Spring 2023

Final Report – sdmay23-40

Design an Interactive Learning Platform to Engage Large Size Classroom

CyClass

cyclass.ece.iastate.edu *(must be on the ISU network)*

Group Members

Tyler Miller	—	*Backend Lead*
Alex Swenson	—	*Backend*
Guan Lin	—	*Backend*
Adam Walters	—	*Frontend Lead*
Jaden Ciesielski	—	*Frontend*
Brandon Burt	—	*Frontend*

Client / Advisor

Dr. Md Maruf Ahamed

Table of Contents

Overview

While large lectures are efficient settings for delivering material to students from professors, they struggle to allow for effective communication. Many, if not all of us, have been a part of lectures where the professor will lecture for 50 minutes and occasionally stop and ask if there are any questions, only to be met with silence. Many students feel as though they can't ask questions for fear of being perceived as "dumb" or "stupid". Additionally, some lecture halls are so large that it is extremely difficult to hear a conversation between the back of the room and the front of the room where the professor is. These sorts of issues occur in all large-size lecture halls across campus. This is an important problem because effective student-professor communication is vital to student success. Not only does communication allow the professor to know what students are struggling with, it also allows for meaningful discussions and clarification on important topics.

Our solution to this problem is building an interactive learning tool to bridge the gap between students and professors. Our application will have chat rooms (or "live lectures") for different lectures led by a professor. In these chat rooms, the students will be able to send messages to the class and professor, virtually raise their hand, anonymously ask questions, and answer in-class polls created by the professor. By allowing students to raise their hand virtually, the professor has a much easier time detecting questions as opposed to scanning a lecture hall of 300+ for a physical hand. To combat the shyness of some students, the anonymous chat feature will allow students to anonymously ask questions and allow them to feel more comfortable in the classroom. The poll feature will allow the professor to get feedback on students' learning and evaluate student participation. The chat rooms will be open 24/7, enabling students and professors to communicate during class as well as outside of class. With these features, our application aims to finally bridge the communication gap in large lectures.

Project Design Progression

Since the first semester of senior design, the vision of our solution did not change significantly. However, our priorities did change: Initially, we were planning on getting the core feature set completed relatively quickly so that we could move on to adding a range of supplemental features we had planned to eventually support in the future, the biggest of which was supporting multiple post types within a course (beyond just a lecture), such as announcements, questions, and possibly assignments. We quickly realized though that this was likely beyond the scope of what we could reasonably accomplish within one semester. Knowing this, we modified our plan to really hone in on the core feature set and polish those features as much as possible, creating a solid foundation for future teams that may end up working on this project. We believe this was a wise decision, resulting in a simpler but far more usable application.

Requirements

Functional Requirements

- Students, Professors, and TA's can create accounts and modify their own credentials
- Students can join a course with a join code.
- Professors can create courses
- Professors can create lectures
- Students can join live lectures and:
    - Raise hand to ask a question to professor (anonymously if they choose)
    - Send messages to the class
    - Edit messages they sent, and delete messages they sent
    - Answer a poll
    - Ask questions in the lecture feed
- Professor can view poll results and export them to a CSV
- Professor can delete any message
- Lectures get archived to be accessed anytime
- Professors, TA's, and Students can send safe attachments in the lecture feed

Nonfunctional Requirements

- Be able to answer questions in a large classroom environment anonymously to inspire more students to ask more often
- Be able to conduct polls with a large classroom, and have the results logged for grading later on. This could also be a measure of attendance in the classroom.
- Students who don't function well in large classrooms should thrive with the ability to interact anonymously and obtain more feedback
- Makes the job of the Professor in a large classroom simpler, and gives them more outreach and ability to impact more students and an opportunity to address everyone at some point.
- Helps Professor/TA relationship through CSV extraction of grading information to speed up grading of polls that can be used as quizzes, etc.
- Everyone is able to go back in time to past lectures and review conversations for useful information to prepare for upcoming assignments and to study.
- Builds a better understanding of coursework through a more complete cohesive learning environment in larger classrooms for students, and improves efficiency of professors.

Standards

We used various standards at different points throughout this project in order to help us create a high-quality product in a professional manner. First, we used IEEE 1016: Software design description when planning out our project. By following this we made data driven decisions to help create the best product. It also guided us towards creating diagrams for our architecture to help give visuals of our projects setup. IEEE 1028: Software Review & ISO/IEC/IEEE 26515:2018 were important standards to use as well during development of our application. IEEE 1028 deals with software review and helps us to ensure quality code by having parties with different perspectives review the code and product to ensure that it is doing what it needs to efficiently. At the same time, ISO/IEC/IEEE 26515:2018 helped us to work as efficiently as nimbly as possible by following an Agile approach to development. This allowed us to quickly iterate on designs and have good communication with our client. IEEE 9274.1.1 dealt with JavaScript Object Notation (JSON), and following it made communication easier between our frontend and backend very simple because both were written in JS and use JSON natively. Finally, IEEE 7002 is a standard that addresses data privacy. This was an extremely important standard to take note of because we need to safeguard students' data so as not to compromise their grades, anonymous questions, etc.

Engineering Constraints and How They Apply

We didn't have many significant constraints, but there are a few. The biggest constraint we faced was the size and power of the server hosting our application. This impacted us in both the number of concurrent requests we could theoretically handle simultaneously, as well as how many media files (PDFs, images, etc.) we would be able to store on the system. This shouldn't be an immediate problem for smaller classes, and we were eventually able to secure a large amount of storage, but it could be a problem for larger classes down the road. Another limitation we ran into was the ability to ensure that uploaded files aren't malicious, as we didn't have any access to a service that could scan them. A tertiary constraint, and one also related to security, had to do with authentication. We originally wanted to try and hook into ISU's Okta Auth system in order to handle logging in and verifying users were who they actually said they were. Unfortunately, we were told that would be unlikely to be allowed and so we implemented our own password-based authentication system in Express.

Security Concerns and Countermeasures

Security of our project has been a high priority since the beginning of even Senior Design 1. Since our application is a web application there were minimal physical security concerns. Our physical security protection is the fact that we are using an Iowa State VM to host our application and that VM is located in a secure building on campus with restricted access.

While we had few physical security concerns we had many cybersecurity concerns. The main concerns for our project were protecting our users data, preventing malicious actors from accessing the site, and preventing API abuse. These categories fall under application security and network security.

Application Security

Authentication

To combat these concerns we have implemented numerous countermeasures. Early on in the project our client/advisor told us that we would be unable to integrate Iowa State's existing Okta OAuth 2.0 authentication system into our project. To keep project costs essentially “free” we did not seek out a different third party authentication, but instead developed our own authentication process.

When a user comes to our application and creates an account their password first goes through a hashing process before it is stored in our database. In an ideal world, we would have multiple environments or in other words versions of the site all running on their own server in order to use lower environments for testing and to completely restrict access to the production database. Our application does not store any extremely sensitive data, but to prevent our development team from knowing users' passwords, we hash the passwords before storing them so that when we are working on the database we will not have any idea what the real plaintext passwords are.

After an account has been created, a user can then use their NetID and password to log into the site. If the login is successful, then on our server a session will be created containing the logged-in user’s info, the different roles they have within courses, and lastly an access token. In order for any API call to be made, a session needs to exist for that user on the server, and the access token needs to be present as a header on the request. We have a middleware process set up that checks those two items before any request is made. If either check fails, the user will receive a 403 forbidden error. This process prevents malicious actors from abusing our APIs as they will not be able to make requests without a valid session and token.

Permission Checks

While the above process may prevent bad actors without accounts from making API calls, a bad actor could easily make an account and gain a valid token and session. As a second layer of security, we have permission checks on our APIs. When a user first creates an account and logs in, their session will contain no roles for them because that user is not a member of any courses within the application. This means that a bad actor would still not be able to access an API they do not have permission to call. The three roles a user can have are STUDENT, TA, and PROFESSOR, and each of these roles is associated with a course (e.g user 1 has a

PROFESSOR role for course 3). All three of these roles have different permission levels and prevent users from accessing content from courses they are not in as well as prevent users from making API calls to gain data only a course owner should see. A few examples of what these permission checks prevent are: a user accessing the answers to a poll, a user accessing lecture/message content from a course they are not in, and user A deleting user B’s messages.

Another security process we have added revolves around users' abilities to join courses. When a course is created a join code will be generated in the form of a UUID. This UUID can then be presented to students, and the students will be able to go to our application and enter the join code to automatically be enrolled as a student within the course. The professor also has the ability to close the course so that the join code will not enroll any more students, preventing unwanted people from joining the course.

One of the last security measures added was for our PATCH endpoints. PATCH endpoints can be dangerous because they revolve around editing data. We only allow specific data fields to be edited to prevent dangerous activity such as a user changing their role to a PROFESSOR level.

Network Security

Private Network

To prevent outside attention, our application runs completely internally within ISU’s network. This means that in order to access the application a user must be connected to the Iowa state network either by VPN or be physically located on campus. This grants us the existing protection of being within Iowa state's network and limits our exposure to the greater internet.

TLS

Our application also utilizes TLS (Transport Level Security) in the form of HTTPS and WSS. We were able to acquire valid certificates from ISU which allowed us to switch from HTTP and WS protocols to HTTPS and WSS which both use TLS. Through HTTPS and WSS, users are able to have an encrypted connection to our server further protecting our users and our application.

Implementation Details

We chose an implementation that is built on JavaScript from end to end, leveraging React for the frontend and Node.js for the backend. This creates a much more uniform and maintainable application and simplifies testing and CI/CD.

Frontend

At its core, the frontend of our application is built around React, giving us the ability to write JSX and use custom-built plug-and-play components throughout the app to accelerate development. Supplemental to React, we use Redux, Axios, and React Router as runtime utilities, and we use Babel and Webpack to power our build process.

Code Structure

The source code of the frontend is organized at a high-level into three categories: components, routes, and utilities.

Components are general-purpose JSX components that can be used generically throughout the application. For example, the most commonly used component is the profile badge component. This component displays a user's name and course role alongside an icon, which can be found on several pages throughout the app. The top bar is also implemented as a component, which can be found at the top of every page.

Routes are a special JSX component which represent a complete page. These components are rendered underneath the top bar of our application. The bulk of the frontend's source code falls under routes.

Utilities are plain JavaScript classes and functions that serve a variety of purposes. The most important utilities are the API classes, which abstract away how the frontend interacts with the backend, allowing components and routes to focus more on the view and less on API-specific details.

These parts all come together inside the file "app.jsx," which defines the routing structure of the website and performs the root call to React to render the application.

Build Process

The source code for our application has to go through a few steps in order to end up as normal HTML, CSS, and JavaScript that browsers can render with high compatibility. This process is handled by Webpack in cooperation with Babel and other minor dependencies.

Webpack uses Babel in order to transpile our JSX and ES6 JavaScript into normal JavaScript that has a higher degree of compatibility than ES6. Webpack then combines the JavaScript files with all CSS files into a singular JavaScript file named "bundle.js." Any assets (i.e. images) used by the components are dumped into an assets directory alongside bundle.js and index.html. With all of these files together in one place, the backend can simply serve the build directory statically like a conventional web server.

Routing Layout

The frontend is generally laid out as shown below in Figure 1. This layout is made possible with the use of React Router, which allows us to structure and perform all routing entirely on the client-side with little to no interaction with the backend (minus the initial load).

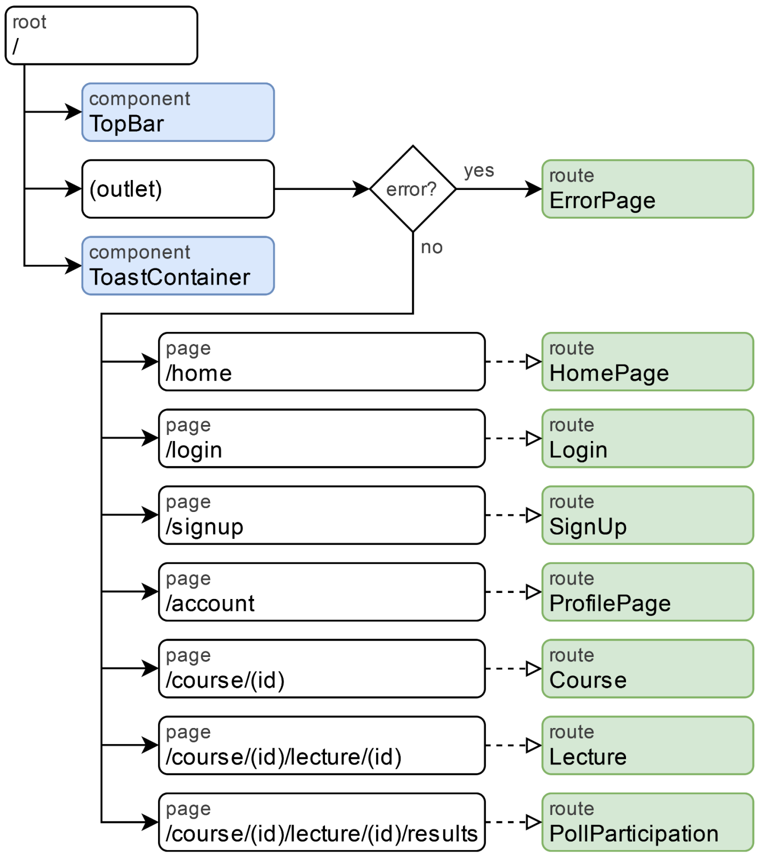


Figure 1 - Frontend's general routing structure

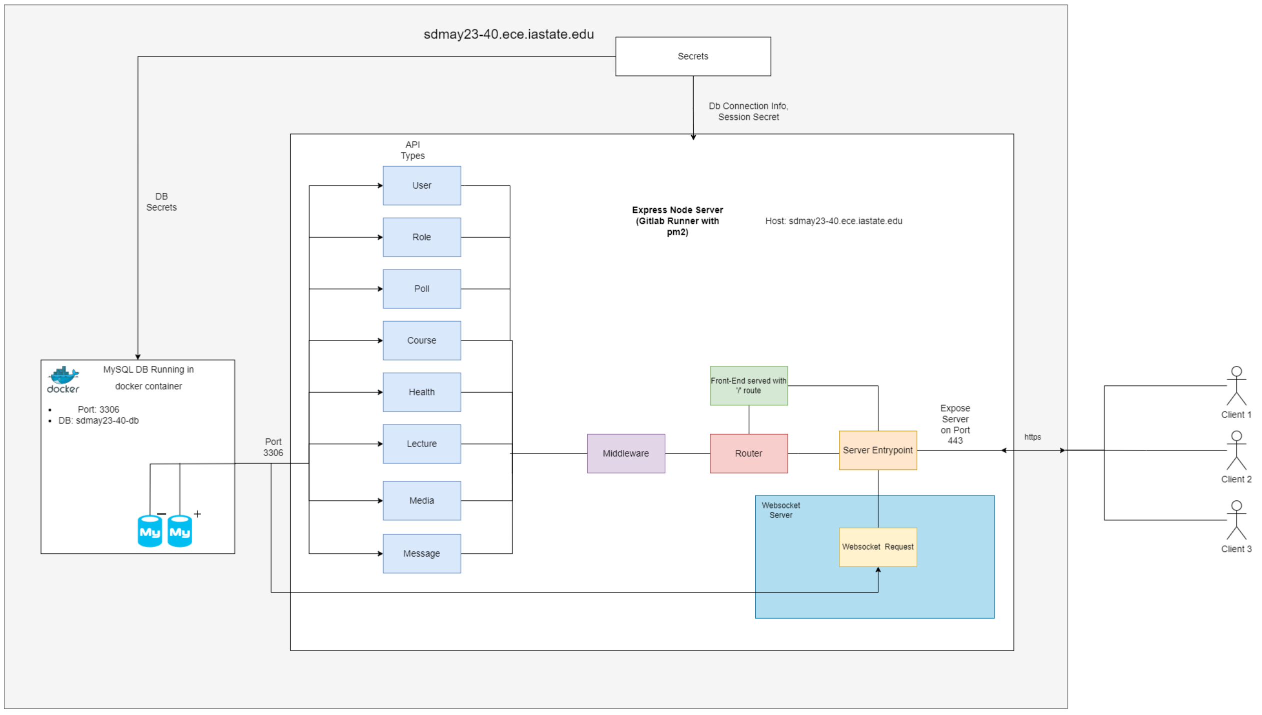
Most of the routes/components shown above are fairly self-explanatory, minus the ToastContainer component. This component is responsible for displaying "toasts," which are essentially just notifications to the user. Toasts are always shown at the bottom of the user's window and persist as the user navigates the application, disappearing after a set period of time.

Backend

Our backend is implemented on an Iowa State virtual machine running Ubuntu which is just a flavor of Linux. This Ubuntu server enables us to run our backend server application which is a Node.js server using the Express library along with supplemental Express packages such as “express-session.” The other main utility packages we use include bcrypt, mysql2, fs, ws, https, body-parser, cookie-parser, and pm2. A full list of dependencies can be found in the package.json file for the backend server code here: (packages). Aside from our Express server, we additionally have a MySQL database running on the same Ubuntu server via Docker.

Architecture

The backend architecture within the Ubuntu server is essentially our Express server interfacing with our MySQL database. By using Express, we are able to run our Websocket server through Express meaning all of our APIs and Websockets can use the same port. The Express server serves a multitude of APIs for our frontend to consume as well as Websockets for persistent connections vital for live messages and data transfer. Figure 2 is a diagram of our application structure within our VM.



sdmay23-40.ece.iastate.edu
Secrets
Db Connection Info, Session Secret
DB Secrets
API Types
User
Role
Poll
Course
Health
Lecture
Media
Message
Express Node Server (Gitlab Runner with pm2)
Host: sdmay23-40.ece.iastate.edu
MySQL DB Running in docker container
Port: 3306
DB: sdmay23-40-db
Port 3306
Middleware
Front-End served with '/' route
Router
Server Entrypoint
Expose Server on Port 443
https
Websocket Server
Websocket Request
Client 1
Client 2
Client 3

Figure 2 - Backend architecture

As described in the frontend section, Express additionally serves the entire frontend code bundle through the “/” route enabling our project to run entirely off of Express instead of needing to run the frontend separately. The API types in Figure 2 encapsulate all of the APIs for that specific data type including all different GET, POST, PATCH, and DELETE requests.

Code Structure

Our backend code layout follows the very popular “Controller Service Model” where essentially each one of our APIs has a controller that will call a service to perform a database operation. This model allows us to reuse code and avoid duplication. Our file hierarchy is structured as follows:

- Server (root)
    - /api folder
        - Folder for each type of API (course, user, message, ect…)
            - Service Folder
                - Services
            - Controllers
            - Subrouter
        - Main Router
    - /middleware folder
    - /utils folder
    - /websockets folder
        - Websocket handler
    - server.js (server entry point)

Additionally, test files sit next to their non-test counterparts. Our server entry point starts up our Node and Websocket server and from there when an API call is made the request will go to our main router → subrouter → middleware → controller → service and then back up again. This code structure has allowed for great organization as well as very little code duplication.

Deployment Process

Since both frontend and backend run off of the Express server, our deployment process just needs to restart one process. Before restarting our application with changes, our deployment pipeline needs to grab our SSH key to connect to the Ubuntu VM as well as run pre-deploy checks that prevent breaking changes from entering the live build. The pm2 Node library is used to handle our server restarts as this library handles this process with next to zero downtime as well as performing load balancing for us. Figure 3 details this process below.

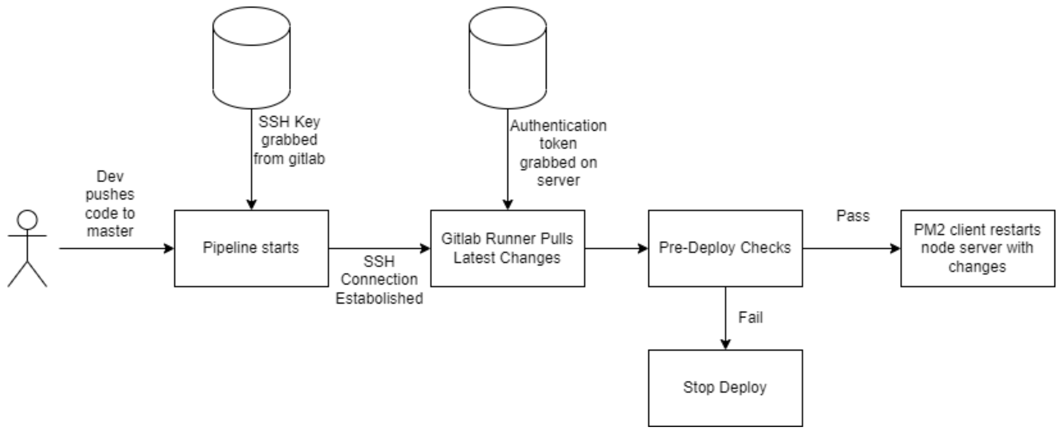

SSH Key grabbed from gitlab
Authentication token grabbed on server
Dev pushes code to master
Pipeline starts
SSH Connection Estabolished
Gitlab Runner Pulls Latest Changes
Pre-Deploy Checks
Pass
PM2 client restarts node server with changes
Fail
Stop Deploy

Figure 3 - Deployment process

Our pre-deploy checks include executing our test suite and running a linting check. These two checks ensure our code is formatted correctly as well as helps detect breaking changes before they enter the live application.

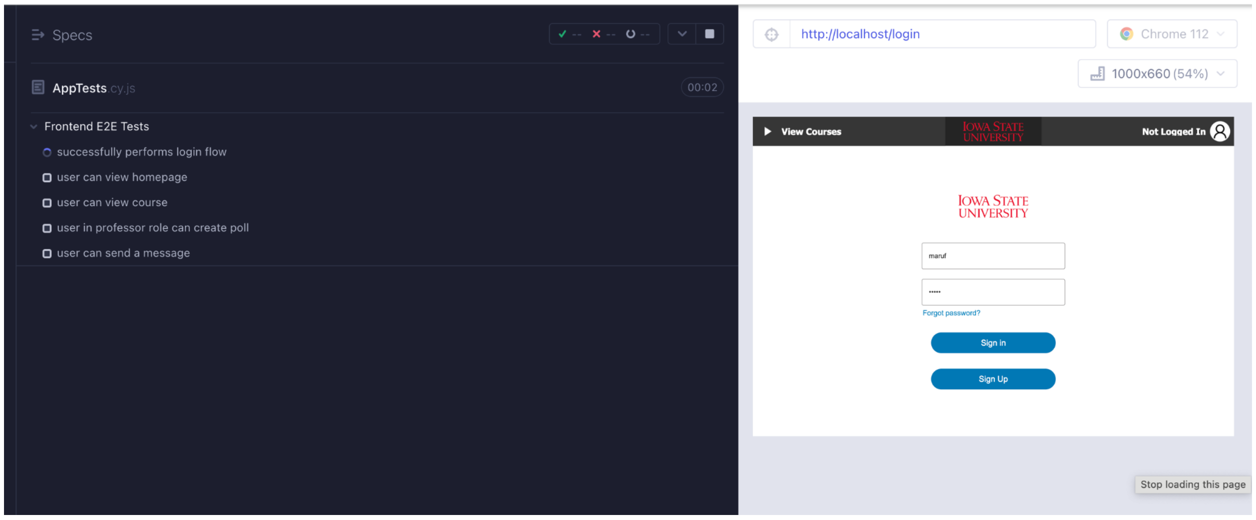
Testing Process and Results

Frontend

The testing of our frontend occurred towards the end of frontend development when most of the user interface, navigation, and user permissions were completed. We decided to focus our testing on end-to-end cases to assure our frontend navigation and processes flowed properly. This also served as a good way to determine if our user roles were working as expected. These tests instilled confidence in our frontend functionality.

End-to-End Testing

For our end-to-end testing we used cypress. Cypress was ideal for this because we were able to test our React components directly with data that emulates every core user movement that occurs within our user interface. We prioritized testing the full navigation of our app, as well as making sure user roles worked properly for security purposes. With Cypress, very similarly to Jest, you can do both positive and negative test cases to assure that certain conditions can and can't happen.



```
AppTests.cy.js                                  00:02
Frontend E2E Tests
  successfully performs login flow
  user can view homepage
  user can view course
  user in professor role can create poll
  user can send a message
```

Figure 4 - End-to-end testing

The tests all passed and successfully cycled through each scenario we posed that encompassed the core of our application. We made sure a user could login, view a course, send a message, and load the home page, and that only a professor can create a poll.

Overall, the results of our frontend testing were successful. The real priority to test on frontend was user navigation and the functionality of elevated user roles. We did so successfully with Cypress, which allowed us to plug in our React components directly into the tests, and then render the application while simultaneously running assertions against the movements of a user in scenarios that would be seen in the everyday use of our application. It proved that our frontend and its user interface ran properly. It also showed that our elevated user role for the professor worked as expected.

Backend

The testing of our backend processes occurred alongside our development throughout the duration of the project. The main testing we focused on for the backend was unit and integration testing. These two types of testing allowed us to have confidence in our code as well as enable us to have automated scripts to prevent application-breaking changes from being deployed.

Unit Testing

For our unit testing we used the Jest framework. By using Jest, we are able to use its excellent mocking tools as well as integrate the entire testing suite into our deployment pipeline. We tested all of our controllers, services, and utilities at the function level allowing us to test functionality without deployment as well as catch breaking changes during deployments.

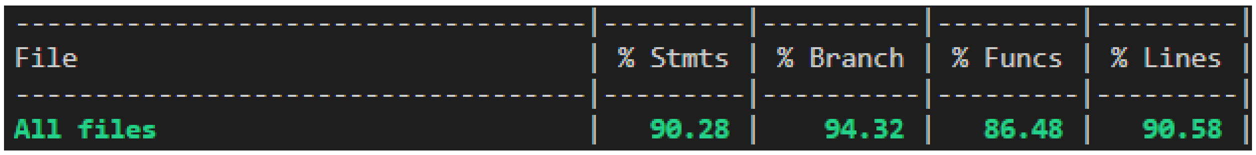
Our deployment pipeline runs a few different checks before officially deploying changes to the live application. The first check is a linting check which ensures all code is formatted correctly. After the linting check passes, our entire Jest suite is run which includes 33 sections and 161 total tests. Additionally, rules are in place to check the code coverage of tests on the backend. If less than 90 percent of the backend code is covered (line, branch, function), a test case fails, or the linting check fails, then the deployment will fail.

Integration Testing

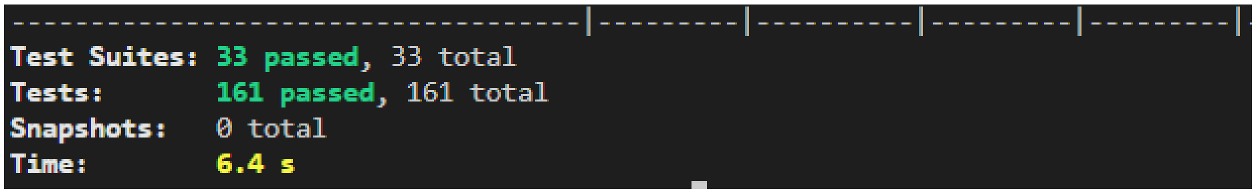
Integration testing is very important for testing our APIs, and the tool we chose to use for this was Postman. We used Postman because there is support for Rest APIs, Websockets, and there is functionality to set up automated test scripts.

Whenever developing our APIs, we start our server locally along with a local instance of our database. We then use Postman to make real API calls/Websocket connections locally, allowing us to test our work before deploying to the live application. Aside from testing our work manually, we also created an automated Postman collection which runs through all of our different APIs using the local server and database and has assertions after each API call.

Results



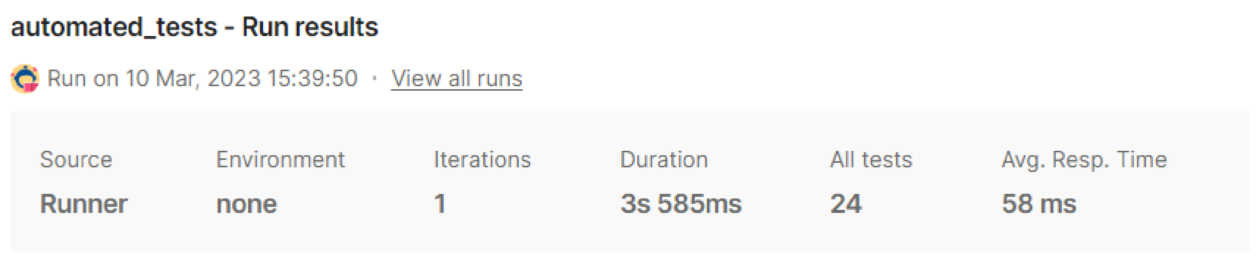
```
------------------------------------|---------|----------|---------|---------|
File                                | % Stmts | % Branch | % Funcs | % Lines |
------------------------------------|---------|----------|---------|---------|
All files                           |   90.28 |    94.32 |   86.48 |   90.58 |
```



```
------------------------------------|---------|----------|---------|---------|-
Test Suites: 33 passed, 33 total
Tests:       161 passed, 161 total
Snapshots:   0 total
Time:        6.4 s
```

Figure 5 - Unit Testing Results

Our unit testing was highly successful and helped find many problems before deployment and even before merging changes. Figure 5 above shows the results of running our Jest test suite which displays very high coverage. This is the same script being run in our deployment pipeline.


automated_tests - Run results

Run on 10 Mar, 2023 15:39:50 · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	1	3s 585ms	24	58 ms

Figure 6 - Postman Automated Tests

Our integration testing was also highly successful. Our entire backend team relied heavily on Postman when developing, utilizing Postman's API tools as well as their Websocket functionality. Postman allowed us to test our APIs/Websockets right after deployment instead of having to wait for the frontend to start consuming. Figure 6 above is an example run of our automated Postman suite which runs through our APIs and performs assertions along the way. Postman additionally helped gauge the speed of our APIs and identify any requests that take too long. Postman gives us the response time of requests so that when we were testing calls on the live version of the server we were able to see requests that take longer than others. Knowing these response times helped to identify a few slow database queries that were then optimized to improve performance.

Overall, the results of our backend testing were successful. By prioritizing testing while we developed, we caught many issues we otherwise would not have known about. We set up processes to allow us to fully run the app locally on our machines which sped up our development tenfold and vastly reduced the amount of breaking changes introduced to the live build of the app. Additionally having our automated scripts allowed us to vastly speed up checking if our code changes were breaking existing functionality.

Project in Context

Our application will affect our stakeholders and others by improving the quality of lectures for students and staff and also improve and encourage communication between professors and students. Students will easily be able to make their voices heard and professors will be able to give feedback effortlessly. Socially, our application can help introverted students. An example of how our application accomplishes this is the application's anonymous chat feature, which allows timid students to have the ability to speak to the professor during class without having the anxiety of asking the question in front of the entire class.

Competing Solutions

Since our application is classified as an interactive learning application, it was designed in a space that had other solutions. The two primary solutions that ISU students are familiar with are Piazza and Top Hat. Both of these applications promote learning through classroom interaction and communication. However, they come with a hefty price tag. Top Hat currently requires a paid subscription and students have to pay an amount for each class that uses it. Similarly, Piazza, while free for students, is quite costly for instructors. Our application would be free, as it is owned by the University, and could replace solutions like Top Hat or Piazza, saving both students and professors money.

Admittedly, other solutions offer many features that our application does not currently have. Examples include automated grading in Top Hat, an emailing feature in Piazza, among others. Keeping this in mind, we have focused our application on improving student-to-professor communication during lectures, which is something the other solutions either do not attempt to accomplish or do not accomplish well. Future teams will be able to further advance this goal as well as extend the application to be more feature-rich, putting the app in a better place to compete with Top Hat and Piazza.

In summary, we have listed the main advantages of our solution compared to others below:

- Gives live, direct feedback from students, TAs, and professors
- Allows students to ask questions during/outside class with option to be anonymous
- Has 24/7 access to past lecture discussions
- Can gather statistics for professors/TAs to use for grading purposes
- Free to use; completely “in-house”

Appendix A: Operation Manual

Frontend Operation Manual

Prerequisites

- Be on ISU's network or connect through the VPN.

Student

Setting Up

1. Go to **cyclass.ece.iastate.edu**.
2. Create an account or login if you already have an account.

Adding a Course

1. Click the plus (+) box join in Figure 7.

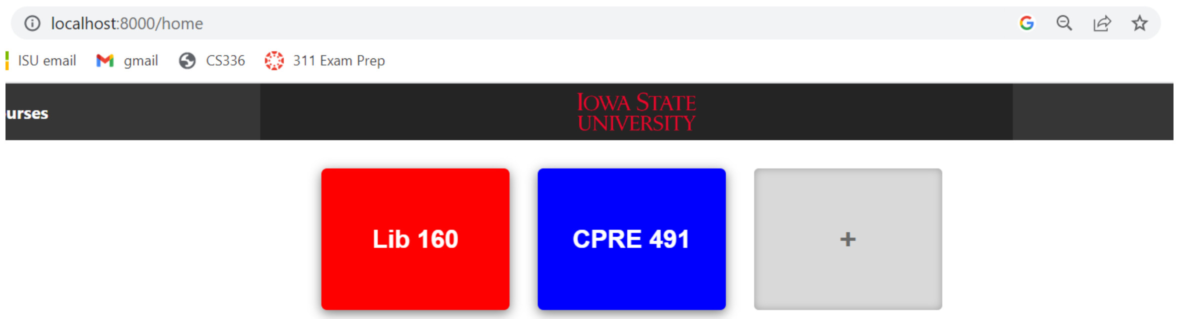


Figure 7 - Home page with icon to add course

2. Copy the join code to the entry box shown in Figure 8.

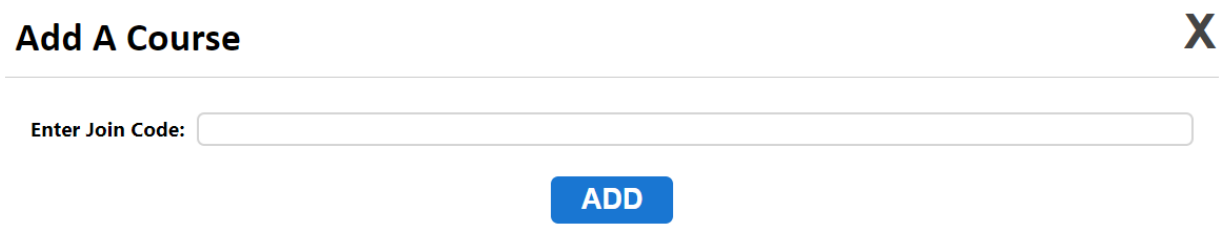


Figure 8 - Add course pop up

3. If successful, the course will now be shown on your home page.

Joining a Lecture

1. Click on the course you want to join the lecture for (e.g. Library 160).

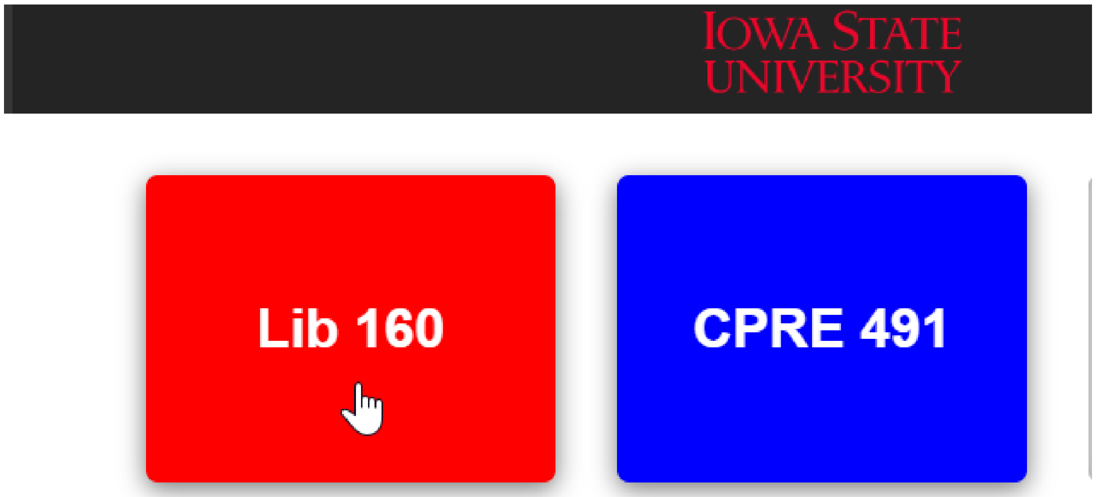
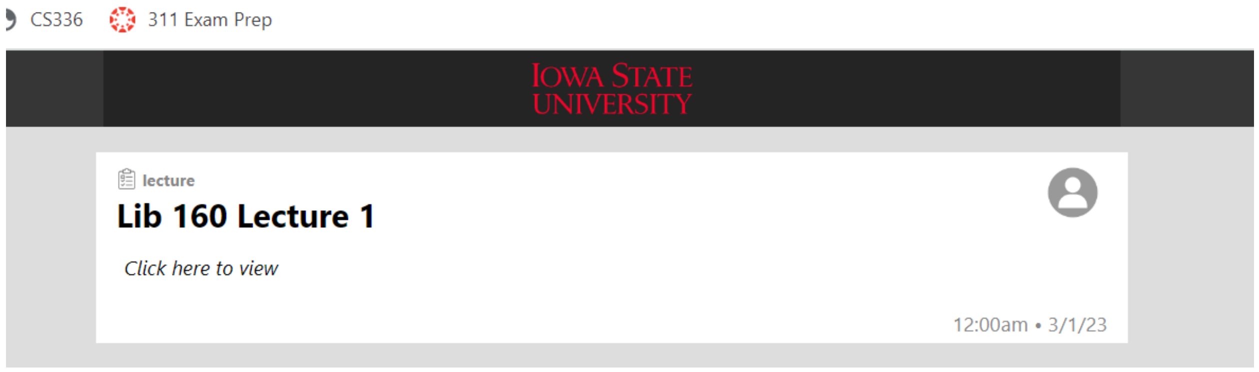


Figure 9 - Select course Lib 160

2. Click the most recent lecture, or any past lecture you'd like.



Lib 160 Lecture 1

Click here to view

12:00am • 3/1/23

Figure 10 - Select lecture in course

3. You will now see the lecture history and be able to contribute to the lecture.

Professor

Setting Up

1. Go to **cyclass.ece.iastate.edu**.
2. Create an account or login if you already have an account.

3. Check to see if you have the “ADMIN” and “CREATE” boxes, as shown in Figure 11. If not, please contact a system administrator to be granted permissions.

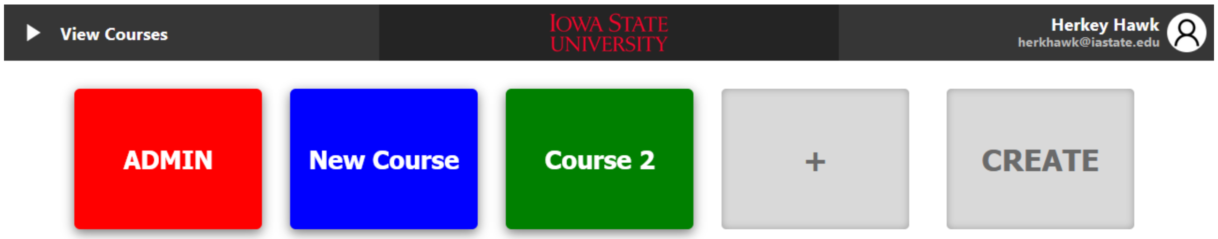


Figure 11 - ADMIN home page

Creating a Course

1. Click the “CREATE” box (shown in Figure 11) to create a course.
2. Click on the new course to enter it. The join code will be displayed at the top, as shown in Figure 12.

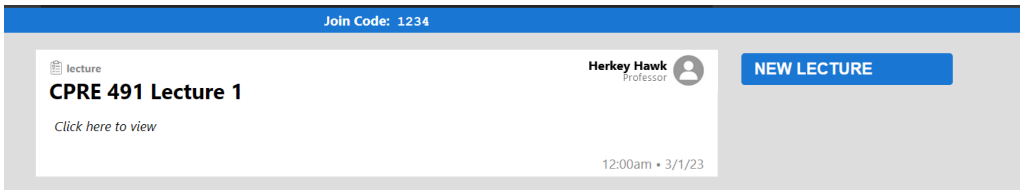


Figure 12 - Course view as professor

Creating a Lecture

1. While in the course view, you can create a new lecture using the button on the right side, as shown in Figure 12.
2. Once inside the new lecture (by clicking on it), you may post a poll or enter a message in the chat using the “POST POLL” button and message box, respectively.

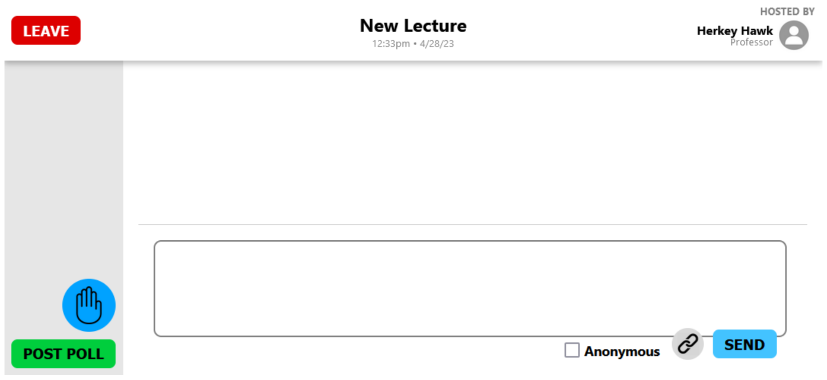


Figure 13 - New live lecture as professor

Backend Operation Manual

Prerequisites

- Docker
- Node 16
- sdmay23-40 repo cloned onto your machine
- .env file created in the /server folder with the following content:

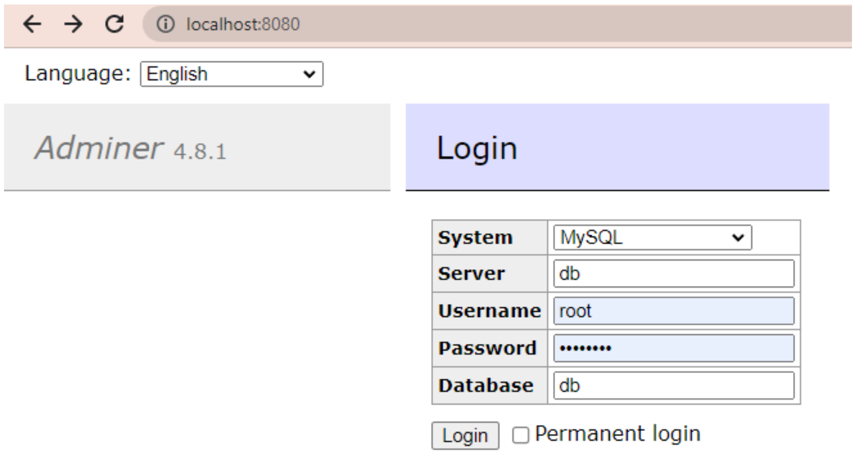
```
PORT=80

DB_HOST=localhost

DB_USER=root

DB_PASSWORD=password

DB_NAME=db

MYSQL_ROOT_PASSWORD=password

MYSQL_DATABASE=db

SESSION_SECRET=test

NODE_ENV=devl
```

Local Database Setup

Our application utilizes mysql for persistent storage. For ease of use both the live application and the local database are running within a docker container using a mysql image. The steps for starting up the mysql database locally are as follows:

1. start up the docker desktop application
2. cd into the /server directory
3. docker compose -f “docker-compose.yml” up -d –build

You will see two new containers appear in docker desktop. One of the containers is the actual database and the other is a web client that can be used to query the local database. To access the web application simply go to localhost:8080 and use the login information from the .env file. You will now be able to run the application using this local database and have access to the local database to directly test queries and modify data.



Language: English

System	MySQL
Server	db
Username	root
Password	••••••••
Database	db

Login ☐ Permanent login

Figure 14 - Adminer Client

The docker-compose.yml essentially contains instructions to create the adminer web client and the local database using a sql file as a template. There is a sql folder containing an init.sql script which the docker-compose uses to create the database as well as populate the database with some initial test data. Any modifications to the database schema can be made in the init.sql file.

Running the Express Server

After creating the local instance of the database you will now be able to successfully start the application which is essentially just an express server. You can simply start the server by the following commands:

1. cd into the /server directory
2. run **npm install** (if you haven't already)
3. Run **node server.js**

This will start the server and work, but if you made code changes you will need to manually restart the server every time. To remedy this you can install the nodemon library. This library will automatically restart the server any time you make changes to a file and save it. You can install the package and run the server with the following commands:

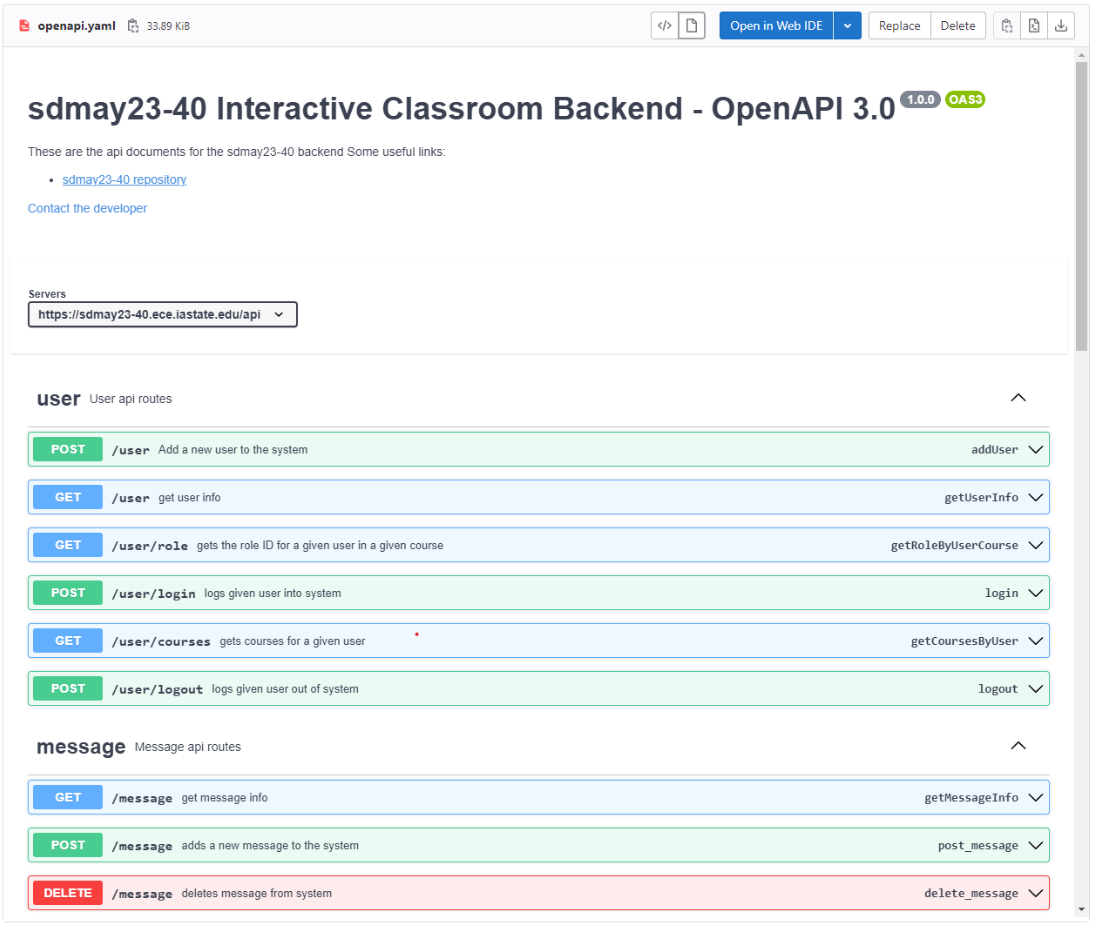
1. **npm install -g nodemon**
2. **nodemon server.js**

Both of the above methods are perfectly valid and after running the commands the server will start up locally on your machine and you will be able to hit the applications endpoints for API's, front-end content, and websockets.

Postman Usage

API

We have all of our api endpoints documented using Open API 3.0 specifications. In the repository in the root directory there is a folder called api-definitions that contains our open api spec .yml file. In the repository on gitlab if you open this file it will show an interactive page with all of our endpoints and details all information about the api. The link to the document is here (OpenApi Specs), but you will need access to the repository. The Document is quite large and you can click to open up each individual api and read more about how to make a request, but a sample visual of what the document looks like is below.



sdmay23-40 Interactive Classroom Backend - OpenAPI 3.0 1.0.0 OAS3

These are the api documents for the sdmay23-40 backend Some useful links:

- sdmay23-40 repository

Contact the developer

Servers
https://sdmay23-40.ece.iastate.edu/api

user User api routes

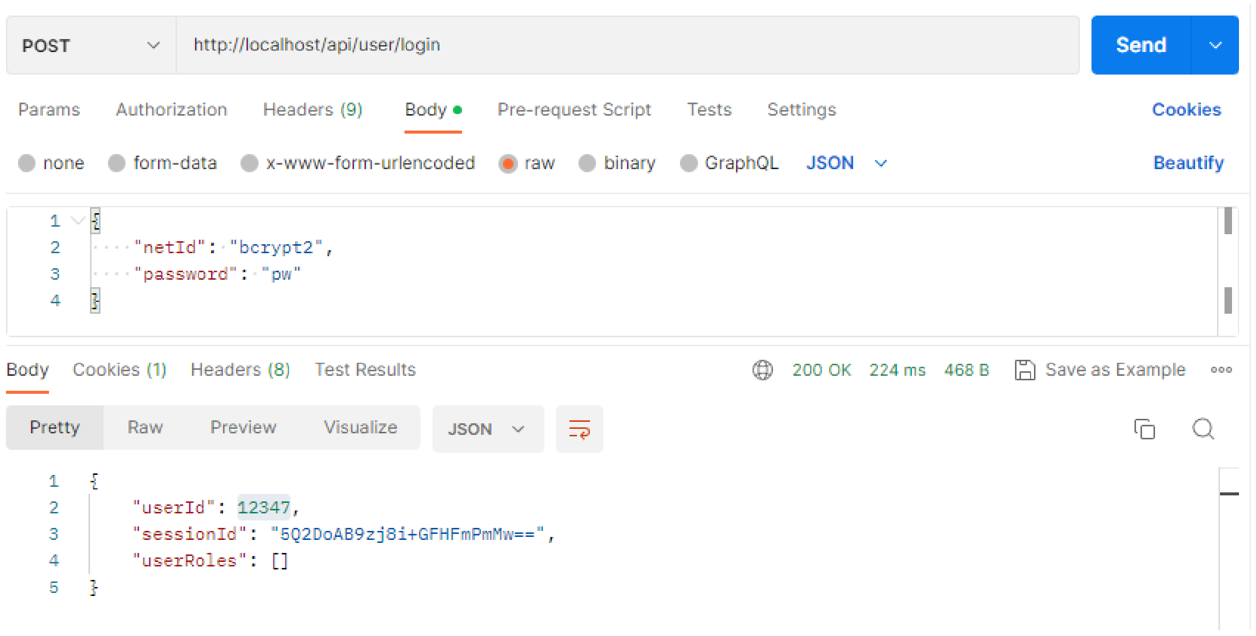
POST	`/user`	Add a new user to the system	`addUser`
GET	`/user`	get user info	`getUserInfo`
GET	`/user/role`	gets the role ID for a given user in a given course	`getRoleByUserCourse`
POST	`/user/login`	logs given user into system	`login`
GET	`/user/courses`	gets courses for a given user	`getCoursesByUser`
POST	`/user/logout`	logs given user out of system	`logout`

message Message api routes

GET	`/message`	get message info	`getMessageInfo`
POST	`/message`	adds a new message to the system	`post_message`
DELETE	`/message`	deletes message from system	`delete_message`

Figure 15 - Open API specification

To make calls to our apis you can use any tools you normally would use to make requests, but we recommend postman for this task as postman supports APIs and websockets. You can install postman, run the server and database locally and make any request as shown in the open API specs. An example is shown below:



http://localhost/api/user/login

```
{
    "netId": "bcrypt2",
    "password": "pw"
}
```

```
{
    "userId": 12347,
    "sessionId": "5Q2DoAB9zj8i+GFHFmPmMw==",
    "userRoles": []
}
```

Figure 16 - Postman Example

Websocket

OpenApi does not support websocket requests so independent documentation was made. Our websockets are how we introduce live communication on the site which is used in our lecture rooms so that users can send messages, answer polls, ask questions and much more. Whenever our application receives a websocket request the user's connection is routed to the correct lecture room so that all content will only be sent to the relevant users. There are multiple types of websocket messages we support which are described in detail in the README.md within the repository (README).

Appendix B: Prior Designs and Versions

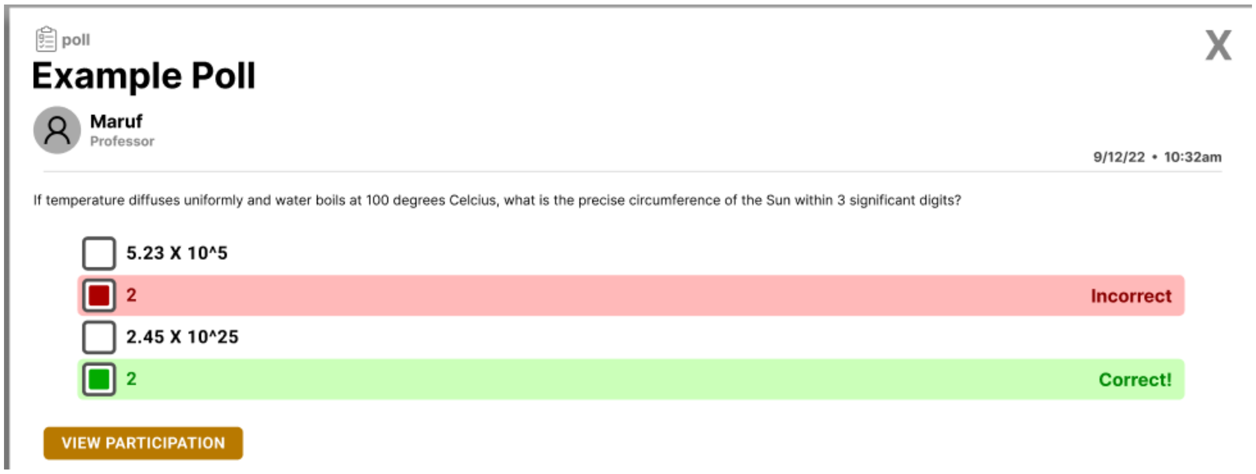
Prior Team's Project

When we first signed up for this project, it was technically in phase II. The original intention was that we would build off from the previous senior design team's work. However, from the very beginning, we were not impressed with the phase I implementation. The choice of framework and design they went with would not have allowed us to extend the application with much ease. The implementation was also very limited, and we felt we could easily reimplement the design with a more modern, versatile framework and a far more extensible design.

When discussing this with our client, he agreed, and he explained that the phase I project was not up his standards either. We decided to scrap that team's project and redo the app from the ground-up, which we believe was a wise decision and allowed us to create a much better application overall.

Poll Form

We created an initial mock up of what the poll message would look like. This was supposed to be similar to the form itself as well, as this was the only poll type pop up we had created in the initial design process.



poll

Example Poll

Maruf
Professor

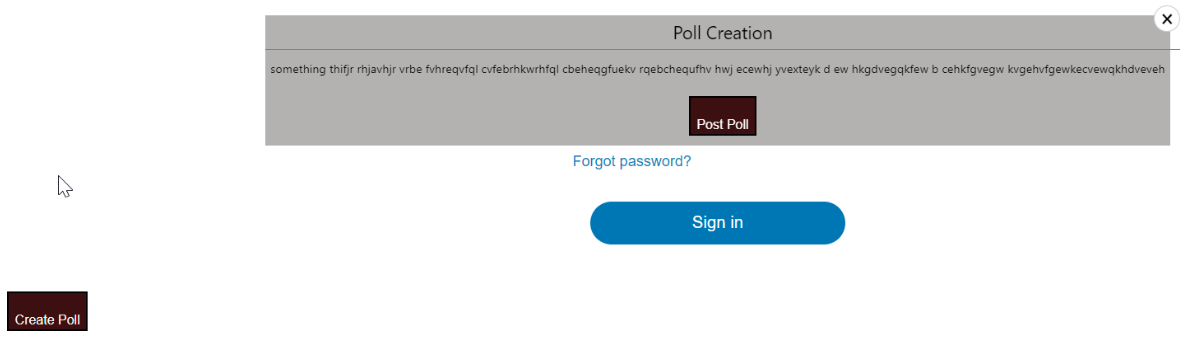
9/12/22 • 10:32am

If temperature diffuses uniformly and water boils at 100 degrees Celcius, what is the precise circumference of the Sun within 3 significant digits?

- ☐ 5.23 X 10^5
- 🟥 2 — Incorrect
- ☐ 2.45 X 10^25
- 🟩 2 — Correct!

Figure 17 - Initial Figma design

We then created an initial poll popup with this title format with a line of separation, which sort of matched the figma, but we had already gone in a slightly different direction to make it easily decipherable as a popup with the alternate color scheme.



Poll Creation

something thifjr rhjavhjr vrbe fvhreqvfql cvfebrhkwrhfql cbeheqgfuekv rqebchequfhv hwj ecewhj yvexteyk d ew hkgdvegqkfew b cehkfgvegw kvgehvfgewkecvewqkhdveveh

Figure 18 - Initial implementation of basic popup form

With strides to follow the figma more closely we updated the answer components to have a similar style to the figma with the square answer boxes and the color scheme of correct vs incorrect.

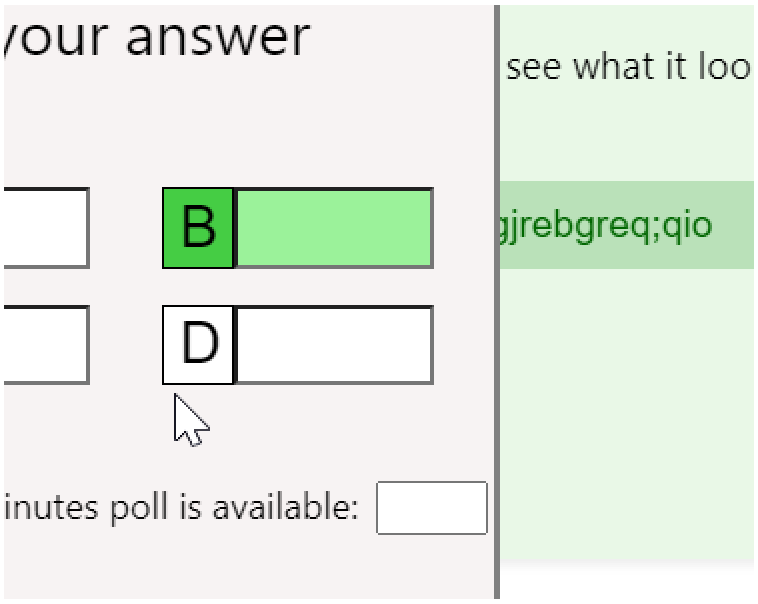
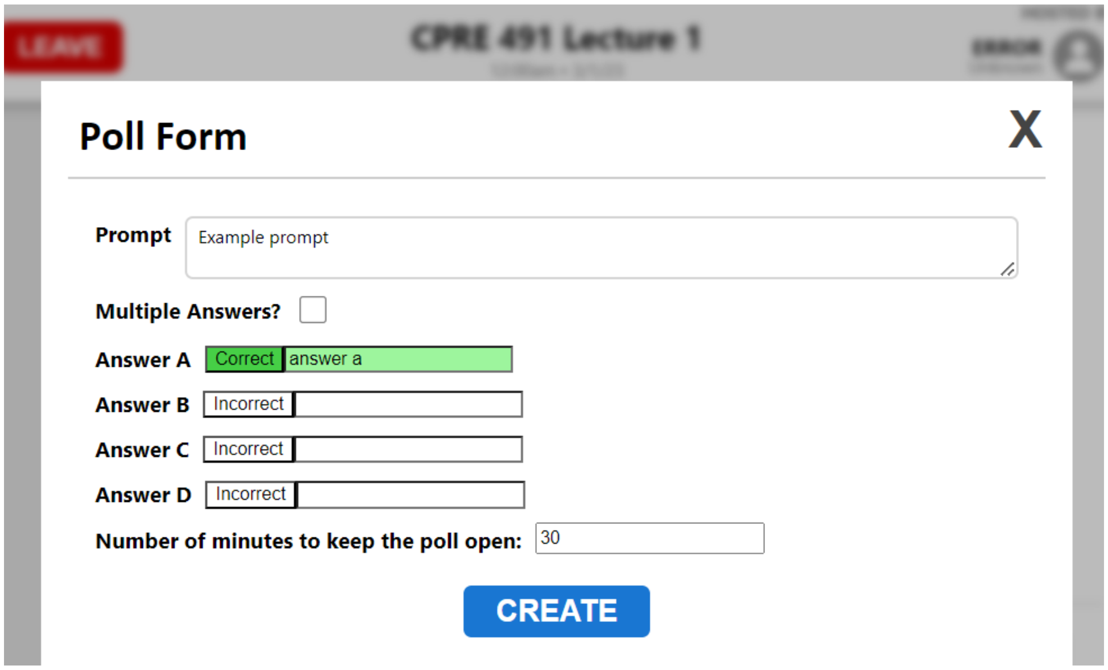


Figure 19 - Figma matching implementation

Finally, we ended up implementing a style that matches the rest of the popup designs, with everything left aligned and a uniform style and size to the components. This design also blurs the back and makes it as though the page is slightly popped up from the rest of the screen. We chose to go with this for the stylistic choices, but we also wanted to make sure everything had a similar feel to it as you move through the website.



Poll Form

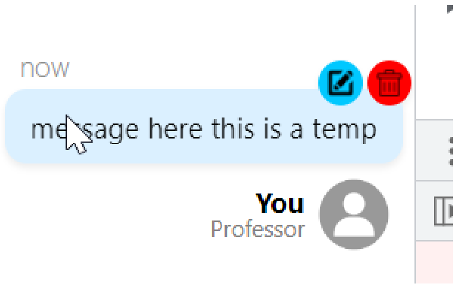
Prompt	Example prompt
Multiple Answers?	☐
Answer A	Correct answer a
Answer B	Incorrect
Answer C	Incorrect
Answer D	Incorrect
Number of minutes to keep the poll open:	30

Figure 20 - Final poll form implementation

Message Editing

Message editing was a feature that was requested at the beginning of the design process, but it was put on the backburner to allow us to focus on more necessary functionality. Once we got to it later in the semester, it was first implemented as a basic show-on-hover HTML edit button to just see if it worked.

However, with the aesthetics not matching the overall design of the page, we ended up changing this to be a small overlapping circular icon, which was much less distracting and did not affect the sizing of the element. Keeping up with the show-on-hover design, we additionally made these buttons pop-up with a clean animation whenever the user's cursor hovers over the message bubble.



message here this is a temp

Figure 21 - Final implementation of edit functionality

Filter Menu

In our initial design phase, we thought there would be a number of different message types a user could send during a lecture (e.g. a poll, a normal message, an announcement, a question, etc.). Having so many messages to sift through, we thought it would be a good idea to add some type of filtering to make it easier to find messages or at least narrow down what the user is looking at. We decided on a filter box that is pictured below in Figure 22.

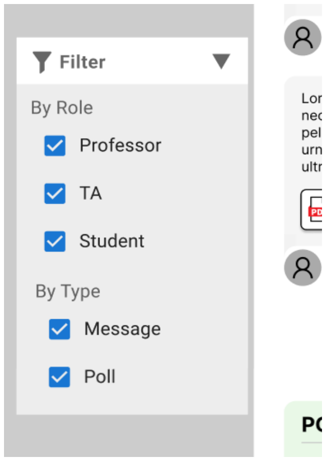


Figure 22 - Initial filter box design

As we began to implement our design, we noticed the lack of necessity for alternative types of messages besides a normal message and a poll. With the poll acting as both a poll and a multiple choice question, as well as it being much larger than a regular message, we didn't see the need to filter for it, as it is extremely easy to find a poll within a lecture. Seeing that this feature wouldn't have that much of an impact, we chose against including it in our final implementation.

Appendix C: Other Considerations

TypeScript

We chose to use plain JavaScript over TypeScript due to the fact that some members on our team had very little experience with TypeScript. Had we chosen to use TypeScript despite this, our development likely would've slowed to a crawl for these members, and it is doubtful that we would have been able to accomplish all that we did. By using plain JavaScript which all of our team members were comfortable with, we were able to focus more on implementation and less on learning a new language.

Cloud Solution

Early on, we considered the possibility of implementing our application in the cloud using a service such as AWS. We decided against this since ISU already has a system in-place for running VMs for these sorts of projects. These VMs are simple to use, have flexible computational capabilities, and, most importantly, are free. Additionally, with how our application has been built, converting it to a cloud application would not be too difficult, as we already have containerized our database and limited the state that the server maintains.

Lessons Learned

There were two noteworthy lessons that we learned as a team from this project. The first was the value of early style planning / coordination through common components. The second was the importance of focusing on really polishing core features before moving to additional features.

While the frontend did try to spend our early weeks making general components, we quickly moved away from this in favor of jumping right into making the various pages that our client was wanting. After a couple months of this, we ended up having a very discontinuous style across the app. Ideally, the app should look as if “one mind” made it. Had our team spent more time planning and creating generic components with a continuous custom style, this likely could have been avoided.

Early in development, we also were planning on getting through our core feature set relatively quickly and then moving on to additional features. However, we ended up finding that doing so would result in an application that would have a lot of features that are all rough, buggy, and/or hard to use. We instead decided to focus on the core feature set for the entire semester. The end result of this was a far more usable and polished application that does what it does very well.

Appendix D: Code

Link to project on GitLab