Part B Project Proposal

Project Name: BruinNotes

Team Name: DJ-JAYS

Team Members: Jinwoo Baik (705378164), Dylan Phe (505834475), Smayra Ramesh (905206685), Aiqi You (405557435), Jack Zhao (405377811), Yunfan Zhong (605313390)

Github: https://github.com/dylanphe/cs130project

1. Design

For the last few weeks, our team has made substantial progress in implementing the BruinNotes application. The development teams have been working in parallel on various sections of the project, which we will go into more detail below.

1.1 Progress on Features

The table below contains information about the current progress of each main feature of our application, as well as our plans moving forward in Part C.

Feature	Current Progress	Future Plans
Account System	Account System UI/UX are completed with workable signup validation, login authentication, and forget password process. The backend API is also complete. User information can be stored and retrieved from the MongoDB database.	Add in password hashing and cookie storage to keep users logged in across pages.
Search System	Started to build the webpage and routing to create a new class page. Created general user interface mockup as well as general working page that has a search bar functionality	Implement some backend connection with search bar when new inputs need to be added to the options users can choose from and make aesthetic same across pages
Course System	The Course System webpages are up and running with workable UI/UX to sort notes and navigate through different professors, and quarters for each course.The communication between backend is still a work of progress.	Implement the backend data model for CourseBucket, CourseData, as well as, APIs to fetch their information and update new classes from user input. Implement responsive design.
Sharing and Requesting Notes	Interface for sharing and requesting forms are both implemented as modals on the course page. User inputs are gathered in the frontend.	Set up communication with the backend to fetch existing notes and requests, update user-inputted notes to the database, and display them on the course note page.
Likes, Dislikes, Comments	Interactive like and dislike buttons are implemented next to the note links. Show comment button, comment list, and input comment field are displayed.	Implement connections to the backend to update and display user inputs.

Figure 1: Progress on various features for BruinNotes.

1.2 Class Diagram

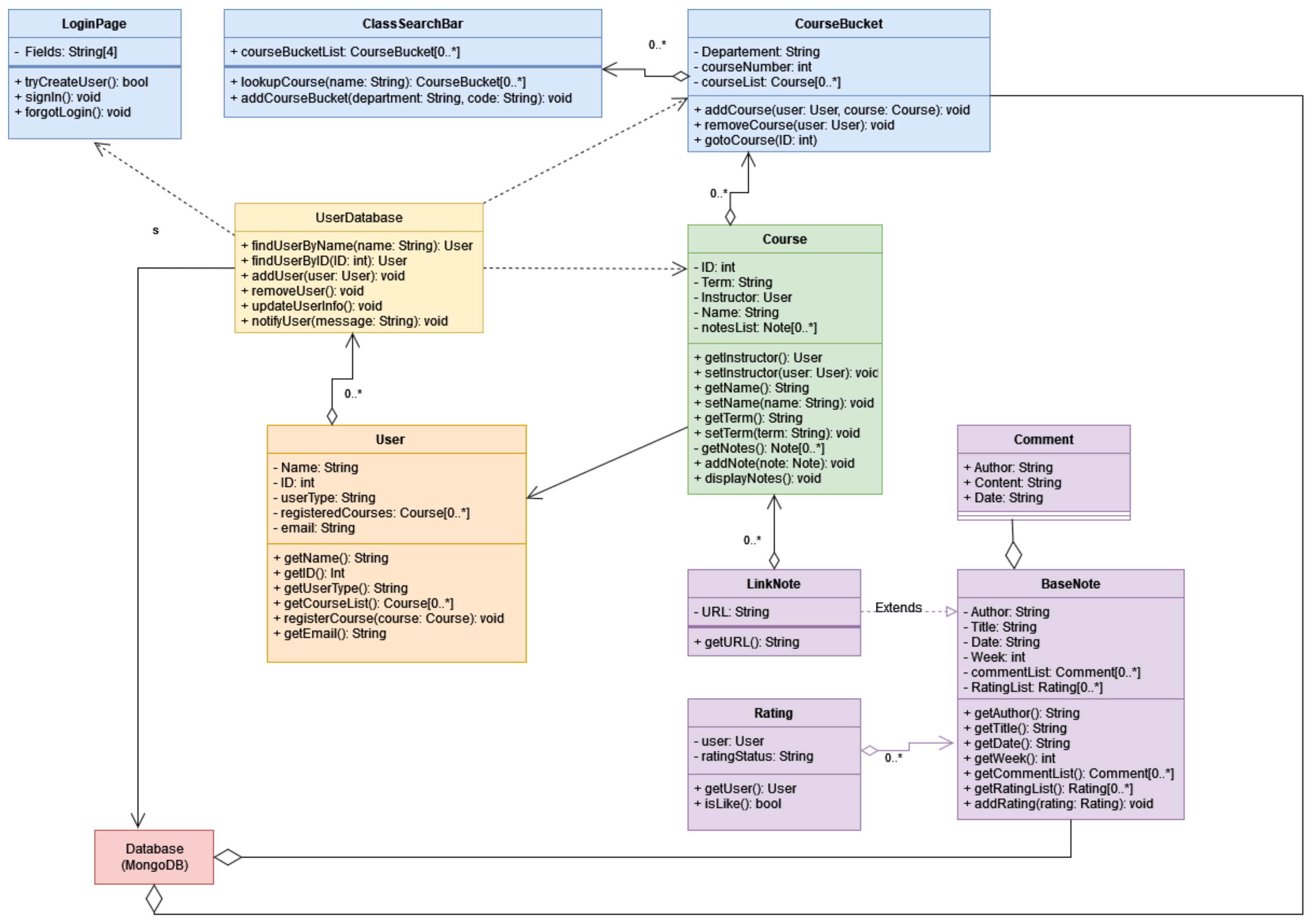


Figure 2: Class diagram for BruinNotes.

1.3 Libraries and Packages

The following table describes all of the major libraries and packages our project uses, as well as their relevance to the project.

Library/Framework/API	Description	Relevance to Project
FastAPI	Python web framework for building high performance, intuitive APIs.	Used to build the backend of the project, and create an API for the frontend to call.
MongoDB	Database that uses JSON-like documents to store data.	Used to store application data (like user info), linked to the FastAPI backend.

PyMongo	Python distribution containing tools for working with MongoDB from Python.	Used to interact with the MongoDB databases from the Python FastAPI backend.
Pydantic	Python library for data validation and type annotations.	Used to define and enforce data types, with helpful error handling.
Typing	Python library that provides runtime support for type hints.	Used to enforce types at runtime, and to make the code more clear and explicit.
Motor	Python library that provides an asynchronous API to access MongoDB.	Used to handle requests from the FastAPI backend to MongoDB in an asynchronous, non-blocking manner.
React	Javascript Libraries for building user interfaces.	Used to build different user interfaces of different web pages of our site.
React-Router-dom	React package for routing through different web pages of the website.	Used to navigate through different web pages with an event onClick of different buttons.
React-axios	React HTTP client library based on promises.	Used to send asynchronous HTTP requests to REST endpoints.
React-Bootstrap	React Library for building different components of user interfaces.	Used to build modal for sharing, adding, and requesting buttons, as well as input forms.

Figure 3: Libraries and packages for BruinNotes.

1.4 User Interface

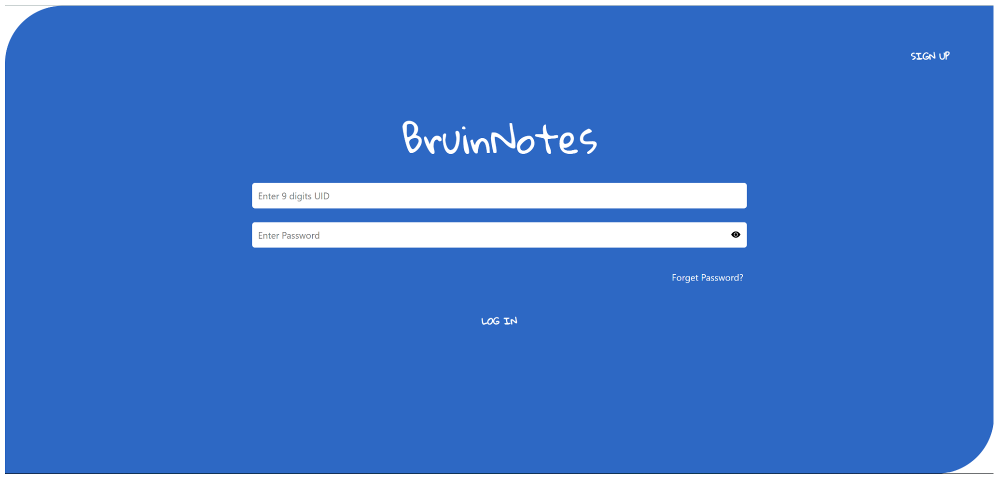
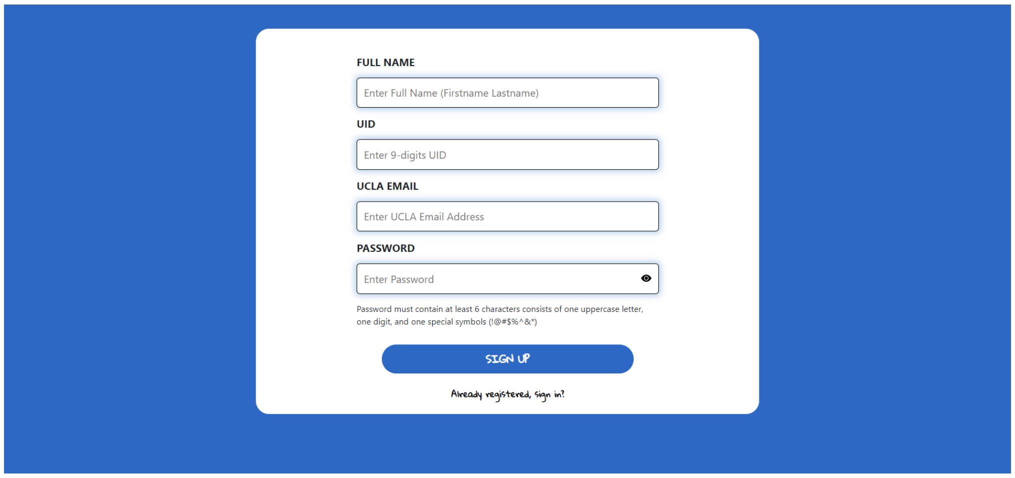


Figure 4: BruinNotes login page.


FULL NAME

Enter Full Name (Firstname Lastname)

UID

Enter 9-digits UID

UCLA EMAIL

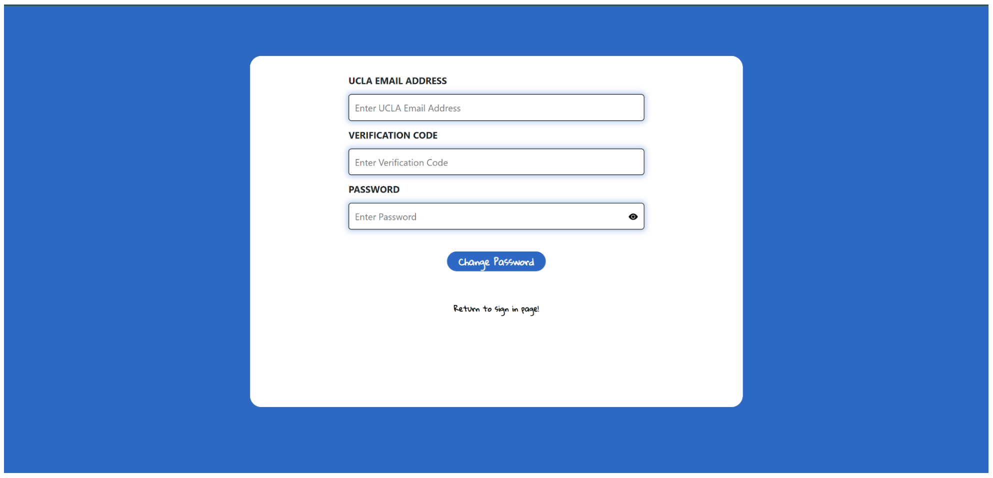
Enter UCLA Email Address

PASSWORD

Enter Password

Password must contain at least 6 characters consists of one uppercase letter, one digit, and one special symbols (!@#$%^&*)

Figure 5: BruinNotes sign up page.


UCLA EMAIL ADDRESS

Enter UCLA Email Address

VERIFICATION CODE

Enter Verification Code

PASSWORD

Enter Password

Figure 6: BruinNotes forgot password page.

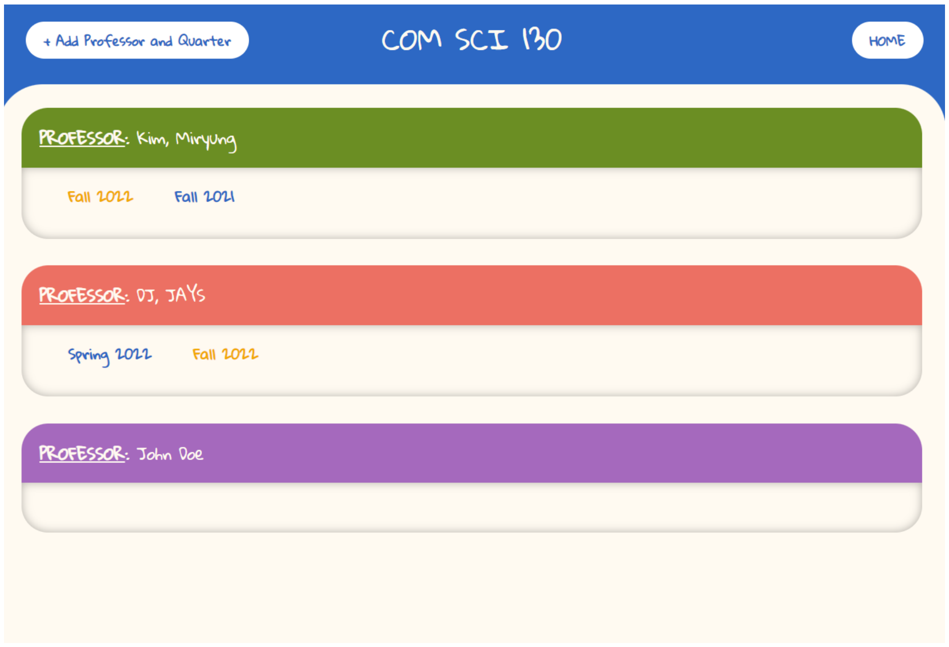
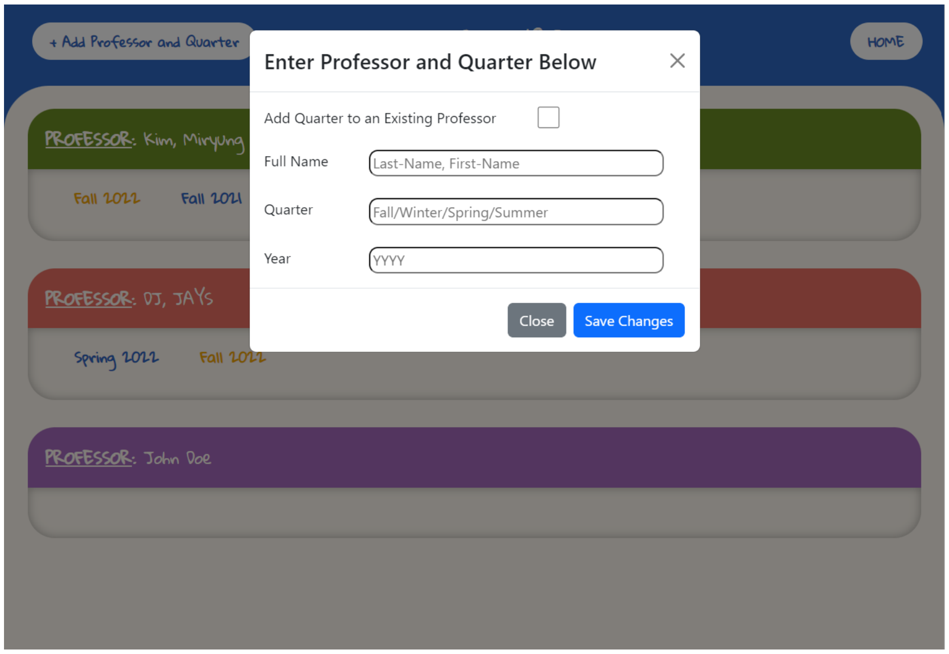


Figure 7: BruinNotes course page.


Enter Professor and Quarter Below

Add Quarter to an Existing Professor

Full Name: Last-Name, First-Name

Quarter: Fall/Winter/Spring/Summer

Year: YYYY

Figure 8: BruinNotes Add Professor and Quarter popup..

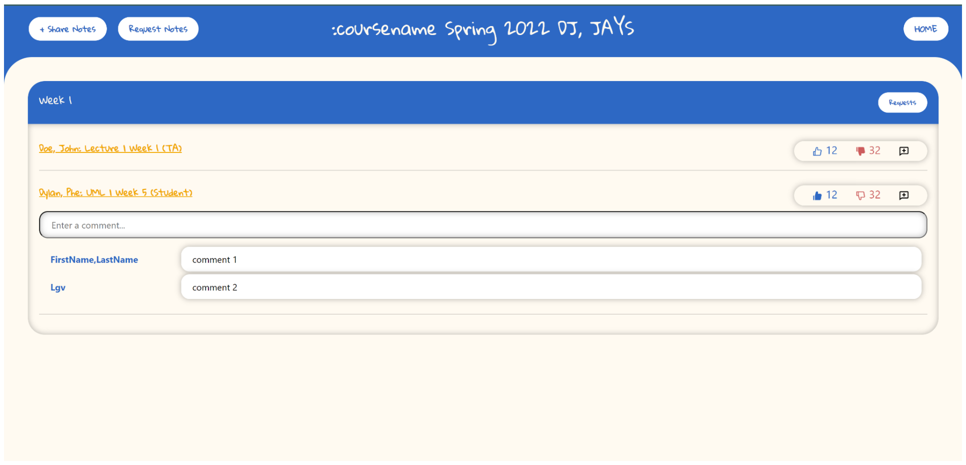


Figure 9: BruinNotes View Notes page.

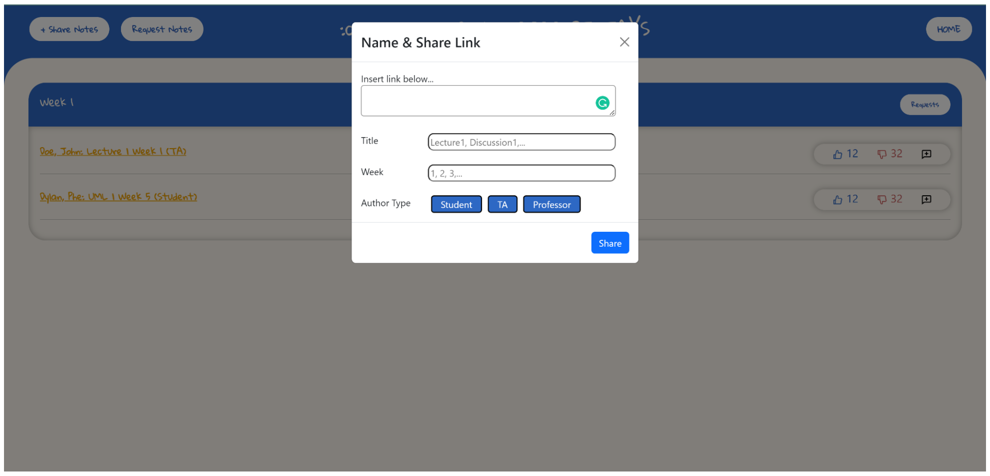


Figure 10: BruinNotes Add Notes popup.

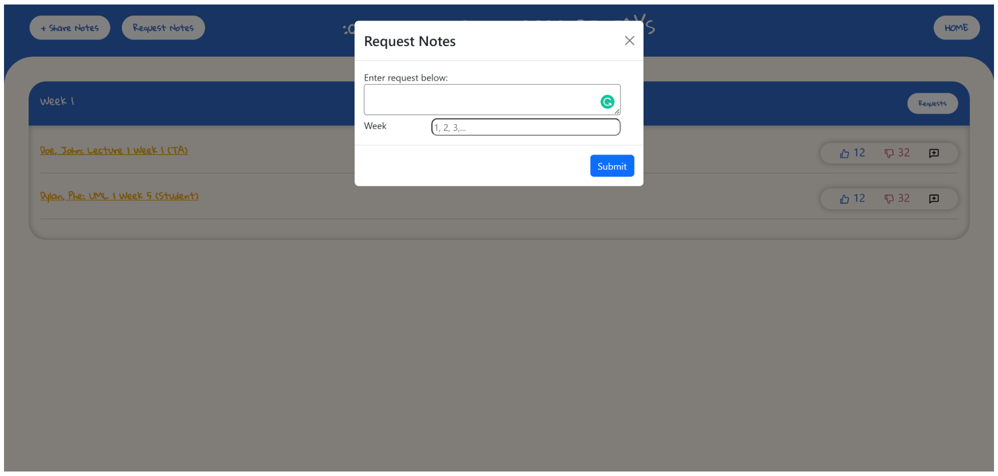


Figure 11: BruinNotes Request Notes popup.

2. Description of API

2.1 API Documentation

Account System API ⇒ Database	
Route	**Input / Output**
POST /adduser	Input: User information (name, email, etc.) / Output: JSON object with new user and HTTP 201 created status
GET /viewallusers	Input: None / Output: List containing all users in database
GET /checkuid/{uid}	Input: UID / Output: True if UID is unique, False otherwise
GET /checkemail/{email}	Input: email / Output: True if email is unique, False otherwise
GET /viewuser/{username}	Input: Username / Output: User with matching username or 404 Not Found
POST /checkpassword	Input: UID and password / Output: True if password matches, False otherwise
PUT /updateuser/{username}	Input: Username and updated user information / Output: Updated user with updated information
DELETE /deleteuser/{username}	Input: Username / Output: Success or failure

Figure 12: BruinNotes account system API documentation.

Information Hiding

The account system API hides all of the internal information on how the backend handles user data, and how the database stores it. While designing the system, we identified the UID as a way to uniquely identify a user, and noted that this could be exposed in the API interface because it was unlikely to change. The output information is also generic enough to be unlikely to change, while still providing necessary and useful information to the client. The way that the backend internally processes and stores user information is hidden from the client, because those mechanisms are the most likely to change in future redesigns.

By using the concept of information hiding, we are able to make changes to modules in a very isolated way. For instance, if we need to split the user information across two databases, it will not require changes to the client. Another potential change concerns whether the database indexes users by UID or email, but we can easily switch between the two without editing the client. All the frontend knows is that it is sending a request containing the user's unique UID, and the backend responds with the relevant data. Moving forward, we will continue to use information hiding as the guiding principle for our module design.

Documentation

In our server code, each of the API functions has a docstring containing a concise description of the function, the input parameters and types, and the return value. An HTML webpage documenting the API can be found here. The documentation contains information about each of the API endpoints, as well as the classes that are used to structure and parse data.

2.2 Sequence Diagram

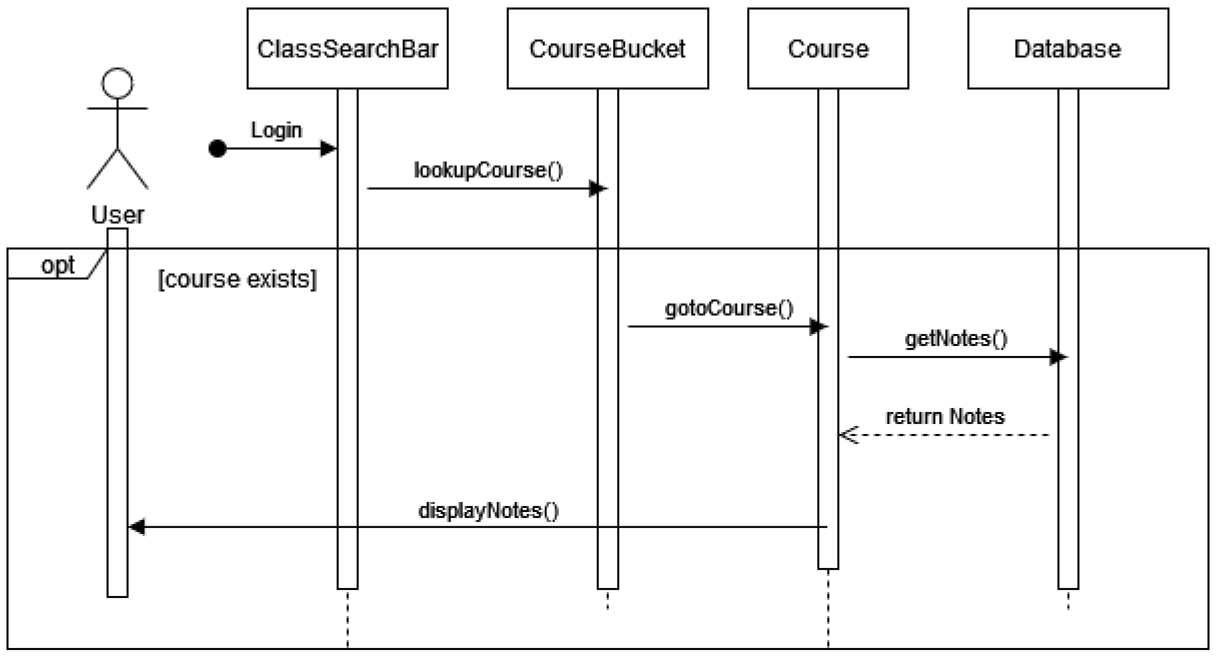


Figure 13: Sequence Diagram for Viewing Existing Notes of a Course

3. Testing

To test the functionality of BruinNotes, we have created this tests directory in our repository that will be used in Part C as a centralized place to hold all our test cases. In our current state of development, our testing has been more manual, to get the basic functionality of our APIs and pages set up and properly connected. For the next stage of our project, we are planning to add tests for all the main features into the above directory. In this next section, we will discuss the testing we have done thus far in our project.

3.1. UX/UI and Frontend Testing

BruinNotes aims at providing simple interfaces for users to share and request notes. Therefore, most of the testing that was done in the frontend development focused heavily on the scaling of the webpages on different devices of different aspect-ratios. To do this, we made use of the “toggle device toolbar” that is available in the inspect mode of any Google Chrome web browser. However, our web applications do not support mobile device aspect-ratios. Other aspects of testing in frontend will be described in detail below within each feature section.

3.2.1. Account System (Signup)

Like every other account system web application, BruinNotes requires user input validation and user authentication. To set up an account, a user must provide:

1. A full name consisting of a first name and a last name, which corresponds to `RegEx = "[A-Za-z]+ [A-Za-z]"`
2. A unique UID consisting of 9 digits (`RegEx = "^\\d{9}$"`)
3. A unique UCLA email address, which is validated using another regular expression (`RegEx="^[\\w-\._]+@([\\w-]+\.)+ucla\.edu$"`)
4. A password that consists of at least 6 characters of at least one uppercase letter, one digit, and one special symbol (`!@#$%^&*`) that equivalent to

`RegEx="^(?=.*[0-9])(?=.*[A-Z])(?=.*[!@#$%^&*])[a-zA-Z0-9!@#$%^&*]{6,16}$"`

To verify the accuracy of our Regular Expressions listed above, we used RegExr, an online tool to build and test a RegEx. To test out each object validation, we used `console.log()` and `alert()`, to output the boolean result for observations on different user input case scenarios. One test scenario was when a user clicks on a sign up button on the Signup Page. If the inputted information matches the regular expression, then the account is created. Otherwise, an error is displayed. Therefore, we checked our test case scenario to see whether they are consistent with the result logged into the console when a user creates an account.

Checking the uniqueness of the UID and the email address requires communications between the frontend and backend through the calling of two APIs, which are *GET /checkuid/{uid}*, and *GET /checkemail/{email}*. These APIs check through the database and return boolean values as responses to whether or not they are unique. Therefore, we have to check for a scenario when a user attempts to sign up for an account using the same email address or the same UID as an existing user. To test the frontend, we made use of `console.log()` and `alert()` to observe correct behavior on two different user input case scenarios of duplicate email/UID and unique email/UID. While writing each of the backend APIs, we did incremental development paired with extensive manual testing. This ensured that each of the functions was working properly, and the changes were being propagated to the MongoDB database.

The success of signing up relies on the POST */adduser* API that adds the inputted information into the database. Therefore, we can check MongoDB to check whether the information in the database is updated.

3.2.2. Account System (Login)

For login, we must verify if the UIDs and the passwords inputted by users match the one they used to create their accounts for authentication purposes. To do this, we made use of the POST */checkpassword* API. This API takes in a dict consisting of the inputted user information, and checks whether it matches the information in the database. If so, it returns true and the login attempt is successful. Otherwise, it returns false and the login fails. Like before, for testing, we used `console.log()` and `alert()` to test the two scenarios that can happen (password matches or doesn't match) for both frontend and backend testing.

3.2.3. Account System (Forget Password)

In a third possible scenario when users forget their password, then the user must go through a process to verify their email and change the passwords for their accounts. The testing is done through checking whether an email is sent on the send verification button clicked, and if the password value is updated in the database, if the users successfully verified their email. We made use of Dylan Phe's UCLA email to do that.

3.3. Search System

For the search system we allow users to search for the class that applies to them. There will be a dropdown menu that the user can interact with and choose the class from a series of options and possibly search for them too as the list gets long. Each time a user needs to add a new class to the search system, they will be able to enter a new page to add the item into the list.

If the item exists already it will be compared against a list of existing classes that have been entered in the system and not allow for the user to enter that class as a "new class."

3.4. Courses System

The course system frontend is tested manually with relevant json data defined in the frontend and manual user inputs.

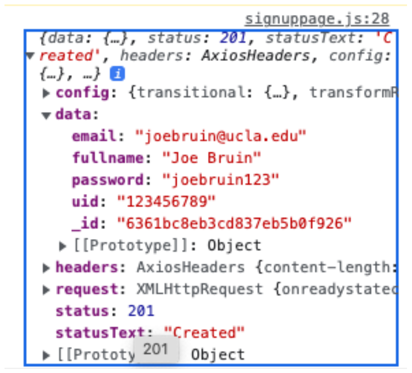
For displaying the course data, one test case scenario is no professor or term is recorded for the course. The outcome for test success is displaying a message that encourages the user to add a professor to that course. Another test case scenario is that a professor exists for the course, but no term is associated with that professor. The outcome for test success is displaying the professor but no term below it, as demonstrated in the course page screenshot in section 1.4. For the case where professors and terms are present, the successful outcome is displaying both professors and their terms as indicated in the screenshot; if more than one professor is displayed, each professor's background color is assigned iteratively among 6 available colors. When the backend communication is implemented, the expected outcome should be correctly displaying all course data on the webpage. For each test case above, outcomes for test failure include incorrect rendering of the page, incorrect display of information, mismatching colors, as well as error or warning messages in the console.

Tests that involve user interactions include pressing buttons, toggling popup modals, and user inputs. One test case scenario is the user clicks the "add professors and quarter" button on the course page, then a popup form is displayed, and then the user clicks somewhere outside the form or clicks the close button on the form. The successful outcome is the form pops up after the first click and disappears after the second click. In the scenario that the user fills out the form and clicks the submit button with all required and valid input, the expected successful outcome would be displaying the newly inputted information in descending order of time. A failed outcome would include unsuccessful rendering of the page, unsuccessful closure of the popup modal, incorrect display of the new information, as well as errors and warnings in the console. In that form, the user can choose to enter a new professor or enter a quarter for an existing professor. A successful outcome would also require these two choices to be mutually exclusive. In the case that the user submits missing or invalid information, such as inputting 202.2 for the year, the expected outcome would continue to show the popup form with all existing inputs and highlight the missing or invalid fields. Outcomes of test failure include disappearance of the popup and the inputs, lack of indicators for invalid fields, and form submission with erroneous inputs.

3.5. Frontend and Backend Communication Testing

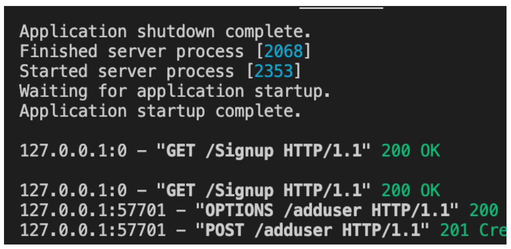
Our web application utilizes RESTful API's to communicate from the React frontend to the FastAPI python backend. Once the backend receives data, such as user information, we can insert that data into our MongoDB cluster.

In order to test this data communication from client to backend to MongoDB, we have created logs along every step. Client logs the post request in the browser console, FastAPI backend logs the HTTP status code and content in the local console, and MongoDB data can be viewed directly in the GUI. The following figures show the process we take for testing.



```
                                    signuppage.js:28
 {data: {…}, status: 201, statusText: 'C
▾reated', headers: AxiosHeaders, config:
 {…}, …}
 ▸ config: {transitional: {…}, transformR
 ▾ data:
     email: "joebruin@ucla.edu"
     fullname: "Joe Bruin"
     password: "joebruin123"
     uid: "123456789"
     _id: "6361bc8eb3cd837eb5b0f926"
   ▸ [[Prototype]]: Object
 ▸ headers: AxiosHeaders {content-length:
 ▸ request: XMLHttpRequest {onreadystatec
   status: 201
   statusText: "Created"
 ▸ [[Prototy 201  Object
```

Figure 14: Client console

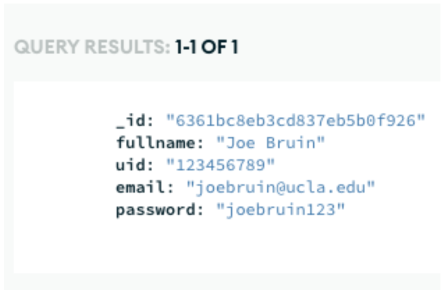


```
Application shutdown complete.
Finished server process [2068]
Started server process [2353]
Waiting for application startup.
Application startup complete.

127.0.0.1:0 - "GET /Signup HTTP/1.1" 200 OK

127.0.0.1:0 - "GET /Signup HTTP/1.1" 200 OK
127.0.0.1:57701 - "OPTIONS /adduser HTTP/1.1" 200
127.0.0.1:57701 - "POST /adduser HTTP/1.1" 201 Cre
```

Figure 15: Server console



QUERY RESULTS: **1-1 OF 1**

```
_id: "6361bc8eb3cd837eb5b0f926"
fullname: "Joe Bruin"
uid: "123456789"
email: "joebruin@ucla.edu"
password: "joebruin123"
```

Figure 16: MongoDB GUI

4. Contributions

Jinwoo Baik helped with the initial setup for the project, by investigating the possible frameworks that could be used to communicate between the frontend and backend. In particular, after some research he set up the basis for the FastAPI Python backend, which communicates with the frontend with GET and POST messages. This is the basis for the current FARM stack of FastAPI, React, and MongoDB. He is currently working on the backend logic on how to deal with the addition and management of courses, and making sure the frontend and backend agree on the functionalities that they implement.

Dylan Phe set up the react frontend components for the web application where he created the required webpages and installed React-Router-Dom for routing through different web pages. He also implemented the user interfaces for the account system and worked with Aiqi to design and implement the user interfaces for the course system webpages. For the account system, he worked on the UIs of the users inputs validation, the uniqueness of the inputted uid and email address, as well as to see if the inputted uid and password matches for their account during account creation and authentication process through the use of React-axios for the communication between frontend and backend. For the course system, he helped Aiqi with the design of the webpage and the UI within those web pages such as the note sharing, requesting, like, dislike, and comment sections.

Smayra Ramesh helped to create the UI groundwork for the design and flow of the applications. She also created mocks to use as a basis for the current app, although it has gone through multiple design iterations from the original mock up design work. For the UI front end components, she is helping to implement the search system in order to be able to both input information into the options of classes as well as choose from an existing set of classes others have already created. Worked with Aiqi to determine the best way to implement search page components.

Aiqi You worked on the course pages, with intensive help from Dylan in the UI design. She set up routing to these pages and used mock-up data to display relevant course information. She used the react-Bootstrap library to implement popup modals for input forms. For part C, she will continue to work on input validation, communication with the backend, and the report feature.

Jack Zhao worked on setting up react frontend communication with FastAPI backend. Using 'Axios' library, he was able to successfully incorporate POST requests in the frontend. He started with the Signup page. He created states to save the user inputs for account creation and passed that information to the backend through a POST request. In the backend, he was able to create and append user classes from the frontend to MongoDB.

Yunfan Zhong set up the MongoDB database for the project, and connected the FastAPI backend to MongoDB using PyMongo. She also created a base model for the user class, as well as a model for updating the user's information. To create a basis for coding all future API endpoints, she wrote functions to add a new database item, view all items, search for a specific item, update an item, and delete an item. As she was writing each of these functions, she also did extensive manual testing to ensure that they worked properly, and the changes were showing up in the MongoDB database. She also created three endpoints that were specific to the account system. The first is to check that an inputted password matches the stored password for a given user. The other two are used to ensure that UIDs and emails are unique to each user, by checking that they do not already exist in the database.