McGill University

Department of Computer Science

COMP 693/ 694/ 695 Research Project

Development of McGill UAsk

A Web-Based Communication Platform for the School of Computer Science

---

Summer Term 2024

Supervisor: Prof. Joseph Vybihal (joseph.vybihal@mcgill.ca)

Student: Yang Kai Yam (kai.y.yang@mail.mcgill.ca)

Project Duration: 13/5/2024 – 31/8/2024

Abstract

McGill UAsk is a full-stack web communication platform that enhances student and faculty interaction at McGill University. Inspired by the functionality of platforms like Ed and Slack, UAsk offers a robust discussion board tailored to the academic environment. The platform integrates essential features such as course management, public and private discussions, and a real-time notification system, all built with a user-friendly and responsive interface that supports light and dark modes.

Our project addresses the limitations of existing solutions by combining their strengths and introducing new features, such as post-management tools, user authentication, course invitation codes, and email verification systems. Leveraging the MERN (MongoDB, Express, React, Node.js) stack, McGill UAsk provides a scalable and secure environment for academic collaboration. The system has been tested across various user roles, ensuring a seamless experience for professors, teaching assistants, and students. It is a module-based design, ensuring that each component is independent, facilitating the future maintenance and enhancements.

This report details the design, implementation, results, and maintenance of the McGill UAsk project, highlighting its potential to improve communication and resource sharing within the university setting significantly. The source code of McGill UAsk can be found at GitHub repository (https://github.com/alan5543/mcgill-cs-chat), and its demonstration can be found here (https://youtu.be/bel31lMmFdE).

Contents

Introduction

Before indicating the infrastructure of the web application, this part will introduce the background and motivation of the project, as well as the design principle about how it can target our goals.

a. Project Motivation

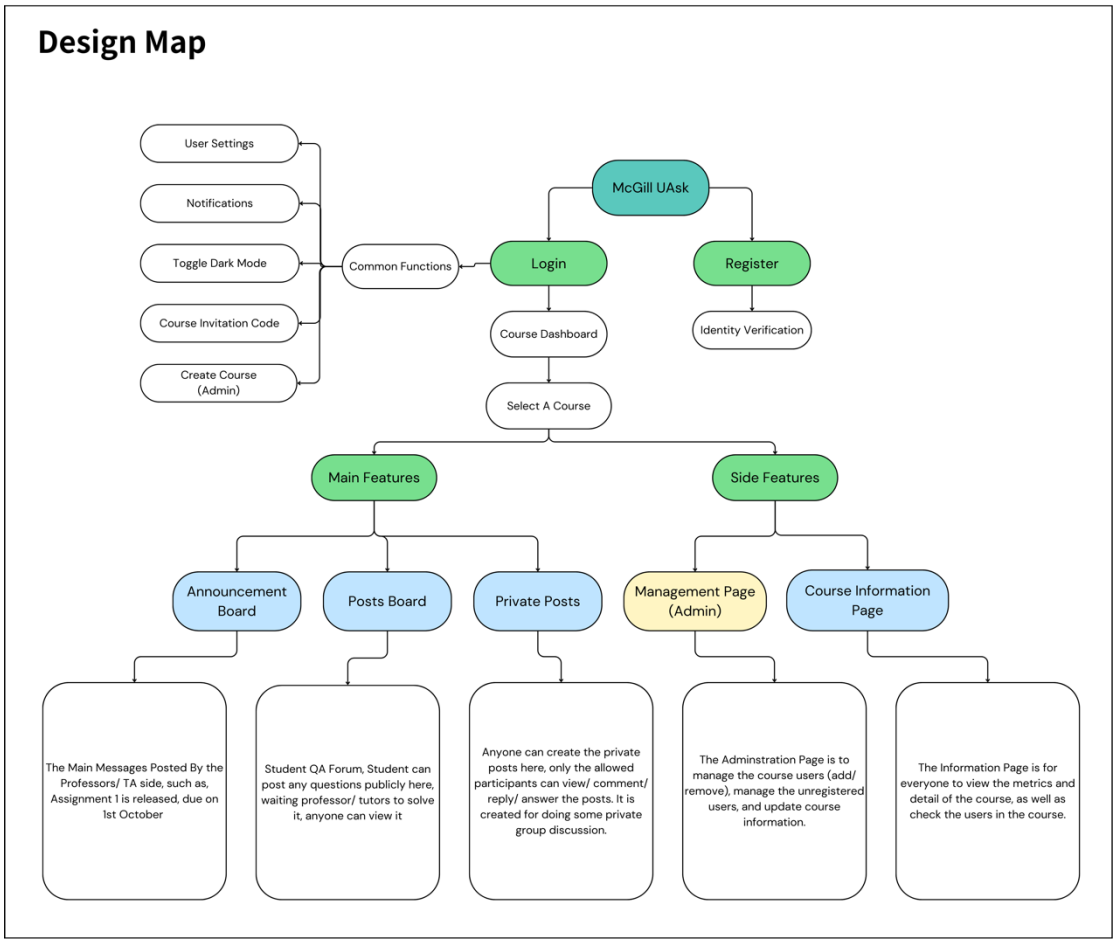
The motivation for the McGill UAsk project stems from the need for a communication platform that is better suited to the academic environment at McGill University. Existing solutions like Ed and Slack, while effective in their respective domains, do not fully address the unique needs of a university setting. Ed provides a structured Q&A forum with discussion threads and category filters, while Slack offers real-time, channel-based communication. McGill UAsk aims to combine the strengths of both platforms, integrating structured discussions with real-time communication features. By introducing new, tailored functionalities, the platform is designed to be scalable, secure, and user-friendly, enhancing collaboration and interaction among students and faculty, and fostering a more engaged and dynamic academic community.

b. Project Background on Existing Solutions

Referencing the existing solutions from COMP 307 [1], there are three projects: *myThreads*, *SOCS Boards*, and *Pi. myThreads* is a channel-based communication tool that supports simple text messaging, while *SOCS Boards* enhances this by incorporating real-time messaging through WebSockets. *Pi*, on the other hand, is a topic-based communication platform with a more structured approach, aligning more closely with McGill's needs by differentiating between roles such as professors, TAs, and students. However, *Pi* lacks key features like a search engine and real-time notifications, and its frontend design is not fully developed. In response to these limitations, McGill UAsk was designed as a post-based communication tool that integrates real-time notifications using WebSockets and offers comprehensive role management (Professors, TA, Students), providing a more complete solution tailored to the university's requirements.

c. Project Objectives and Scope

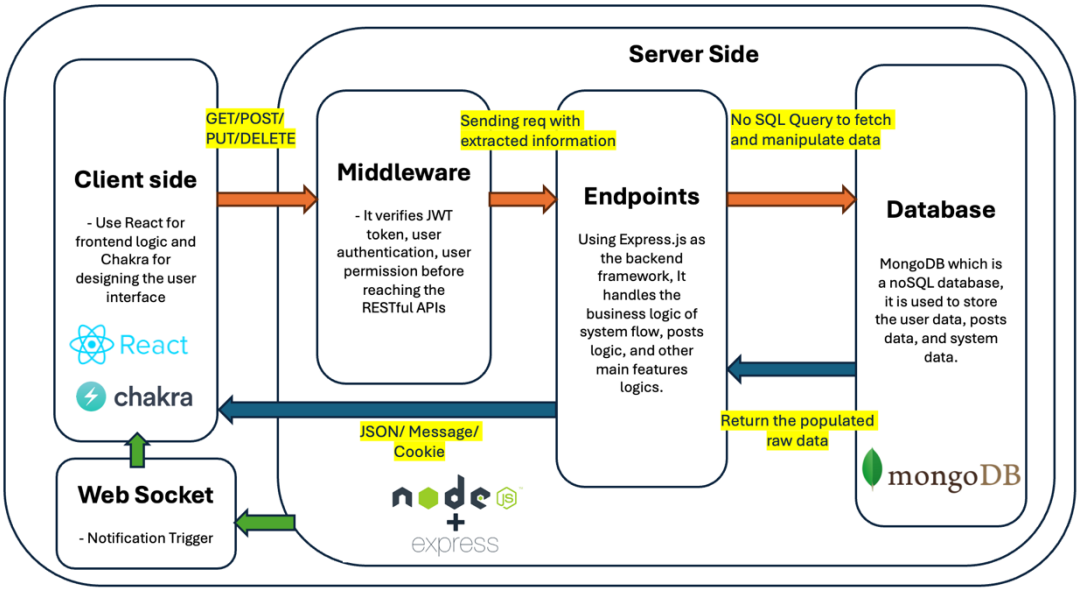
McGill UAsk is built around three core features, as shown in Figure 1. **Announcement Board** allows professors and TAs to post important messages, such as exam grade releases, all users can receive key information. **Posts Board** serves as a public Q&A forum where users can ask question in the post, receive comments from anyone, and the posts can only be resolved by professors or TAs. **Private Posts** enables users to create private group discussions with specific participants, ensuring confidentiality. These features are all supported by a notification system that promptly informs users of any updates, ensuring effective and timely communication.



(Figure 1 – The Features Graph of the Design)

The application also supports the user and course managements. So the professors and TAs can act as the admin to add or remove the course users, manage the unregistered users, and update any course information.

d. Project Frameworks



(Figure 2 – The McGill UAsk Web Architecture)

The McGill UAsk application is built using the MERN stack, comprising React and Chakra UI on the client side, and Express with Node.js on the server side, all connected to a MongoDB database. When a client makes a request to the server through GET, POST, DELETE, or PUT methods, middleware is employed to verify the JWT token stored in the client's cookie. This ensures that the user has the appropriate permissions (such as whether the user is authenticated, or whether the user is the course admin). Upon successful verification, the server logic processes the request, with data from MongoDB being populated as needed before returning the response, shown in Figure 2.

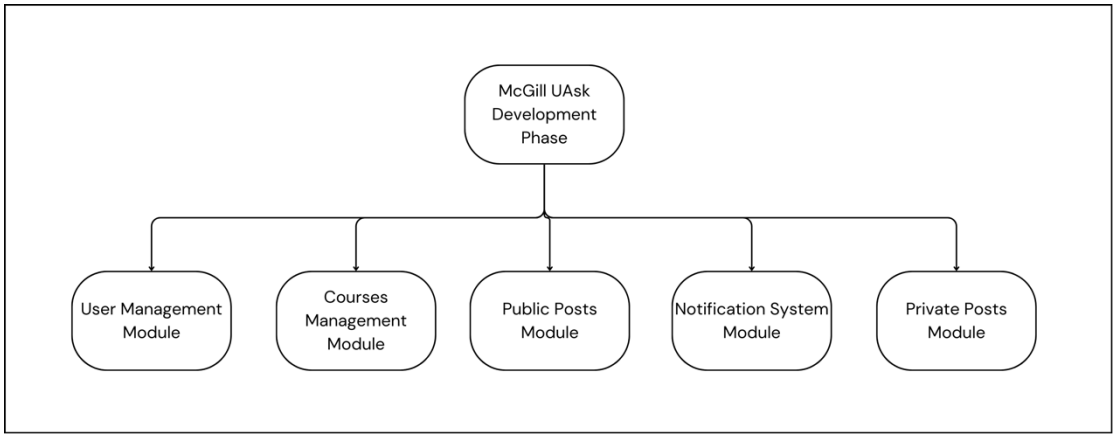
Additionally, WebSocket is utilized for real-time communication between the server and client. This allows the client to receive instant notifications whenever there are updates, such as when a post is commented on, ensuring timely and responsive communication within the platform.

Implementation

The McGill UAsk Design is implemented in different independent modules. Firstly, the modularization and web architectures will be discussed. Then, the implementation of each module will be detailed with the logic flow diagram.

a. Design Modules

Our application logic is separated into five modules. The first is the User Management module, which handles user authentication, login, and registration. The Course Management module is responsible for course creation and retrieval functionalities. The Public Posts module manages public discussions, including post creation, comments, and related features. The Private Posts module builds on the Public Posts module, adding a layer of confidentiality for private group discussions. Lastly, the Notification System module supports real-time communication, ensuring timely updates and interactions.



(Figure 3 – McGill UAsk Modularization)

b. Web and Database Architecture

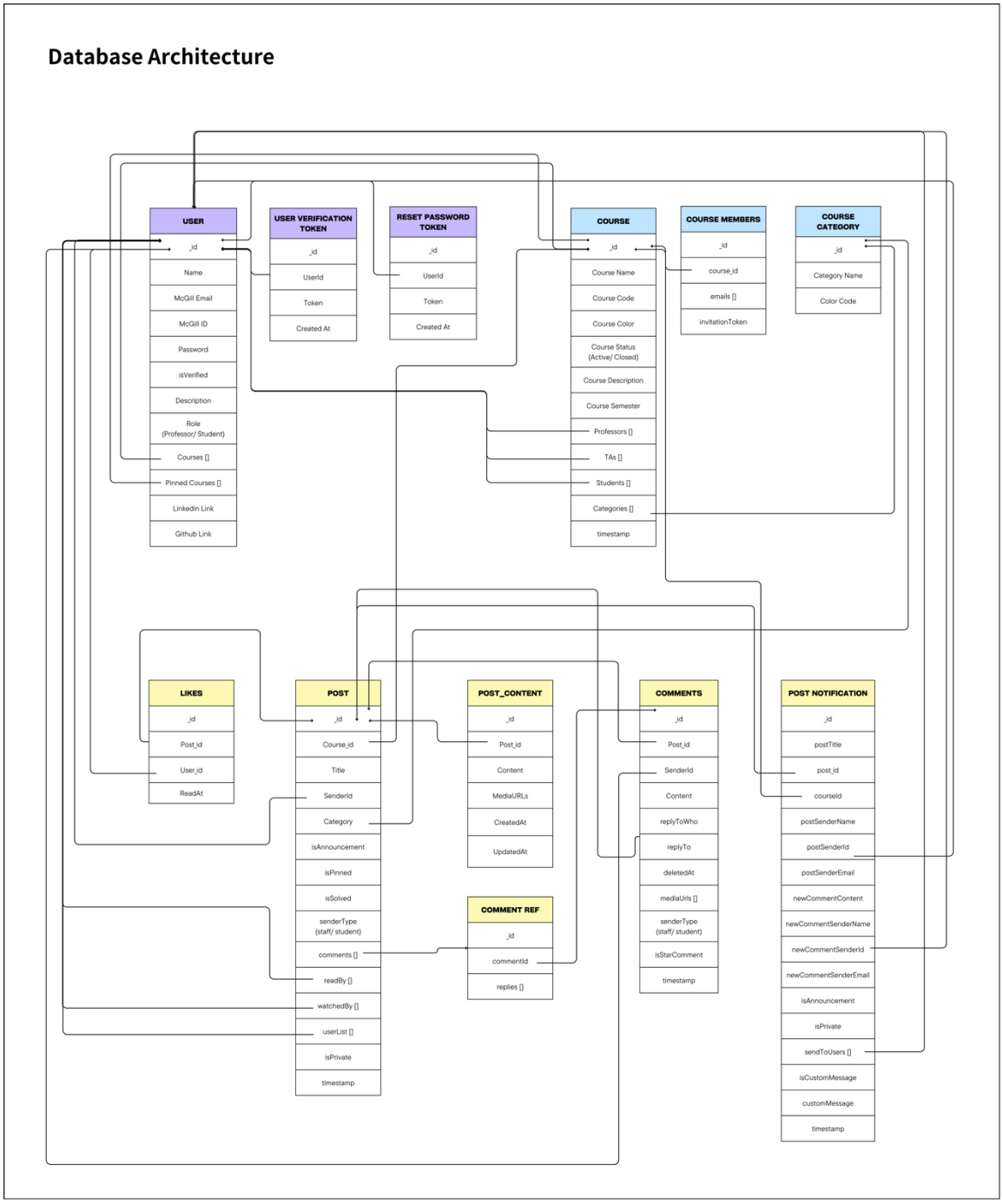
Database Structure

The MongoDB database for McGill UAsk is organized into three main types of collections: user-related, course-related, and post-related.

User-Related Collections

The primary user collection stores crucial user information, such as enrolled

courses, roles, and McGill email addresses. The unique user ID from this collection is used to track various user-specific data, including the posts created by that user. Additionally, two token collections are used to store unique tokens for each user. These tokens are essential for generating specific verification or password reset links, which are then sent to the user's email for account verification or password recovery.



(Figure 4 – The Database Architecture)

Course-Related Collections

The course collection serves as the main repository for course information, including comprehensive lists of students, TAs, and professors. This setup is particularly useful for role management and enhances the efficiency of the user search engine. Additionally, the course members collection stores the emails of all course participants, both registered and unregistered. This setup allows unregistered users to join courses easily by entering the course invitation token after registering in the system.

Post-Related Collections

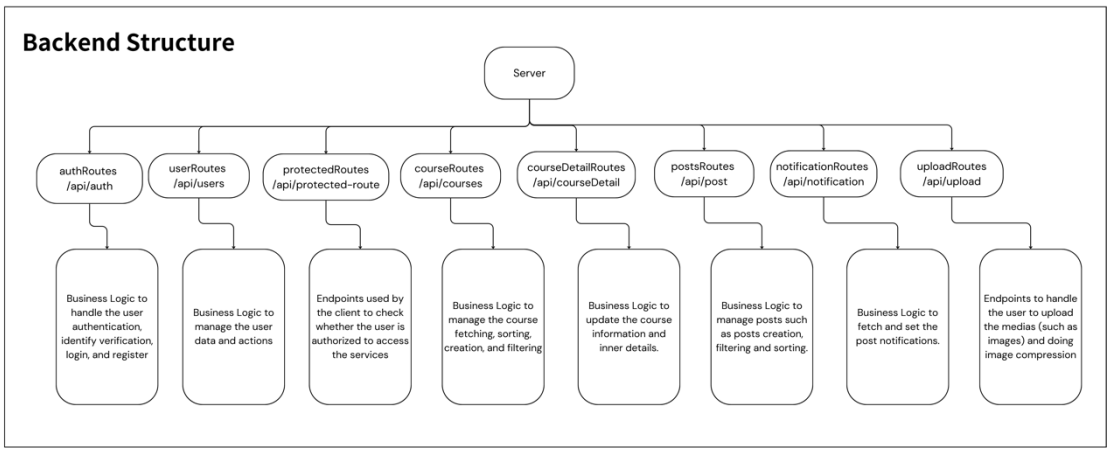
The primary post collection stores key post metrics such as the sender's ID and a list of users who have read the post. Importantly, the content of each post is stored separately in the post content collection, which is referenced by the post ID. This approach reduces the size of post data and increases the flexibility of the database. The post collection also contains a comments list that maps comment IDs to their respective replies, clarifying the hierarchy of comments and replies. This structure supports not only commenting on posts but also replying to comments and even replying to replies, enabling a more dynamic discussion flow.

Comment Collection

The comment information itself is stored in the comments collection, which includes details such as the post ID, sender ID, comment content, and the user being replied to. Additionally, the post notification collection stores updates related to different posts, including new comments. This collection works in conjunction with WebSocket to power the real-time notification system, ensuring that users are promptly informed of new activity on the platform.

Backend Structure

The backend of McGill UAsk is implemented using the Express framework and is structured into routes, controllers, and services to ensure a clean separation of concerns and efficient handling of requests. The application's various feature endpoints are organized into distinct routes, such as authentication routes, user routes, course routes, and posts routes. When a request is made, it first hits the corresponding route, which may invoke a middleware function to perform tasks such as verifying a JWT token to ensure the user is authenticated. Once the middleware check is complete, the request is passed to the appropriate controller function, which handles the business logic associated with that request. If the request involves accessing or modifying data, the controller will call a service function responsible for interacting with the database or performing local operations.



(Figure 5 – The Backend Architecture)

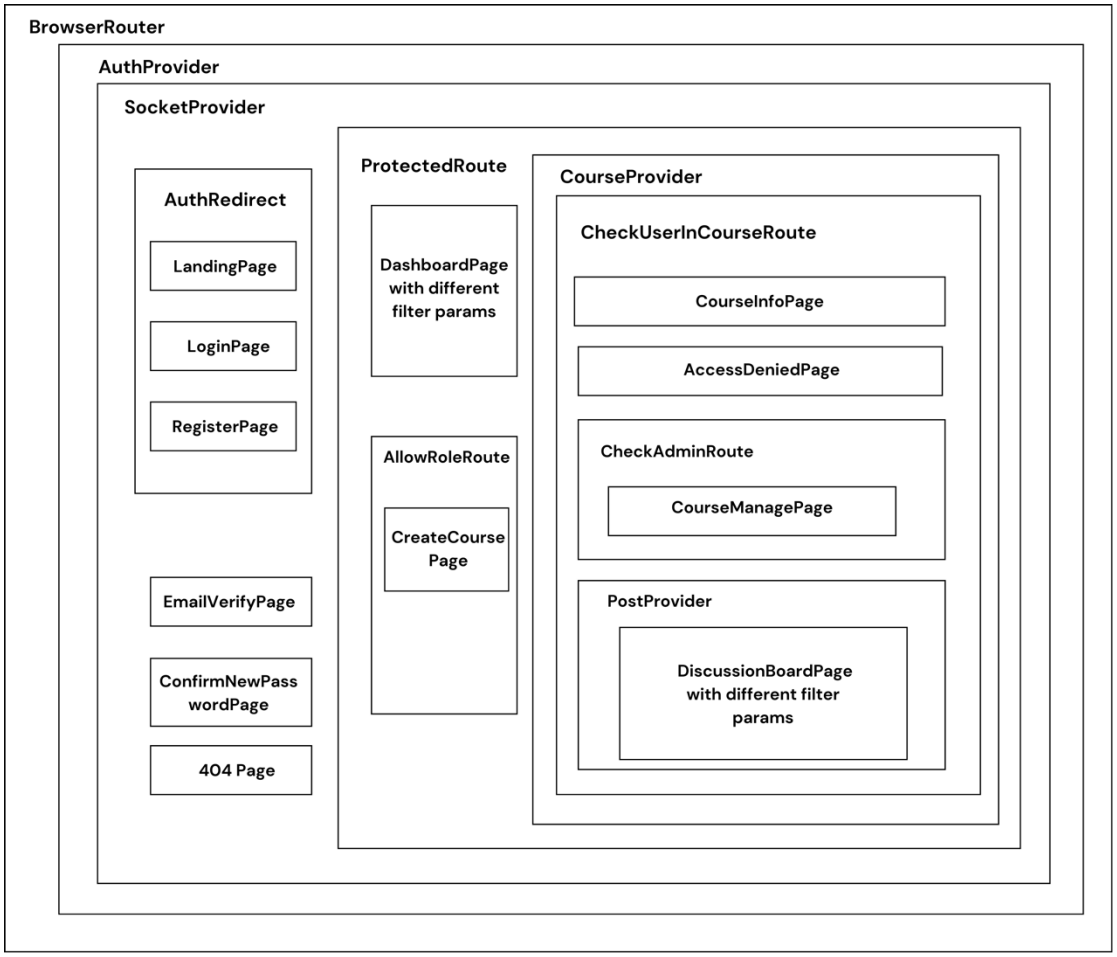
For instance, when a user wants to create a post, they would make a request to the endpoint /api/post/:courseId/post. This request is routed to the post route, where middleware first verifies that the user is logged in by checking the user ID stored in the client cookie. Additionally, the middleware checks whether the user is enrolled in the specified course by validating their presence in the course's participant list. If these checks pass, the request is forwarded to the createPost controller function, which handles the logic for creating a new post. Finally, the newly created post is stored in the database through a service function, completing the request process.

Frontend Structure

The client-side of McGill UAsk is built using React and Chakra UI, organized into layers for efficient management of routing, authentication, real-time updates, and access control.

BrowserRouter and AuthProvider

At the top is BrowserRouter, which handles routing within the application. Below it, AuthProvider manages user authentication, checking for a valid JWT token in cookies. If the token is valid, the user is redirected to the dashboard; otherwise, they are directed to the login page.



(Figure 6 – The Frontend Architecture)

SocketProvider

The SocketProvider layer establishes a WebSocket connection with the server for real-time notifications. It triggers Chakra UI's toast notifications when posts are commented on or updated, based on server broadcasts.

ProtectedRoute

The core features of the system are wrapped within the ProtectedRoute layer. This layer ensures that only authenticated users can access the main features of the application. It checks for the existence of user information before allowing access to the feature pages.

AllowRoleRoute

For role-based access control, the AllowRoleRoute layer checks if the current user holds the highest admin role (Professor) before granting access to specific pages, such as the course creation page.

CheckUserInCourseRoute

The CheckUserInCourseRoute layer verifies if the user is a member of the course by calling a server endpoint with the course ID. If the user is not part of the course, they are redirected to an access denied page.

CheckAdminPage

The CheckAdminPage layer is used to determine whether the current user is a TA or Professor for a specific course, ensuring they have the appropriate permissions to access and manage the course administration board, where they can update course details and manage course participants.

PostProvider

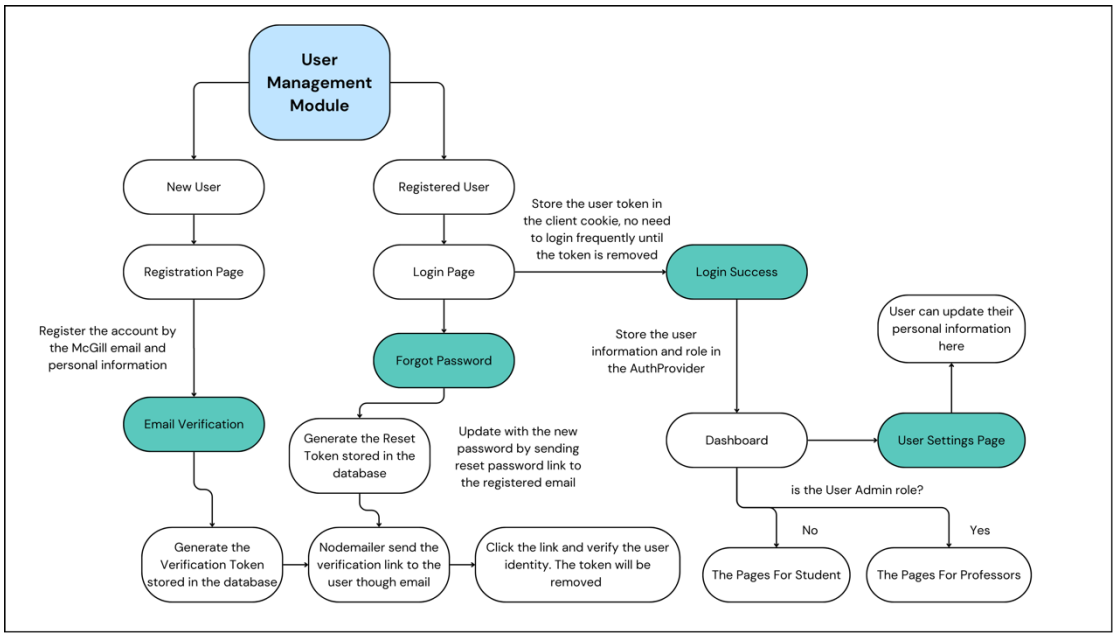
The PostProvider layer stores information related to the currently selected post, the current page (whether it's announcements, posts, or private posts), the applied post category, and any active post filters. This layer is crucial for maintaining UI consistency and improving the efficiency of post content retrieval across the application.

c. User Management Module

Figure 7 illustrates the logic design of the user management module, which primarily handles user authentication and registration. This module manages two scenarios: new users and returning users.

For new users, they must register their accounts using their McGill email. Upon registration, the user's data is stored in the user collection, but the isVerified flag is set to false. Simultaneously, a random verification token is generated using the Crypto library and stored in the user verification token collection along with the user ID. An email containing the verification link is then sent to the new user by the library 'nodemailer'. The user must click this link to verify their identity, which triggers the '/api/auth/:userId/verify/:token' endpoint, setting the isVerified flag to true and removing the token from the collection. The verification link in the email will be structured as follows:

"${process.env.BASE_URL}/users/${token.userId}/verify/${token.token}"



(Figure 7 – Visualization of the User Management Module)

For the returning users, they need to login in the system first. If they forget the passwords, a reset token will also be generated and stored in the forget password token collection, thereby send the reset password link to the user by email, the server will check the existence reset token and update the user collection with the new password. If they login successfully, they will direct to our dashboard page. For professors, additional options, e.g. course creation, will be available.

d. Course Management Module

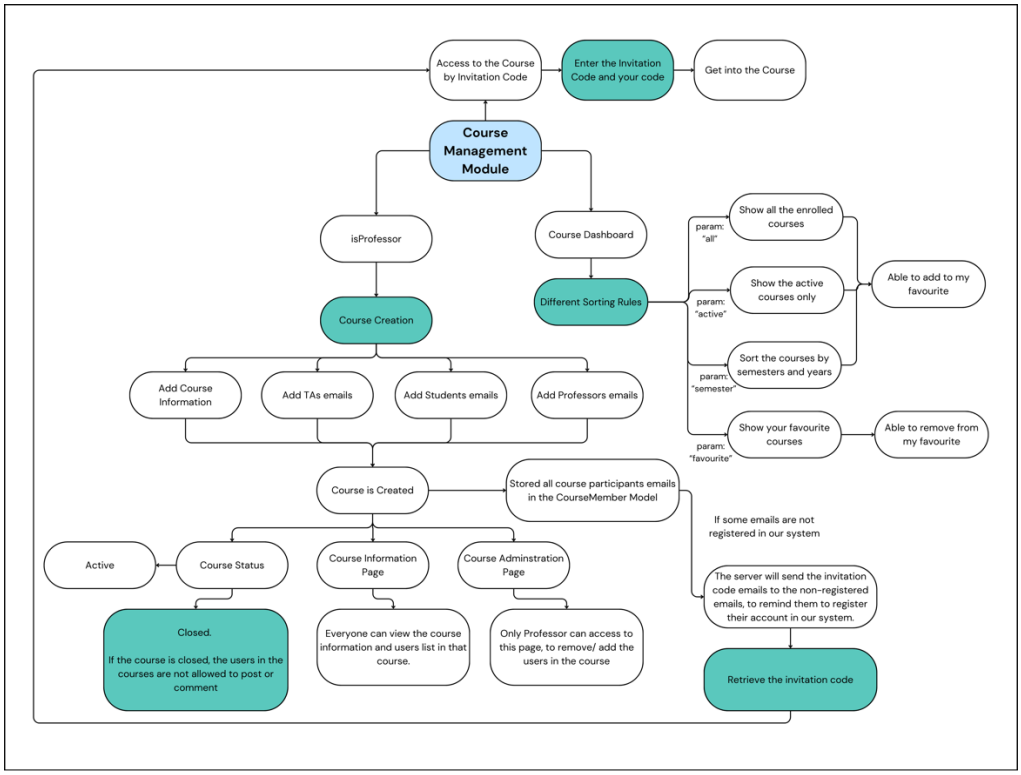
It manages course creation, course fetching, and course invitation codes.

Course Creation

Only professors can create courses by providing course information and user emails (for TAs, students, and other professors). Once a course is created, the data is stored in the course collection with a status of 'active,' and a unique course invitation code is generated. The course status can be either 'active' or 'closed'; when a course is closed, users are unable to create discussions within it. Additionally, if a user email provided during course creation is not registered in the system, the server will automatically send a course invitation code to that non-registered email address.

Course Invitation Code

After receiving the course invitation code, the user can request to join the course by submitting the code, their role, and their email in the request body. The server checks if the code exists in the course members collection. If it does, the server then verifies whether the provided email matches the role (TA, student, or professor) associated with that course. Once the verification is successful, the dashboard's course list is refreshed to include the newly added course.



(Figure 8 – Visualization of the Course Management Module)

Courses Fetching

The client provides different sorting rules for displaying the courses list in the dashboard. The dashboard URL is designed as below supporting different params:

"${process.env.BASE_URL}/dashboard/${param}"

When the param is "all" or the URL has no param, it is the default setting to fetch all your courses without any filtering. For each course, the user can add the course to 'my favourite' by adding the course id in to the favorite course list in his user collection.

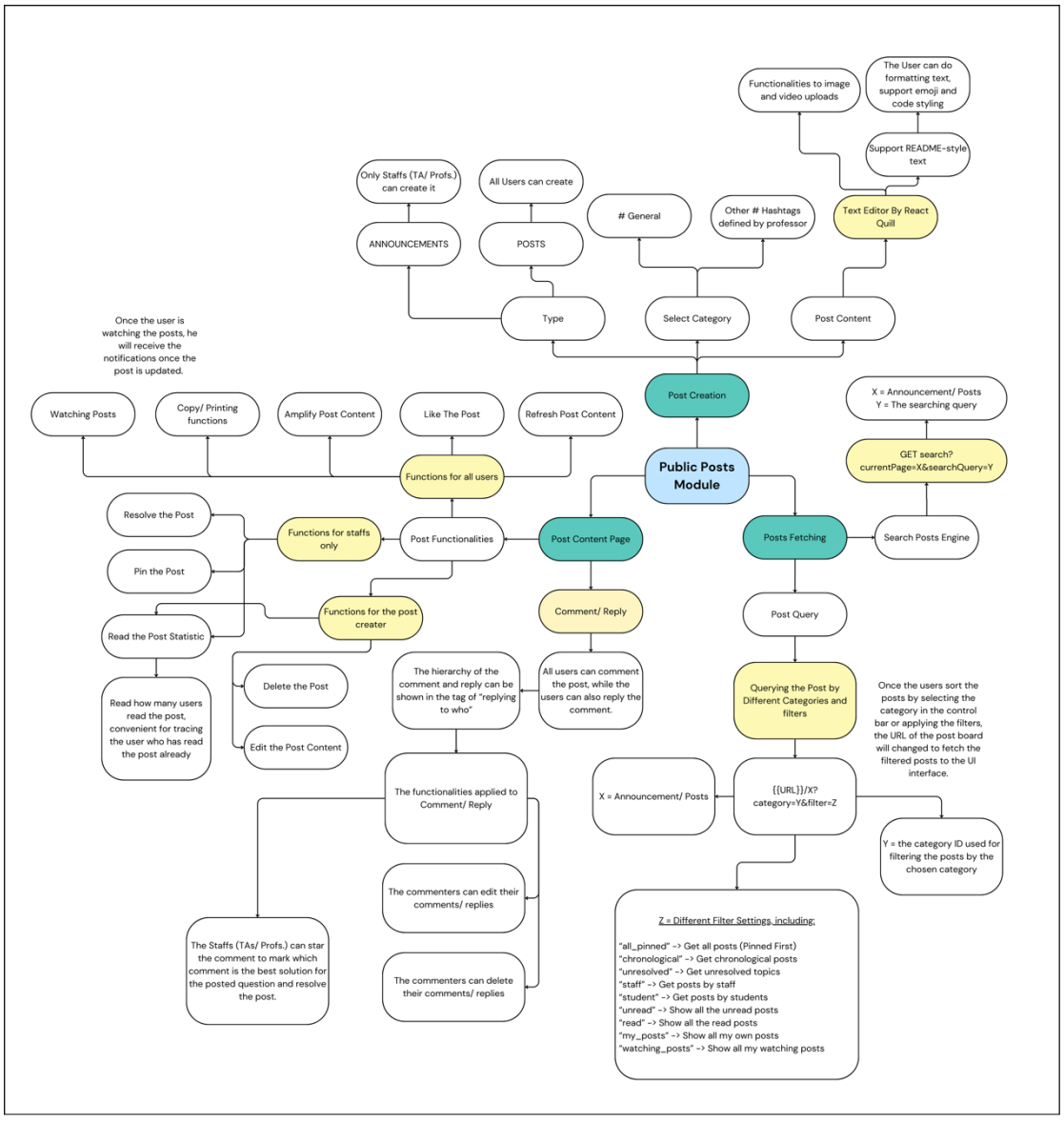
When the param is "favourite", the client will call the endpoint "/api/course/ getPinCourses" instead of "/api/course/ getAll", to retrieve and populate the favourite course list in their user collection. For each my favourite course, they have options to remove from "my favourite" too.

When the param is "active", the endpoint of "/api/course/getActive" will be reached, that only the courses with the flag "isActive" = "true" are fetched. This feature can help the user to view their active courses, without any expired courses.

When the param is "semester", the endpoint of "/api/course/getSemester" will be called, the output is a map (not a list now). The map is collecting the courses in the particular semester, e.g. {"Fall 2024": [course1, course2], "Winter 2023": [course3]"}. The UI presentation is different for this sorting, the client will sort the semesters from latest first, and showing their related courses.

e. Public Posts Module

The Public Post module is the largest feature in our application, relating to the main features. There are three components of it, including Post Creation, Posts Fetching, and Post Content.



(Figure 9 – Visualization of Public Posts Module)

Post Creation

There are two types of public posts in McGill UAsk: Announcements and Posts. Only staff members (Professors and TAs) have the permission to create announcements. The server enforces this through the middleware "checkDiscussionAdmin," which verifies the user's role before allowing the creation of announcements.

When creating a post, three main pieces of information are required: the Post Title, Hashtag, and Post Content. The Post Title is where users can input their questions, while the Hashtag allows users to categorize their questions, such as "exam question" or "assignment question." These hashtags are predefined by the professor during the course creation process.

The Post Content is managed through a multi-functional text editor implemented using the Quill library. This editor supports README-style text editing, allowing users to format text, align it, and change fonts or colors. Users can also include code snippets, LaTeX, math syntax, or equations directly within the content. Additionally, the editor allows users to upload images by either clicking the upload button or using drag-and-drop functionality. When an image is uploaded, it is first compressed using the "browser-image-compression" library before being stored on the server in the directory “/uploads/:courseCode/”. The image file name is generated using a combination of the user ID, course ID, and timestamp. The text editor also supports embedding YouTube videos through an HTML iframe. Also, the library DOMPurify is applied to sanitize the HTML input from Quill to prevent the XXS attack risk.

Posts Fetching

The application can query different posts by giving the current page (Announcement or Posts), different category Id (Predefined by the professor), different filter setting (such as “staff” to retrieve all the posts created by the staff). The choices of the filter settings are shown in the Figure 9. The querying is implemented in the URL like below:

“${DISCUSSION_BOARD_URL}/ ${current page}/ ${category id}/ ${filter}”

Based on the URL, it will retrieve the posts by calling the endpoint “/api/posts/ /:courseId/posts”. The endpoint is to handle posts fetching by inputting the params ‘isAnnouncement’, ‘filter’, and ‘category’, then filter and populate the related posts to the client. Moreover, the Infinite Scrolling is applied in the fetching design, initially only the first ten posts will be fetched, afterwards by giving the

cursor (the last post id we fetched) and limit (The number of posts we needed), the endpoint will fetch the next ten posts to the client until all the filtered posts are fetched.

Moreover, there is a post searching engine for the user to search the post by the post title and author name. The searching is to GET the endpoint of 'search? currentPage & searchQuery", where currentPage is announcements or posts, while the searchQuery is the search key entered by the user.

Post Content

After clicking on a post from the list, the user is directed to the post content page, where they can view the full details of the post, leave comments or replies, and access additional features such as liking or pinning the post. The page provides different functions depending on the user's role:

- **For All Users:**

  They can like the post, which adds their user ID to the like list in the post collection. Users can also choose to watch the post, adding the post ID to their watching list in their user collection. This makes it easier to filter watched posts in future queries, and users will receive notifications when a watched post is updated.

- **For Staff Members (Professors and TAs):**

  Staff can pin and resolve posts. Pinning sets the *"isPinned"* flag to true, and resolving a post sets the *"isResolved"* flag to true in the post model. Marking a post as resolved indicates that the post has been verified and the issue has been addressed, helping staff manage and identify unanswered questions more efficiently.

- **For Post Creators:**

  The original post creator has additional privileges. They can view post statistics to see how many users have read their post, which is useful for determining whether staff members have seen it. They also have the option to

delete or edit their post. Editing the post involves calling the endpoint *"/api/post/:courseId/post/:postId/edit"* with the updated HTML content in the request body. The server verifies that the *senderId* matches the post creator before modifying the post content in the post content model using the post ID.

- **Comment and Reply:**

The user can add comments using the Quill text editor located below the post, and they can also reply to existing comments. Both comments and replies are part of the same Comment Model. Replies include additional attributes such as replyTo (the ID of the comment being replied to) and replyToWho (the name of the commenter being replied to). For example, the reply data likes this:

{
  postId: “postId”,
  senderId: “Who sends this comment”,
  content: “The comment content by Quill”,
  replyToWho: “Replying to which person”,
  replyTo: “Replying to which comment”,
  senderType: “Are you staff or student”,
  IsStarComment: “Is the comment starred”
}

Also, once the comment and reply architecture is defined in the Post Model as a map “comments”, storing the level of different comment id, like this:

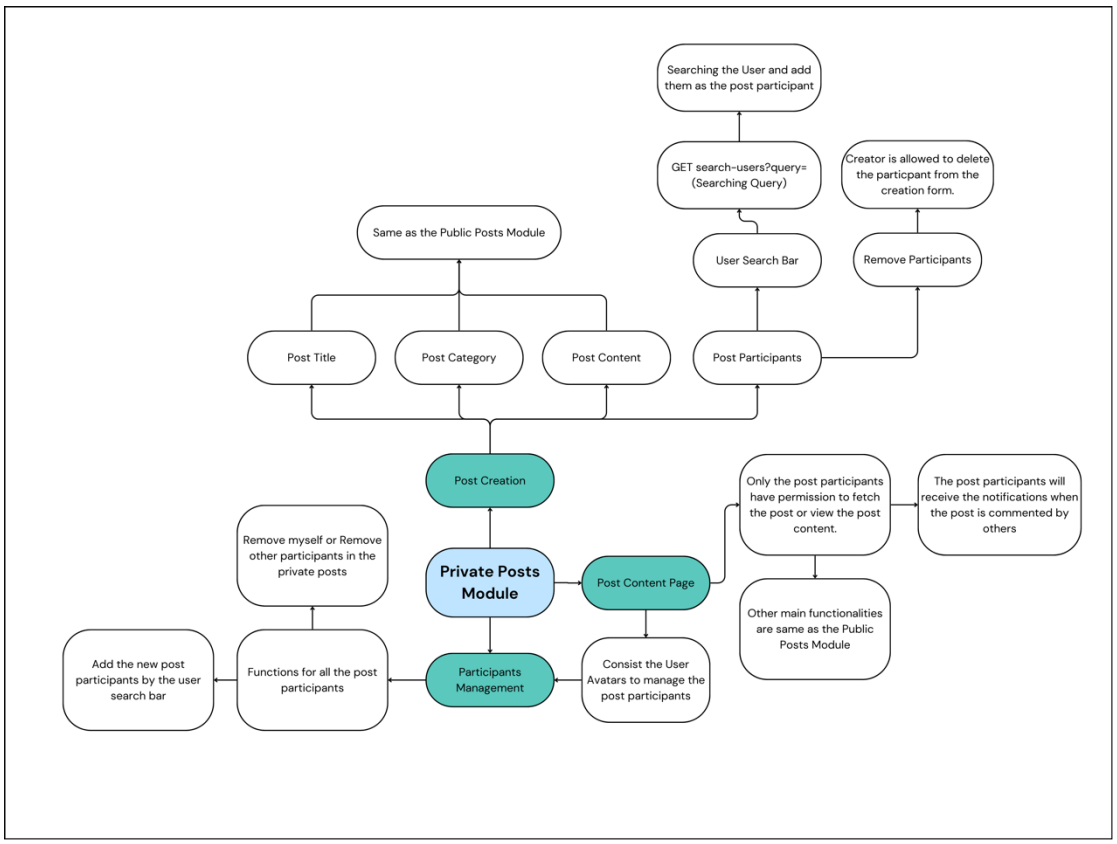
{
  “Comment Id 1”: [“reply Id 1”, “reply Id 2”],
  “Comment Id 2”: [“reply Id 3”],
  “Comment Id 3”: []
}

From the above example, they are will populated and find the comment content by the comment id, and then fetch the comments to the client side. There are 2 replies in the first comment, 1 reply in the second comment, and

no reply in the third comment. Moreover, the staffs have the permission to star the comment by changing the flag “isStarComment” to be true.

f. Private Posts Module

The Private Posts Module is built upon the Public Posts Module, with an additional layer of user permission checks to ensure only authorized users can access and interact with private posts, shown in the Figure 10.



(Figure 10 – Visualization of Private Posts Module)

Private Post Creation

Same as the public post creation, post title, category and post content are required. But it will additionally require the Post Participants, it can add the user to the post by searching the username or email. After the post is created, the private post data will be stored in the this structure in the post collection.

{

… (Same as the Public Post Structure)

isPrivate: true,

userList: [.userA, userB, userC]

}

The above example shows the flag of isPrivate set to be true, and only userA, userB, and userC are allowed to view and access to the post.

Private Post Fetching and Content

For the post fetching function, it is same as the public posts module calling the endpoint of *"/api/posts /:courseId/posts",* however, it will have an additional request body (isPrivate). If there is isPrivate in the request, the fetching controller will filter out the posts in which the isPrivate = true and the userList containing the request user Id.

For the post content retrieval, when the user is accessing to that private post, the client will call *"/api/posts /:courseId/posts/:postId"* to get the post detail by post Id. If the post's isPrivate is true, it will also check whether the request userId is existed in the userList. If not, they will redirect to the access denied page, otherwise they can view the private discussion.

Participants Management

There is an additional user avatar showing in the private post content page. The user avatar is to manage the participated users in that post. The user in the post can add the more user by the search engine, or they can remove the current participant or remove themselves from the post. Once the participant is removed, he will not be allowed to access the private post anymore. If the user is added, it will push the new user id to the userList from the Post Model. If the user is deleted, it will pop the user Id from the list.

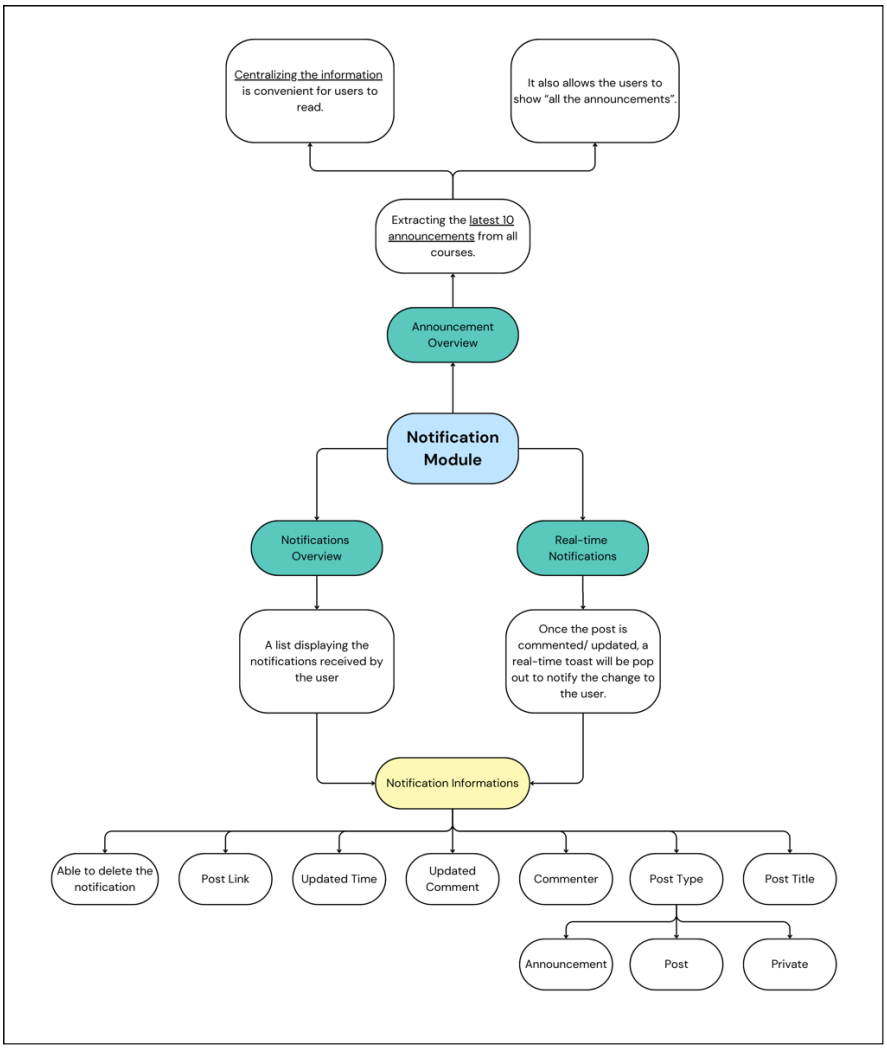
For all participants in a private post, whenever the post is commented on or updated, it triggers the notification mechanism. The server's WebSocket then broadcasts the new message to the participants' rooms. Upon receiving the message, the client displays a real-time notification informing the user that someone has commented on their post.

g. Notification System Module

The notification mechanism, which is implemented by the WebSocket library, has three components, including Announcements Overview, Real-time Notifications, and Notifications Overview, shown in the Figure 11.

Announcements Overview

This feature centralizes all announcements from the user's courses and displays them on the notification page. To enhance user experience, only the latest 10 announcements are shown, allowing users to easily read important updates without needing to navigate to each course's discussion board. The /get-announcements endpoint in the User Routes handles this by fetching announcements based on the user's enrolled courses and a specified limit, currently set to 10.



(Figure 11 – Visualization of Notification System Module)

Real-time Notifications

The notification mechanism is activated by the comment and reply functions, sending notifications to each user's room via WebSocket, based on the userList in the post data. For instance, if the userList for a post includes [UserA, UserB], and UserC comments on the post, UserA and UserB will receive notifications. Additionally, UserC is added to the userList, updating it to [UserA, UserB, UserC]. If UserA then replies to UserC's comment, both UserB and UserC will receive notifications.

These notifications are real-time and are sent to the room named User_${userId} via WebSocket. Each user registers to their respective room, User_${userId}, upon logging in. When a message is broadcast to User_${userId}, the user will receive it. The message data will appear like this and stored in the Notification Model Collection with a list *"sentToUsers"* of storing which users should receive this notification.

{
PostLink: "The Post URL for the user to click and redirect to the post",
UpdateTime: "The comment time",
UpdatedComment: "The content of the new comment",
Commenter: "The name of the comment sender",
PostTitle: "The Post Name which is updated",
PostType: "Is the post announcement/ private/ public. Different Color marked."
…
}

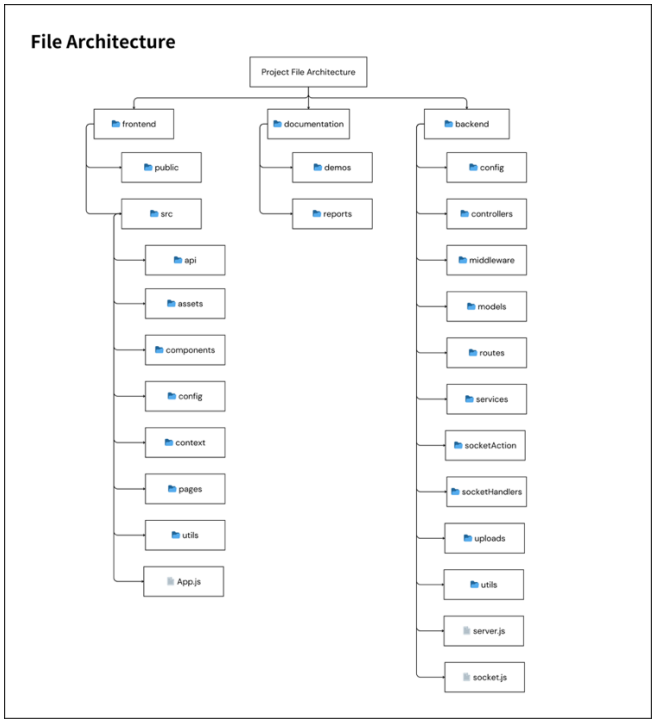
Notifications Overview

There is a "Notifications Overview" in the page of Notification, showing all the unread notifications that the user received. It is a list of notifications, with a URL button for the user to click and redirect the target post. Once the updated post is read, the *"sentToUsers"* in the Notification Model will be updated and your User ID will be pop from it. Or the user can also click the delete button of the notification, to remove the notification.

Results

The result section will include the future maintenance of this application, the user guide on how to use this app, and the enhanced consideration of the development process, such as UX/UI design.

a. How to use the program



(Figure 12 – File Structure of Design)

Documentation

The program file structure is mainly separated into three folders. The documentation is for showing the UI and reports of the application.

Backend Folder

For the backend folder, the concept of separation of concern is applied. There are few main folder including routes (defining the Endpoints), middleware (The middleware or verification needed in the specific endpoints), controllers (The Business Logic from the Route), service (The handlers of querying the database data), models (The data structure of the MongoDB), utils (The common functions), socketHandlers (The socket connections and broadcasting name), socketAction (The wrapper functions without calling the WebSocket directly).

Frontend Folder

For the frontend side, the UI is modularized in the components folders that will be used in the page folders. The context folder is defined for the background storage, service, and routing techniques. The api folder is the wrapper function to define the service function to call the endpoints. Moreover, the util function is some common function such as styling, formatting date, while the config function is the folder to define the connection between the client and the server.

Backend Configuration

To run the application, first define the necessary configurations in the .env file within the backend. This includes setting values for PORT, MONGO_URI, CLIENT_URL, JWT_SECRET, and SALT for encryption. Additionally, you'll need to configure email settings in the .env file for the Nodemailer library to enable email functionality. The configurations of env. file need to pre-defined:

PORT = Specifies the port on which the server will run
MONGO_URI = Defines the connection string for the MongoDB database
BASE_URL = Sets the base URL for the frontend application
EMAIL_SERVICE = Configures the email service provider, e.g. gmail
EMAIL_PORT = Specifies the port number used by the email service for SMTP
EMAIL_SECURE = whether the email service should use a secure TLS/SSL.
EMAIL_SENDER = The email address used as the sender for outgoing emails
EMAIL_PASSWORD = The password or API key used to authenticate
JWT_SECRET = The secret key used to sign JSON Web Tokens (JWTs)
SALT = The number of salt rounds used in hashing passwords

Frontend Configuration

On the frontend side, update the “proxy” URL in the package.json file to point to your production or development URL, such as http://localhost:5000.

How to start the application

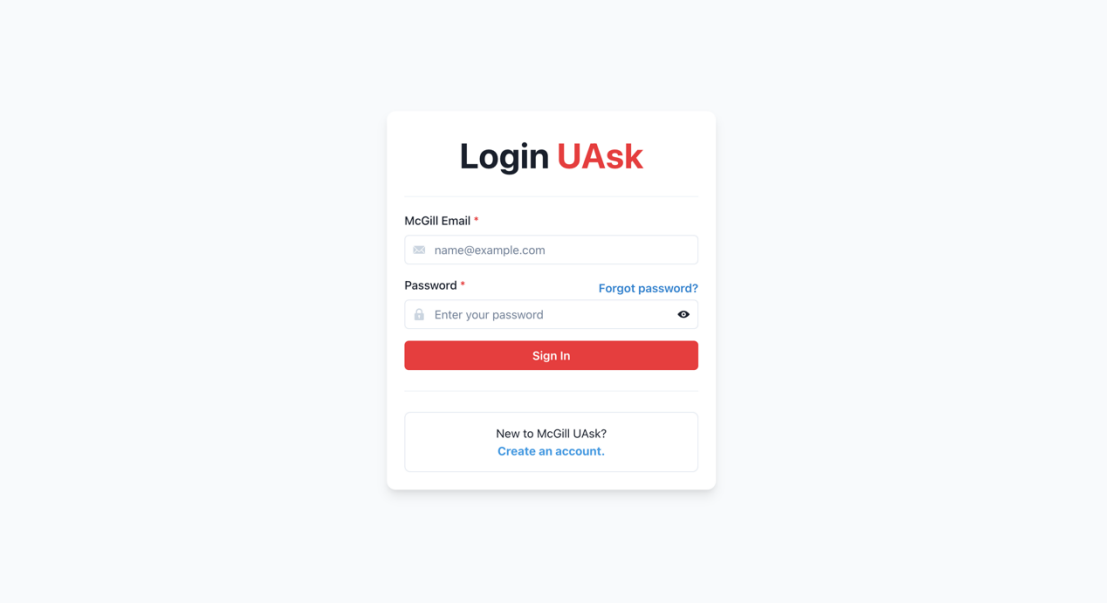
Once the configurations are set, you can start the backend and frontend separately by running the command npm start in the development environment.

b. User Guide

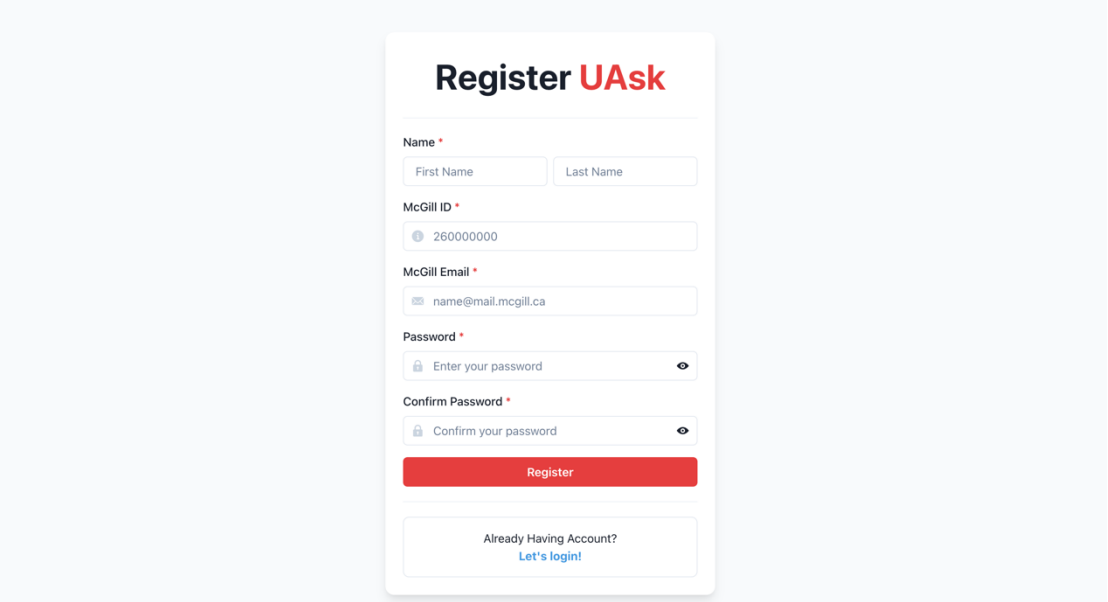
Here is the demonstration on how to use the McGill UAsk and create the discussion between the students and the faulty, with the User Interface flow.

I. Login and Registration

If the users are registered already, they can login to accounts by the McGill email and password. There is a forget password and create account option.



If the new user arrives our platform, they can register accounts by McGill ID and email. If the email suffix ends with "@mcgill.ca", it will set it as the professors account, otherwise they are all student accounts.



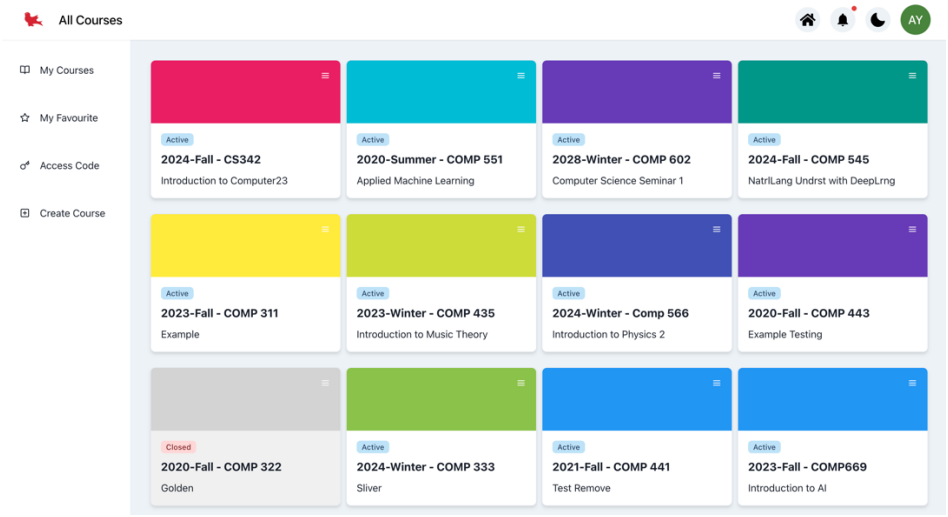
(Figure 13 – Login and Registration Page)

II. Courses Dashboard

After the users are successfully login, they will direct to the course dashboard, where they can view their enrolled courses, enter the invitation code, view the notifications, and create the course if they are professors.

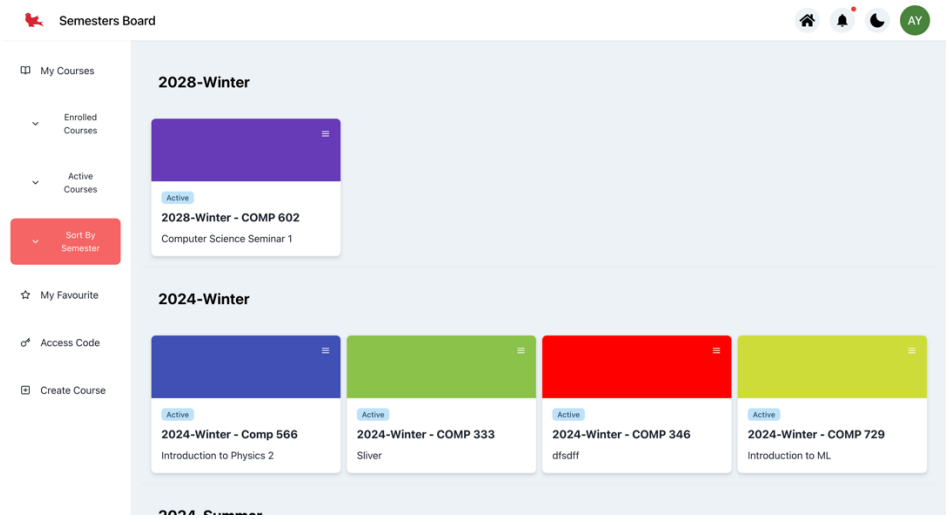
i. Courses Display

The users will see all their enrolled courses by default first.



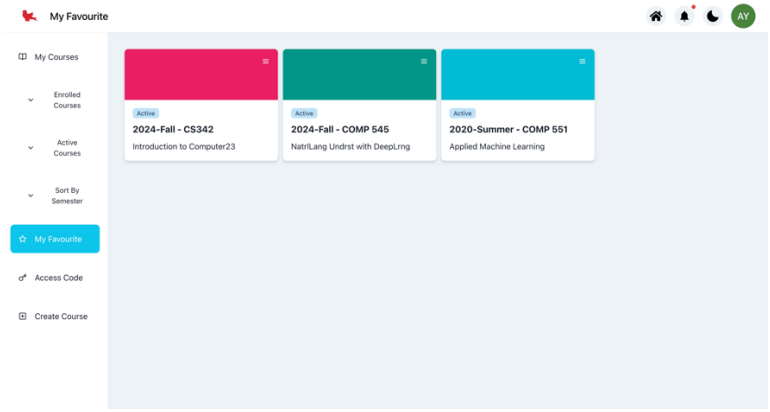
(Figure 14 – The Default Course Dashboard Page)

The users can select different sorting rules in sidebar to present the courses efficiently, such as to show active courses only (Appendix 2) or showing the courses sorted by different semester.



(Figure 15 – The Course Dashboard Page Sorting by Semester)

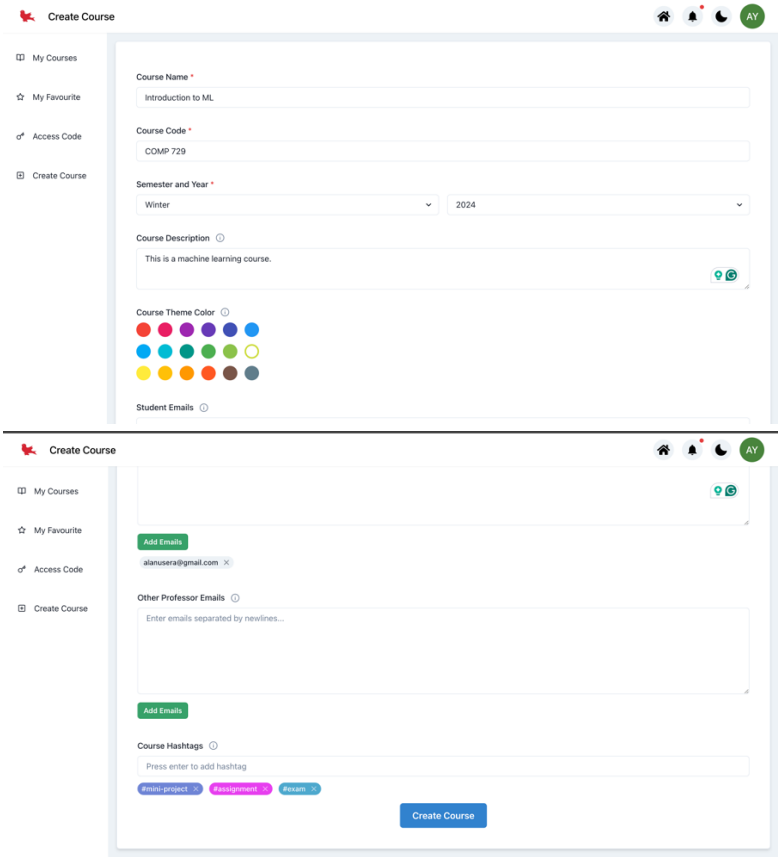
The users can select to show my favourite courses only in sidebar, and they can also remove from/ add to the favourite by clicking the menu of each course card.



(Figure 16 – The Course Dashboard Page Showing My Favourite)

ii. Create Course

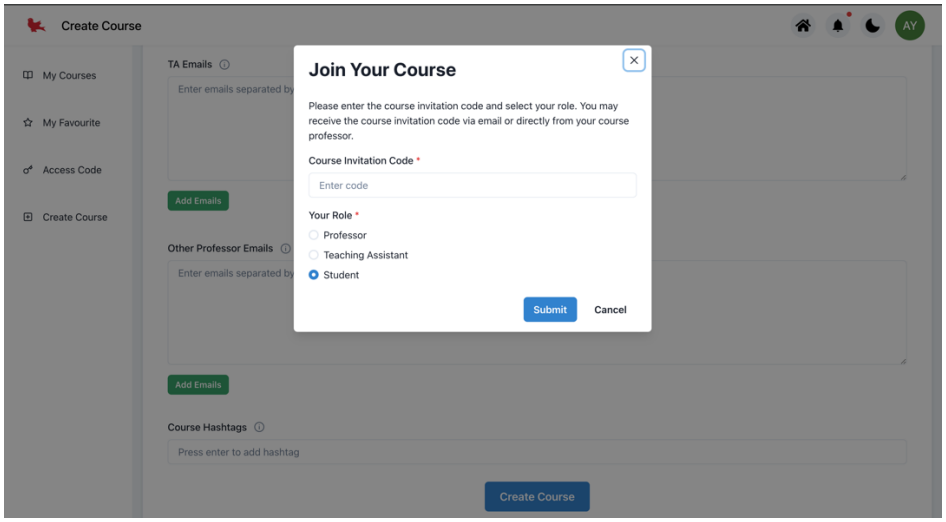
For the professors, they can create the course by filling the form with the course information, course theme color (for styling the course discussion board), students list, TAs list, other professors list, and they can define different hashtags (for the post creation and filtering usage).



(Figure 17 – The Course Creation Form)

iii. Course Invitation Code

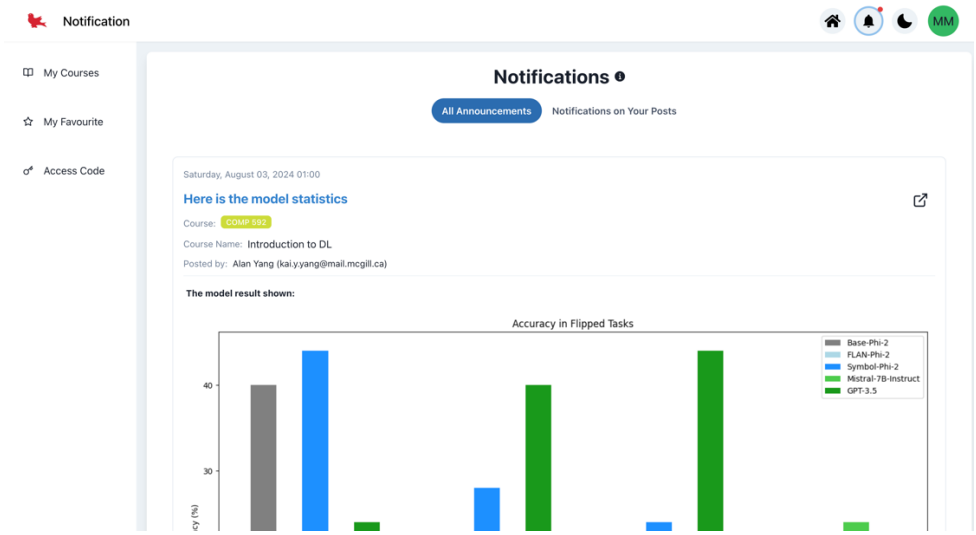
After the course is created, the professor can receive a course invitation code (shown in Appendix 3). For the students who took the course but not yet enrolled in the course discussion, they can enter the invitation code with their related role in the modal, get into the course directly.



(Figure 18 – The Course Invitation Code Access Page)

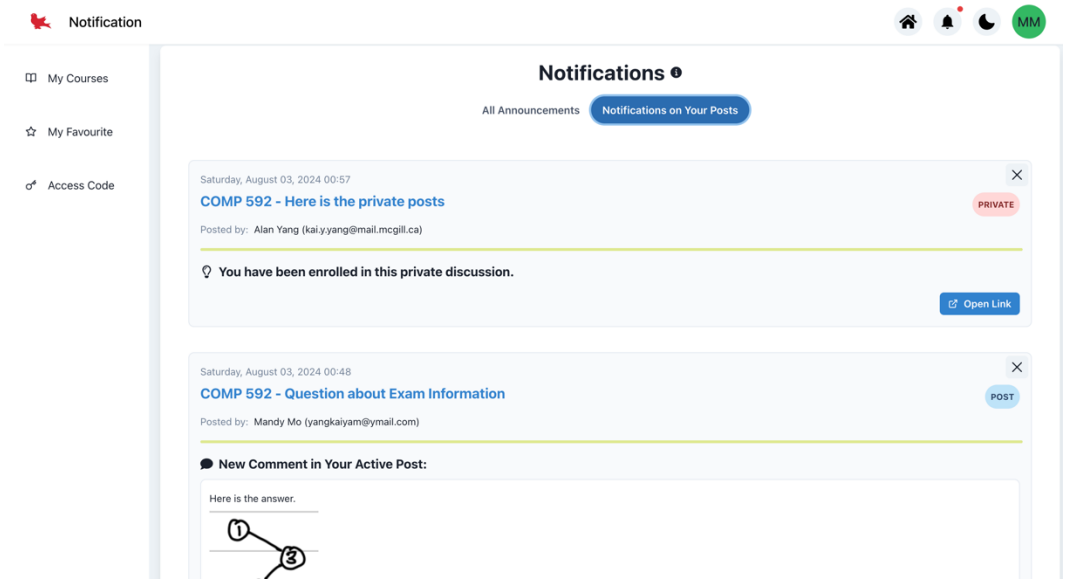
iv. Notifications Page (Bell Icon)

Users can access the notifications page by clicking the bell icon on the navigation bar. Here, they can view the latest 10 announcements they have received and can click the links to view more details.



(Figure 19 – The Announcements Overview in Notification Page)

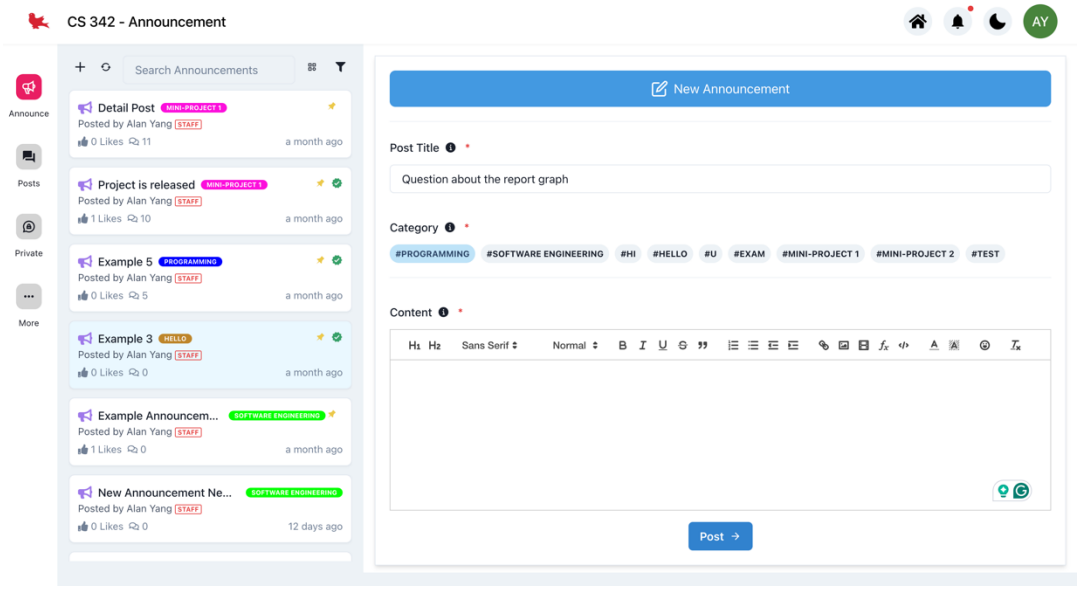
Users can also view the unread notifications here and can click the links to view more details. The notification has shown where does it come from, the message type, and the new message they received.



(Figure 20 – The Notifications Overview in Notification Page)

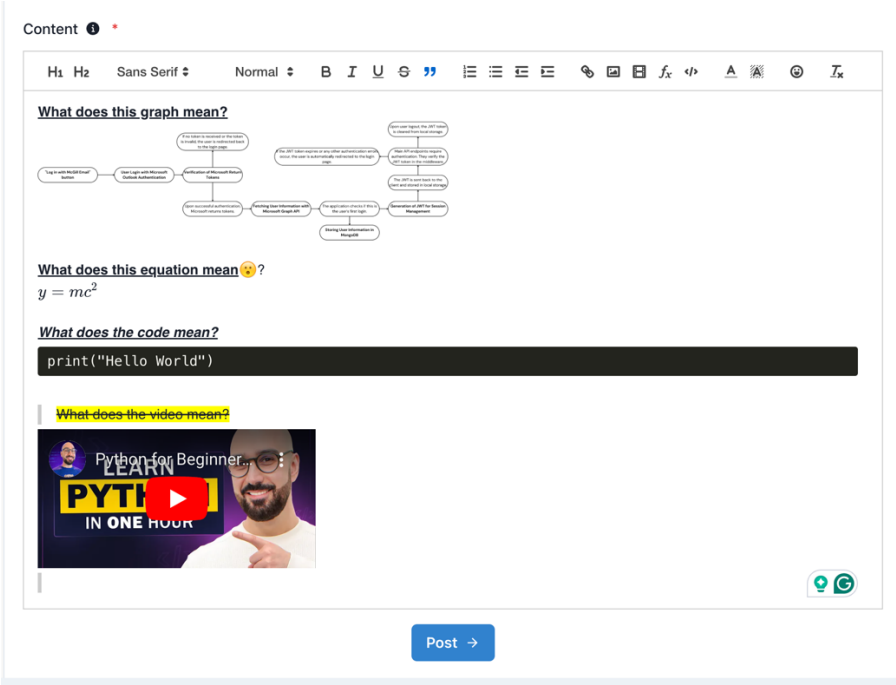
III. Create Posts and Announcements

After entering to the course discussion board, the users can see the three main features: Announce, Posts, and Private in the left sidebar. The below is the announcement board, only staffs (TAs and Professors) are allowed to click the "plus" icon to create the announcements. For the posts board, everyone is allowed to create the post.



(Figure 21 – The Announcement Board)

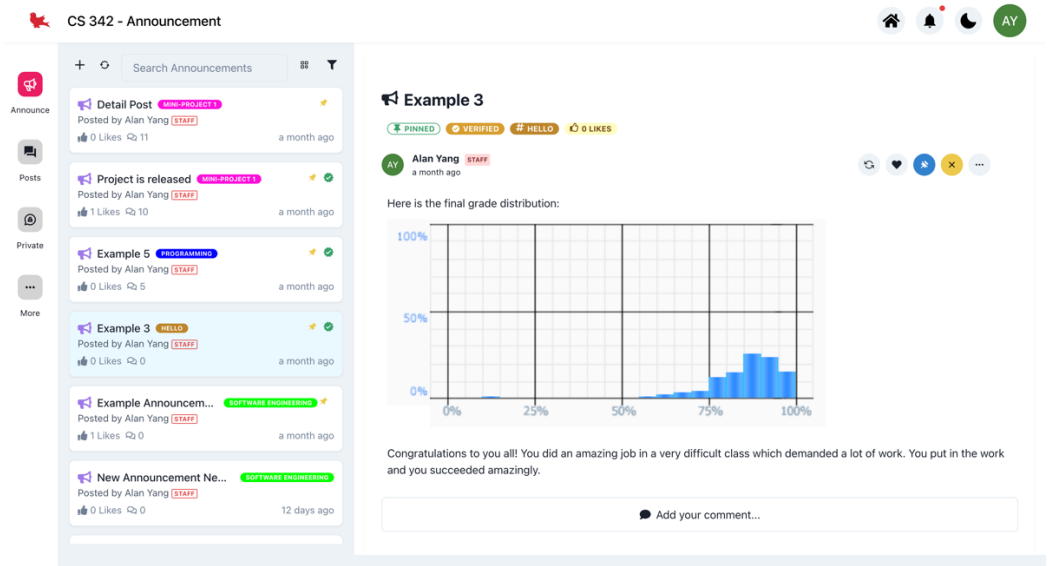
According to figure 21, the post creation requires the user to enter the post title, select the post category, and input the post content. The content box is a text editor supporting multi-media input such as image upload, equation syntax, code syntax, color styling, and YouTube iframe, shown in figure 22.



(Figure 22 – The Text Editor of the Post Content)

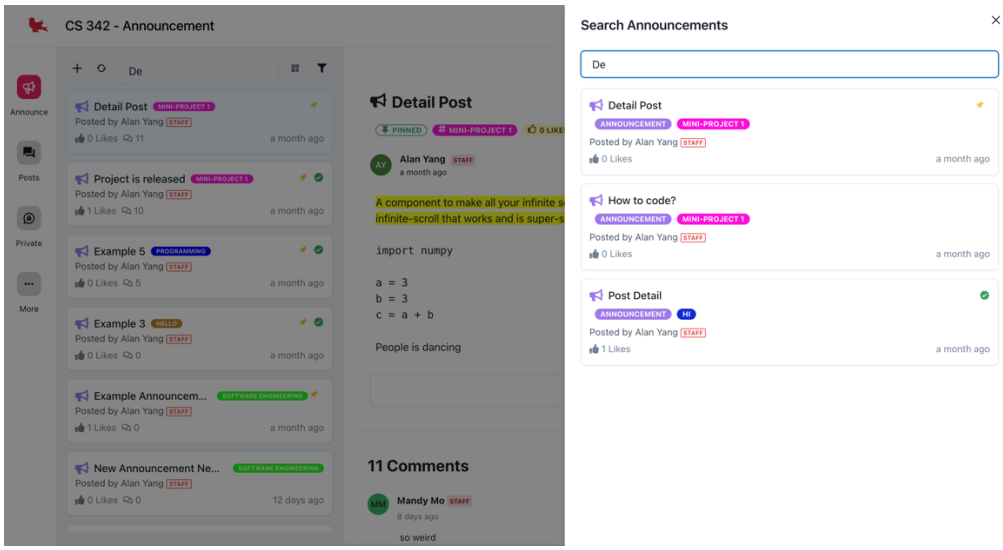
IV. Posts Fetching

The layout for Announcements and Posts is the same. Users can click on a post in the list on the left, and the post content will be displayed in the area on the right.



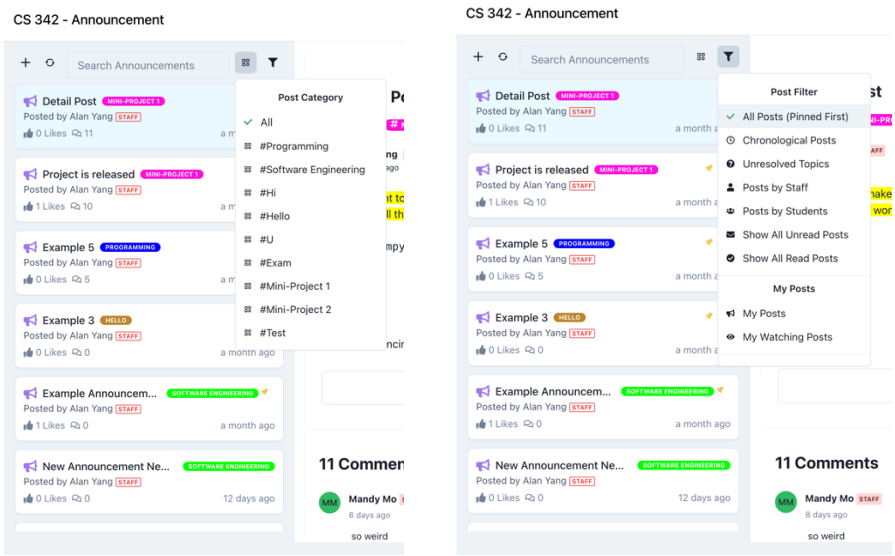
(Figure 23 – The Post Fetching Layout)

For the posts list on the left, we have a search bar to search the post by the title and author name. The searched posts will be shown on the right drawer.



(Figure 24 – The Post Searching)

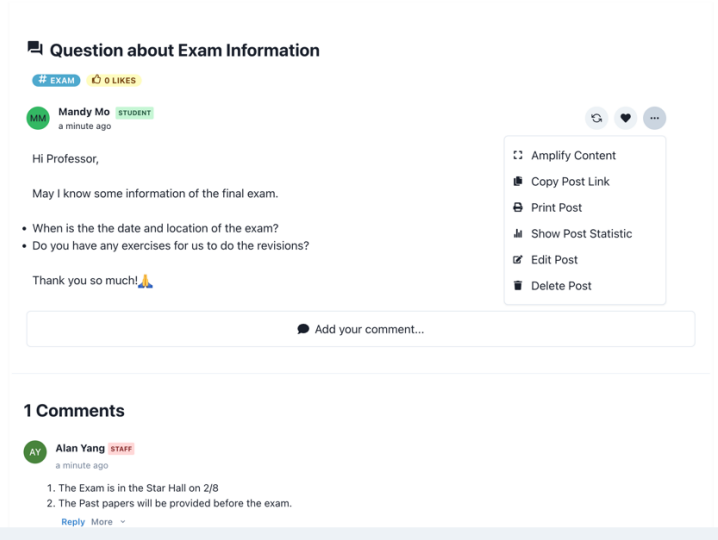
The control bar on the left side of the posts list allows for category filtering and sorting options. Users can filter posts by selecting a specific category (e.g., #Mini-Project 1), which will display all posts related to that category. For sorting, users have several options, such as displaying pinned posts first, sorting posts from latest to oldest, or showing only unread posts, etc.



(Figure 25 – The Post Filtering and Sorting)

Post Creator Actions

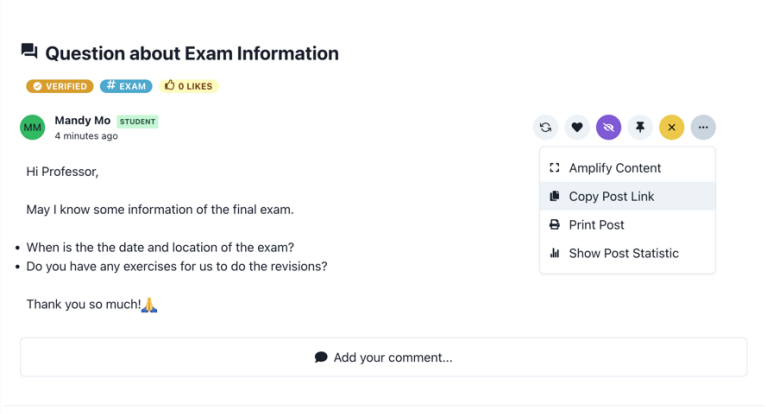
In the Post Content area, all users can like the post, refresh the post, amplify the post content, copy the post link, watch the post, and print the post details. The post creator has additional permissions, allowing them to edit and delete the post, as well as view post statistics.



(Figure 26 – The Post Creator Actions in the Post)

Staff Actions (TAs and Professors)

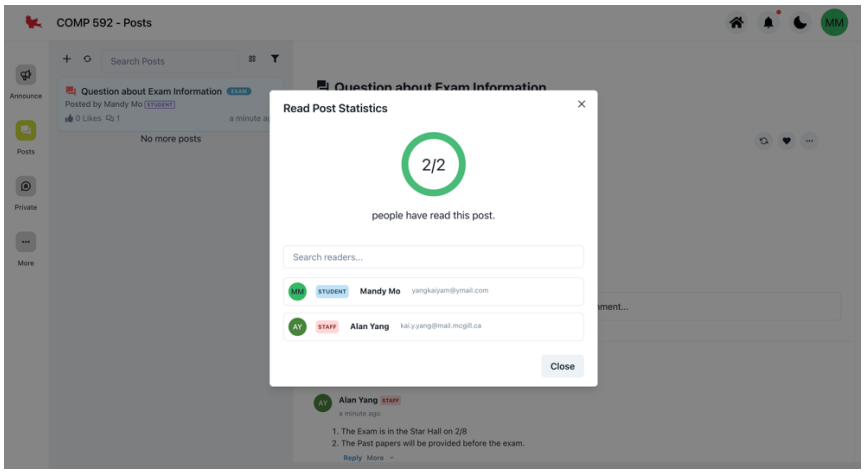
For staff members, they have the highest level of permission on the post. They can pin the post, mark the question as resolved, and view the post statistics.



(Figure 27 – The Staff Actions in the Post)

Post Statistic

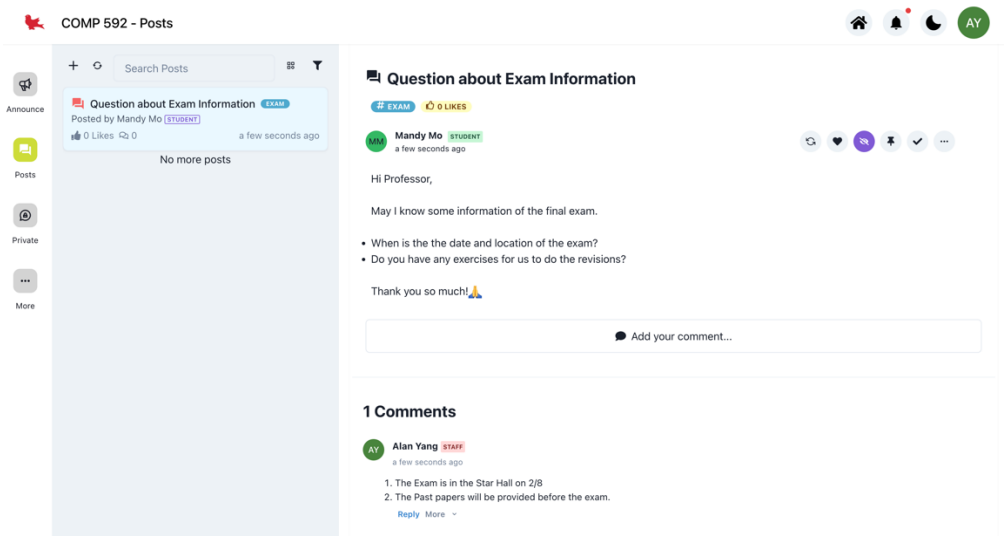
This feature shows how many users have read your post and allows you to search for specific users who have read it. It is designed to help you easily track whether staff members have seen your questions.



(Figure 28 – The Post Statistic Modal)

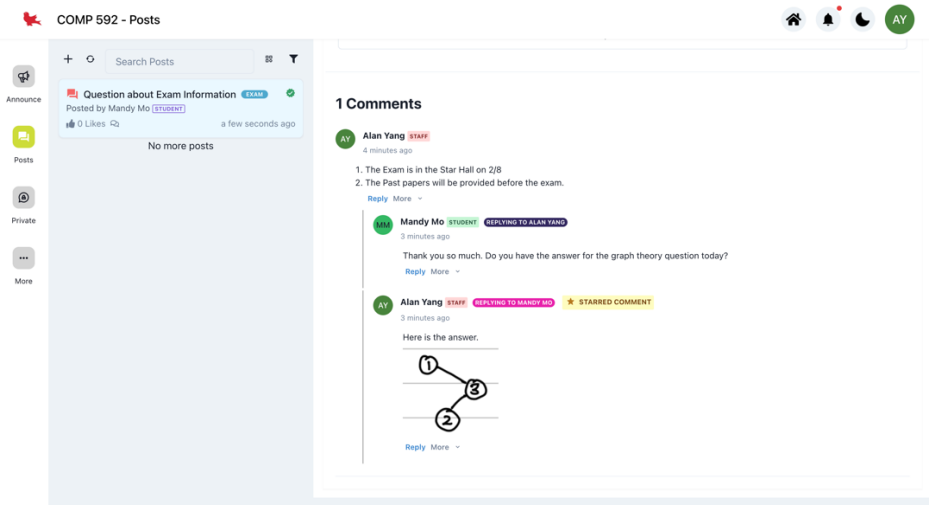
V. Comment and Reply

In each post content, all users can leave a comment using the Quill text editor. After submitting a comment, the post content page automatically refreshes to display the latest post details, including your new comment. Each comment shows the commenter's name, the time elapsed since the comment was posted, and the comment content. Additionally, anyone can reply to a comment.



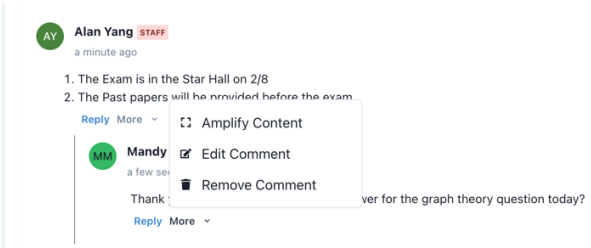
(Figure 29 – Making the Comment in the post)

For each comment, users can reply directly to it. Comments and replies are displayed in a two-level format, where each reply indicates who it is "replying to" and whether it was made by staff or a student. If a comment or reply is starred by the staff, it is tagged with "STARRED COMMENT."



(Figure 30 – Making the Reply on the comment)

Comment creators can edit or delete their comments. If a comment is edited, it will display the "(edited)" label at the end. If the comment is deleted, it will be replaced with the text "(Deleted)."

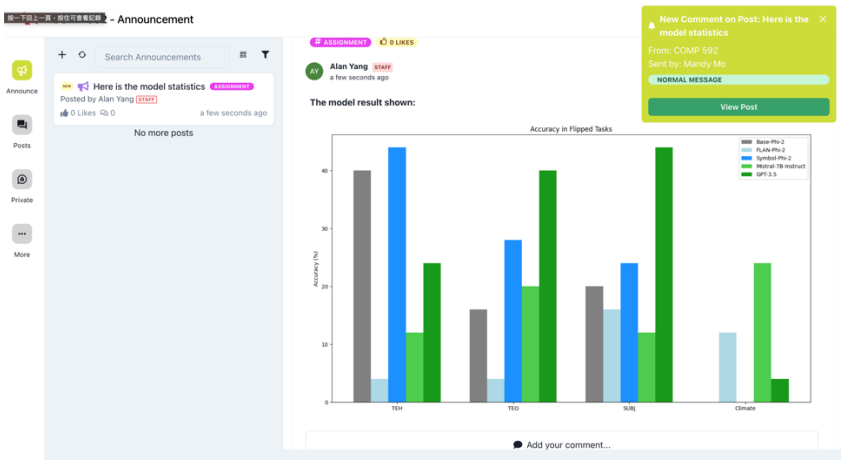


For the staffs, they are allowed to star the comment, which means they think this comment has solved the question of the post. After they star the comment, the post can be resolved.

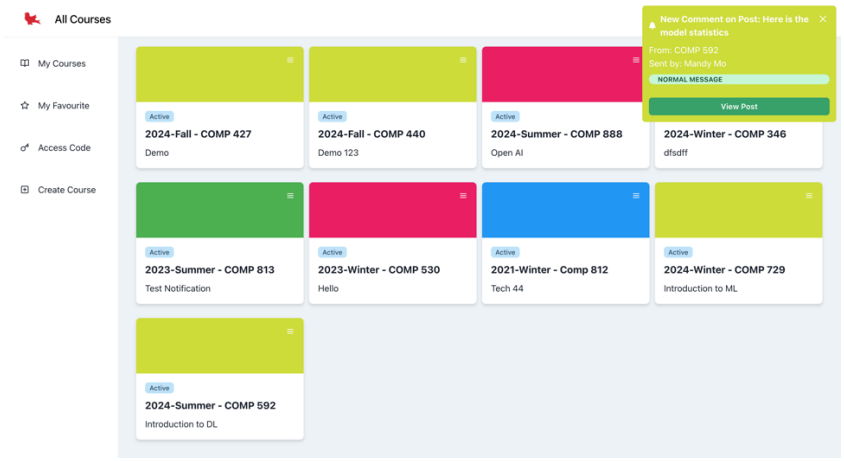
(Figure 31 – The actions on the comment)

Real-Time Notification

When the user is making comment or reply under the post, the participants in the post will receive the notification of the new comment from a pop-out toast, shown in Figure 32.



For example, if the userA made a post, userB made a comment under the post from userA, thereby the userA will be notified by a toast in the current page. The userA can click the “View Post” to see the new comment.



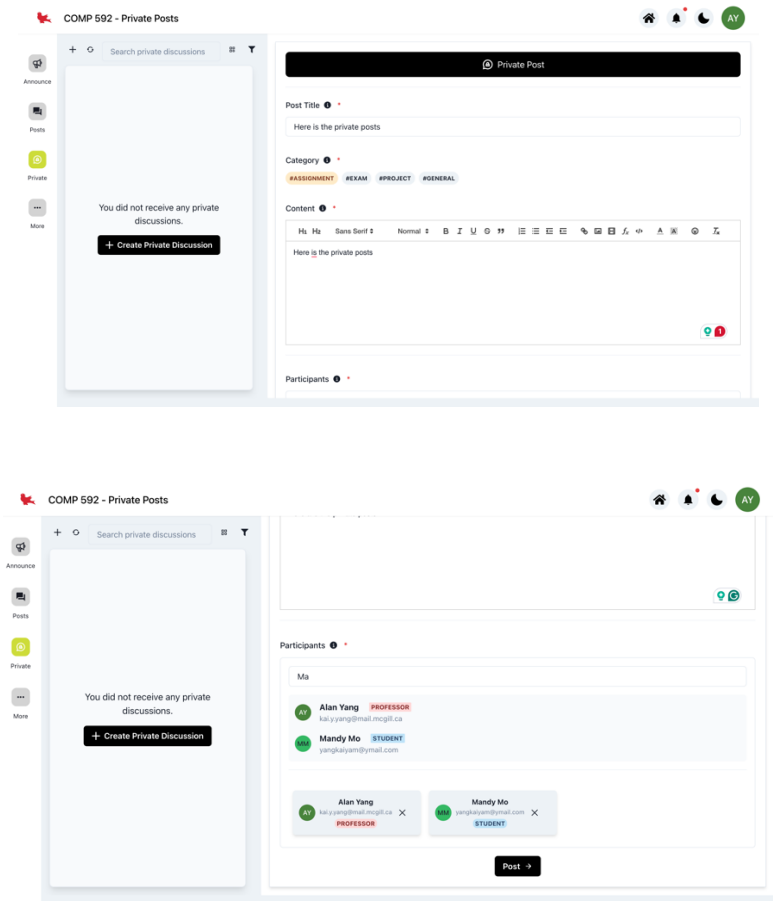
(Figure 32 – The real-time notification of the comment)

VI. Private Post Discussion

Private Post Creation

The Private Post Page can be accessed by clicking the “lock” icon from the left sidebar. Every user can create the private discussion, they will provide the post title, category, and content, as well as the discussion participants. They will search the users by email or name and add them to be the post

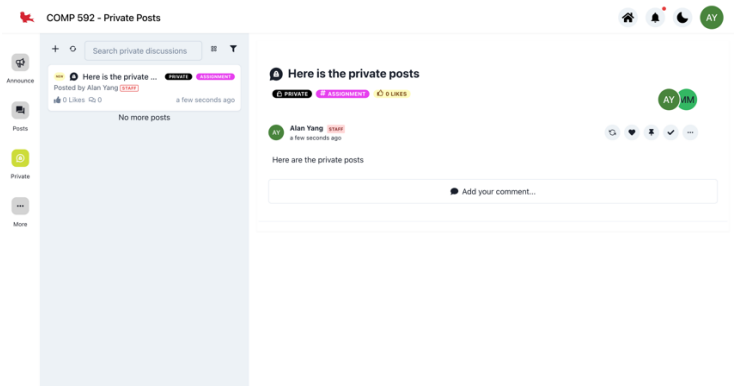
participants. Only that post participants can view or access to the post.



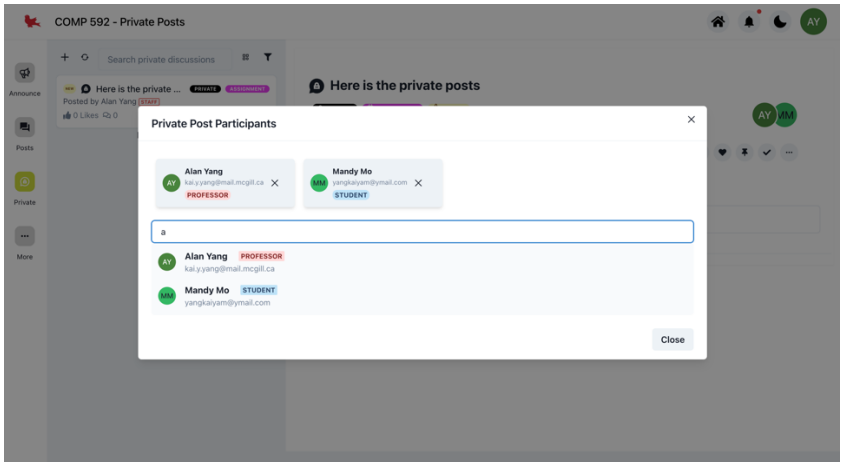
(Figure 33 – The Private Post Creation Page)

Private Post Content

For the content of the private post, it is similar as the public post, but it will have a participant avatar on the top right corner. The avatar is to access the participant management modal, the user can add more users to the posts, remove the users from the posts, shown in the Figure 35.



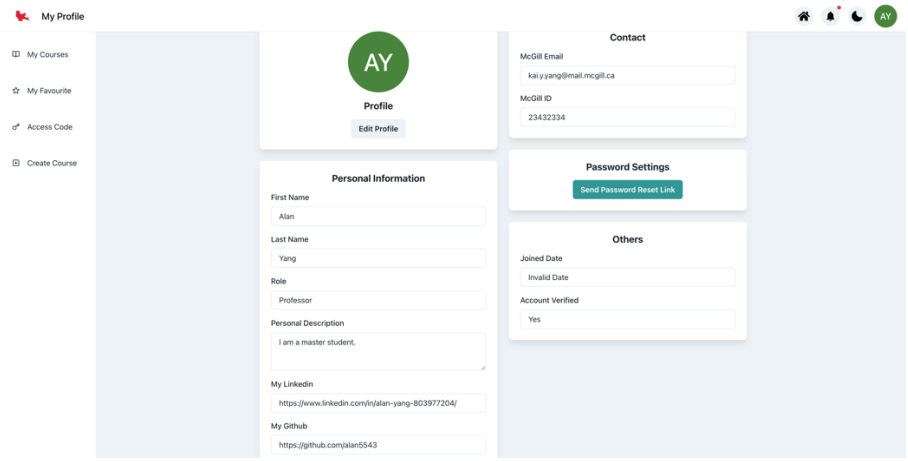
(Figure 34 – The Private Post Content Page)



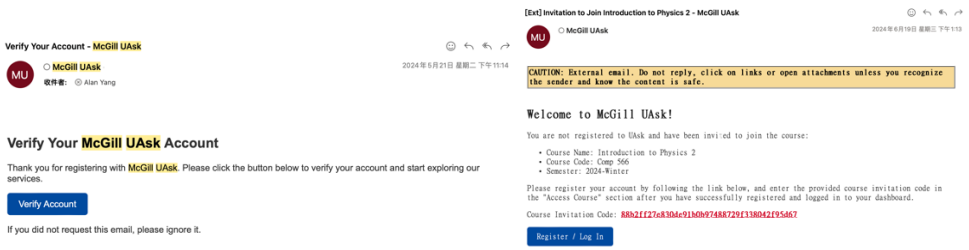
(Figure 35 – The Private Post Participants Management)

VII. User Managements

The users can view their profile by clicking the avatar on the navigation bar. The profile page is allowed to modify the user information, or adding some personal connection (such as GitHub and LinkedIn links), or adding the personal description for more interaction among students and faculty. Also, the user can reset the password in the page, shown in Figure 36.



(Figure 36 – The Personal Profile)

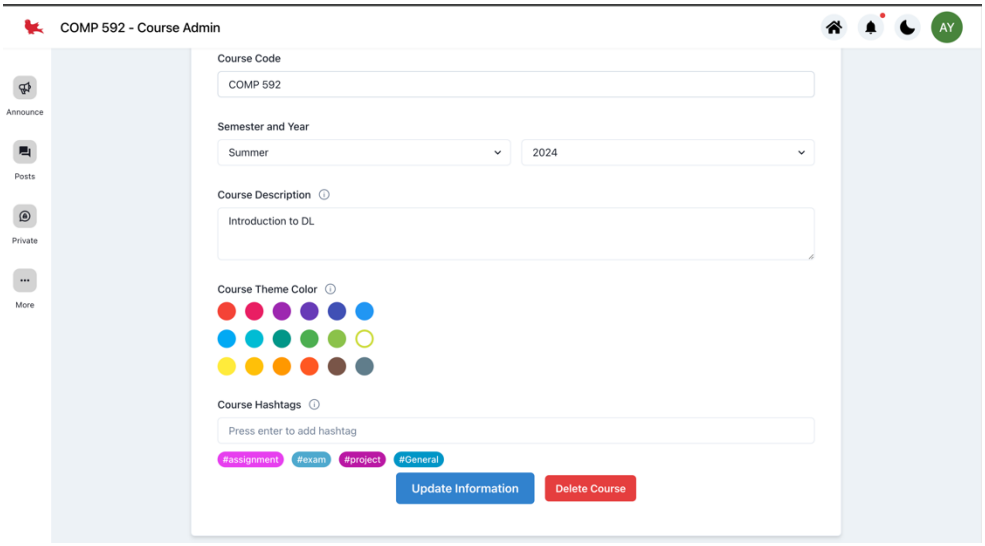


(Figure 37 – The Verification Links sending via email)

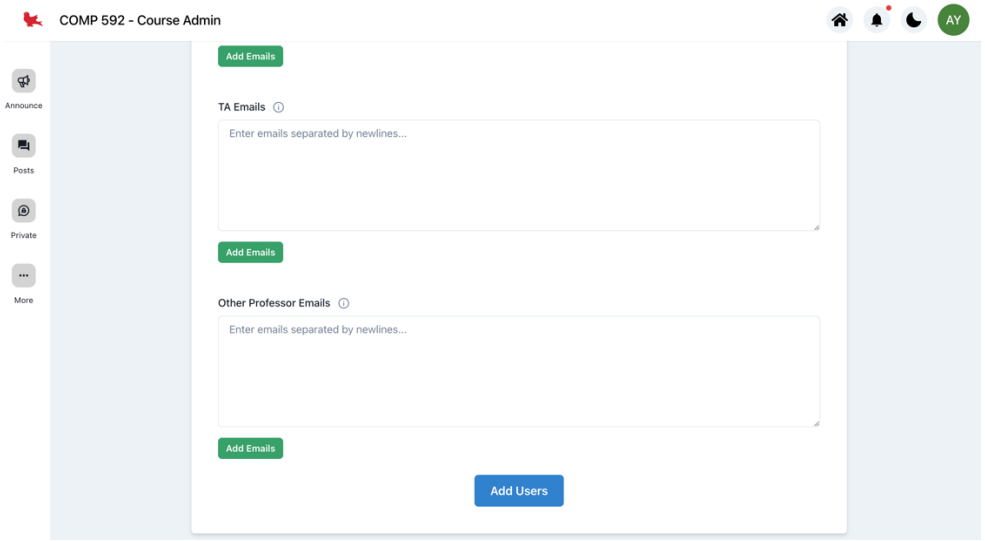
In Figure 37, they show the demonstration emails sending to the users, when they need to do the account verification or receive the course invitation code.

VIII. Course Managements

Each course discussion board can be managed by the staffs (TAs and Professors) in the course admin page. The staffs can update the course information, add more hashtags, and delete the course. Once the course is deleted, all the related posts of the course are deleted too.

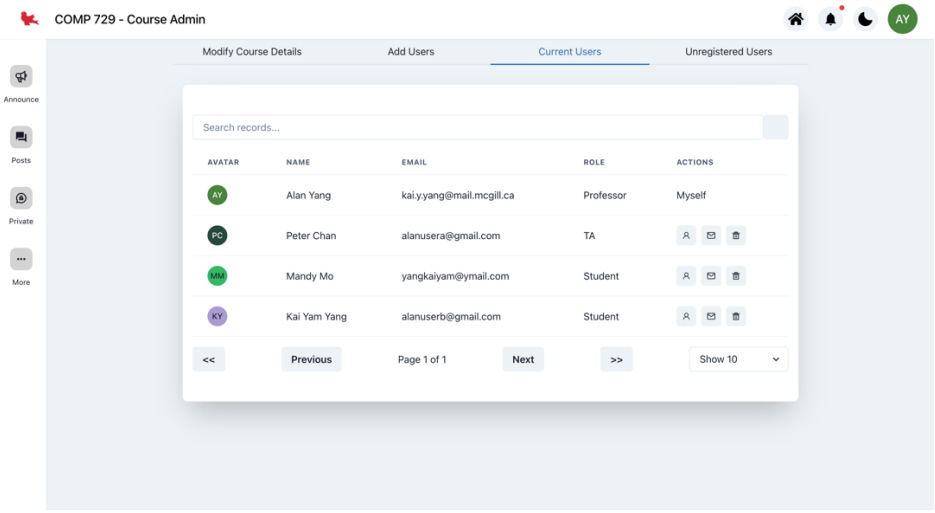


(Figure 38 – The Course Information Update in Course Admin Page)



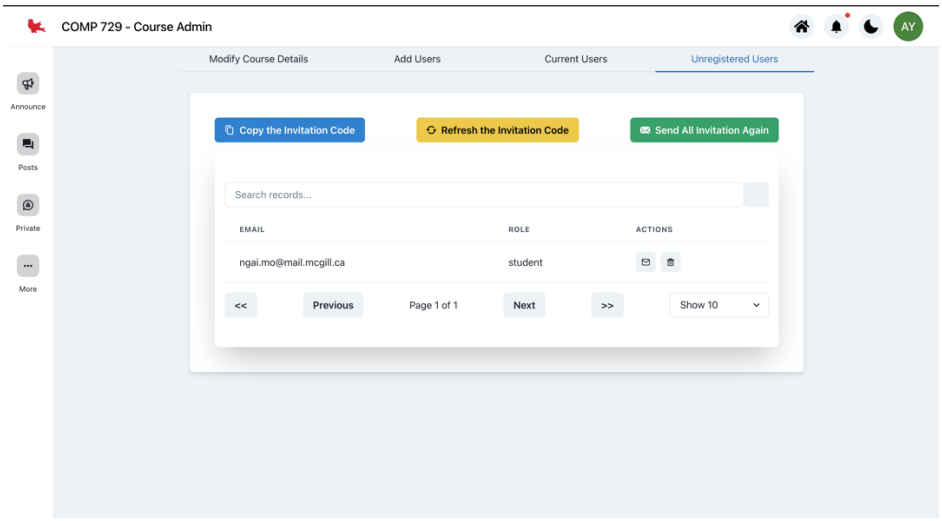
(Figure 39 – The Add Users in Course Admin Page)

Staff members can also add more users to the course, as shown in Figure 39. If the user's email is not registered in the system, an invitation code will be sent to that email. As illustrated in Figure 40, staff can manage enrolled course members by finding user information, viewing their profiles, and sending emails directly to users. Staff have the authority to delete users from the course. If a user is deleted, they are immediately removed from the dashboard and redirected to an access denied page via WebSocket.



(Figure 39 – The Add Users in Course Admin Page)

Additionally, the staff can manage the unregistered user email in Figure 40. They can send all the invitation email to the unregistered user again, or they can remove the unregistered emails. Also, they can copy the invitation code or refresh code here.



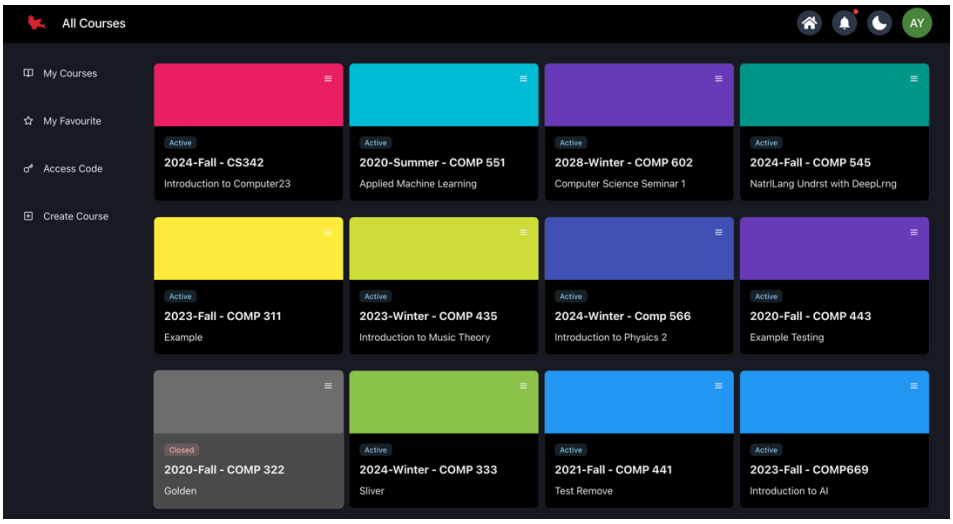
(Figure 40 – The Add Users in Course Admin Page)

c. UI/UX Design

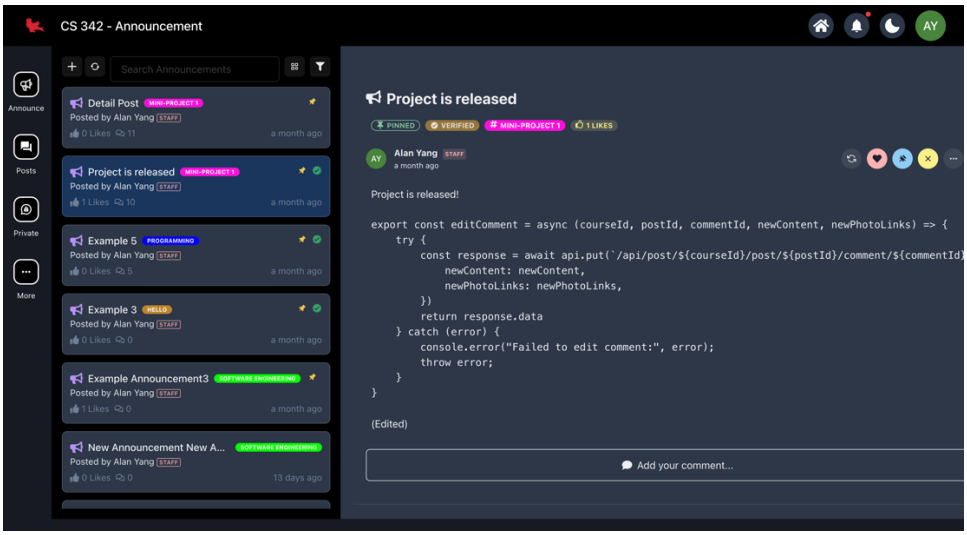
The McGill UAsk application is crafted with a strong focus on user experience (UX) and user interface (UI) design to ensure it is both intuitive and visually appealing across all devices.

I. Dark Mode

A standout feature of the UI design is the support for dark mode, implemented using Chakra UI. This feature not only caters to user preferences but also helps the user to use the app in low-light conditions. Dark mode is seamlessly integrated, ensuring that all elements, from text to buttons, maintain optimal contrast and readability.



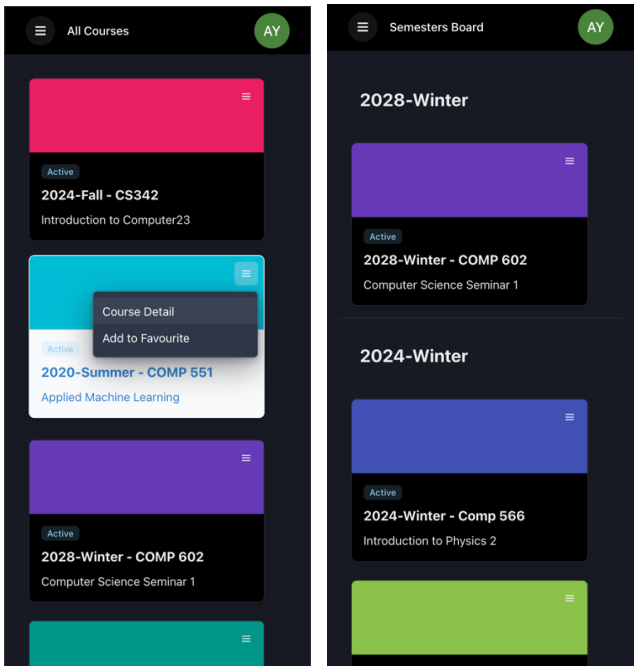
(Figure 41 – The Course Dashboard in Dark Mode)



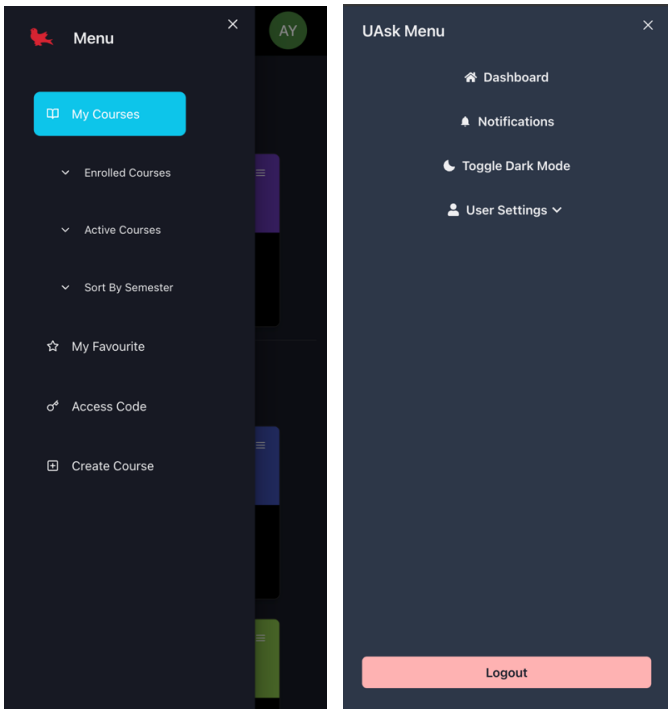
(Figure 42 – The Discussion Board in Dark Mode)

II. Responsive UI Design

The application is fully responsive, that the UI layout adapts smoothly to different screen sizes, including phones, tablets, and laptops. This ensures that users can access the platform from any device without losing functionality or ease of use. The layout adjusts dynamically, keeping navigation simple and ensuring all features are easily accessible.

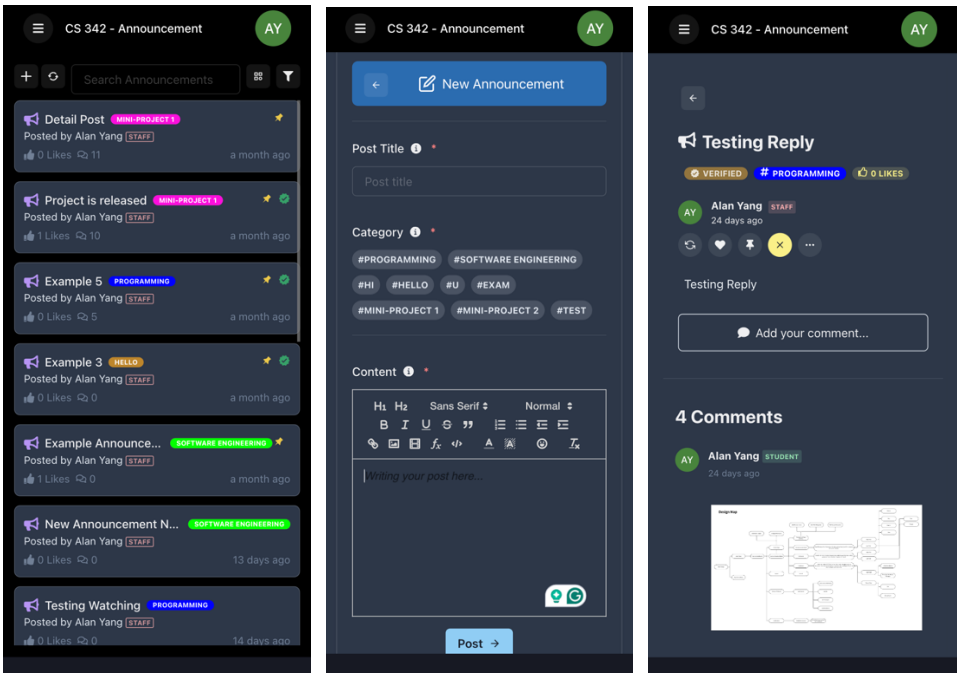


(Figure 43 – The Courses Dashboard in Mobile Layout)

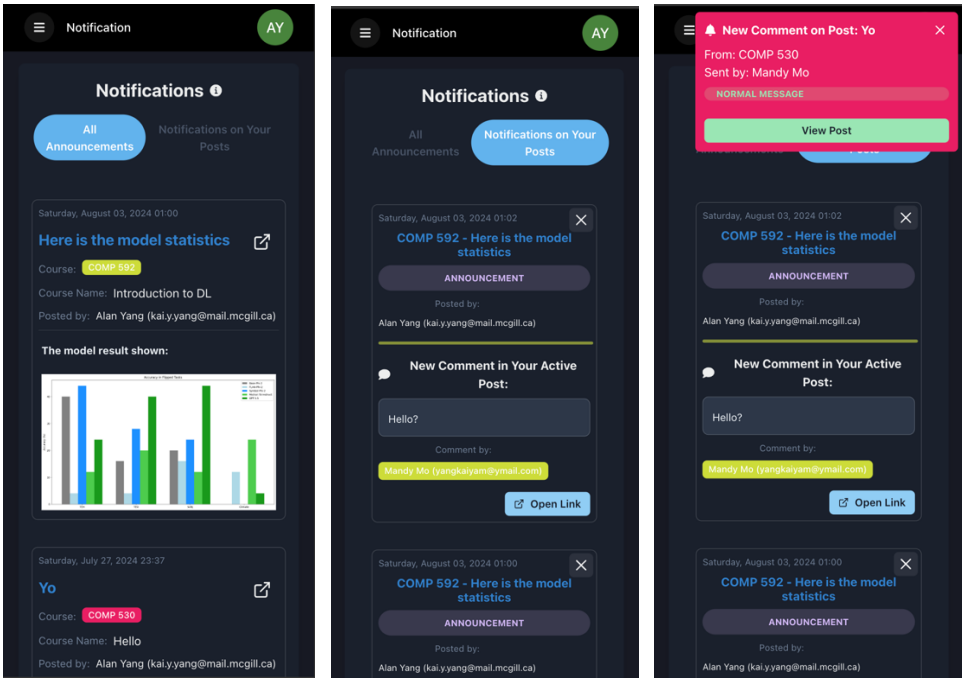


(Figure 44 – The Navigation Bar in Mobile Layout)

In Figure 44, the layout adapts based on screen size. On a laptop-sized screen, a sidebar displays the main features, while a top navigation bar shows common functions, such as viewing user profiles. When the screen size decreases, the sidebar transforms into a left drawer, and the navigation bar becomes a top full drawer for easier access on smaller devices.



(Figure 45 – The Posts Board in Mobile Layout)

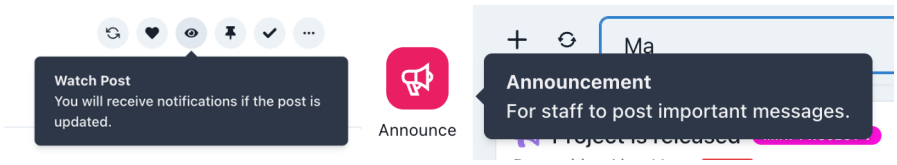


(Figure 46 – The Notifications in Mobile Layout)

In Figure 45, the original posts board on the laptop screen is split into two sections: the posts list on the left and the post content on the right.

III. CTA and feedback

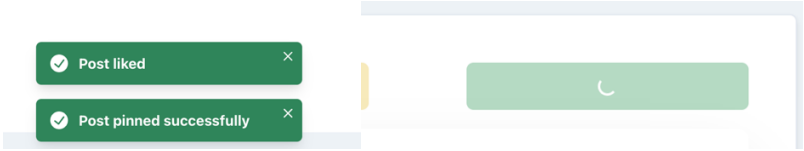
Call to Action (CTA) elements are a crucial part of the design, strategically placed to guide users through key tasks such as creating posts, managing courses, and accessing notifications. These CTAs are designed to be easily noticeable and encourage user interaction, enhancing the overall efficiency and user-friendliness of the application. For example, tooltips are implemented around buttons to help users understand how to use different features of the application.



(Figure 47 – The Tooltips Usage in McGill UAsk)

Feedback is also our consideration of the UI/UX app design. Users receive immediate, clear feedback for their actions, such as submitting a post, liking a comment, or adding a course. This feedback helps confirm that actions have been completed successfully, enhancing user confidence and satisfaction. The toasts and modals are acted as the feedback after the user did the important actions.

Also, for some buttons to call the endpoint service, it might have some delay, it will show a loading icon during the server is processing for the user to know the situation of the app.



(Figure 48 – The Feedbacks Design in McGill UAsk)

Future Works

The McGill UAsk platform has successfully implemented essential discussion and interaction tools for students and faculty. However, several enhancements could further elevate the platform's capabilities.

TAs Performance Tracking

This feature is for professors to monitor TA performance that could be beneficial. The system can log and count metrics such as the number of times TAs comment, how frequently they are online, and how often they assist students. The data could be visualized on a management page with graphs to show the TAs performance across different courses. The feature may be created by implementing a new data collection to store each TAs performance schema. Such insights would help professors assess the effectiveness and engagement of TAs, ensuring that students receive the best possible support.

Chatroom Integration

A chatroom feature, like Slack channels, was originally planned but replaced with private posts due to server load concerns. Reintroducing chatrooms could significantly benefit casual communication. Integrating third-party services like Firebase, SendBird, or Twilio would manage chat functionalities more efficiently, reducing the burden on the server and ensuring smooth performance.

Enhanced Communication Tools

Expanding communication tools to include tagging, polling, and collaborative features like shared drawing boards would enhance user interaction. Also, for more efficient communication, multilingual support is also important. Implementing multilingual support would make the platform more inclusive.

AI Integration

Incorporating AI could transform the platform by introducing a chatbot to answer common student questions, reducing reliance on professors and TAs. Additionally, AI could monitor and flag inappropriate content, ensuring a safe environment. These AI features would enhance both user support and content moderation.

Advanced Search Functionality

Our current search engine supports basic queries based on post titles, author names, or emails. However, enhancing this functionality to allow more advanced searches could significantly improve usability. For example, implementing a full-text search across all post content, including comments and replies, would allow users to find specific information more easily.

Scalability and Performance Optimization

To handle increased traffic, migrating to cloud-based services and implementing load balancing are essential. These measures will ensure the platform remains responsive and efficient as it grows.

Expanded Notification System

Our current application only supports the in-app notification alert, but the notification system can be more comprehensive. Enhancing the notification system to include email alerts would keep users better informed. We might try to use the services like AWS SNS or SendGrid would manage large volumes of notifications without straining the server, ensuring timely updates.

Conclusion

The McGill UAsk project offers a comprehensive communication tool designed for the academic environment at McGill University. By combining the strengths of existing platforms like Ed and Slack with new, innovative features specifically developed for students and faculty, McGill UAsk delivers a powerful and user-friendly platform for academic discussions.

In this project, the key features such as course management, public and private discussions, and a real-time notification system are successfully implemented, all within a responsive and accessible user interface. Applying the framework of MERN stack, coupled with the UI/UX design, has resulted in a platform that not only meets the current needs of its users but also lays a strong foundation for future enhancements.

Looking ahead, there are numerous opportunities to further improve McGill UAsk. Suggestions such as the reintroduction of chatrooms, AI integration, advanced search capabilities, and scalability optimizations will ensure the platform continues to evolve in line with user demands and technological advancements. By continuously refining and expanding the platform's features, McGill UAsk can become a more well-rounded application for the communication between students, TAs, and professors.

To sum up, McGill UAsk has successfully met its primary objectives of enhancing communication and resource sharing at McGill University. It is hoped that it can be a valuable tool for both students and faculty in the future.

Reflection

Working on the McGill UAsk project has been a progressive experience, taking an idea to a concept graphs, then taking a concept from inception to a fully functional application. This journey allowed me to apply and expand my full-stack development skills while bringing my creativity to life. I learned the importance of brainstorming, modular design, and careful planning in phases to keep the project on track. Finding solutions within resource constraints and balancing ambition with practicality were key lessons, especially as I aspire to roles in project management or starting my own company.

Using third-party libraries and services enhanced my technical skills and deepened my understanding of implementing custom solutions. The project also highlighted the importance of creativity in software development, teaching me how to focus ideas to create features that align with customer needs and project goals. This experience has not only strengthened my problem-solving abilities but also inspired me toward future entrepreneurial endeavors.

Bibliography

1. The Project Winners from COMP 307 Course Resources (MyThreads, Pi, SOCSBoards) Retrieved from https://mcgill-my.sharepoint.com/personal/joseph_vybihal_mcgill_ca/_layouts/15/onedrive.aspx?ga=1&id=%2Fpersonal%2Fjoseph%5Fvybihal%5Fmcgill%5Fca%2FDocuments%2FMcGill%20Desktop%2FArchive%2F2023%20Fall%2FCOMP%20307%2FAssignments%2FProject%2FWinners

2. Chakra UI - build accessible React apps with speed.
   Retrieved from https://chakra-ui.com/

3. React - A JavaScript library for building user interfaces.
   Retrieved from https://reactjs.org/

4. Express - Fast, unopinionated, minimalist web framework for Node.js.
   Retrieved from https://expressjs.com/

5. MongoDB - The most popular database for modern apps.
   Retrieved from https://www.mongodb.com/

6. Node.js - Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine.
   Retrieved from https://nodejs.org/

7. Quill -Your powerful rich text editor.
   Retrieved from https://quilljs.com/

8. Nodemailer - Send e-mails with Node.JS
   Retrieved from https://nodemailer.com/

9. WebSocket - The WebSocket API (WebSockets)
   Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

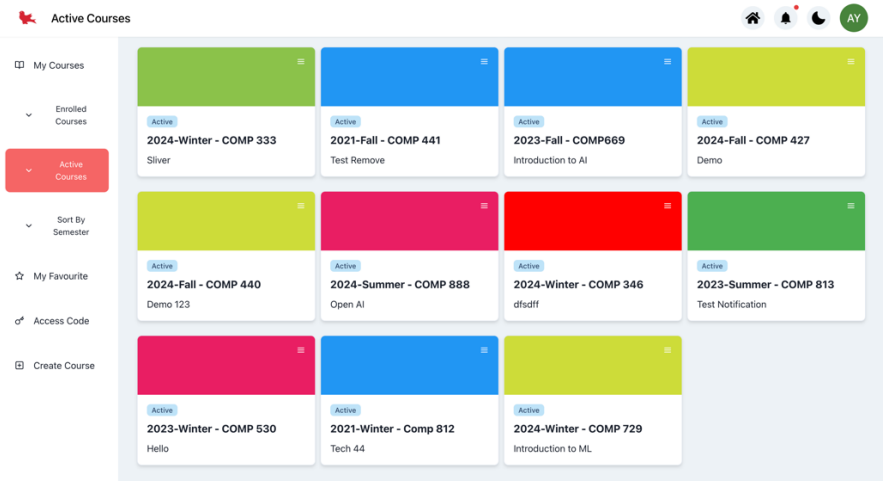
10. Browser Image Compression - JavaScript library for image compression in the browser
    Retrieved from https://www.npmjs.com/package/browser-image-compression

11. JWT - JSON Web Tokens
    Retrieved from https://jwt.io/

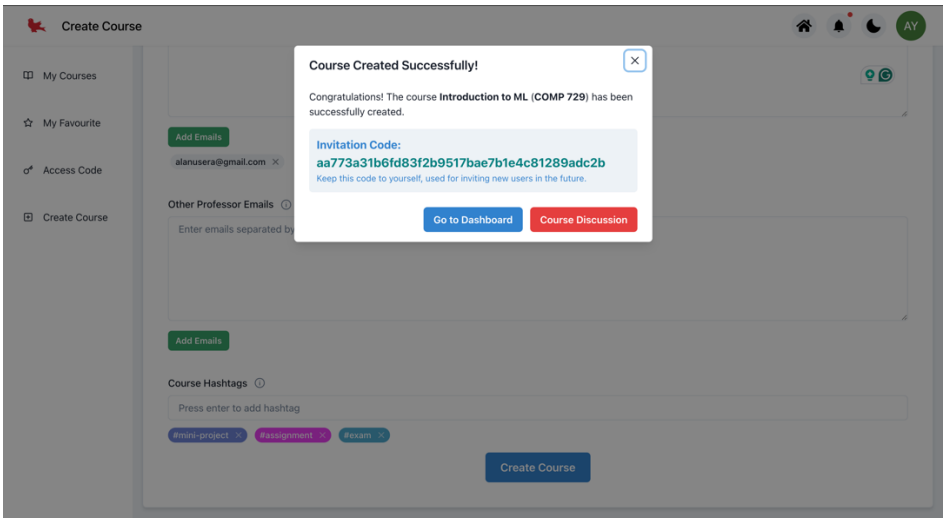
Appendix

1. The Landing Page of McGill UAsk

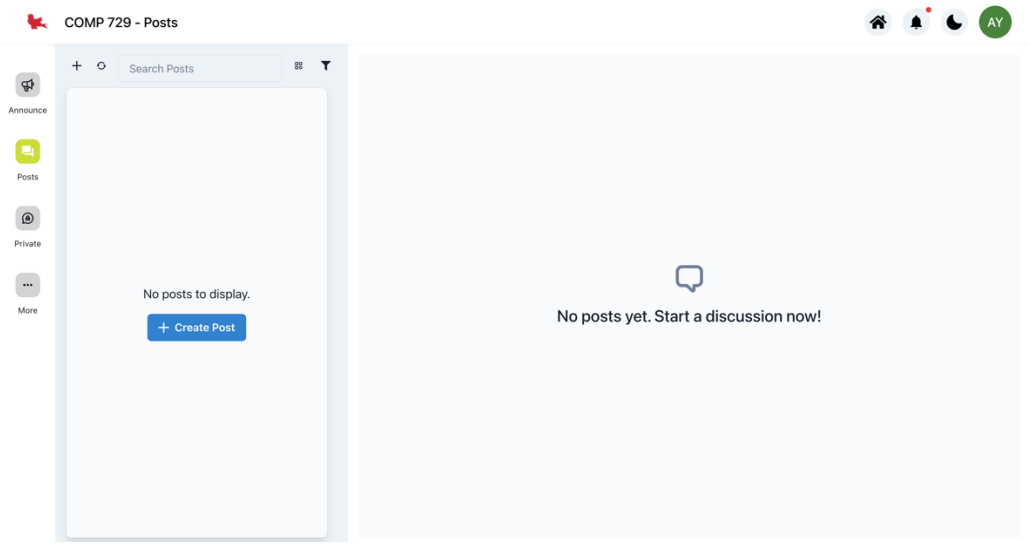
2. The Course Dashboard Showing Active Courses



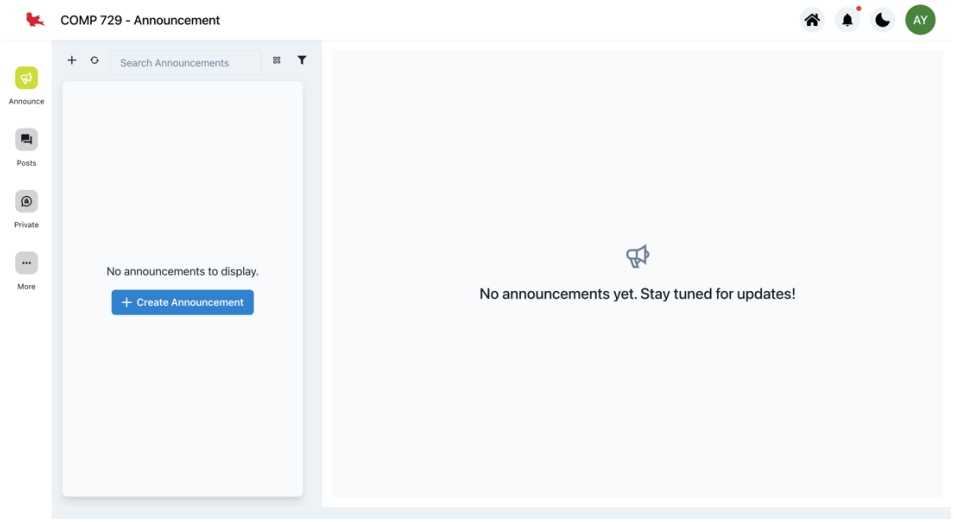
3. The Course Invitation Code Return After Course Creation is successful



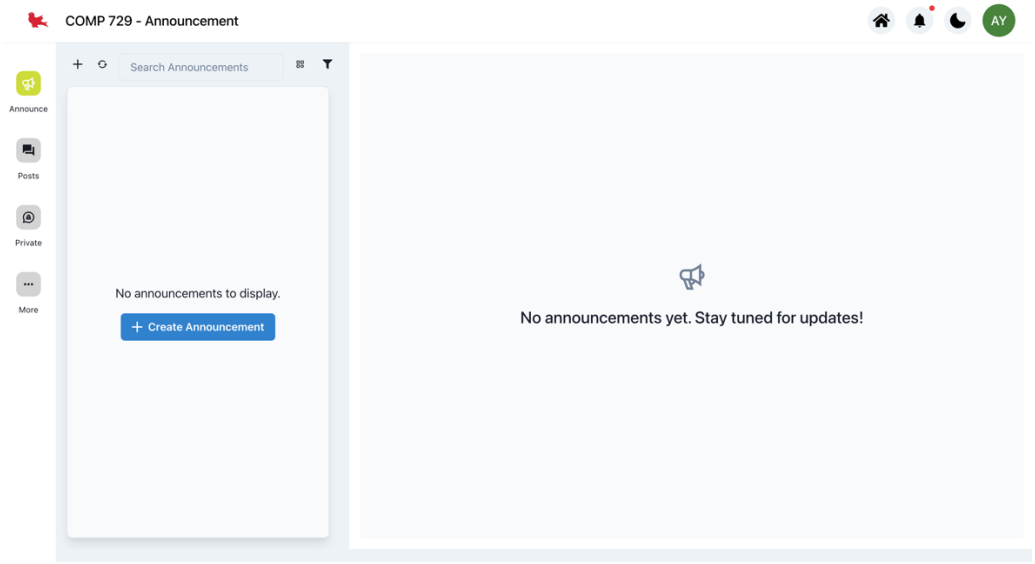
4. The Posts Board when the posts are empty



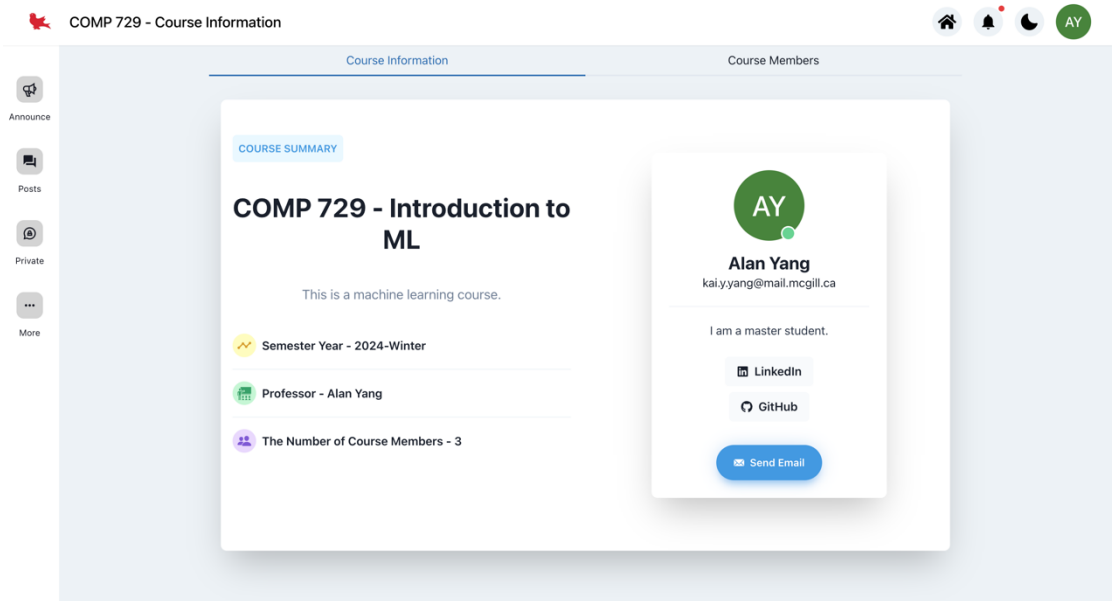
5. The Announcement Board when the posts are empty (Staff View)



6. The Announcement Board when the posts are empty (Student View)



7. The Course Information Page



8. API Endpoint Specifications for Authentication Routes

This section outlines the API endpoints used for user authentication. The endpoints are mainly related to the login and registration features.

Auth Routes			
Endpoint Name	**Endpoint Body**	**Usage**	**Success Response**
POST /api/auth/register	{ user data }	Registers a new user by validating and encrypting user data, creating the user, generating a verification token, and sending a verification email.	201 Created, {message: 'An Email sent to your account please verify'}
GET /api/auth/verify	N/A	Verifies user email by checking user ID and token, updating verification status, and removing the token.	200 OK, {message: 'Email verified successfully'}
POST /api/auth/login	{ login data }	Logs in a user by validating login data, checking user existence and verification, validating password, generating a JWT token, and setting a cookie.	200 OK, { message: 'Login successfully', user: { user data } }
POST /api/auth/forgetPassword	{ email }	Initiates password reset by validating email, checking user existence, generating a reset token, and sending a reset password email.	201 Created, { message: 'An Email sent to your account to reset your password.' }
GET /api/auth/verifyForgetPassword	N/A	Verifies password reset token by checking user ID and token.	200 OK, { message: 'The reset password token is valid' }
PUT /api/auth/updateUserPassword	{ userId, tokenId, new password }	Updates user password by validating new password, verifying reset token, encrypting new password, updating user password, and	200 OK, { message: 'Password updated successfully.' }

		removing the reset token.	
POST /api/auth/logout	N/A	Logs out the user by clearing the authentication cookie.	200 OK, { message: 'Logged out successfully' }

9. API Endpoint Specifications for Protected Routes

Those API endpoints are used for checking the user is authenticated before access the main APIs in the system.

Protected Routes			
Endpoint Name	**Endpoint Body**	**Usage**	**Success Response**
GET /api/protected	N/A	Accesses a protected route that requires user authentication. **Uses the 'protect' middleware to ensure the user is authenticated before returning user data.**	200 OK, { message: 'This is a protected route', user: { user data } }
GET /api/protected/check	N/A	Checks if the user is authenticated by verifying the presence and validity of a JWT token in the cookies.	200 OK, { authenticated: true } if token is valid, { authenticated: false } if token is missing or invalid

10. API Endpoint Specifications for User Routes

This section outlines the API endpoints used for user managements. The endpoints include the retrieval of user profile and user announcements.

User Routes			
Uses the 'protect' middleware to ensure the user is authenticated.			
Endpoint Name	**Endpoint Body**	**Usage**	**Success Response**
PUT /api/users/profile (protect)	{ updated user data }	Updates the authenticated user's profile with the provided data.	200 OK, { message: 'User profile updated successfully', user: { id, firstName, lastName, mcgillEmail, mcgillID, verified, picture, description, role, courses, joinedDate, linkedin, github } }
GET /api/users/profile/ (protect)	N/A	Fetches the profile information of another user by their user ID.	200 OK, { user: { id, firstName, lastName, mcgillEmail, mcgillID, verified, picture, description, role, courses, joinedDate, linkedin, github } }
GET /api/users/my-posts (protect)	N/A	Fetches the authenticated user's posts.	200 OK, { post: [{ isAnnouncement, title, postTime, courseName, category, isVerified, openLink }] }
GET /api/users/unread-announcements (protect)	N/A	Fetches the unread announcements for the authenticated user.	200 OK, { announcements: [{ title, code, from, postedByEmail, postedBy, postTime, link, courseLink, courseColor, content }] }

11. API Endpoint Specifications for Course Routes

The API endpoints used for the course management are outlined here. The feature includes the course creation and courses fetching.

Course Routes			
Uses the 'protect' middleware to ensure the user is authenticated.			
Endpoint Name	**Endpoint Body**	**Usage**	**Success Response**
POST /api/courses/create (protect, checkProfessor)	{ course data }	Creates a new course. The middleware 'checkProfessor' is to ensure the user has the correct role	201 Created, { message: 'Course created successfully', courseId, courseName, courseCode, nonExistentUsers, invitationToken }
POST /api/courses/access (protect)	{ invitation token, courseRole }	Accesses a course using an invitation token.	200 OK, { message: 'You have successfully enrolled in the course {courseCode} as {role}.' }
GET /api/courses/getAll (protect)	N/A	Gets all courses associated with the authenticated user.	200 OK, { courses: [{ courseName, courseCode, courseStatus, courseColor, courseSemester }] }
GET /api/courses/getActive (protect)	N/A	Gets all active courses associated with the authenticated user.	200 OK, { activeCourses: [{ courseName, courseCode, courseStatus, courseColor, courseSemester }] }
GET /api/courses/getSemester (protect)	N/A	Gets all courses sorted by semester for the authenticated user.	200 OK, { semesterCourses: { semester: [{ courseName, courseCode, courseStatus, courseColor, courseSemester }] } }
GET /api/courses/getPinCourses (protect)	N/A	Gets the pinned courses for the authenticated user.	200 OK, { pinnedCourses: [{ courseName, courseCode, courseStatus, courseColor, courseSemester }] }
PUT /api/courses/:courseId/pin	N/A	Pins a course for the authenticated user.	200 OK, { message: 'Course pinned successfully', pinCourses:

(protect)			[{ courseName, courseCode, courseStatus, courseColor, courseSemester }] }
PUT /api/courses/:courseId/unpin (protect)	N/A	Unpins a course for the authenticated user.	200 OK, { message: 'Course unpinned successfully', pinCourses: [{ courseName, courseCode, courseStatus, courseColor, courseSemester }] }
GET /api/courses/:courseId (protect)	N/A	Gets the details of a specific course by course ID.	200 OK, { id: courseId, name: courseName, code: courseCode, semester: courseSemester, status: courseStatus, description: courseDescription, lecturers: [{ firstName, lastName, mcgillEmail }], tas: [{ firstName, lastName, mcgillEmail }] }

12. API Endpoint Specifications for Course Detail Routes

The API Endpoints are related to the Course Detail control. The endpoints can get the status and information of the particular course.

Course Details Routes			
✧ **Uses the 'protect' middleware to ensure the user is authenticated.** ✧ **Uses the 'checkUserInCourse' middleware to ensure the user is enrolled in the course already**			
Endpoint Name	**Endpoint Body**	**Usage**	**Success Response**
GET /api/courseDetail/:courseId/check (protect, checkUserInCourse)	N/A	Gets the user's permission for the course discussion board.	200 OK, { id, name, code, color, status, semester, description, hashtags, role }
GET /api/courseDetail/:courseId/getInfo (protect, checkUserInCourse)	N/A	Gets the information of the course	200 OK, { name, code, semester, description,

		discussion board.	professors: [{ name, email }], numOfMembers }
GET /api/courseDetail/:courseId/getMembers (protect, checkUserInCourse)	N/A	Gets the user list in the course discussion board.	200 OK, { members: [{ userId, fullName, email, role }] }
PUT /api/courseDetail/:courseId/removeMember/me (protect, checkUserInCourse)	N/A	Removes the authenticated user from the course.	200 OK, { message: 'Successfully removed from the course' }
GET /api/courseDetail/:courseId/search-users (protect, checkUserInCourse)	N/A	Searches for users in the course.	200 OK, { users: [{ userId, name, email, role }] }

13. API Endpoint Specifications for Course Detail Admin Routes

The endpoints are related to the course detail administration, and they are only restricted for the usage from TA and professors. The students accounts are not allowed to use the endpoints.

Course Detail Admin Routes			
✧ **Uses the 'protect' middleware to ensure the user is authenticated.** ✧ **Uses the 'checkUserInCourse' middleware to ensure the user is enrolled in the course already.** ✧ **Uses the 'checkDiscussionAdmin' middleware to check the user is the course admin (Professors/ TAs) in the given course**			
Endpoint Name	**Endpoint Body**	**Usage**	**Success Response**
POST /api/courseDetail/:courseId/admin/update (checkDiscussionAdmin)	{ name, code, color, status, description, semester, hashtags }	Updates the course discussion information.	200 OK, { message: 'Course updated successfully', course: { updated course data } }
POST /api/courseDetail/:courseId/admin/addUser (checkDiscussionAdmin)	{ user data }	Adds users to the course.	200 OK, { message: 'Users processed successfully', addedUsers, invitedUsers, errorMessages }
PUT /api/courseDetail/:courseId/admin/removeMember /users/:userId (checkDiscussionAdmin)	N/A	Removes a user from the course.	200 OK, { message: 'User removed from course successfully' }
PUT /api/courseDetail/:courseId/admin/removeMember /unregisteredUsers (checkDiscussionAdmin)	{ removeEmail }	Removes an unregistered user from the course.	200 OK, { message: 'Email {removeEmail} removed successfully' }

GET /api/courseDetail/:courseId/admin/getUnregisteredUsers (checkDiscussionAdmin)	N/A	Gets the list of unregistered users.	200 OK, { unregisteredUsers : [{ email, role }] }
PUT /api/courseDetail/:courseId/admin/close (checkDiscussionAdmin)	N/A	Closes the course.	200 OK, { message: 'The Course {courseCode} is closed, thank you.' }
PUT /api/courseDetail/:courseId/admin/refreshInvitationCode (checkDiscussionAdmin)	N/A	Refreshes the course invitation code.	200 OK, { message: 'Course invitation code refreshed successfully', code: { newInvitationCode } }
GET /api/courseDetail/:courseId/admin/invitationCode (checkDiscussionAdmin)	N/A	Gets the course invitation code.	200 OK, { invitationCode }
POST /api/courseDetail/:courseId/admin/sendInvitation (checkDiscussionAdmin)	{ invitedEmail }	Sends the course invitation code via email.	200 OK, { message: 'Course invitation code is sent to {invitedEmail} successfully' }
DELETE /api/courseDetail/:courseId/admin/delete (checkDiscussionAdmin)	N/A	Deletes the course.	200 OK, { message: 'Course successfully removed' }

14. API Endpoint Specifications for Post Routes

Those endpoints are used for post management. All the posts features are described, including the public and private posts creation, comments and replies, as well as posts side functions.

Post Routes			
✧ **Uses the 'protect' middleware to ensure the user is authenticated.** ✧ **Uses the 'checkUserInCourse' middleware to ensure the user is enrolled in the course already.**			
Endpoint Name	**Endpoint Body**	**Usage**	**Success Response**
POST /api/post /:courseId/post (protect, checkUserInCourse)	{ post data }	Creates a new post in the course.	201 Created, { message: 'Post created successfully', post: { post data } }
GET /api/post /:courseId/posts (protect, checkUserInCourse)	N/A	Fetches all posts in the course.	200 OK, { posts: [{ post data }] }
GET /api/post /:courseId/post/:postId (protect, checkUserInCourse)	N/A	Fetches the details of a specific post including comments.	200 OK, { post: { post data }, comments: [{ comment data }] }
PUT /api/post /:courseId/post/:postId/pin (protect, checkUserInCourse)	N/A	Pins a post in the course.	200 OK, { message: 'Post pinned successfully' }
PUT /api/post/:courseId/post/:postId/unpin (protect, checkUserInCourse)	N/A	Unpins a post in the course.	200 OK, { message: 'Post unpinned successfully' }
PUT /api/post/:courseId/post/:postId/resolve	N/A	Marks a post as	200 OK, { message:

(protect, checkUserInCourse)		resolved.	'Post marked as resolved' }
PUT /api/post/:courseId/post/:postId/unresolve (protect, checkUserInCourse)	N/A	Marks a post as unresolved.	200 OK, { message: 'Post marked as unresolved' }
PUT /api/post/:courseId/post/:postId/like (protect, checkUserInCourse)	N/A	Likes a post.	200 OK, { message: 'Post liked successfully' }
DELETE /api/post/:courseId/post/:postId/like (protect, checkUserInCourse)	N/A	Unlikes a post.	200 OK, { message: 'Post unliked successfully' }
PUT /api/post/:courseId/post/:postId/watching (protect, checkUserInCourse)	N/A	Starts watching a post for updates.	200 OK, { message: 'Started watching post' }
PUT /api/post/:courseId/post/:postId/unWatching (protect, checkUserInCourse)	N/A	Stops watching a post.	200 OK, { message: 'Stopped watching post' }
GET /api/post/:courseId/search (protect, checkUserInCourse)	N/A	Searches for posts in the course.	200 OK, { posts: [{ post data }] }
POST /api/post/:courseId/post/:postId/comment (protect, checkUserInCourse)	{ comment data }	Adds a comment to a post.	201 Created, { message: 'Comment added successfully', comment: { comment data } }
PUT /api/post/:courseId/post/:postId/comment/:commentId/star (protect, checkUserInCourse)	N/A	Stars a comment.	200 OK, { message: 'Comment

			starred successfully' }
PUT /api/post/:courseId/post/:postId/comment/:commentId/delete (protect, checkUserInCourse)	N/A	Deletes a comment.	200 OK, { message: 'Comment deleted successfully' }
DELETE /api/post/:courseId/post/:postId/delete (protect, checkUserInCourse)	N/A	Deletes a post.	200 OK, { message: 'Post deleted successfully' }
GET /api/post/:courseId/post/:postId/readStat (protect, checkUserInCourse)	N/A	Fetches the read statistics of a post.	200 OK, { readStatistics: { readStat data } }
PUT /api/post/:courseId/post/:postId/comment/:commentId/edit (protect, checkUserInCourse)	{ comment data }	Edits a comment.	200 OK, { message: 'Comment edited successfully', comment: { comment data } }
PUT /api/post/:courseId/post/:postId/edit (protect, checkUserInCourse)	{ post data }	Edits a post.	200 OK, { message: 'Post edited successfully', post: { post data } }
GET /api/post/:courseId/post/:postId/participants (protect, checkUserInCourse, checkParticipantPermission)	N/A	Gets the participants of a private chat in a post.	200 OK, { participants: [{ participant data }] }
DELETE /api/post/:courseId/post/:postId/participants/:userId (protect, checkUserInCourse, checkParticipantPermission)	N/A	Removes a participant from a private chat	200 OK, { message: 'Participant removed

		in a post.	successfully' }
POST /api/post/:courseId/post/:postId/participants (protect, checkUserInCourse, checkParticipantPermission)	{ participant data }	Adds a participant to a private chat in a post.	201 Created, { message: 'Participant added successfully', participant: { participant data } }

15. API Endpoint Specifications for Upload Routes

The API is for image and documentation uploading to the server. There is an image compression processing when the user is uploading their images.

Upload Routes			
✧ **Uses the 'protect' middleware to ensure the user is authenticated.** ✧ **Uses the 'checkUserInCourse' middleware to ensure the user is enrolled in the course already.**			
Endpoint Name	**Endpoint Body**	**Usage**	**Success Response**
POST /api/uploads/	FormData with a single file (key: 'file')	Uploads a file for the specified course. Requires user authentication and course membership.	200 OK, { "path": "file upload URL" }

16. API Endpoint Specifications for Notification System Routes

Those endpoints are related to the notification system and they includes the notification fetching, and the removal of the notification.

Notification Routes			
✧ **Uses the 'protect' middleware to ensure the user is authenticated.**			
Endpoint Name	**Endpoint Body**	**Usage**	**Success Response**
GET /api/notifications/get	None	Fetches all notifications for the authenticated user.	200 OK, Array of notification objects, including post details, course details, and comments if any.
PUT /api/notifications/remove/ :notificationId	None	Removes a specific notification by its ID for the authenticated user.	200 OK, { "message": "Notification removed successfully", "notificationId": "ID of removed notification" }