

Requirements Analysis

Overview:

StudentStudy.com is an online interactive webpage designed to help students better engage with course material on their own. We are designing an application that makes it easy and unique for students to engage with one another online, sharing documents and studying outside the classroom. Study groups are some of the best way to engage and learn the material presented in class, and with StudentStudy.com, we plan to make it even easier.

However, this can seem impossible when you aren't familiar with anyone in your course, and it may be too intimidating to suggest doing so to a stranger. Though uncommon, some students will use different ways to study as a class, for reasons including: asking questions about the course material, comparing notes taken from lectures, collaborating on assignments (when allowed), and to plan/initiate group study sessions. Our interactive webpage is designed to help assist students in joining study groups and expanding their own learning.

Purpose:

The purpose of this document is to provide a detailed analysis of the requirements for the development of Studentstudy.com, a web application that connects university students who are studying for the same courses. This document will outline the objectives of the project, the functionality and roles of users, and the technical requirements for the development of the application.

Objectives:

- To provide a platform for students to easily find compatible study partners.
- To encourage collaboration and resource sharing among students.
- To provide teachers with a tool to monitor and support student study groups. ‘

Stakeholders:

- Our individual users (Students and Teachers seperately)
- Us, the devleopers

Users:

Student:

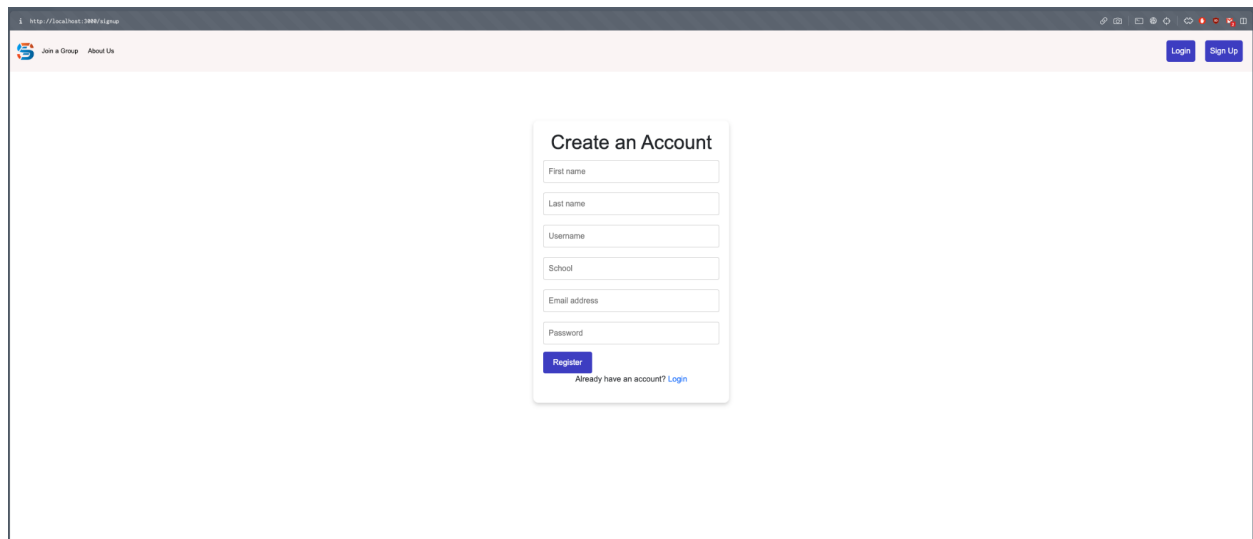
- Can create an account and complete a profile with their information.
- Can search for courses and connect with other students studying the same courses.
- Can create study groups and invite other students to join.
- Can upload and share documents and resources with other group members.
- Can communicate with group members.
- Can leave groups and join other groups.

Technical Requirements:

- Front-end development using HTML, CSS, and JavaScript.
- Back-end development using Express JS and SQL based on topics covered in class.
- Implementation of a database system to store user and group information.
- Implementation of a document sharing system for users to upload and share resources.
- Implementation of a user authentication system for security and privacy purpose.

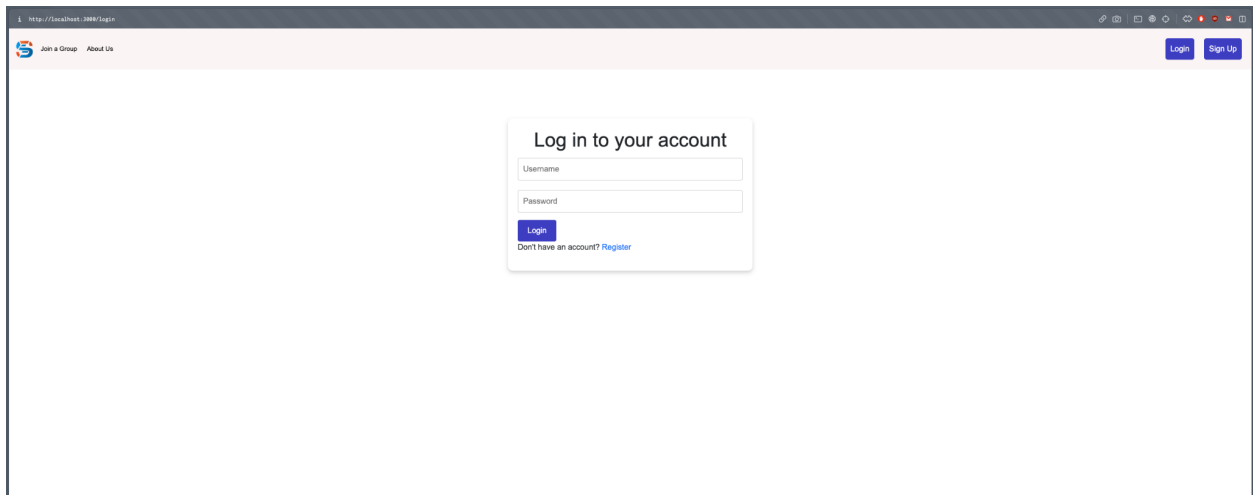
Implementation:

When implementing users, we took heavy inspiration from the user implementation we saw in the CMS application, however we added some additional parameters. Things like the users email, school, their first and lastname, and so on. You can directly see this with our sign up page,

A screenshot of a web browser displaying a 'Create an Account' form. The browser's address bar shows 'http://localhost:3000/signup'. The page has a light pink header with a logo on the left and 'Login' and 'Sign Up' buttons on the right. The main content area is white and contains a centered form titled 'Create an Account'. The form has six input fields: 'First name', 'Last name', 'Username', 'School', 'Email address', and 'Password'. Below these fields is a blue 'Register' button and a link that says 'Already have an account? Login'.

When we create a new user account, a POST request is sent out to our index router, where we will create a new user and establish these values as such. We will also automatically sign our user in with the newly created account.

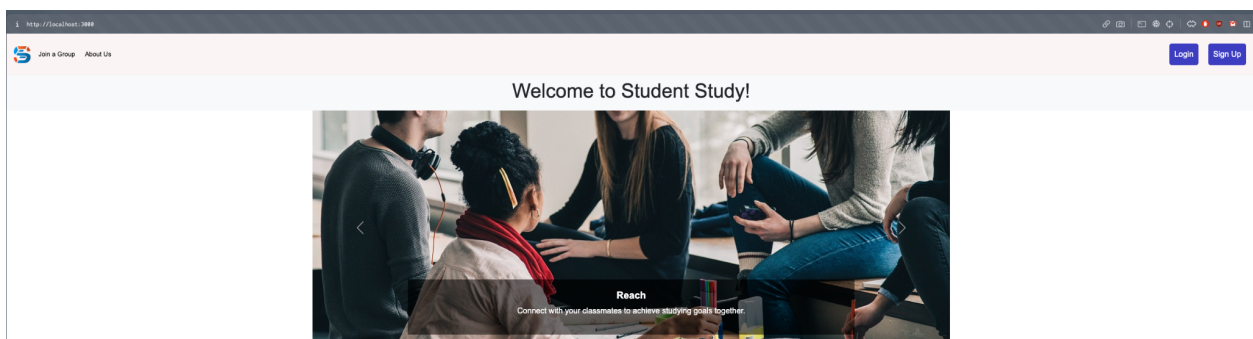
Our login page is a similar format to our sign up page.

A screenshot of a web browser showing a login page. The browser's address bar displays 'http://localhost:3000/login'. The page has a light pink header with a logo on the left and 'Login' and 'Sign Up' buttons on the right. The main content area is white and features a centered white box with the title 'Log in to your account'. Inside this box are two input fields labeled 'Username' and 'Password', a blue 'Login' button, and a link that says 'Don't have an account? Register'.

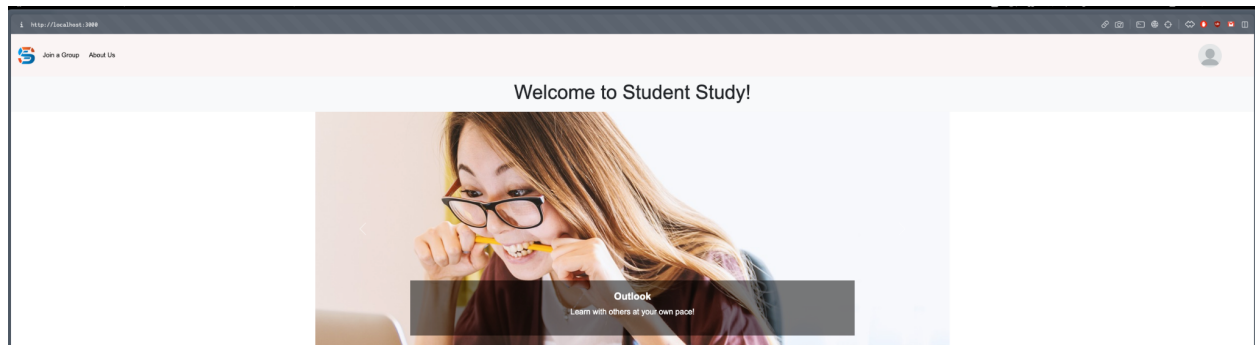
Once again, a POST request is sent out to our index router, and we check if there is a user that matches our requirements. If so, then we sign our user in and keep our user until our session expires.

When a user logs into our webpage, we wanted there to be a distinct signifier that they we're logged in, so our header bar will adjust accordingly.

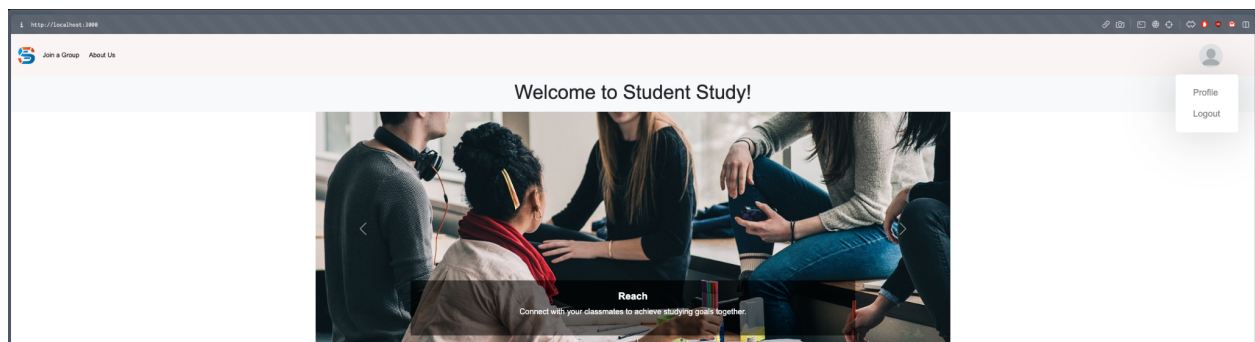
Below is an example of our home-page without a user signed in:



And here we have our home-page with our user signed in:

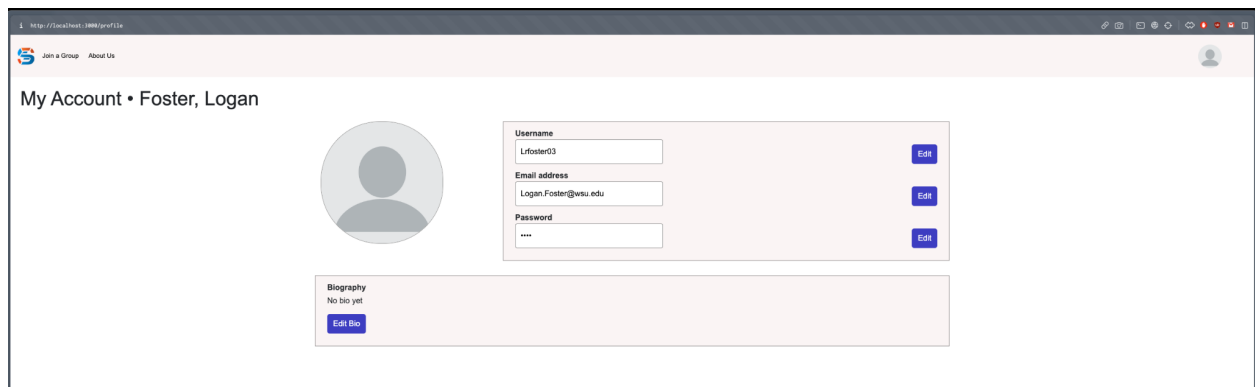


Selecting the user icon in the top right of our header will also showcase a dropdown allowing the user to access different information:



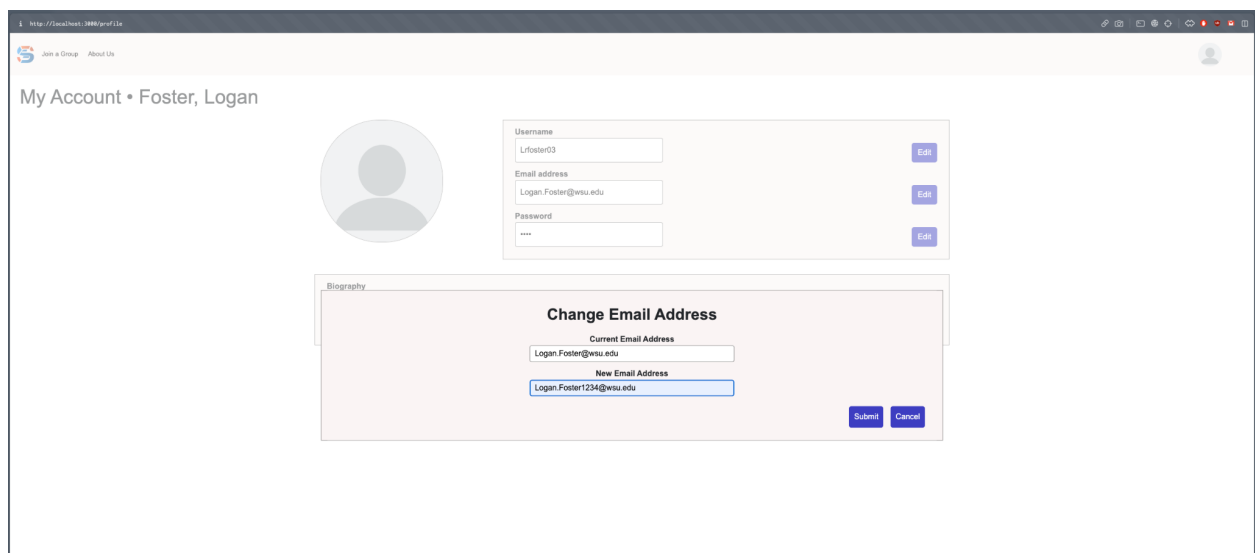
Selecting profile will bring them to their profile page where they can review and adjust some information, and groups will bring them to the myGroups page, signifying the groups the user has joined.

In the profile section, the user will see the following



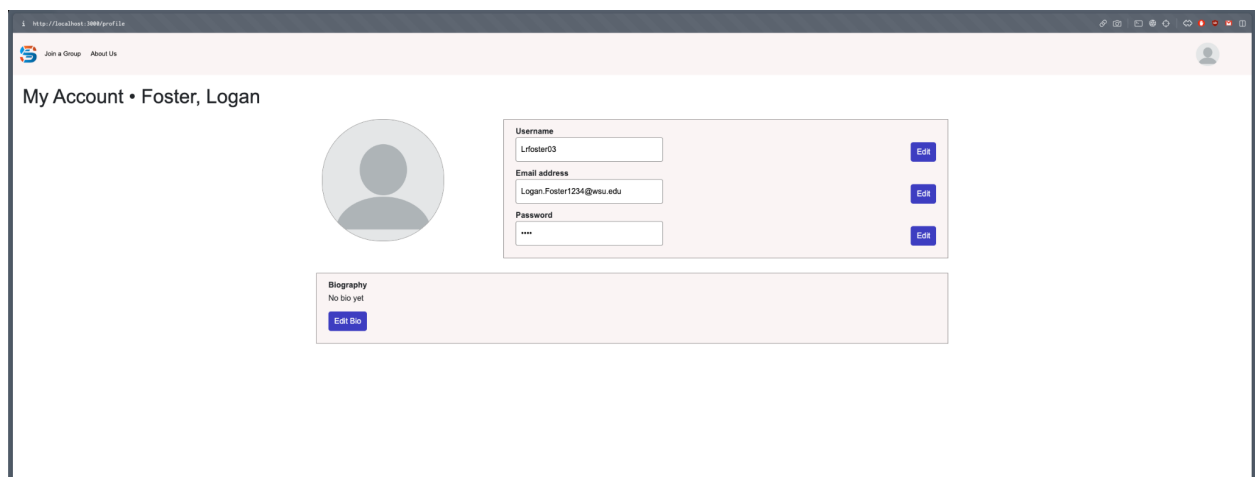
Before showing any information, the GET request for '/profile' compares local variable 'res.locals.loggedIn,' which is set to 'true' after a successful sign up/login. If it is true, then the user is logged in and is therefore allowed to view their profile information.

Clicking edit allows us to edit any of the respective fields. For example, editing our email shows us the following:



The screenshot shows a web browser window with the URL 'http://localhost:3000/profile'. The page title is 'My Account • Foster, Logan'. On the left, there is a circular profile picture placeholder. To the right, there is a form with three fields: 'Username' (lrfoster03), 'Email address' (Logan.Foster@wsu.edu), and 'Password' (masked with four dots). Each field has an 'Edit' button to its right. Below the form, there is a 'Biography' section with the text 'No bio yet' and an 'Edit Bio' button. A modal dialog titled 'Change Email Address' is open, showing the 'Current Email Address' (Logan.Foster@wsu.edu) and a 'New Email Address' field (Logan.Foster1234@wsu.edu) with 'Submit' and 'Cancel' buttons.

And when I change email to the following, we see that it updates:



The screenshot shows the same web browser window, but the 'Email address' field now displays 'Logan.Foster1234@wsu.edu'. The 'Change Email Address' modal is no longer visible. The 'Biography' section remains the same with 'No bio yet' and an 'Edit Bio' button.

This is once again handled through the use of POST requests, where we will specifically adjust the one element that we are requesting to change. Below you can see an example provided of how we receive those requests, but also do some backend verification to ensure that what we are changing is correct.

```

router.post('/chg_usrname', async function (req, res, next) {
  try {
    const user = await User.findUser( req.session.user.username, req.session.user.password)
    if (user !== null) {
      const placeholder = await User.usernameExists(req.body.newusername)
      if (placeholder === false) {

        const test = await User.create({ username: req.body.newusername, password: user.password, firstname: user.firstname })
        await user.destroy()

        res.locals.username = req.body.newusername
        req.session.user = test

        res.redirect("/profile")
      } else {
        res.redirect("/profile/?msg=username+exists")
      }
    } else {
      res.redirect("/profile/?msg=fail")
    }
  } catch (error) {
    console.log(error);
    res.redirect("/profile/?msg=error")
  }
});

router.post('/chg_pswrd', async function (req, res, next) {
  try {
    const user = await User.findOne({ where: { username: req.session.user.username, password: req.session.user.password } })

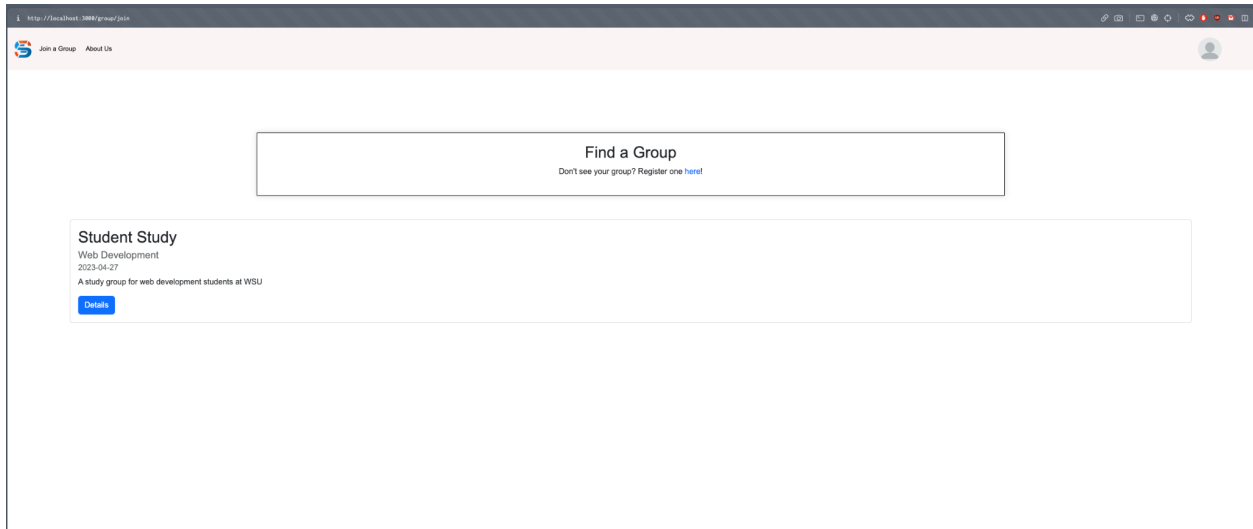
    if (user !== null) {
      if (req.body.newpassword === req.body.reppassword) // passwords match
      {
        req.session.user.password = req.body.newpassword
        res.locals.password = req.body.newpassword
        user.password = req.body.newpassword
        await user.save()
        req.session.user = user

        res.redirect("/profile")
      } else {
        res.redirect("/profile/?msg=pswrds+not+match")
      }
    } else {
      res.redirect("/profile/?msg=fail")
    }
  }
});

```

Unfortunately, at this time, we do not have the capability to allow the user to adjust their profile picture, so they are stuck with the default profile picture. The rest of the information provided is available to update however.

Once again in our header, selecting Join a Group will take us to our groups join page, where we will showcase all the groups we have available. We ultimately chose to consolidate both of our join group pages into one, as we saw it as being redundant to have both one for courses and one for individual schools, especially given that we want to help students engage inside and outside of their designated universities.

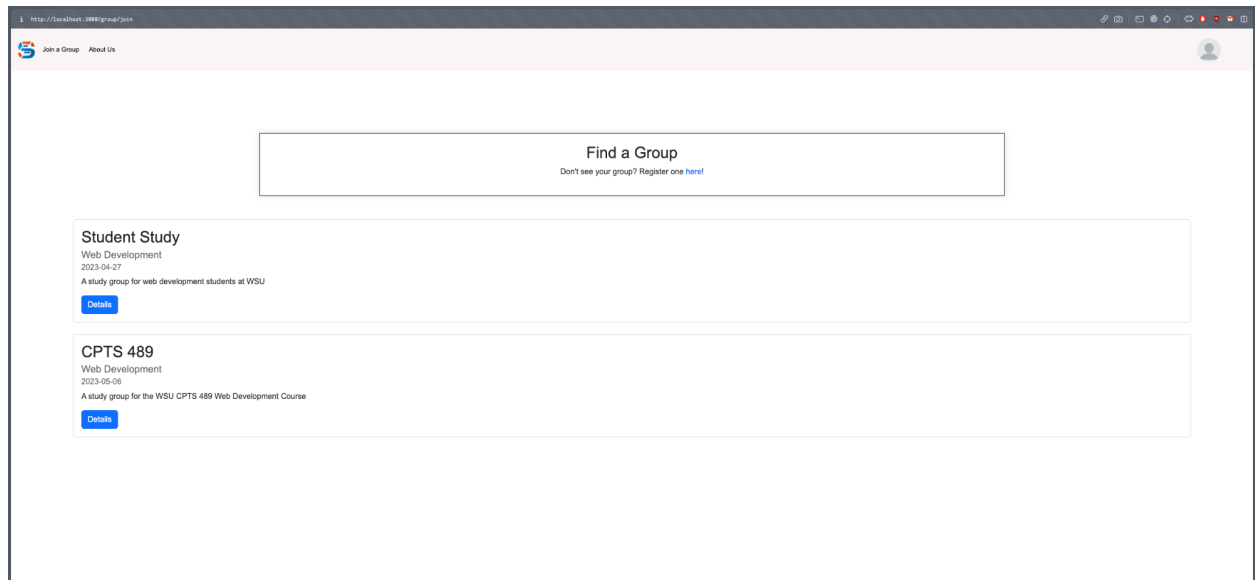


We display this information through the use of bootstrap cards and the Group Model that we have established, we display all of our currently established groups.

If you do not see the group you want to join, we can register a new group through the hyperlink provided in our box. This will bring us to our group registration page, allowing us to enter in details for our group. Below is an example:

A screenshot of a web browser showing a page titled 'Create a Group'. The browser's address bar shows 'http://localhost:3000/group/register'. The page has the same header as the previous screenshot. The main content area features a white box with the title 'Create a Group'. It contains four input fields: 'Group Name' with the value 'CPTS 489', 'Subject' with the value 'Web Development', 'Exp. date' with the value '05/06/2023', and a text area with the value 'A study group for the WSU CPTS 489 Web Development Course'. A blue 'Register' button is at the bottom of the form.

And upon clicking register, we can see that it takes us back to our groups page with our newly registered group

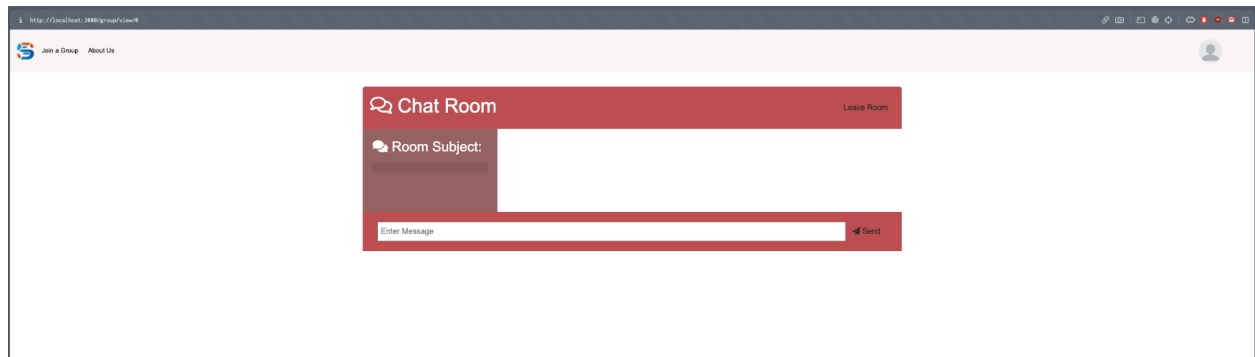


You can see we do this through a similar mean to the user creation. We create a new Group Model and then set each of the respective values with the values that the user has entered. Its rather simple.

```
router.post('/register', async function(req, res, next) {
  try {
    await Group.create(
      {
        name: req.body.groupname,
        subject: req.body.subject,
        expdate: req.body.expirationdate,
        groupbio: req.body.description,
        groupid: groupid,
      }
    )
  }
})
```

And then we create the new model and redirect our user back to our groups page. The group ID is an identification number we use to help us find our group. It is a simple 0, 1, 2... etc. integer token that we input and it helps us to find the specific group we want to pull values from.

Clicking on details leads us into our groups page, where we have our messaging functionality.



Unfortunately, we were not able to get all of our bugs with messaging fixed prior to our due date, so as of now, this page serves as a proof of concept.

An aspect that we thought was important for our web page was the ability to always navigate around the page. That is why in each page, we have our header visible and the ability to navigate to nearly every aspect of the web application simultaneously.