontend Code

I implemented the dynamic class management feature on the frontend. My `renderClasses` function updates the UI by displaying current classes and allowing users to add or remove them dynamically. I organized components cleanly: HTML structure under `.current-classes`, JavaScript logic separated, and smooth CSS transitions for UI updates.

The frontend integrates with the backend through POST and DELETE requests to `/settings/classes`, enabling real-time updates without page reloads. I maintained a clear separation between data handling and UI, following class-taught practices and using only vanilla JavaScript.

Challenges included keeping the dropdown filtered to exclude already-added courses and making UI changes smooth. I solved these by filtering course lists dynamically and applying CSS transitions.

`/pages/Settings/index.html`

```
<h4>Current classes</h4>
<div class="current-classes">
  <div class="add-btn-container">
    <button class="add-btn">+</button>
    <div class="dropdown-menu">
      <div class="dropdown-group">
        <label for="subject-select">Subject:</label>
        <select id="subject-select">
          <option value="">— Select a subject —</option>
        </select>
      </div>
      <div class="dropdown-group">
        <label for="course-number-select">Number:</label>
        <!-- course-number-select will be added by JavaScript -->
      </div>
    </div>
  </div>
</div>
</div>
```

`/pages/Settings/script.js`

```
function renderClasses(classes) {
  const classesContainer = document.querySelector('.current-classes');

  const addBtnContainer = classesContainer.querySelector('.add-btn-container');
  classesContainer.innerHTML = '';
  classesContainer.appendChild(addBtnContainer);

  if (classes && classes.length > 0) {
    classes.forEach(cls => {
      const classElement = document.createElement('div');
      classElement.className = 'class-item';
      classElement.innerHTML = `
        <span>${cls.subject} ${cls.number}</span>
        <button class="remove-class-btn" data-id="${cls.id}">✕</button>
      `;
      classesContainer.insertBefore(classElement, addBtnContainer);
    });

    document.querySelectorAll('.remove-class-btn').forEach(btn => {
      btn.addEventListener('click', async () => {
        try {
          const classElement = btn.parentElement;
          const classId = btn.getAttribute('data-id');
          const result = await settingsService.removeClass(classId);

          if (result) {
            classElement.style.opacity = '0';
            setTimeout(() => classElement.remove(), 300);
          }
        } catch (error) {
          showMessage(errorMessage, 'Failed to remove class');
        }
      });
    });
  }
}
```

# Backend Code

I developed the backend Settings API using an organized MVC structure. The SettingsController handles profile updates, ensuring proper validation and consistent error handling. Models manage data structure and storage, controllers handle business logic, and routes define API endpoints. Middleware functions validate request formats and required fields, improving error management and maintaining clean code. The backend integrates with the frontend through RESTful endpoints, allowing settings to be updated dynamically. My implementation follows class-taught patterns: using Express.js without extra frameworks, maintaining clear separation between models, controllers, and routes, and applying reusable middleware for validations. Challenges included enforcing consistent request validation and managing in-memory data safely. I solved these by writing modular middleware and structuring models carefully to prevent data inconsistencies.

/backend/src/controllers/settings.js

```
export class SettingsController {
  async updateProfile(req, res) {
    try {
      const profile = req.body;
      if (!profile || Object.keys(profile).length === 0) {
        return res.status(400).json({ error: 'Profile data is required' });
      }

      const result = await this.model.updateProfile(profile);
      res.status(200).json({
        message: 'Profile updated successfully',
        data: result
      });
    } catch (error) {
      if (error.message.includes('must be') ||
        error.message.includes('is required')) {
        res.status(400).json({ error: error.message });
      } else {
        res.status(500).json({ error: 'Internal server error' });
      }
    }
  }
}
```

/backend/src/routes/settings.js

```
const router = express.Router();
const controller = new SettingsController();

const validateRequest = (req, res, next) => {
  if (req.method === 'POST' || req.method === 'PUT') {
    if (!req.is('application/json')) {
      return res.status(415).json({
        error: 'Content-Type must be application/json'
      });
    }
  }
  next();
};

const validateUserId = (req, res, next) => {
  const userId = req.params.userId || req.body.userId;
  if (!userId) {
    return res.status(400).json({ error: 'User ID is required' });
  }
  if (typeof userId !== 'string' && typeof userId !== 'number') {
    return res.status(400).json({ error: 'Invalid User ID format' });
  }
  next();
};
```

# Challenges and Insights

During the development of the settings screen, several technical challenges came up, each leading to important lessons. One key challenge was designing the backend to handle different types of settings (preferences, profile, account, and classes) while keeping the code organized. To solve this, separate endpoints were created for each settings type, with their own controller methods and validation middleware. Another challenge was making sure data validation and error handling worked well across many different endpoints. Each endpoint had its own data needs. To fix this, reusable middleware functions like `validateRequest` and `validateUserId` were built to handle common checks. On the frontend, connecting with the backend brought its own set of problems, especially with managing complicated state updates and server communication for different settings. To address this, an EventHub pattern was used to separate components and handle events in a consistent way. This made the code easier to maintain and made it simpler to add new features later on.



# Future Improvements and Next Steps

- Reconfigure settings page layout using multiview UI
- Turn settings page HTML and CSS into components
- (Potential) implement an undo button for changes made to user profile
- Changing email (new page)/password (link to forgot password page?)

