A blue and white logo

Description automatically generated

Software Engineering Department

Ort Braude College

Capstone Project Phase B

Supervisor:

Natali Levi- Ph.D.

Students:

Shymaa Titi

Carolin Mansour

**Code: 23-2-D-5**

**Notifications management system**

**Contents**

**Abstract------------------------------------------------------------------------------------------------- 2-3**

**1. Introduction--------------------------------------------------------------------------------------------4**

# **2.Solution Description--------------------------------------------------------------------------------4-8**

**2.1 Activity Diagram--------------------------------------------------------------------------------7**

**2.2 Package diagram---------------------------------------------------------------------------------8**

# **3.Development Process Description-------------------------------------------------------------- 9-13**

**3.1Tools Used and Client Interface During Development-----------------------------------9**

**3.2 architecture of the system---------------------------------------------------------------- 10-13**

**4.Testing--------------------------------------------------------------------------------------------- 13-15**

**5.Results and Insights--------------------------------------------------------------------------------- 16**

**6.Challenges Faced and Solutions-------------------------------------------------------------------17**

**7.User guide------------------------------------------------------------------------------------------18-28**

# **8.Maintenance Guide------------------------------------------------------------------------------ 28-45**

**8.1 installation instructions--------------------------------------------------------------------- 28**

**8.2 Front-end------------------------------------------------------------------------------------29-31**

**8.3 Backt-end-----------------------------------------------------------------------------------32-45**

**Abstract**

The notification management system described here empowers users to send notifications to targeted recipients without the need for handling or processing responses. It serves as a user-centric platform that enables individuals or groups to disseminate important information, updates, or alerts through various communication channels such as email, SMS, push notifications, or in-app messages. The system provides a user-friendly interface through which users can compose and customize notification messages. They can specify the desired timing or schedule for sending the notifications, select the intended recipients or recipient groups, and choose the appropriate delivery channels. The system ensures that notifications are sent according to the users' preferences and predefined schedules. Administrators or system operators play a role in configuring the system settings and granting users the necessary permissions to send notifications. They oversee the overall functionality and ensure the smooth operation of the system, but do not directly engage in handling the responses to the notifications. Recipients of the notifications receive the messages through their preferred communication channels. Although they may have the ability to respond to the notifications, the focus of this system is on enabling users to broadcast information rather than managing the ensuing interactions. By providing users with the capability to send notifications without the burden of response handling, the system streamlines communication processes, improves information dissemination, and empowers users to efficiently reach their target audience. It simplifies the sending of notifications, enhancing user engagement and facilitating effective communication within the specified scope of the system's capabilities.

Index Terms: Messages, sending messages, message log, Notification.

# Introduction

In the fast-paced landscape of online communication, staying updated on crucial notifications and updates poses a significant challenge, often resulting in users feeling overwhelmed and inundated by the sheer volume of incoming messages from various sources. The struggle to manage these notifications amidst the constant influx can lead to critical information being overlooked or missed entirely, hampering productivity and collaboration efforts. Moreover, coordinating the timing and delivery of messages becomes increasingly complex, particularly when orchestrating communication across diverse teams or groups. Enter our web notifications management system, meticulously crafted to tackle these pressing challenges head-on. At its core, our platform empowers users with a suite of robust features designed to streamline communication processes and elevate the overall user experience. By enabling users to send notifications in channels, facilitating seamless interaction and collaboration among team members, and providing real-time updates on important developments, our system ensures that critical information is efficiently disseminated and received. Additionally, our platform facilitates user-to-user live chat functionality, fostering direct and immediate communication channels to enhance collaboration and productivity further.

One of the standout features of our system is its ability to schedule messages for delivery at a later time, offering unparalleled flexibility and convenience for users. Whether coordinating project timelines, sending reminders for upcoming events, or ensuring that important announcements reach the intended audience at precisely the right moment, our scheduling feature empowers users to optimize their communication strategies with precision and efficiency. In essence, our web notifications management system serves as a comprehensive solution to the challenges of modern communication, providing users with the tools and capabilities they need to navigate the digital landscape effectively. With our platform, users can take control of their communication channels, mitigate information overload, and foster collaboration and productivity across teams and groups. Experience the transformative power of our system and revolutionize the way you communicate online today.

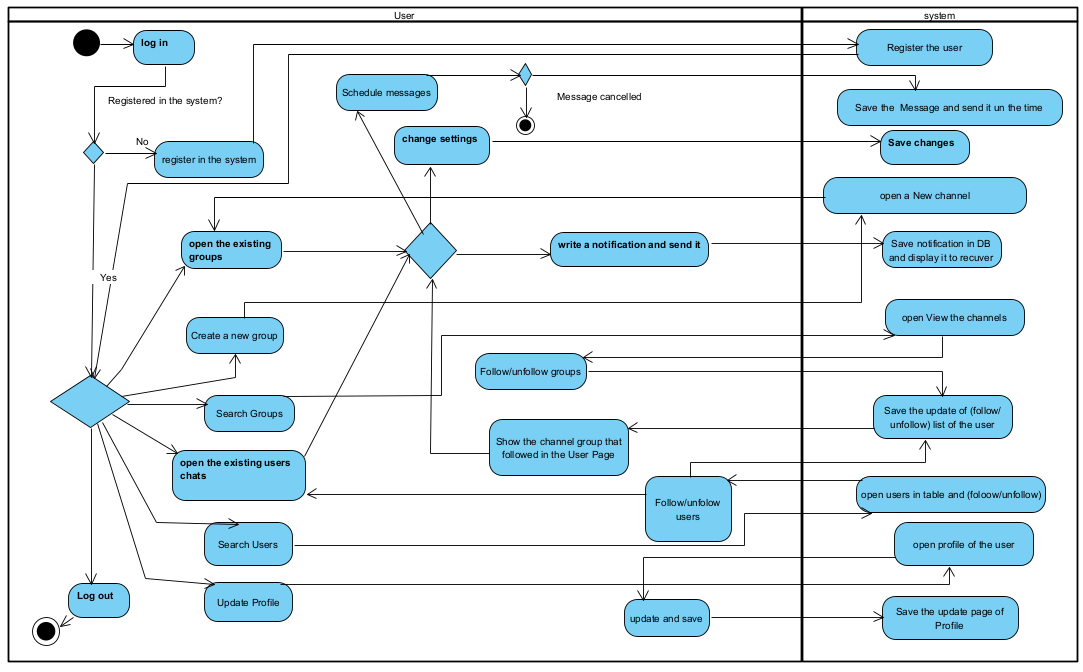
# Solution Description

Our system is a comprehensive messaging platform with several key features:

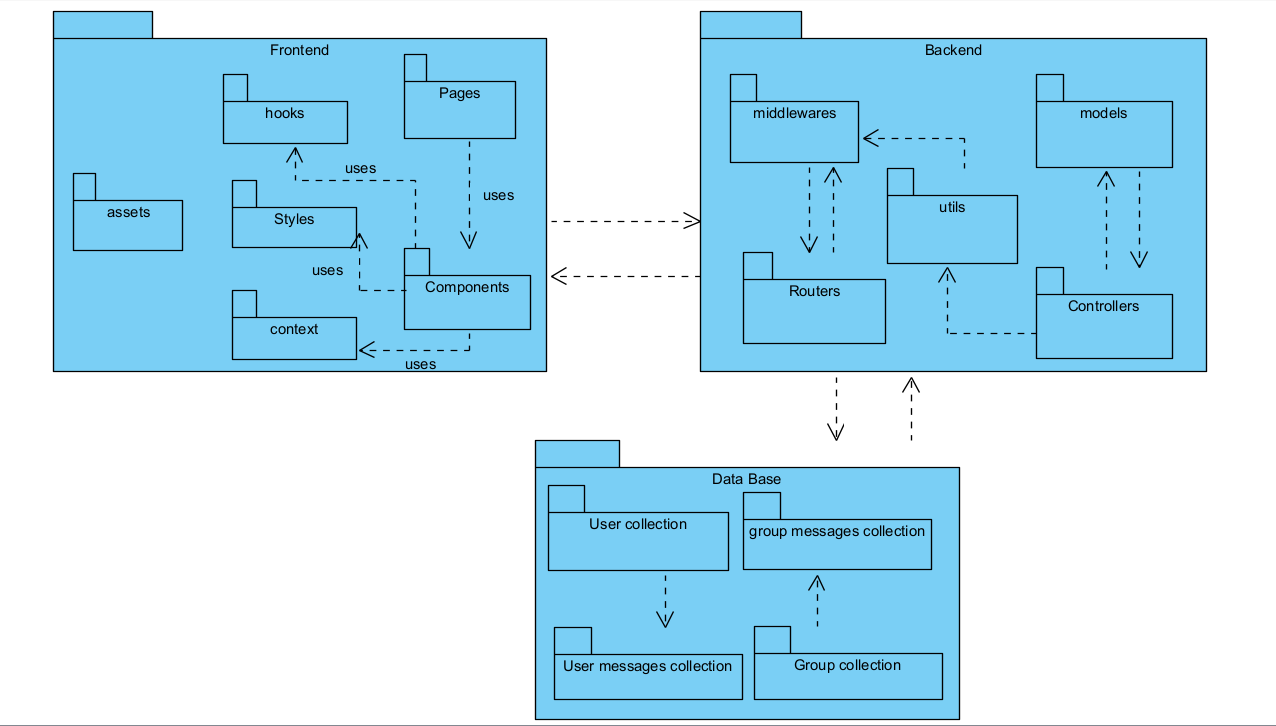
* **Channel Creation**: Users have the ability to create channels upon entering the system. This process typically involves providing a name and possibly a description for the channel. Users may also have the option to customize the channel with features such as a profile picture or banner.
* **Real-Time Messaging**: Once created, channels serve as platforms for real-time messaging. Users can post messages within the channel, and these messages are immediately visible to other users who have access to the channel.
* **Followers and Notifications**: Other users can choose to follow channels that interest them. By following a channel, users receive notifications for every message published within that channel. This ensures that followers stay updated on the latest content and discussions happening within their areas of interest.
* **View-Only Access for Followers**: Followers of a channel have view-only access, meaning they can see the messages posted in the channel but cannot post messages themselves. This distinction helps maintain the integrity of the channel and prevents unauthorized users from cluttering the channel with irrelevant content.
* **Admin Control**: Administrators (admins) have control over channel settings. These settings may include options such as allowing all followers to post in the channel, restricting posting privileges to certain users or roles, moderating content before it's published, or enabling features like threaded conversations or polls.
* **Editing and Rescheduling**: Admins have the ability to edit or reschedule scheduled notifications before they are sent. This allows admins to make adjustments to the content or timing of notifications based on changing circumstances or user feedback.
* **Search Functionality**: Users can access a search functionality within the system that allows them to discover channels created by other users. This search functionality may include filters or sorting options to help users find channels based on various criteria such as keywords, channel names, or categories.
* **Follow Action**: Once users find a channel they're interested in, they have the option to follow it. This action signifies their interest in receiving notifications for new posts made within that channel. Following a channel essentially subscribes the user to updates from that channel.
* **Membership Status**: When a user follows another user's channel, they become a member of that channel. As a member, they gain access to the content posted in the channel and receive notifications whenever new messages are published. However, being a member typically does not grant additional privileges beyond receiving notifications.
* **Notification Settings**: Users may have the option to customize their notification settings for each channel they follow. This could include choosing the types of notifications they wish to receive (e.g., all posts, only posts from specific users, mentions, etc.) and how they receive them (e.g., push notifications, email notifications, in-app alerts, etc.).
* **Unfollowing Channels**: Users also have the ability to unfollow channels at any time. This action stops the user from receiving notifications for new posts in the channel and removes their membership status from that channel.
* **Initiating Chat**: After a user follows another user, they gain the ability to initiate a direct messaging (DM) conversation with them. This action could be performed from various areas within the platform, such as the user's profile, a list of followers, or within a channel where the other user has posted.
* **User Interface**: The direct messaging interface typically includes a chat window where users can exchange text messages in real-time. This window may also support multimedia elements such as images, videos, or file attachments, depending on the system's capabilities.
* **Notifications**: Users receive notifications for new direct messages, ensuring that they are aware of incoming messages even when they are not actively using the platform. These notifications may be delivered via various channels such as push notifications, email alerts, or in-app notifications.
* **Privacy and Security**: Direct messages are typically private and only visible to the users participating in the conversation. The system may implement encryption and other security measures to protect the confidentiality of messages exchanged between users.
* **Message Management**: Users have the ability to manage their direct messages, including actions such as deleting messages, archiving conversations, or blocking/unblocking other users. These features provide users with control over their communication and privacy within the platform.
* **Presence Indicators**: The system may display presence indicators to indicate whether a user is currently online or offline. This helps users gauge the availability of their contacts for real-time communication.
* **Group Chats**: In addition to one-on-one conversations, the system may also support group chats where multiple users can participate in a single conversation thread. Group chats can be initiated by users and may have similar features to one-on-one chats, such as multimedia support and message management options.
* **Personal Information Update**: Users can access a dedicated section within the system to update their personal information. This section typically includes fields for basic details such as name, email address, profile picture/avatar, and any other relevant information needed for user identification or customization within the system.
* **Security Measures**: To ensure data security and user privacy, the system may implement authentication mechanisms before allowing users to update their personal information. This could include requiring users to log in with their credentials or providing additional verification steps such as two-factor authentication.
* **Privacy Settings**: Users may have the option to manage their privacy settings, controlling who can view certain aspects of their profile or personal information. This could include options to make their profile public, visible to only their followers, or completely private.
* **Password Management**: Users might also be able to change their passwords from the profile management section. This is crucial for maintaining account security and preventing unauthorized access.
* **Log Out Functionality**: Users can log out of the system from any page within the platform. This action terminates the user's current session and clears any active authentication tokens, ensuring that the user's account remains secure, especially when using shared or public devices.
* **Activity History**: Optionally, the system might provide users with access to their activity history or logs, allowing them to review past interactions, changes to their profile, or login sessions for security and auditing purposes.
* **Email Preferences**: Users may also have the ability to manage their email preferences from their profile settings, such as opting in or out of promotional emails, newsletters, or system notifications sent via email.

Overall, this system provides a versatile and interactive messaging experience, allowing users to create, follow, and engage with channels, receive timely notifications, and communicate with each other directly.

Activity Diagram:



Package diagram:



# Development Process Description:

We started by carefully looking into what the notification management system needed to do. This involved figuring out who would be using it, what kinds of notifications it would handle, and what other features it should have for managing notifications well. After we had a clear picture of what was needed, we moved on to planning and designing how the system would work. We worked out how the system would be built, what kind of data it would store, and how users would interact with it to make sure it would be easy to use.

Once the planning was done, we began building the system. We used React technologies

Testing was a big part of making sure the system worked correctly. At every step of building the system, we tested it thoroughly to make sure it did what it was supposed to. We checked that different parts of the system worked well together, and we asked real users to try it out to see if it met their needs.

Once we were confident that the system functioned correctly, we made it available for people to use. This involved deploying it for users to access and interact with. After deployment, we monitored its performance to ensure continued functionality. If any issues arose, we were prepared to address them promptly. Additionally, we gathered feedback from users to identify potential areas for improvement and enhance the system's overall quality.

Tools Used and Client Interface During Development:

Throughout the development process, we leveraged a diverse set of tools to optimize efficiency and maintain seamless communication with our clients. Version control was meticulously managed using Git, allowing for organized collaboration and tracking of code changes. Project management and communication were facilitated through platforms like Slack and Zoom, enabling real-time discussions, progress updates, and task management. Additionally, collaborative coding sessions were conducted using IDEs such as Visual Studio Code, fostering teamwork and efficient code development. To ensure the smooth integration and functionality of our APIs, we extensively utilized Postman for testing and validation, allowing us to fine-tune endpoints and address any issues proactively. This comprehensive approach to tool utilization enabled us to effectively gather feedback, discuss project progress, and promptly address any concerns or modifications required, ultimately ensuring alignment with client expectations and the timely delivery of the project.

architecture of the system:

**Frontend (React-based)**:

**1.React Components:**

In our messaging project, React components were instrumental in structuring our user interface. We divided our application into modular components, such as schedule messages, Chat for users, and User list, group list, chat for groups, each responsible for rendering a specific part of the UI. By breaking down our UI into components, we ensured code reusability and maintainability, allowing us to easily add new features or modify existing ones without affecting other parts of the application.

**2.Routing (React Router):**

React Router played a crucial role in defining the routes of our messaging application. We used it to create routes for different views, such as the home page, chat rooms, and user profiles, groups channels, schedule messages . React Router enabled seamless navigation between these views, allowing users to switch between pages without a full page reload. This enhanced user experience and facilitated smooth transitions between different parts of the application.

**3. Styling:**

Styling was an essential aspect of our messaging project to ensure a visually appealing and user-friendly interface. We adopted a flexible styling approach that suited our project requirements. For instance, we utilized CSS for global styles and basic styling elements. CSS Modules were employed to encapsulate styles locally within specific components, preventing style conflicts and making our code more maintainable. Additionally, we leveraged Styled Components to create dynamic and reusable styled components directly within our JavaScript code. This allowed us to apply component-based styling, making our UI more responsive and adaptable to different screen sizes.

**Backend (Node.js-based)**:

* + **Express.js**: Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. It simplifies the process of creating RESTful APIs and handling HTTP requests.
  + **Routing**: We define routes in our Express application to handle different types of requests. Express Router allows us to organize our route handlers into modular and reusable units, improving code organization and maintainability.
  + **Authentication and Authorization**: We are implementing authentication and authorization mechanisms to protect our application's resources and enforce access control policies. And we used JWT for stateless authentication or session-based authentication for server-side session management.
  + **Database Integration**: We choose a database solution that best fits our application's needs. MongoDB is a NoSQL document database that offers flexibility and scalability.

**Database**:

* + **MongoDB**: MongoDB is a document-oriented NoSQL database that stores data in flexible, JSON-like documents. It's suitable for applications with rapidly changing schema or unstructured data.

In the structure of our database, we constructed 4 collections, each containing specific data:  
**User Collection**:

* + Contains information about users, including:
    - Username: The unique identifier for each user.
    - Email: The email address associated with the user's account.
    - Encrypted password: The hashed and encrypted password to ensure security.
    - Role: The role or level of access granted to the user (e.g., admin, regular user).
    - Notifications: Notifications received by the user from other users or system events.
    - Followers: An array of followers, where each object within the array contains details about the users who are following this user. Each follower object may include information such as username, email, etc.

**Group Collection**:

* + Stores data about each newly created group, including:
    - Admin: The user who created the group and has administrative privileges.
    - Group name: The name or title of the group.
    - Message history: A log of messages exchanged within the group, including sender, timestamp, and message content.
    - Last message: Details of the most recent message sent in the group, including sender, timestamp, and content.
    - Last message time: The timestamp of the last message sent in the group.
    - Muted users: Information about users who have been muted in the group, preventing them from receiving notifications or messages.

**Message Collection**:

* + Contains data of individual messages, including:
    - Sender: The user who sent the message.
    - Receiver: For private messages, the recipient user's identifier. For group/channel messages, the identifier of the group or channel where the message was sent.
    - Timestamp: The time when the message was sent.
    - Scheduled: Indicates whether the message is scheduled to be sent at a later time (true/false). If scheduled, includes the time to send.
    - Content: The content of the message, which may include text, images, files, etc.

**User Message Collection**:

* + Specifically for user-to-user chats, storing all messages exchanged between two users, including:
    - Sender: The user who sent the message.
    - Receiver: The recipient user.
    - Content: The content of the message.

These collections form the backbone of our database structure, organizing user data, group information, and message exchanges within our application.

**Deployment**:

* + **Hosting Services**: We have chosen a hosting provider that meets our deployment needs and budget. Firebase offers a serverless approach for building and deploying web applications.

**Testing and Debugging**:

* + **Unit Tests**: We write tests to validate the behavior of individual units or components of our application. We use testing Jest framework with assertion Chai libraries built-in assertions.
  + **Integration Tests**: We write tests to verify the interactions between different components or modules of our application. Using Supertest for testing HTTP endpoints and Sinon.js library for mocking dependencies.
  + **Debugging Tools**: We use browser developer: Chrome DevTools for debugging client-side JavaScript code. For server-side debugging, We use logging Winston library to capture and analyze application logs.

# Testing

*Login and user registration*

|  |  |  |  |
| --- | --- | --- | --- |
| Test# | Description | Expected Result | Pass/Fail |
| 1 | Start system | Main system page opened | Pass |
| 2 | Enter the correct user name and password. then click login button | Accept user Name and password and display | Pass |
| 3 | Enter the invalid Username and Password and click login button | Error message “Login Failed” | Pass |
| 4 | In adding form enter new details click Submit button | Added user Contact detail to database. | Pass |
| 5 | Signup new user by using Register form Clicking Signup button | Appear login form | Pass |
| 6 | In change of adding Details to register Form When enter invalid Email address unequal password for Signup | Display “Enter valid Email Address and passwords should be same. | Pass |
| 7 | Resetting password by using login form Clicking Forgot Password button | Appear resetting password form | Pass |
| 8 | Enter the Email Address or Phone number and click Continue button | Accept Email Address or Phone number and display resetting option | Pass |
| 9 | Resetting passwordbyEntering new password and confirmation code Clicking Reset Password button | Appear login form | Pass |

*Send Scheduled notification*

|  |  |  |  |
| --- | --- | --- | --- |
| Test# | Description | Expected Result | Pass/Fail |
| 1 | Click “send notification” | Open send notification page | Pass |
| 2 | Choose time in the past to send the notification in it | Error “the time is not relevant | Pass |
| 3 | Choose time of the day but with past hours | Error “the time is not relevant | Pass |
| 4 | Do not choose the people who will receive the message | Error message “choose people to receive message” | Pass |
| 5 | Click confirm | Message that the notification sent successfully | Pass |
| 6 | Sent notification without content | Error message “please fill the content of your notification” | Pass |
| 7 | Cancelled notification | The notification deleted and not received | Pass |

*User main actions*

|  |  |  |  |
| --- | --- | --- | --- |
| **Test#** | **Description** | **Expected Result** | **Pass/Fail** |
| **1** | Enter the correct user name and password. then click login button | Appear main page | Pass |
| **2** | Options Menu by Clicking Menu button | Appear Menu Options | Pass |
| **3** | Main Window Option by Clicking Main Window Item | Appear main page | Pass |
| **4** | Logout Option by Clicking Logout Item | Appear login page | Pass |
| **5** | Search Groups by clicking Search Group | Appear Groups table and search the | Pass |
| **6** | Update details Option by Clicking update details Item | Appear update user details page | Pass |
| **7** | Enter new user name. then click update button | Username updated | Pass |
| **8** | Enter old password and new password. then click update password | Password updated | Pass |

*Admin actions*

|  |  |  |  |
| --- | --- | --- | --- |
| Test# | Description | Expected Result | Pass/Fail |
| 1 | Sent notification | Open send notification page | Pass |
| 2 | Delete members from the group | Click in delete button in the settings | Pass |
| 3 | Change the name of the group | Change it and click save | Pass |
| 4 | Open the group to be like the chat (let the member send messages in his group) | Change the settings | Pass |
| 5 | Sent notification by mail | in | Pass |
| 6 | Follow group by follow click | See in Home page the channel of this group | Pass |
| 7 | Unfollow group by unfollow click | The channel will disappear in Home page | Pass |
| 8 | Search User by click in search user | See all users and can follow them | Pass |
| 9 | Mute the group notifications by click “on” in Mute group button | Did not see the notification when admin sent | Pass |

# Results and Insights

Working on a React and Node.js project to develop an information management system provided valuable learning lessons. Overall, the project progressed effectively, but there are areas where improvements could have been made.

One aspect that worked well was our background in React and Node.js, which facilitated smooth development. Additionally, utilizing modern web development technologies allowed us to create a user-friendly interface and implement complex functionalities efficiently.

However, in retrospect, there are a few areas where we could have improved:

Project Planning: We could have spent more time upfront on detailed project planning, including defining clear objectives, requirements, and milestones. This would have provided better guidance throughout the development process and helped in managing expectations more effectively.

Communication: While we maintained regular communication, there were instances where communication gaps occurred, leading to misunderstandings or delays. In the future, implementing more structured communication channels and ensuring everyone is on the same page would enhance collaboration and productivity.

Testing: Although we conducted testing throughout the development process, we could have implemented more comprehensive testing strategies, including unit testing, integration testing, and user acceptance testing. This would have helped in identifying and resolving issues earlier, reducing the likelihood of bugs or inconsistencies in the final product.

Overall, we believe we approached the project in the correct manner. However, in hindsight, there are areas we would address differently. Firstly, we would enhance our initial requirement gathering process to ensure a more comprehensive understanding of user needs and expectations from the outset. Additionally, we would improve our project documentation to facilitate smoother collaboration and handover processes. Lastly, we would allocate more time for thorough testing and quality assurance to identify and address potential issues earlier in the development cycle, ultimately improving the overall stability and reliability of the system. These changes would contribute to more efficient and successful project execution in future endeavors.

# Challenges Faced and Solutions

Throughout the development process, we encountered various challenges and implemented solutions to overcome them.

Managing the complexity of data structures for user profiles, messages, and media storage was a significant challenge during the project. To overcome this challenge, we focused on implementing optimized data structures and database optimizations. This approach allowed us to efficiently handle large volumes of user data and messages while ensuring optimal performance and scalability of the system.

Implementing real-time messaging functionality using WebSockets presented another technical challenge. Leveraging the socket.io library for WebSocket implementation, we optimized message handling and minimized latency. This enabled us to provide seamless real-time communication between users, enhancing the overall user experience.

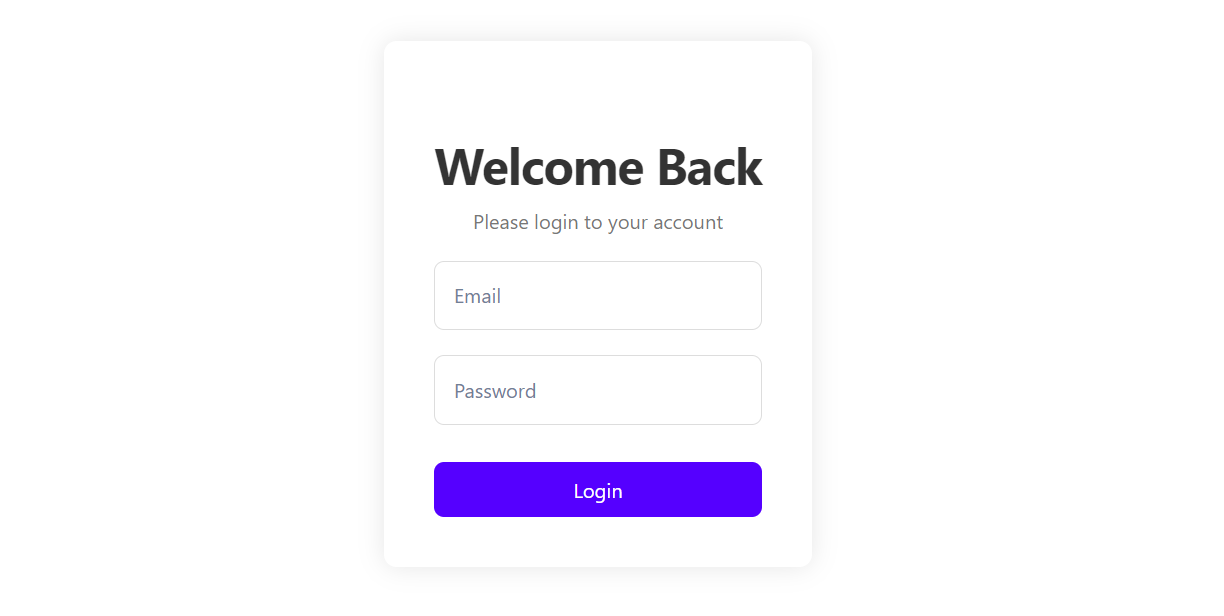
Designing a secure authentication system while maintaining scalability was crucial for the project's success. To address this challenge, we adopted industry-standard authentication protocols and best practices. This ensured secure user authentication and authorization, safeguarding the integrity and security of user accounts and data.

An additional challenge was implementing scheduled messages, which required effective management of message queues and scheduling mechanisms. To address this, we utilized the nod-schedule library to implement scheduled message delivery. Additionally, we implemented a notification bell feature to notify users of new messages, enhancing the user experience and providing easy access to recent message updates.

1. User guide:

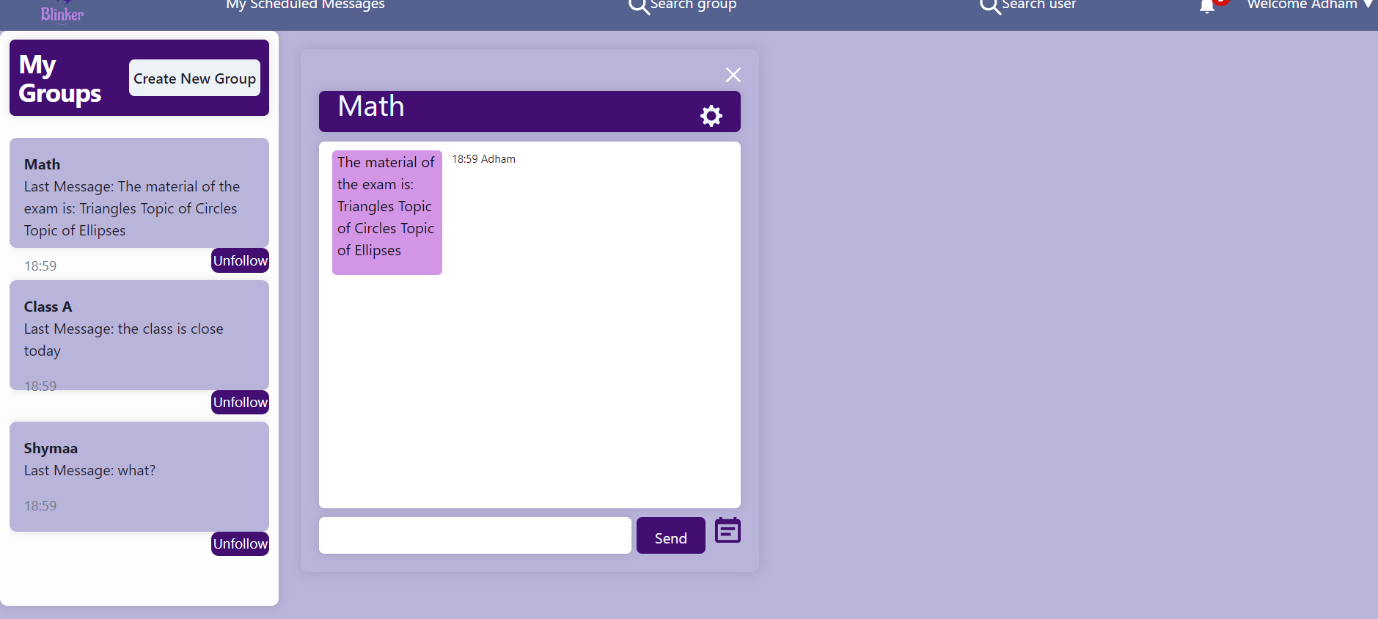
**Screens of our system:**

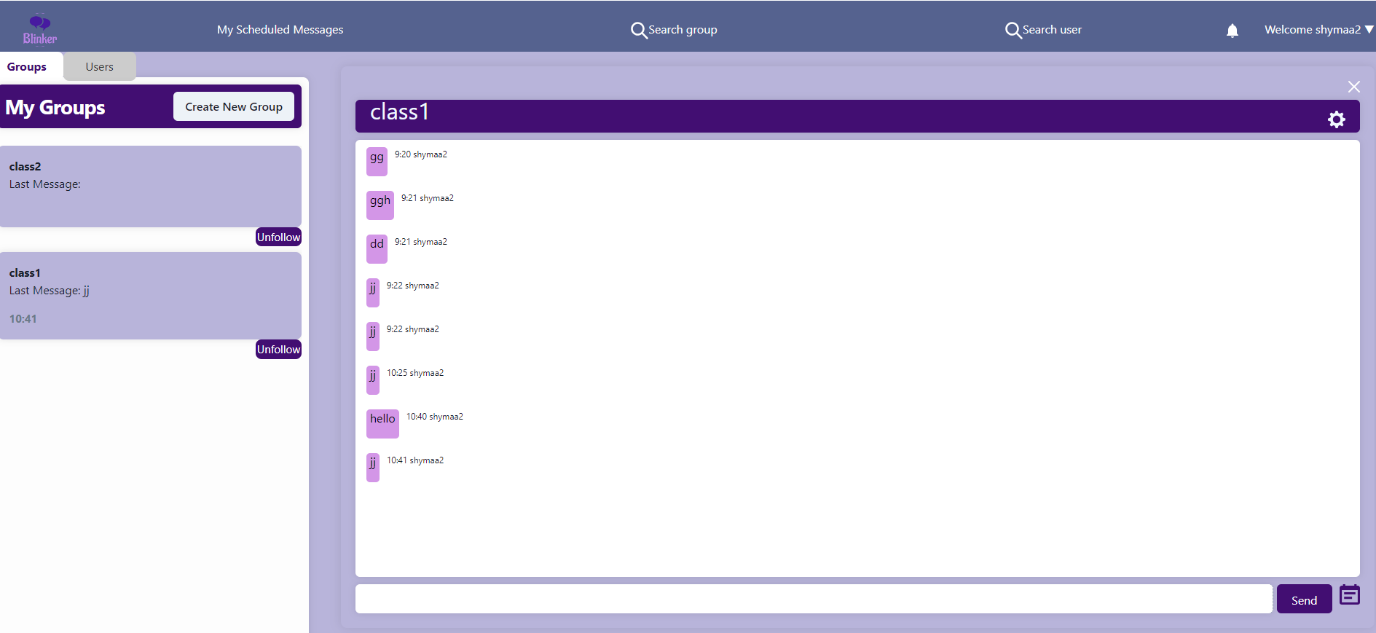
**Login page:**

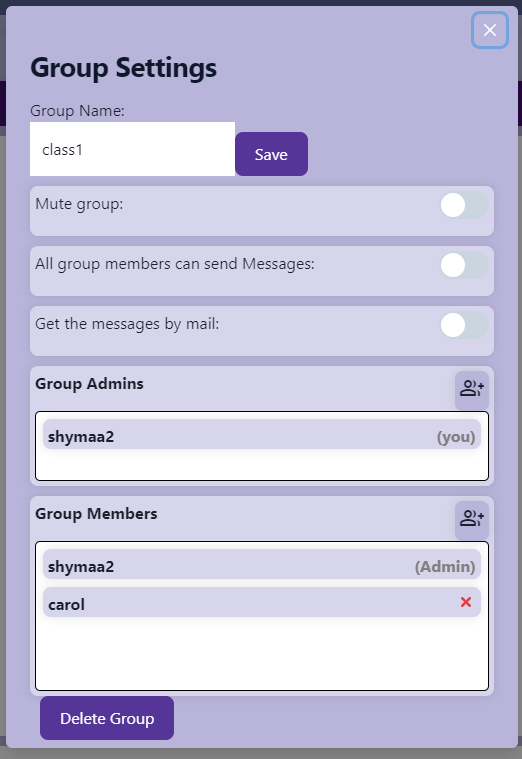


**Home Page:**

**"This is the main screen after logging in."**

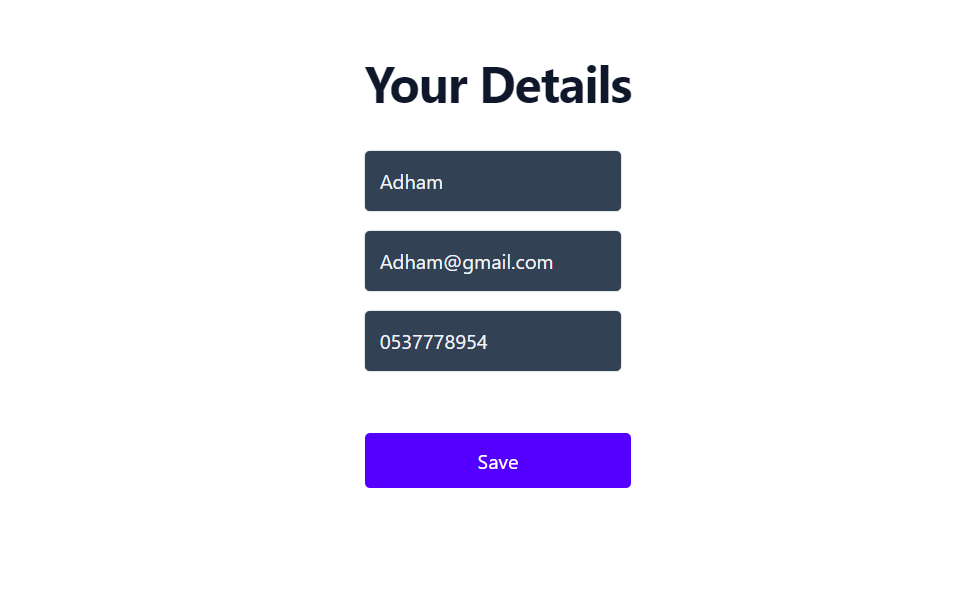






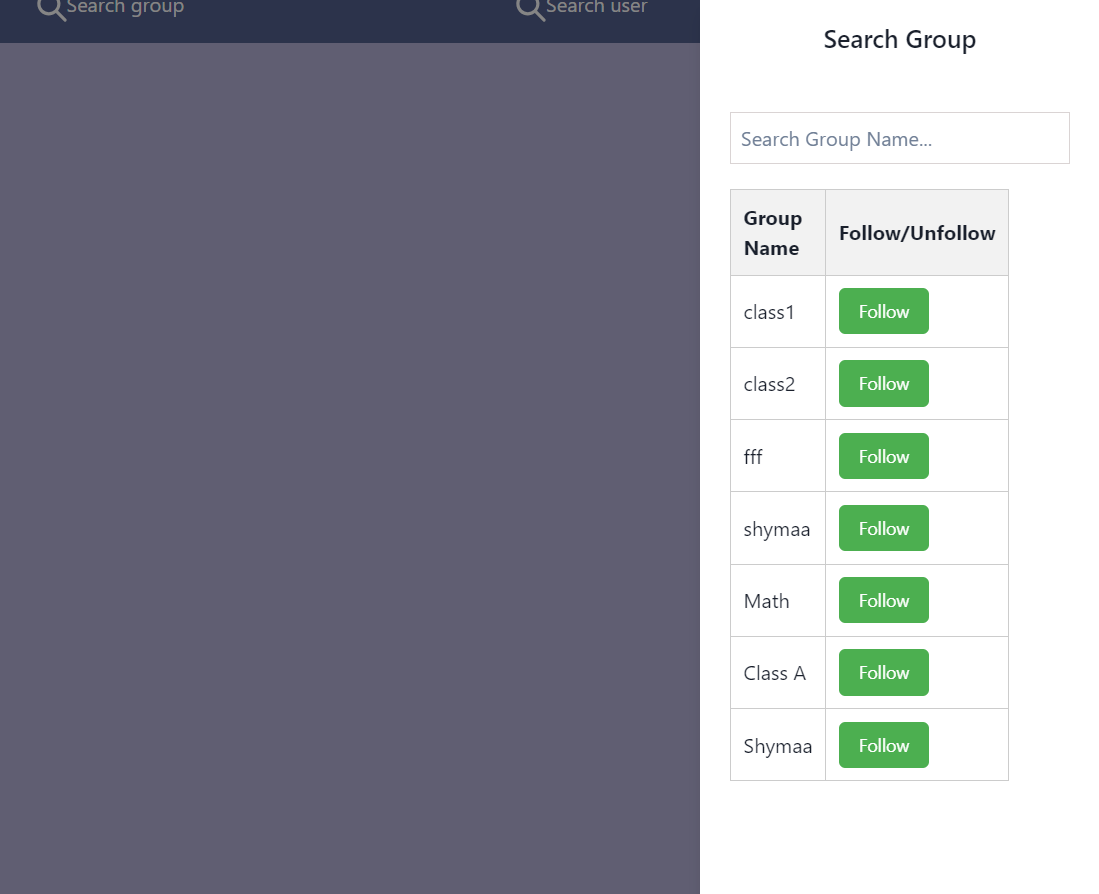
**Profile Managment:**

**"The user profile settings allow users to update their personal information."**



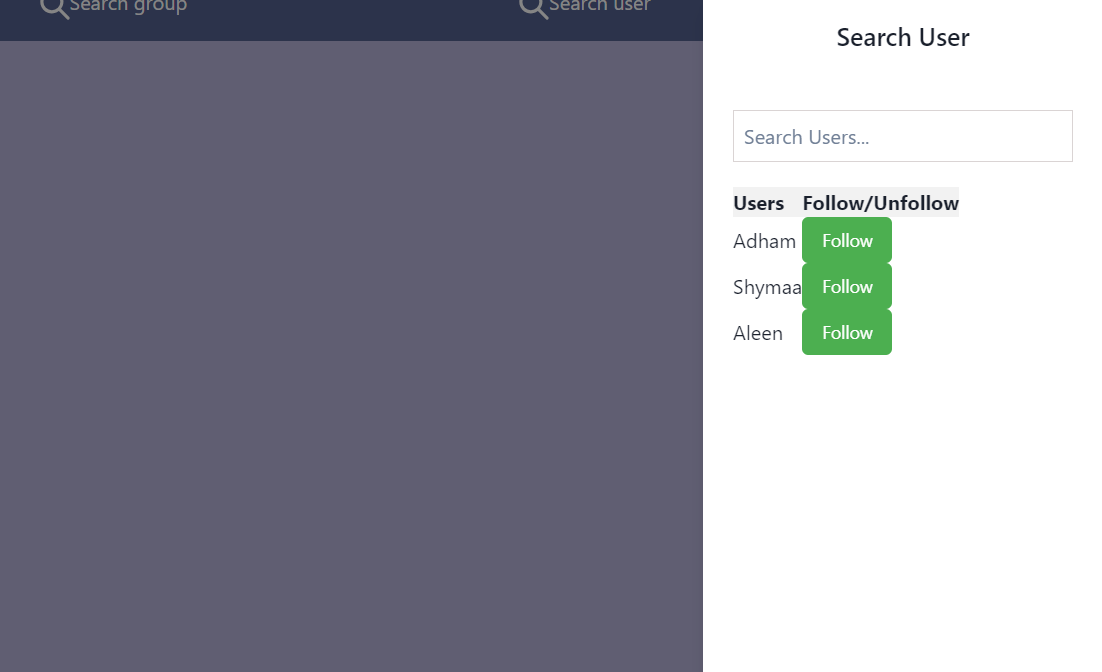
**Search Group:**

**"The user can search for a group they're interested in and choose to follow it or not."**



**Search User:**

**"The user can search for a user they're interested in and choose to follow it or not."**

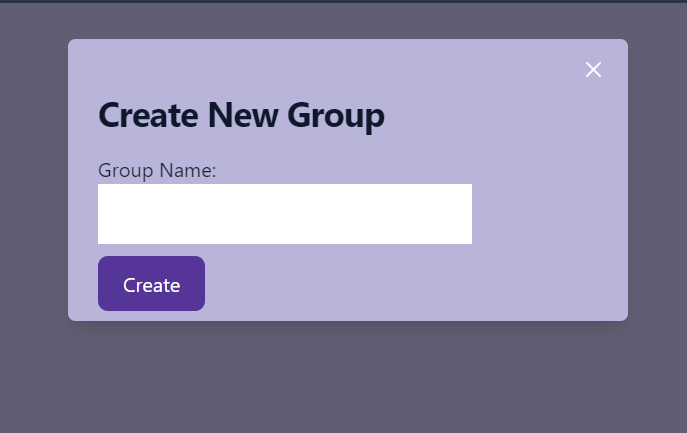


**Signup Page:**



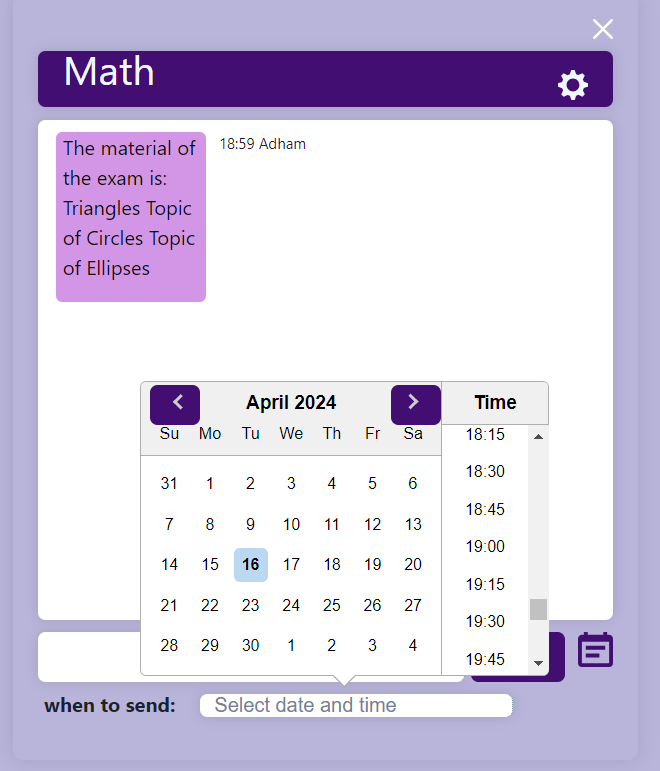
**Create new group:**

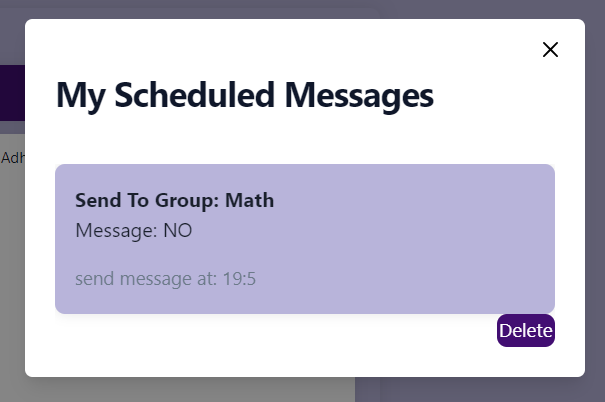
**"The user can be an admin and create a new group."**

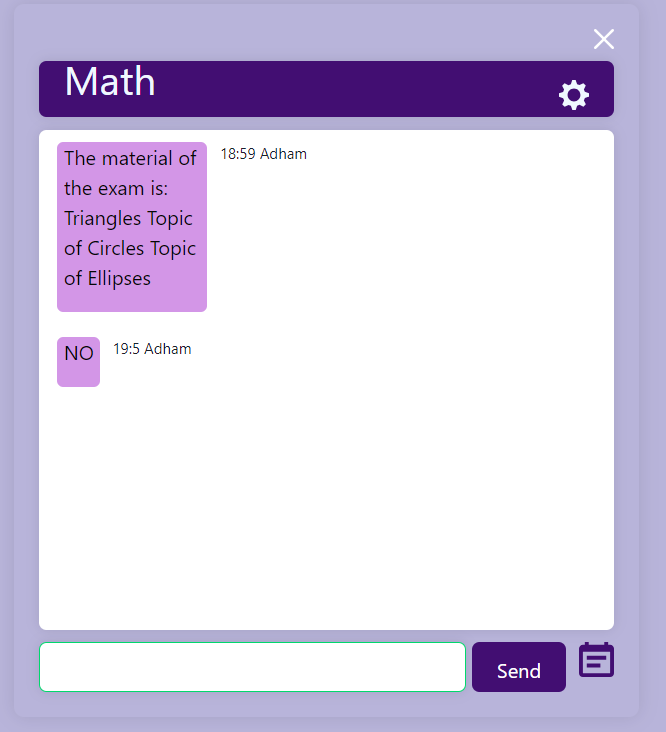


**Schedule message:**

**" User can schedule a message for any time and date he want. "**







# Maintenance Guide:

* 1. **installation instructions:**

To open and run the project, follow these steps:

1. git clone <https://github.com/Shymaatiti26/Notification-\_Managment\_System>
2. Navigate to the "server" directory by typing cd server.
3. Run  **npm install**
4. Start the server by running **npm start** in the console.
5. Navigate to the "client" directory by entering cd client in the terminal.
6. Run **npm install --force**
7. Run the client side by executing **npm run dev**.
8. Cope the link and past it in your browser : http://localhost:5173/Home

\*\*you should to open two terminals in VC ine of client and one of server.

\*\*first the DB will be empty so you should to register two users at least.

\*\* if you want to run the files of the installation Scripts you should do this command:  
bash install\_react.sh for react terminal in VC , and bash install\_node.sh for server (node.js) in

Server terminal in VC.  
In addition you should install MogoDB if doesn't exit in your computer.

* 1. **Front-end:**

1. **Components:**
   1. **Groups.jsx**
      * 1. handleFollow: add/remove user from group.users db
   2. **Chat.jsx**
      * 1. saveMessageToServer: send grpup message to backend to save on db
        2. setSenLaterToFalse: set the group sendLater to false in group db
        3. saveNotificationToServer: save notification to the db
        4. getGroupMessages: get group LastMessages from the database.
        5. setLatestMessage: set the latest message in group db
        6. sendMessage: check if to send the message now or it is scheduled and send the message with sokot.io according to that.
        7. joinGroup: join to socke.io group.
        8. setUserRoom: join user room for notification in socket.io.
        9. checkAdmin: check if the logged in user is the group admin.
        10. handleDateChange: handle date change from the date picker
        11. turnDatePicker: show date picker on/of according to send later.
        12. playSound: play notification sound.
   3. **CreateGroup.jsx**
      1. Submit :create new group
   4. **Login.jsx**
      1. handleSubmit:login to the
   5. **Notifications.jsx**
      1. deleteNotification: delete notification from user notifications db
      2. getUserNotifications: get use notification from the database.
   6. **Profile.jsx**
      * 1. handleInputChange: handle name input change and save it in User.
        2. handleSubmit: save the profile changes in the database.
   7. **SchedulesMsgList.jsx**
      * 1. getScheduledMsgs: get the user scheduled messages from db
   8. **Settings.jsx**
      * 1. handleSubmit: handle group name input change and save in db
        2. deleteGroup(groupId): delete Group from db by group id
        3. checkAdmin: check if the loged in user is the group admin.
        4. getGroupUsers:get the group users names from db
        5. deleteGroupMember: deletre group member from the group db
        6. handleGroupSendersChange: handle the all group members can send switcher change
        7. handleMuteGroupChange: handle the mute group switcher change.
   9. **Signup.jsx** 
      * 1. handleInputChange: handle form data changes and save it to FormData
        2. registerUser:send FormData to db
        3. handleSubmit: send form to db by registerUser function and check the response.
   10. **UserGroupsList.jsx**
       * 1. getUserGroups: get all the user groups from db and store them on groups.
         2. unfollorGroup: delete user from group.users in db
         3. getGroupSenders:set groupSenders in group db true/false
         4. IsGroupMuted: check if group muted.
2. **Context:**
3. **Hooks :**
4. **Pages:**
   1. Home.jsx
   2. UserPage.js
      * 1. handleLogout: logout from the app
   3. **BackEnd:**

**Controllers**:

* UserController.js:

**registerUser:**

* Name: registerUser
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: JSON response with username, token, and user ID, or an error message in case of failure.

**loginUser:**

* Name: loginUser
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: JSON response with username, token, and user ID upon successful login, or an error message in case of failure.

**getUserProfile:**

* Name: getUserProfile
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: JSON response with user details retrieved from the database based on the provided user ID.

**updatePassword:**

* Name: updatePassword
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: JSON response indicating success or failure of the password update operation.

**updateProfile:**

* Name: updateProfile
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: JSON response indicating success or failure of the user profile update operation.

**logout:**

* Name: logout
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: None (commented out), but typically would clear the authentication token from the client.

**allUsers:**

* Name: allUsers
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: JSON response containing details of all users in the system.

**getUserDetails:**

* Name: getUserDetails
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: JSON response with details of a specific user based on the provided user ID.

**updateUser:**

* Name: updateUser
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: JSON response indicating success or failure of the user profile update operation.

**deleteUser:**

* Name: deleteUser
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: JSON response indicating success or failure of the user deletion operation.

**saveNotification:**

* Name: saveNotification
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: None (asynchronous operation to save a notification in the user's database).

**getUserNotifications:**

* Name: getUserNotifications
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: JSON response containing notifications for the specified user.

**deleteNotification:**

* Name: deleteNotification
* Input: HTTP request (req), HTTP response (res), Next function (next)
* Output: None (asynchronous operation to delete a notification for the specified user).

* Group Conroller:

**createGroup:**

* Name: createGroup
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response indicating success or failure of group creation, along with the group ID and whether the group already exists.

**getUserGroups:**

* Name: getUserGroups
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing all groups associated with the provided user ID.

**deleteGroup:**

* Name: deleteGroup
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: None, but logs success or failure message.

**setLatestMessage:**

* Name: setLatestMessage
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: None, but updates the latest message and its sending time for a specific group.

**addUserToGroup:**

* Name: addUserToGroup
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output:None (Functionality incomplete, currently empty).

**getAllGroups:**

* Name: getAllGroups
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing details of all groups.

**UpdateGroup:**

* Name: UpdateGroup
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response indicating success or failure of adding a user to a group.

**RemoveUserFromGroup:**

* Name: RemoveUserFromGroup
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response indicating success or failure of removing a user from a group.

**ChangeGgoupName:**

* Name: ChangeGgoupName
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response indicating success or failure of changing the group name.

**UnfollowGroup:**

* Name: unfollowGroup
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response indicating success or failure of removing a user from a group.

**getGroups:**

* Name: getGroups
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing details of all groups.

**getGroupByID:**

* Name: getGroupByID
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing details of a group based on the provided group ID.

**getGroupUsers:**

* Name: getGroupUsers
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing users belonging to a specific group.

**setGroupSender:**

* Name: setGroupSender
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: None, but updates the GroupSender status for a specific group.

**getGroupSenders:**

* Name: getGroupSenders
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing the GroupSender status for a specific group.

**setMuteGroup:**

* Name: setMuteGroup
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response indicating success or failure of adding/removing a user to/from the muteOnUsers array.

**checkUserExistInMute:**

* Name: checkUserExistInMute
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response indicating whether the user exists in the muteOnUsers array of a specific group.
* Message Controller:

**getMessage:**

* Name: getMessage
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing the ID of the saved message in the database.

**sendMessages:**

* Name: sendMessages
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing all messages for a specific group that are not scheduled for later sending.

**getScheduledMsgs:**

* Name: getScheduledMsgs
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing all messages scheduled for later sending by a specific user, along with their associated group names.

**setSenLaterToFalse:**

* Name: setSenLaterToFalse
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing the updated message with the `sendLater` flag set to false.

* UserController:

**getAllUsers:**

* Name: getAllUsers
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response containing details of all users.

**followUser:**

* Name: followUser
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response indicating success or failure of creating a user's followers array.

**removeUserFromUser1:**

* Name: removeUserFromUser1
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response indicating success or failure of removing a user from another user's followers array.

**UpdateUser1:**

* Name: UpdateUser1
* Input: HTTP request (`req`), HTTP response (`res`), Next function (`next`)
* Output: JSON response indicating success or failure of adding a user to another user's followers array.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Models:**

* User Model:

**Schema Definition:**

* The schema defines the structure of documents in the `User` collection.
* It has several fields:

1. `username`: A string field representing the username of the user.

2. `email`: A string field representing the email address of the user. It is required and unique.

3. `phone`: A string field representing the phone number of the user. It is required and validated using the `validator.isMobilePhone` function.

4. `password`: A string field representing the password of the user. It is required and encrypted using bcrypt before saving to the database. It is not returned in query results (`select: false`).

5. `role`: A string field representing the role of the user. It defaults to 'user'.

6. `createdAt`: A date field representing the creation date of the user document.

7. `notifications`: An array of objects representing notifications for the user. Each notification object contains sender, group, message, timeSent, sendLater, and groupName fields.

**Pre-Save Middleware:**

* The schema defines a pre-save middleware function to encrypt the password using bcrypt before saving the user document.

**Instance Method:**

* The schema defines a method `comparePassword` to compare the entered password with the hashed password stored in the database.

**Instance Method:**

* The schema defines a method `getJwtToken` to generate a JSON Web Token (JWT) for the user, which includes the user's ID and expires in 7 days.

**Model Definition:**

* The model is created using `mongoose.model('User', userSchema)`, which compiles the schema into a model.
* The model name is `'User'`, and it corresponds to the `User` collection in the MongoDB database.
* This model will have methods for interacting with documents in the `User` collection.
* Message Model:

**Schema Definition:**

* The schema defines the structure of documents in the `Message` collection.
* It has several fields:

1. `sender`: A string field representing the sender of the message.

2. `group`: An array of ObjectIds referencing documents in the `Group` collection. This **field** represents the group to which the message belongs.

3. `message`: A string field representing the content of the message.

4. `timeSent`: A string field representing the timestamp when the message was sent.

5. `sendLater`: A string field representing whether the message is scheduled to be sent later.

**Model Definition:**

* The model is created using `mongoose.model("Message", messageSchema)`, which compiles the schema into a model.
* The model name is `"Message"`, and it corresponds to the `Message` collection in the MongoDB database.
* This model will have methods for interacting with documents in the `Message` collection.

* Group Model:

**Schema Definition:**

* The schema defines the structure of documents in the `Group` collection.
* It has several fields:

1. `groupName`: A string field representing the name of the group.

2. `users`: An array of ObjectIds referencing documents in the `User` collection. This field represents the users who are members of the group.

3. `muteOnUsers`: An array of ObjectIds referencing documents in the `User` collection. This field represents users who are muted in the group.

4. `Messages`: An array of ObjectIds referencing documents in the `Message` collection. This field represents the messages sent within the group.

5. `groupAdmin`: An array of strings representing the usernames of group administrators.

6. `latestMessage`: A string field representing the latest message in the group.

7. `latestMessageTime`: A string field representing the timestamp of the latest message in the group.

8. `groupSenders`: A string field representing whether all group members can send messages.

**Model Definition:**

* The model is created using `mongoose.model("Group", groupSchema)`, which compiles the schema into a model.
* The model name is `'Group'`, and it corresponds to the `Group` collection in the MongoDB database.
* -This model will have methods for interacting with documents in the `Group` collection.
* User1 Model:

**Schema Definition:**

* The schema defines the structure of documents in the `User1` collection.
* It has two fields:

1. `username`: A string field that is required and must be unique.

2. `followedUsers`: An array of ObjectIds referencing documents in the `User` collection. This field represents the users that the current user is following.

**Model Definition:**

* The model is created using `mongoose.model('User1', userSchema1)`, which compiles the schema into a model.
* The model name is `'User1'`, and it corresponds to the `User1` collection in the MongoDB database.
* This model will have methods for interacting with documents in the `User1` collection.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Routers:

* AuthRouter:

**Register User:**

* Endpoint: POST /register
* Controller: registerUser from authController
* Description: Allows a user to register by providing user information such as username, email, phone, and password.

**Login User:**

* Endpoint: POST /login
* Controller: loginUser from authController
* Description: Allows a user to login by providing email and password.

**Logout User:**

* Endpoint: POST /logout
* Controller: logout from authController
* Description: Logs out the currently authenticated user.

**Get User Profile:**

* Endpoint: GET /me
* Controller: getUserProfile from authController
* Description: Retrieves the profile of the currently authenticated user.

**Get All Users:**

* Endpoint: GET /getAllUsers
* Controller: getAllUsers from userController
* Description: Retrieves a list of all users.

**Follow User:**

* Endpoint: POST /followUser
* Controller: followUser from userController
* Description: Allows a user to follow another user.

**Save Notification:**

* Endpoint: POST /saveNotification
* Controller: saveNotification from authController
* Description: Saves a notification for a user.

**Get User Notifications:**

* Endpoint: POST /getUserNotifications
* Controller: getUserNotifications from authController
* Description: Retrieves notifications for a user.

**Delete Notification:**

* Endpoint: POST /deleteNotification
* Controller: deleteNotification from authController
* Description: Deletes a notification for a user.

* GroupRouter:

**Create Group:**

* Endpoint: POST /createGroup
* Controller: createGroup from groupController
* Description: Allows a user to create a new group.

**Get User Groups:**

* Endpoint: GET /UserGroupList
* Controller: getUserGroups from groupController
* Description: Retrieves all groups associated with a user.

**Delete Group:**

* Endpoint: POST /deleteGroup
* Controller: deleteGroup from groupController
* Description: Deletes a group.

**Set Latest Message:**

* Endpoint: POST /setLatestMessage
* Controller: setLatestMessage from groupController
* Description: Sets the latest message for a group.

**Get All Groups:**

* Endpoint: GET /getAllGroups
* Controller: getAllGroups from groupController
* Description: Retrieves all groups.

**Update Group:**

* Endpoint: PUT /UpdateGroup
* Controller: UpdateGroup from groupController
* Description: Updates a group.

**Remove User From Group:**

* Endpoint: PUT /RemoveUserFromGroup
* Controller: RemoveUserFromGroup from groupController
* Description: Removes a user from a group.

**Change Group Name:**

* Endpoint: POST /changeGgoupName
* Controller: ChangeGgoupName from groupController
* Description: Changes the name of a group.

**Unfollow Group:**

* Endpoint: POST /UnfollowGroup
* Controller: UnfollowGroup from groupController
* Description: Unfollows a group.

**Get Groups:**

* Endpoint: GET /getGroups
* Controller: getGroups from groupController
* Description: Retrieves all groups.

**Get Group By ID:**

* Endpoint: POST /getGroupByID
* Controller: getGroupByID from groupController
* Description: Retrieves a group by its ID.

**Get Group Users:**

* Endpoint: POST /getGroupUsers
* Controller: getGroupUsers from groupController
* Description: Retrieves users belonging to a group.

**Set Group Sender:**

* Endpoint: POST /setGroupSender
* Controller: setGroupSender from groupController
* Description: Sets the group sender property.

**Get Group Senders:**

**Error Handling:**

* If there's an error in the execution of func(req, res, next), it's caught in the .catch(next) block.
* The next function is called with the error, passing it to the next error-handling middleware in the stack.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Utils:
* Errorhandler:

**Class Definition:**

* The class is defined using the class keyword.

**Constructor Method:**

The constructor method takes two parameters: message and statusCode.

* message: A string that describes the error.
* statusCode: An integer representing the HTTP status code for the error.

Inside the constructor, super(message) is called to pass the message parameter to the parent class (Error).

The statusCode parameter is assigned to this.statusCode.

**Error Stack Trace:**

* Error.captureStackTrace(this, this.constructor) captures the stack trace of the error, making it easier to debug.

**Export Statement:**

* The module.exports statement exports the ErrorHandler class, making it available for use in other parts of the application.
* JwtToken:

**Importing Dependencies:**

* The module imports the jsonwebtoken library to generate JWTs.

**Function Definition (createToken):**

* The createToken function takes the user ID (\_id) as its parameter.

**Inside the function:**

* jwt.sign({\_id},'gghgbjkdbbmmnmcfhjlsdjla',{expiresIn: '3d'}) generates a JWT using the user ID (\_id) as the payload.
* 'gghgbjkdbbmmnmcfhjlsdjla' is used as the secret key for signing the token.
* {expiresIn: '3d'} sets the expiration time of the token to 3 days.

**Export Statement:**

* The module.exports statement exports the createToken function, making it available for use in other parts of the application.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* connectDB:
* It imports necessary modules: mongoose for MongoDB interaction and dotenv for loading environment variables.
* It configures dotenv to load environment variables from a .env file.
* The mongoOptions object contains various options for configuring the MongoDB connection. These options include settings for connection parameters like useNewUrlParser, useUnifiedTopology, autoIndex, connectTimeoutMS, socketTimeoutMS, minPoolSize, and maxPoolSize.
* The connectDB function is defined, which returns a promise. Inside this function:
* It constructs the MongoDB connection URL (mongoURL) using the mongodb:// protocol, the host (127.0.0.1), port (27017), and the name of the database (NotificationsSystem).
* It uses mongoose.connect() to establish a connection to the MongoDB database using the URL and options specified in mongoOptions.
* If the connection is successful, it resolves the promise with the connection object (conn).
* If there's an error during the connection process, it rejects the promise with the error.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Index.js:

**Imports:**

* express, mongoose, cors, dotenv, jwt, body-parser, nodemailer, http, socket.io, node-schedule: Necessary modules for creating and configuring the server, handling HTTP requests, and enabling real-time communication.
* connectDB: A function to establish a connection to the MongoDB database.
* loginRouter, signup, newMessage, Profile, auth, groupRouter, messageRouter: Route handlers for different endpoints.

**Express App Setup:**

* Middleware setup for JSON parsing and cookie parsing.
* CORS setup to allow requests from specific origins with credentials.
* Definition of the HTTP server using http.createServer.
* Configuration of socket.io to work with the HTTP server.

**Socket.IO Event Handling:**

* Event handlers for various socket events like connection, userRoom, joinGroup, send-message, and disconnect.
* These event handlers manage real-time communication between clients, such as sending and receiving messages.

**Routes Setup:**

* Setup of routes for handling various API endpoints.
* Routes are mounted under different base paths (/, /api/v1) depending on their functionality.

**Database Connection:**

* Calls connectDB function to establish a connection to the MongoDB database.
* Uses mongoose to interact with the database.
* Server Configuration and Startup:
* Starts the server on a specified port (PORT) or a default port (3001).
* Logs a message indicating the server has started successfully.