palisadoes

User Tag Management - Talawa Admin

Google Summer of Code 2024

Basic information

Name	Shekhar Patel
GitHub	 duplixx
E-mail	duplixxx20@gmail.com
Nationality	Indian
University	Galgotias University
Degree	Bachelor of Technology
Timezone	IST (UTC +5:30)

About Me

My name is Shekhar, and I'm currently pursuing my third year of engineering at Galgotias University, Greater Noida. As a full-stack developer deeply fascinated by machine learning and AI, I also possess a solid background in cybersecurity. Additionally, I have a profound passion for culinary exploration and enjoy experimenting with various dishes.

Having served as a community leader in The Hacker's Meetup and as a Google Developers Group Noida team member, I have experienced firsthand the challenges involved in community events. These experiences have further reinforced my commitment to community building and problem-solving, and I am eager to bring my skills and enthusiasm to the Talawa initiative.

Prior Experience

I've had the privilege of working on various frontend and backend projects, utilizing a diverse range of technologies such as React.js, Next.js, Vue.js, Django, React Native, Node.js, and Express. About a year ago, I began contributing to open-source projects, which has been a deeply enriching experience. The supportive and engaging community within the open-source realm has continuously inspired and motivated me.

During my time as a Software Developer Intern at EliteKods, I had the opportunity to work on over a dozen projects over six months. Despite being remote, I collaborated closely with a diverse team spread across different regions. This experience not only honed my technical skills but also enhanced my ability to communicate and collaborate effectively in a team environment. I primarily worked with technologies such as Next.js, React.js, ShadCN, Supabase, MongoDB, and Web3connect.

Overall, my experiences have shaped me into a flexible and quick learner, capable of thriving in diverse and challenging environments. I'm excited about the opportunity to bring my skills and enthusiasm to new projects and continue growing both personally and professionally.

Contributions to Palisadoes:

- All the Issues opened by me in Talawa-admin can be found here.
- All the PRs (Status: open) by in Talawa-admin me can be found here.
- All the Issues opened by me in Talawa-API can be found here.
- All the PRs (Status: open) by me in Talawa-API me can be found here.

Merged/Under review PRs in Talawa-Admin & Talawa-Api repository:

S. No.	Pull Request	Status
01	Feat: tagColor in UserTags	Merged
02	Feature: Allow Single Object Upload Photo/video	Merged
03	MVP Talawa Admin : Hiding Chat Feature from Talawa Web App	Merged
04	Improper positioning of Event Statistics in the mobile view	Merged
05	GitHub Action: Fix authorized-changes-detection	Merged
06	test: SecuredRouteForUser component 100% Test Coverage and fixed uncovered lines	Merged
07	Setting Up Codecov Environment (Documentation Update)	Merged
08	Organization People Revamp design	Merged
09	Add Event Image Support with validation	Merged
10	SideTweaks/Changes in leftdrawer and title section	Merged
11	Revamp and Modular Event Screen	Merged
12	Bug Report: Join Organization Error on "View All Organizations" Page Fixed	Merged
13	Manage Button UI Fixed	Merged
14	Update PR_WORKFLOW for delta errors/warning checks	Merged
15	test: SecuredRouteForUser component 100% Test Coverage and fixed uncovered lines	Merged

16	[userTypeFix] Error encountered in Settings page (MemberDetail file) as superadmin.	Under Review

Issues Opened:

S. No.	Issues	Status
1.	Talawa-Admin Docker Support Issue	Open
2.	Add Image upload functionality in Create Event	Open
3.	Funds and Actions Items design Enhancements	Open
4.	GraphQL validation failed errors	Open
5.	Add Password Strength Bar in Signup Form	Closed
6.	Code Coverage: Create tests for src/components/CheckIn/CheckInModal.tsx	Closed
7.	Layout and Input Field Issues in Post Details Modal	Closed

Project - User Tag Management Talawa Admin

Project duration: **350 Hours**

Description:

We need a comprehensive user tag management system for Talawa Admin to address current limitations and enhance community organization. This system will introduce both automatic and manual tag allotment, hierarchical tag structures, and CRUD operations.

a. The current lack of a robust tag management system hinders effective organization and communication within the Talawa community. The proposed system aims to streamline user categorization, facilitating targeted communication and enhancing community engagement.

b. This should not be confused with hashtags or tagging someone in a chat. This is for applying a label/category to User and Event objects

c. Support for this feature was added to Talawa API in the past year. We welcome GitHub issues to verify and rectify the expected functionality of this API support during the GSoC evaluation period.

Expected Outcomes:

- Automatic tag allotment based on predefined criteria.
- Manual tag allotment by Admins and/or Super Admins.
- Hierarchical tag structures with parent and nested sub-tags.
- CRUD operations for tags (Create, Read, Update, Delete).
- Grouping of users based on tags.
- Broadcasting emails/messages to a group of users with certain tags.
- Filtering of users based on tags for targeted communication.

Mentors	Rishav Jha Mech, Nitya Pasrija, Noble Mittal

Current Scenario

The current Talawa-Admin platform lacks a dedicated interface for managing tags and integrating them into existing functionalities. This hinders efficient member filtering and potentially limits the value of tags for admins.

Solution:

I propose a two-pronged approach that leverages the existing backend work on tags in Talawa-API and Talawa-Admin integration by implementing intuitive designs and logic: (I have past experience working on user-tags and I have opened issue and PR)

1. **Talawa-Admin Integration:**

Develop a dedicated "Tags" screen within Talawa-Admin for comprehensive tag management. This screen should allow admins to:

- View a list of all existing tags.
- Create new tags as needed.
- Create Child tags or Sub-tags of Parent Tags
- Edit or delete existing tags.
    - ➔ Integrate tag functionality into member search and filtering options within Talawa-Admin. This will empower admins to find members based on relevant tags efficiently.
- Implement these user tags seamlessly within Talawa-admin and Talawa-user portal ensuring consistent tag functionality across all platforms.

2. **Talawa-API Enhancements :**

Explore potential improvements to core tag functionality within Talawa-API, considering aspects like:

- **Tag Types:** If tags can be categorized (e.g., user status, interests, skills), consider implementing a tagging system that supports this structure. This can provide further granularity in member filtering.
- **Tag Descriptions:** Allow admins to add brief descriptions to tags, providing context for their meaning and purpose.

Benefits:

- **Improved Admin Efficiency:** A dedicated Tags screen and integrated search/filtering by tags will significantly streamline member management for admins.
- **Scalability and Flexibility:** The proposed integration lays the foundation for future enhancements to tag functionality, ensuring the system can adapt to evolving needs.

Additional Considerations:

- **Security:** Ensure proper access controls are in place within Talawa-Admin to restrict tag management actions to authorized admins.
- **User Interface Design:** Design the Tags screens in Talawa-Admin to be intuitive and user-friendly for admins.

Proposed Features and Changes

★ Automatic Tag Allotment Based on Predefined Criteria

Objective

Develop a system for assigning initial tags to users within a community structure, adhering to predetermined criteria. These tags will be readily viewable by Superadmins and Admins for enhanced user management.

Proposed Approach

1. **User Data Model:**

   Establish a user data model that captures essential user attributes relevant to tag assignment. This may include:

   - `adminApproved` (boolean): Indicates whether the user has been approved for full membership.
   - `pluginCreationAllowed` (boolean): Indicates whether the user has plugin

installation privileges. Plugins allow users to install and uninstall features like events, funds, and posts.

2. **Tag Definition:**

   Define a set of tags that accurately represent user statuses and permissions based on the chosen criteria. Examples:

   - `unapproved` (assigned to users who haven't been approved)
   - `Plugins disabled` (assigned to users without plugin creation privileges)

3. **Automatic Tag Assignment Logic:**

   Implement an automated process that assigns tags to users upon registration or based on attribute changes. This logic can be expressed as a series of rules or decision trees:

   - If `adminApproved` is `False`, assign `unapproved`.
   - If `pluginCreationAllowed` is `False`, assign `Plugins disabled`.

User Journey: Account Registration and Tag Assignment

Scenario: A new user registers on the Talawa platform.

1. **Registration and Initial Tags:**
   - The user completes the registration process and submits their information.

- **System Action:** The system automatically assigns two tags to the user's profile:
    - unapproved
    - Plugins disabled

2. **User Status:**
    - The user's account is in a pending state, awaiting approval from an admin or superadmin.
    - The user cannot access features requiring approval, such as plugin creation.
3. **Admin/Superadmin View and Actions:**
    - Superadmins and admins can access a list of users or individual user profiles.
    - In the requests, the assigned tags (`unapproved` and `Plugins disabled`) are clearly visible.

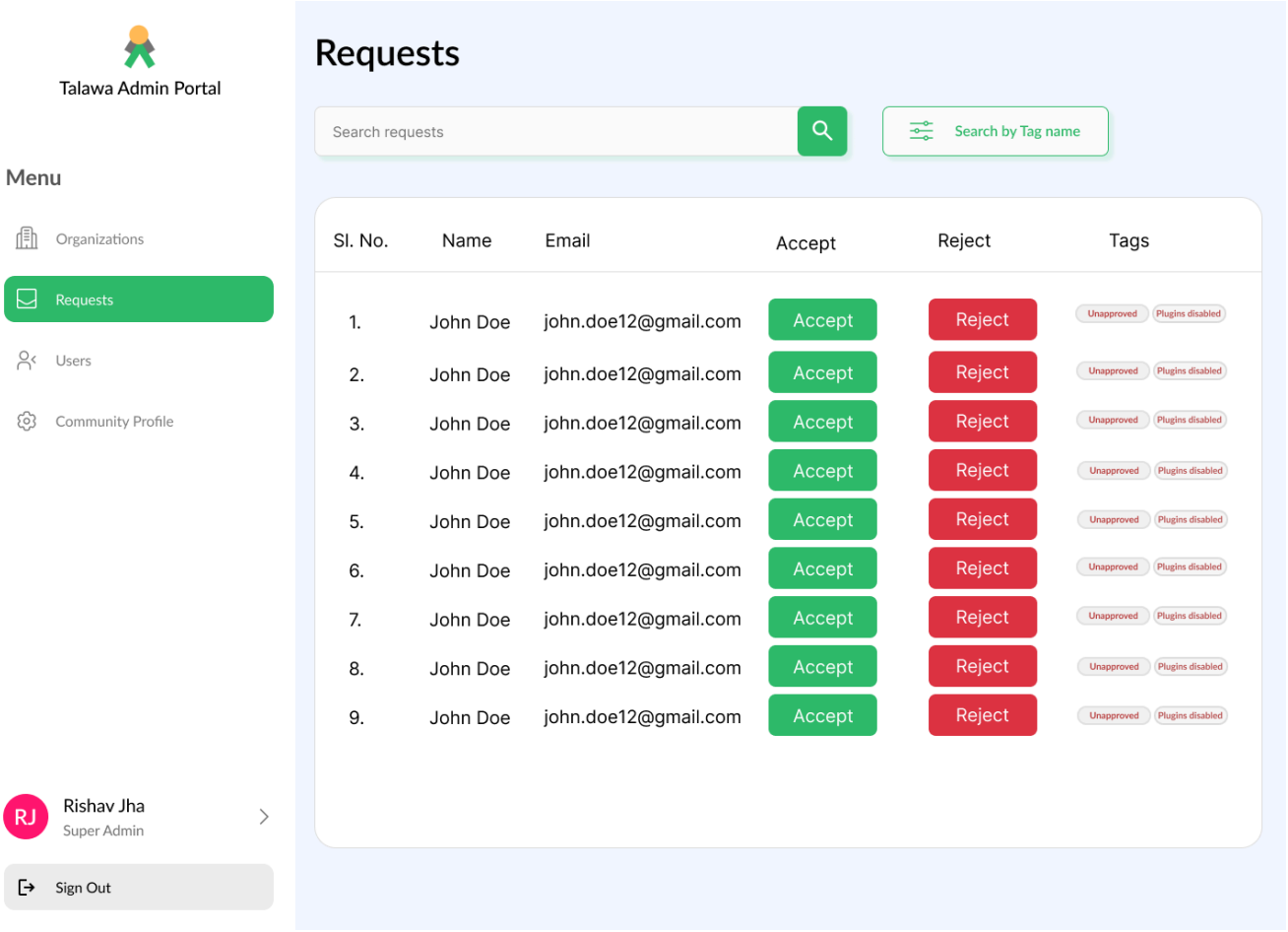


Figure 0 : Proposed UI design for automatic tag's

4. **Based on these tags admins/superadmins can:**
    - **Approve the user:** This grants the user full access to the platform and removes the `unapproved` tag.
    - **Deny the user's request:** The user's account may be disabled or require further information.
    - **Leave the user pending:** This may be appropriate if additional information or verification is needed.

★ Manual Tag Allotment by Admins and Super Admins

Objective

Enhance user management and enable targeted communication within the Talawa community by empowering Admins and Super Admins to manually assign tags to users.

Proposed Approach:

I propose to develop a user-friendly interface within Talawa-Admin for manual tag assignment. This interface will allow Admins and Super Admins to assign tags

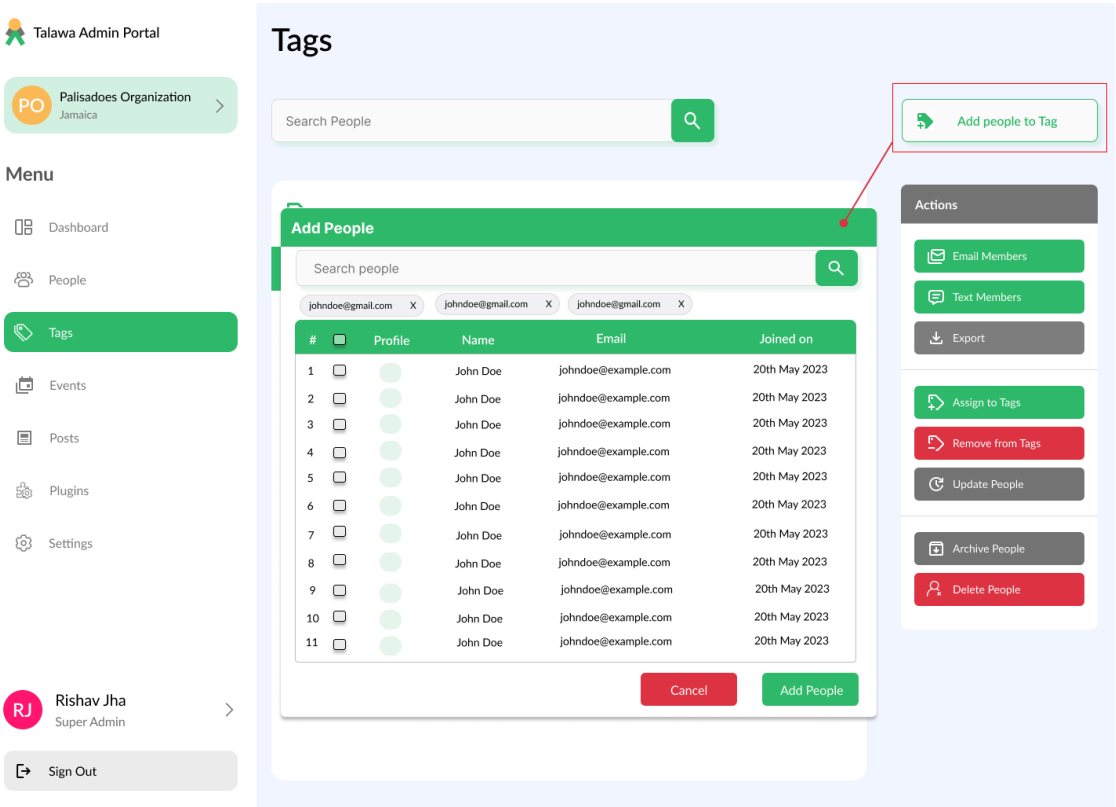


Figure 1 : UI design for Manual Tag Allotment

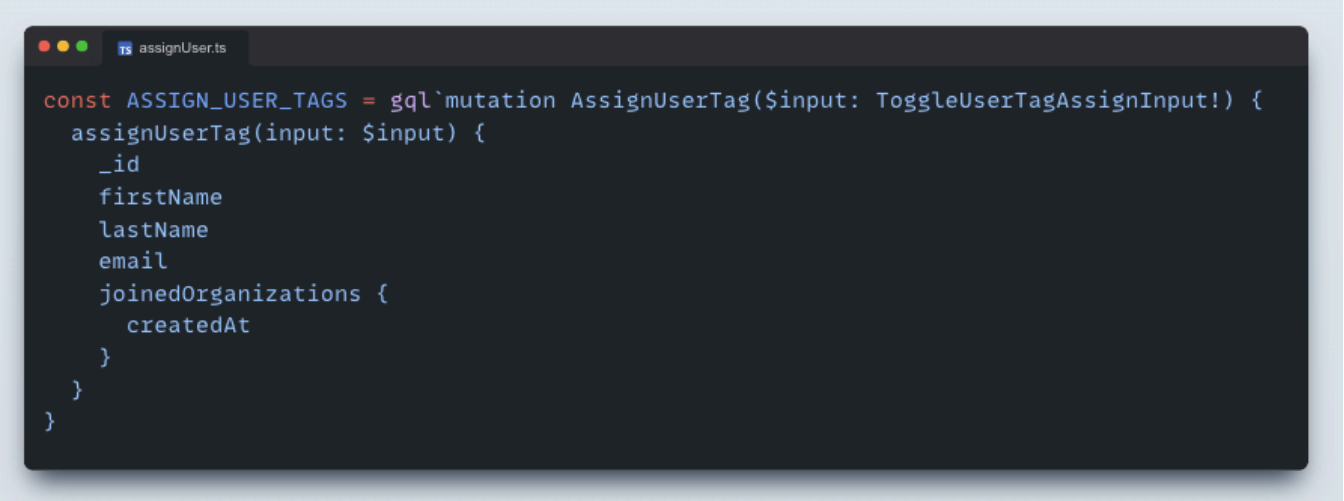


Figure 2 : The mutation for assign user tag in Talawa-admin

User Journey for Manual Tag Assignment:

- **Initiation:**
  - The Admin/Superadmin navigates to the Tags screen in the Talawa-Admin dashboard.
  - They locate the desired tag in the "View Tags" column.
  - Upon identifying the tag, they click the "Manage" button associated with it. (**Fig 7**)
  - After clicking the "Manage" button a Manage People screen appears.
  - Inside the “Manage People” screen, they are presented with a comprehensive list of tag members.
  - To manually add people to tag admin/superadmin can click on “Add people to Tag” button (**Fig 1**)
- **Modal Interaction:**
  - After clicking "Manage," a modal window promptly appears on the screen.
  - Inside the modal, they are presented with a comprehensive list of organization members.
  - To streamline their search process, they utilize the search bar functionality to find specific individuals efficiently.
  - Upon finding the desired individuals, the Admin/Superadmin selects

them using checkboxes or opts for a convenient "Select All" option.

- **Confirmation and Assignment:**
  - With the selection process completed, the Admin/Superadmin proceeds by clicking the "Add People" button.
  - Following this action, the application confirms their selection by displaying a confirmation prompt.
  - Upon confirmation, the application triggers the ASSIGN_USER_TAGS GraphQL mutation to assign the selected tag(s) to the chosen users.
- **Success or Error Handling:**
  - In case of a successful tag assignment, the application provides a success message to inform the Admin/Superadmin about the successful operation.
  - Conversely, if any errors occur during the assignment process, the application promptly notifies the user, enabling them to take appropriate actions to resolve the issue.

★ Hierarchical Tag Structures with Parent and Nested sub-tags

Objective

The objective of hierarchical tag structures with parent and nested sub-tags is to:

- **Organize tags in a categorized manner:** This allows for a more granular classification of users and events within the Talawa platform.
- **Improve information discovery:** By having parent and nested sub-tags, Admins and Superadmins can easily browse and locate relevant tags for specific needs.
- **Enhance user and event management:** With hierarchical tags, admins can categorize users and events based on broader and more specific criteria, facilitating targeted communication and organization.

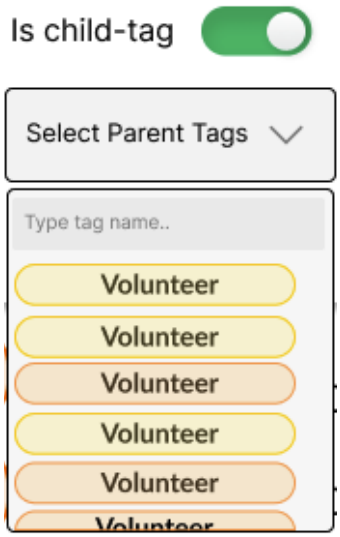
Proposed Approach

- **For creating child tags:**

  I propose incorporating an "Is child-tag" toggle within the proposed Create Tag Modal (**Fig 3**).

  This toggle would allow Admins/Superadmins to designate whether the tag being

created is a child tag. By enabling this toggle, Admins/Superadmins can select a parent tag from a dropdown list of existing tags before creating a new tag, effectively making it a child tag of the chosen parent. This feature enhances organization and hierarchy within the tag system, providing more structured management of tags.



➔ CreateUserTagInput will look like this:

```
export type CreateUserTagInput = {
  name: Scalars['String']['input'];
  organizationId: Scalars['ID']['input'];
  parentTagId?: InputMaybe<Scalars['ID']['input']>;
  tagColor: Scalars['String']['input'];
};
```

➔ Input that will be accepted for creating child tags:

```
"input": {
  "name":"....."
  "tagColor": "#.....",
  "parentTagId": 3
}
```

➔ Mutations for creating child tags:

mutations.ts

```ts
import { gql } from '@apollo/client';

const CREATE_USER_TAG = gql`
  mutation CreateUserTag($input: CreateUserTagInput!) {
    createUserTag(input: $input) {
      name
      parentTag {
        name
      }
    }
  }
`;
```

➔ Frontend Implementation:

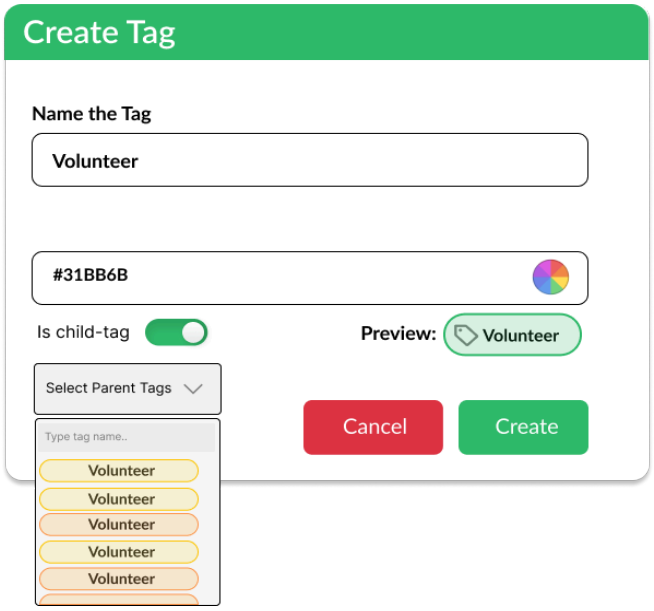
```ts
const handleCreateTag = async () => {
  try {
    const response = await createUserTag({
      variables: {
        input: {
          parentTagId: selectedParentId,
          name: tagName,
          color: tagColor,
        },
      },
    });

    if (response.data?.createUserTag) {
      handleCloseCreateTagModal();
      setToastMessage('Tag created successfully!');
      setShowToast(true);
    } else {
      throw new Error('Failed to create tag');
    }
  } catch (error) {
    console.error('Error creating tag:', error);
    setToastMessage('Error creating tag!');
    setShowToast(true);
  }
};

const handleSelectParentTag = (tagId) => {
  setSelectedParentId(tagId);
};
```

```
return (
  <div>
    <div>Search...</div>

    <Dropdown>
      <Dropdown.Toggle variant="success" id="dropdown-basic">
        Actions
      </Dropdown.Toggle>
      <Dropdown.Menu>
        <Dropdown.Item onClick={handleOpenCreateTagModal}>
          Create Tag
        </Dropdown.Item>
        <Dropdown.Item
        onClick={handleOpenUpdateTagModal}
        eventKey="update-tags" >Update Tags</Dropdown.Item>
        <Dropdown.Item
        onClick={handleOpenDeleteTagModal}
        eventKey="delete-tags">Delete Tags</Dropdown.Item>
      </Dropdown.Menu>
    </Dropdown>

    {/* Tag Creation Modal */}
    <Modal show={isCreateTagModalOpen} onHide={handleCloseCreateTagModal}>
      <Modal.Header closeButton>
        <Modal.Title>Create Tag</Modal.Title>
      </Modal.Header>
      <Modal.Body>
        <InputGroup className="mb-3">
          <InputGroup.Text id="tagName">Tag Name</InputGroup.Text>
          <FormControl
            placeholder="Enter tag name"
            aria-label="Tag name"
            aria-describedby="tagName"
            value={tagName}
```

```
            onChange={(e) => setTagName(e.target.value)}
          />
        </InputGroup>
        <InputGroup className="mb-3">
          <InputGroup.Text id="tagColor">Tag Color</InputGroup.Text>
          <FormControl
            placeholder="Enter tag color"
            aria-label="Tag color"
            aria-describedby="tagColor"
            value={tagColor}
            onChange={(e) => setTagColor(e.target.value)}
          />
        </InputGroup>
        <ToggleButtonGroup
          type="checkbox"
          value={isChildTag}
          onChange={(value) => setIsChildTag(value)}
        >
          <ToggleButton value={true}>Is Child Tag</ToggleButton>
        </ToggleButtonGroup>
        {/* Parent Tag Dropdown */}
        {isChildTag && (
          <Dropdown>
            <Dropdown.Toggle variant="success" id="dropdown-basic">
              Select Parent Tag
            </Dropdown.Toggle>
            <Dropdown.Menu>
              {tags.map((tag) => (
                <Dropdown.Item
                  key={tag.id}
                  onClick={() => handleSelectParentTag(tag.id)}
                >
                  {tag.name}
                </Dropdown.Item>
              ))}
```

```
            </Dropdown.Menu>
          </Dropdown>
        )}
      </Modal.Body>
      <Modal.Footer>
        <Button variant="secondary" onClick={handleCloseCreateTagModal}>
          Cancel
        </Button>
        <Button variant="primary" onClick={handleCreateTag}>
          Create Tag
        </Button>
      </Modal.Footer>
    </Modal>
  </div>
  );
};

export default TagsScreenHeader;
```

➔ Proposed UI design for Creating child tag:



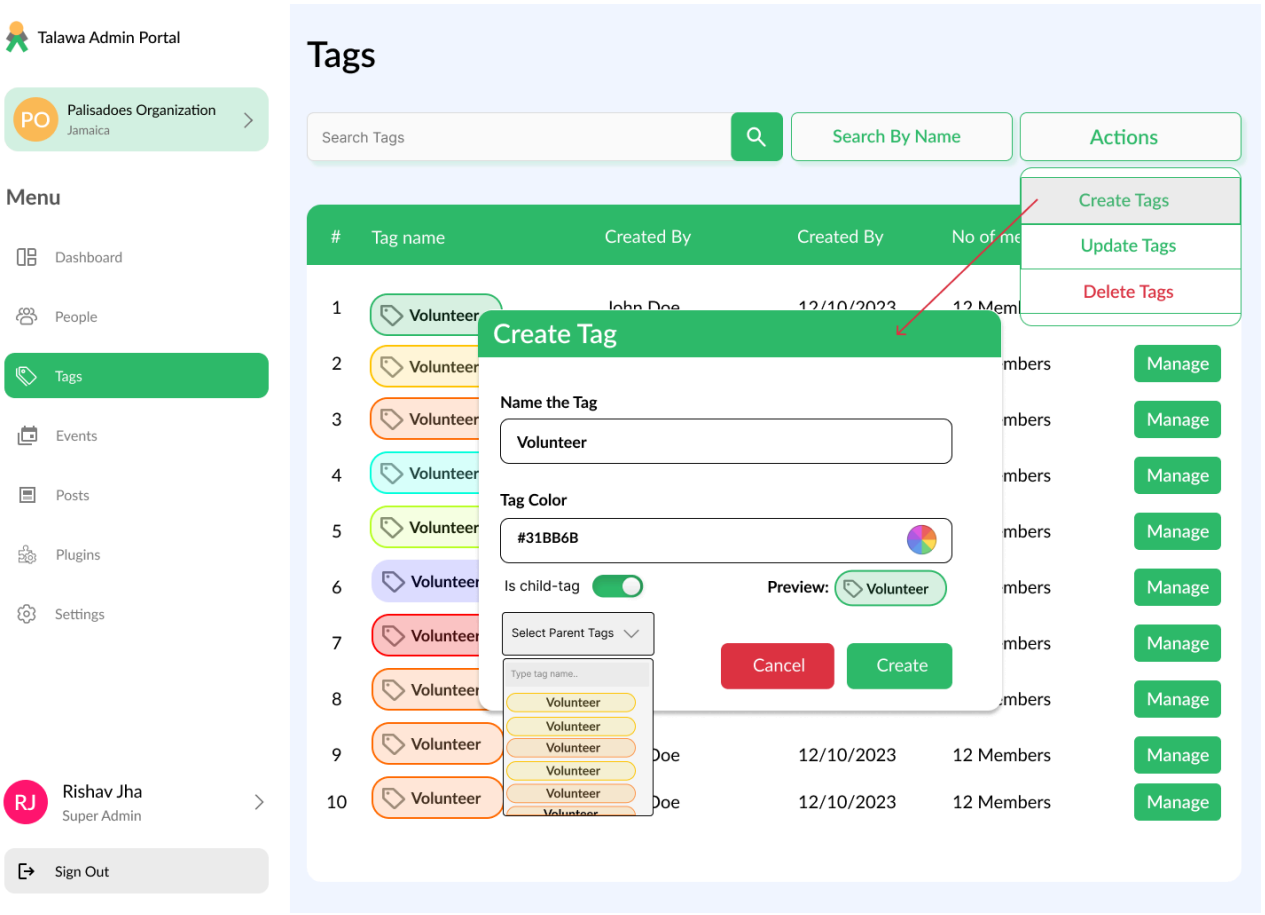


Figure 3 : UI design for Creating Subtags

★ CRUD Operations for tags (Create, Read, Update, Delete).

This proposal takes advantage of the already-developed Create, Read, Update, and Delete (CRUD) functionalities for tags within Talawa-API. We'll leverage the existing code base to seamlessly integrate these operations into the Talawa-Admin user interface, providing a streamlined experience for managing tags.
(**tagColor field is added in this PR**)

OrganizationTagUser.ts

```ts
// A User Tag is used for the categorization and the grouping of related users
// Each tag belongs to a particular organization, and is private to the same.
// Each tag can be nested to hold other sub-tags so as to create a heriecheal structure.
const organizationTagUserSchema = new Schema({
  name: {
    type: String,
    required: true,
  },
  organizationId: {
    type: Schema.Types.ObjectId,
    ref: "Organization",
    required: true,
  },
  parentTagId: {
    type: Schema.Types.ObjectId,
    ref: "OrganizationTagUser",
    required: false,
    default: null, // A null parent corresponds to a root tag in the organization
  },
  tagColor:{
    type: String,
    required: true,
    default: "#000000",
  }
});
```

Figure 4 : Tag's Schema

UserTag Creation Conditions (CREATE):

- **Database Verification:** The system will verify if the user creating the tag exists in the MongoDB database before proceeding.
- **Organization Validation:** The provided organization ID will be validated to ensure its existence.
- **Privilege Checks:** Only Admins and Super Admins will be authorized to create tags. This can be achieved using role-based access control (RBAC) mechanisms in Talawa-Admin.
- **Hierarchical Tag Checks:**
  - ➔ If parent tags are used, the system will verify that the provided parent ID belongs to the specified organization.
  - ➔ It will also check for duplicate tag names within the same parent tag

GraphQL Mutation for createUserTags:

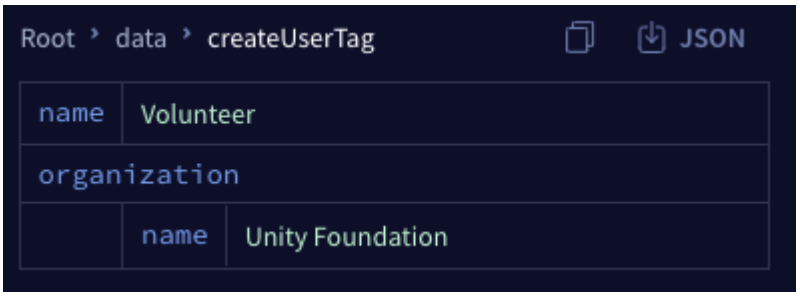
A GraphQL mutation named `CREATE_USER_TAG` will be implemented in Talawa-Admin for tag creation. This mutation will look like this:

input.ts

```
{
  "input": {
    "name": "Volunteer",
    "tagColor": "#FFFFFF",
    "organizationId": "6737904485008f171cf29924"
  }
}
```

- Accept an input object based on the `CreateTagInput` schema.
- Perform the necessary validation and authorization checks as outlined above.
- Call the appropriate Talawa-API endpoint to create the new tag.
- If there's any errors then display in the form of toaster using react-toastify.

➔ The response will look somewhat look like this:



➔ Here is the UI design that will be appropriate for implementation in frontend:

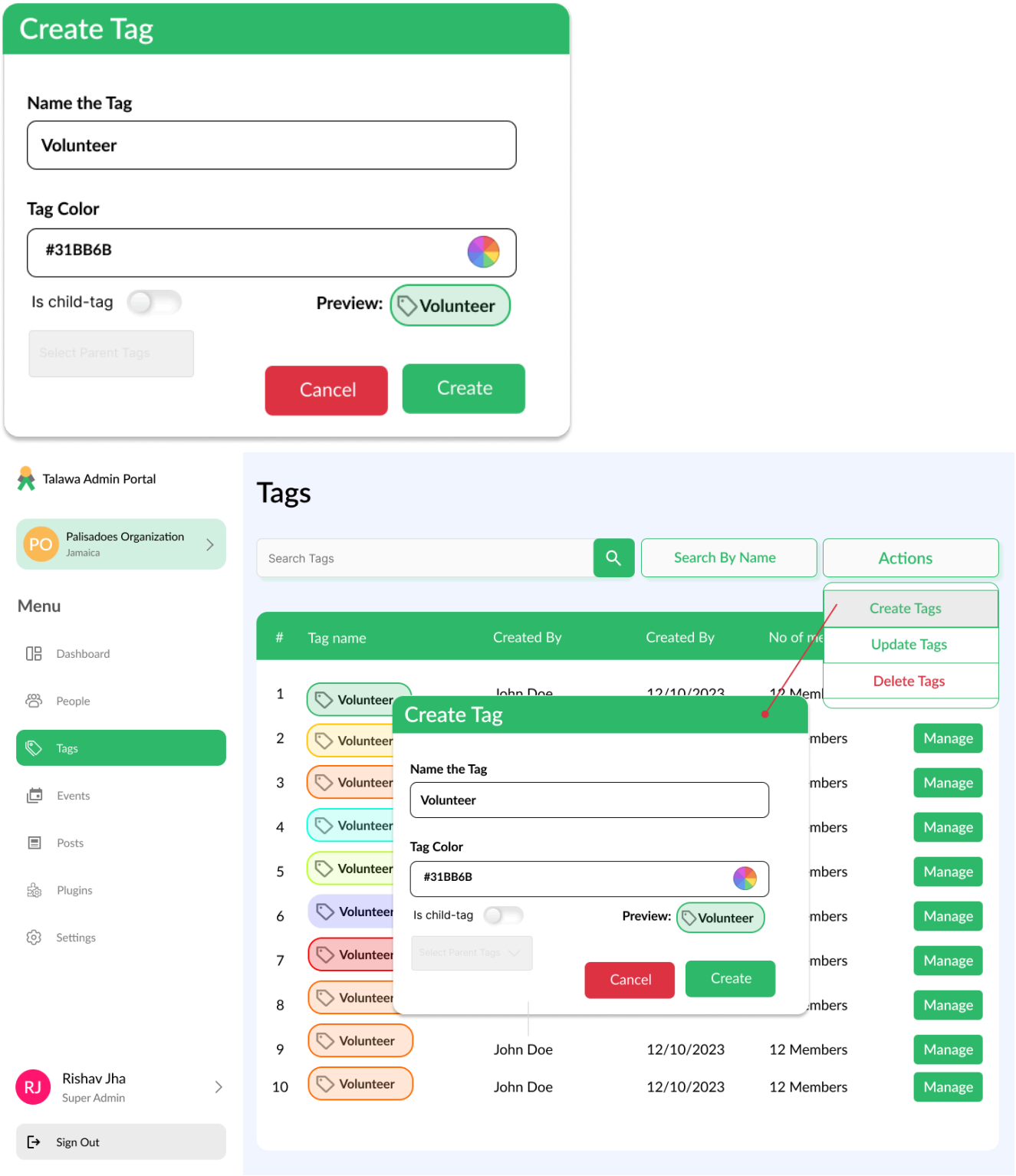


Figure 5 : Actions Dropdown includes CRUD operations

READ:

For displaying tags, we will present them in a specific table format. To retrieve data from the backend, we will implement a GraphQL query that will fetch the necessary information.

query.ts

```
query getTags($filter: TagFilterInput, $sort: TagSortInput,
   $pagination: PaginationInput) {

  tags(filter: $filter, sort: $sort, pagination: $pagination) {
    totalCount
    pageInfo {
      hasNextPage
      endCursor
    }
    edges {
      node {
        id
        name
        description
        organization {
          id
          name
        }
        parentTag {
          id
          name
        }
      }
      cursor
    }
  }
}
```

Figure 6 : Add getTags query in Talawa-Admin

➔ Explanation:

- **Query Name:** `getTags`
- **Arguments:**
  - `filter` : Allows filtering tags based on criteria like name, organization, etc.
  - `sort` : Enables sorting results by fields like name or creation date.
  - `pagination` (optional): Manages large tag lists with pagination parameters like

`first` and `after`.

- **Return Type:** `TagsConnection`

- **Output:**
  - `totalCount`: Total number of matching tags.
  - `pageInfo`: Pagination information (if pagination is used).
  - `edges`: Array of `TagEdge` objects:
    - `node`: The actual `UserTag` object with its fields.
    - `cursor`: Pagination cursor for that tag (if pagination is used).

➔ Additional Considerations:

- **Efficiency:** Consider defining `TagFilterInput`, `TagSortInput`, and `PaginationInput` types for robust filtering, sorting, and pagination capabilities.
- **Error Handling:** Implement error handling mechanisms within your application to handle unexpected API responses or query errors.

➔ Custom Types:

- **TagFilterInput**: Define filter criteria fields (e.g., `name`, `organizationId`, etc.).
- **TagSortInput**: Define sorting fields and directions (e.g., `name: ASC`, `createdAt: DESC`).
- **PaginationInput**: Define pagination parameters (e.g., `first`, `after`, `last`).

schema.graphql

```
type Query {

  getTags(filter: TagFilterInput,
    sort: TagSortInput, pagination: PaginationInput): TagsConnection

}
```

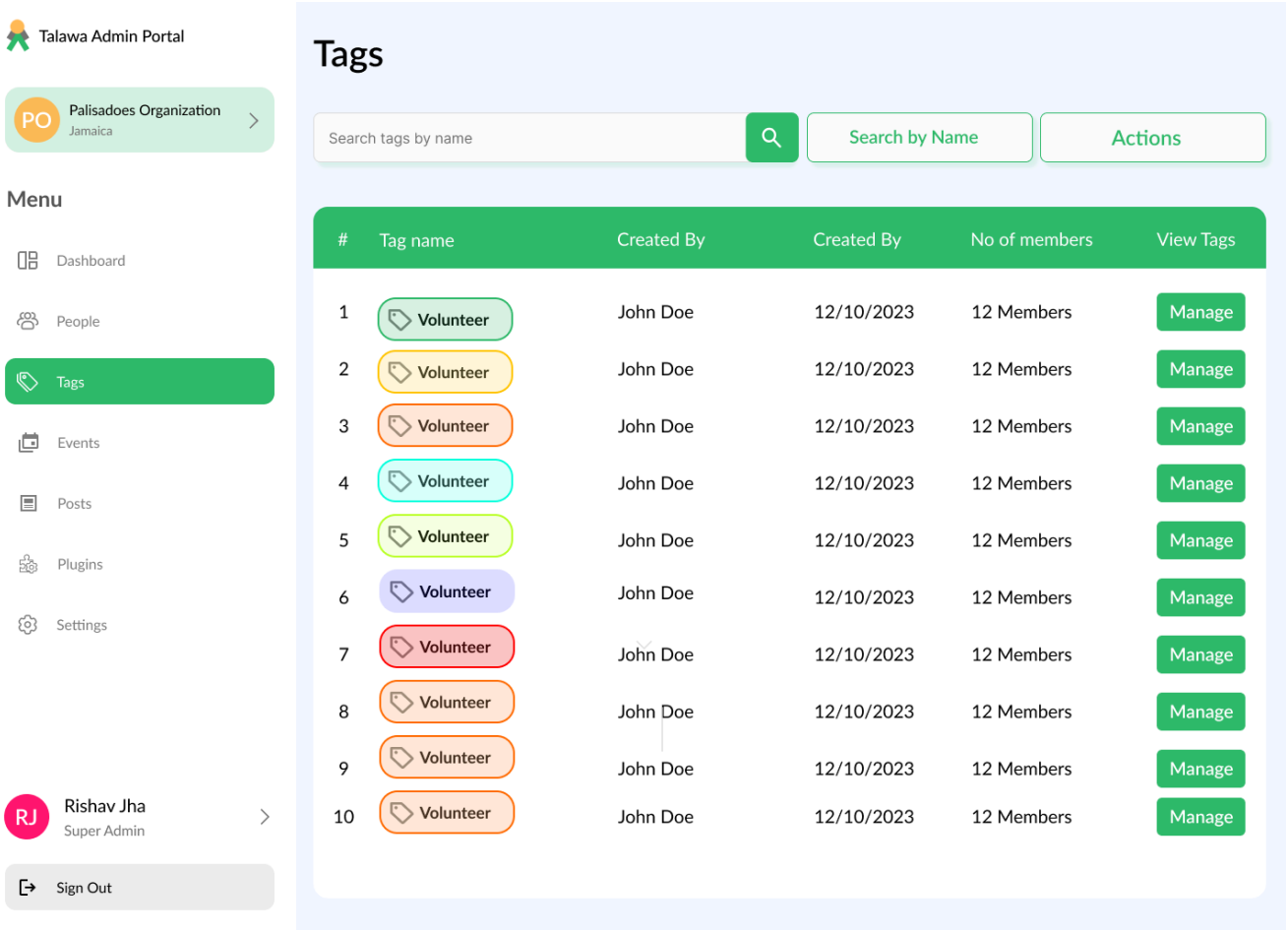


Figure 7 : UI for viewing Tags

➔ Admin/Super Admin can also view tag members by clicking the “Manage” button:

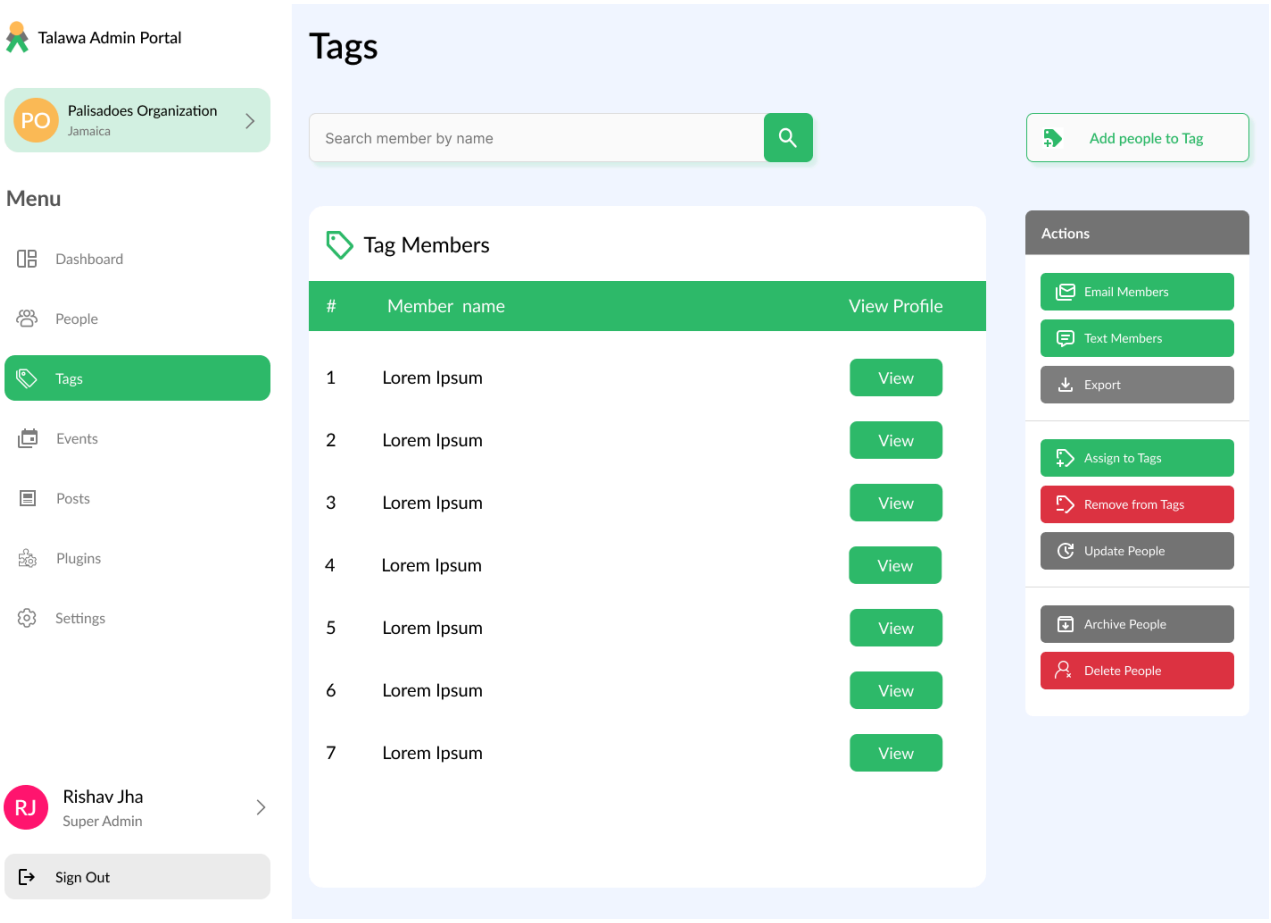
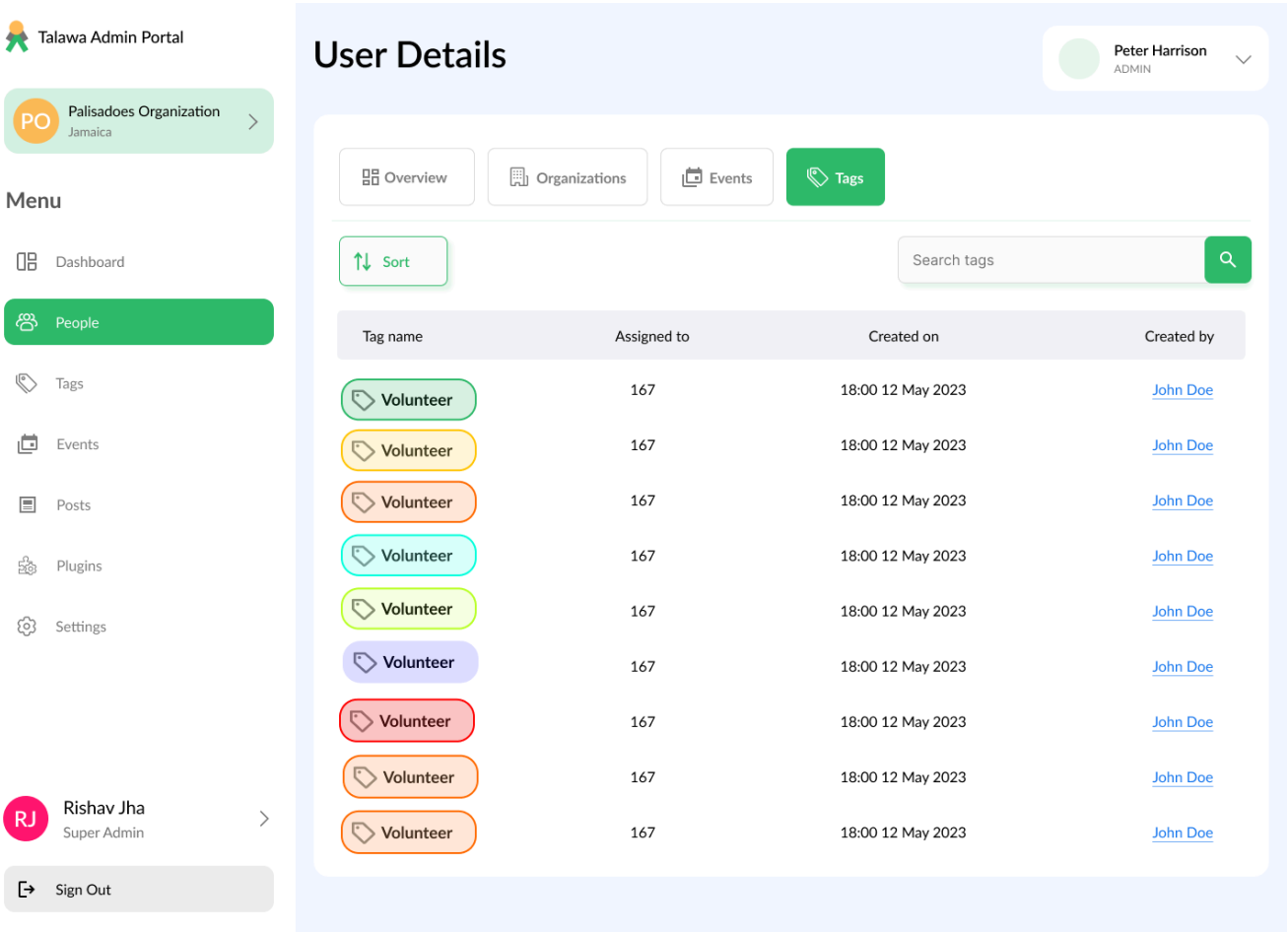


Figure 8 : UI for viewing members in specific selected tag

View Tags in User details screen:

I propose adding a feature on the people screen where you admin can see if members have any specific tags they have within the organization. This way, admin will have a clearer picture of each person's role and connections within the org.



Update Functionality

The update functionality in the Tags screen allows Admins to modify both the name and color of the tags.

Accessing Update Tag

- Navigate to the Tags screen in your application.
- Click on the action button in the top-right corner of the screen.
- In the dropdown menu that appears, select "Update Tags".
- To update a tag name, simply click on the existing name and enter the new name.
- To update a tag color, click on the color swatch next to the tag name and select a new color from the color picker.
- Click on the "Update" button to save your changes

➔ The `updateTag` mutation accepts inputs such as the tag's name and color.

updateTag.ts

```
const UPDATE_TAG_MUTATION = gql`
  mutation UpdateTag($id: ID!, $data: UpdateTagInput!) {
    updateTag(id: $id, data: $data) {
      id
      name
      tagColor
    }
  }
`;

async function updateTag(tagId, updateData) {
  const { data } = await client.mutate({
    mutation: UPDATE_TAG_MUTATION,
    variables: { id: tagId, data: updateData },
  });

  return data.updateTag;
}
```

➔ **UI design**

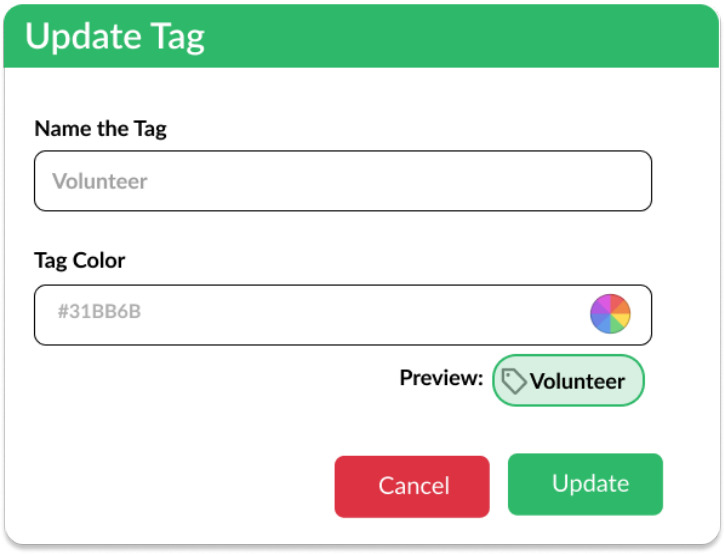


Figure 9 : Updating tag Modal

Delete Tags(DELETE):

Similarly to the other functionalities, the delete functionality for CRUD operations will operate in the same manner. We will utilize the removeUserTags mutation, which accepts an ID as input.

deleteUserTag.ts

```
const REMOVE_USER_TAG = mutation RemoveUserTag($removeUserTagId: ID!) {
  removeUserTag(id: $removeUserTagId) {
    name
    _id
  }
}
```

removeUserTag.ts

```
import { useMutation } from '@apollo/client';
import { REMOVE_USER_TAG } from 'GraphQl/Mutations/UserMutations';

async function removeUserTag(userTagId) {
  const [removeUserTagMutation, { loading, error }] = useMutation(REMOVE_USER_TAG);

  if (loading) {
    console.log('Removing user tag...');
    return;
  }

  if (error) {
    console.error('Error removing user tag:', error);
    // Handle errors appropriately (e.g., display error message to user)
    return;
  }

  try {
    const { data } = await removeUserTagMutation({
      variables: { removeUserTagId },
    });

    if (data?.removeUserTag) {
      console.log('User tag removed successfully:', data.removeUserTag);
    } else {
      console.error('Unexpected response from mutation');
    }
  } catch (error) {
    console.error('Error removing user tag:', error);
    // Handle unexpected errors
  }
}

// Example usage:
const userTagIdToRemove = '123';
removeUserTag(userTagIdToRemove);
```

Figure 10 : Remove user tag logic

The deletion of parent user tags involves iterating through the associated child tags as seen on code present in Figure 10.

Upon initiating the delete action, a confirmation modal will appear to confirm the deletion. Subsequently, the `REMOVE_USER_TAG` mutation from Talawa-Api will be invoked. This mutation encapsulates the logic for iteration and will handle any errors that may arise during the process.

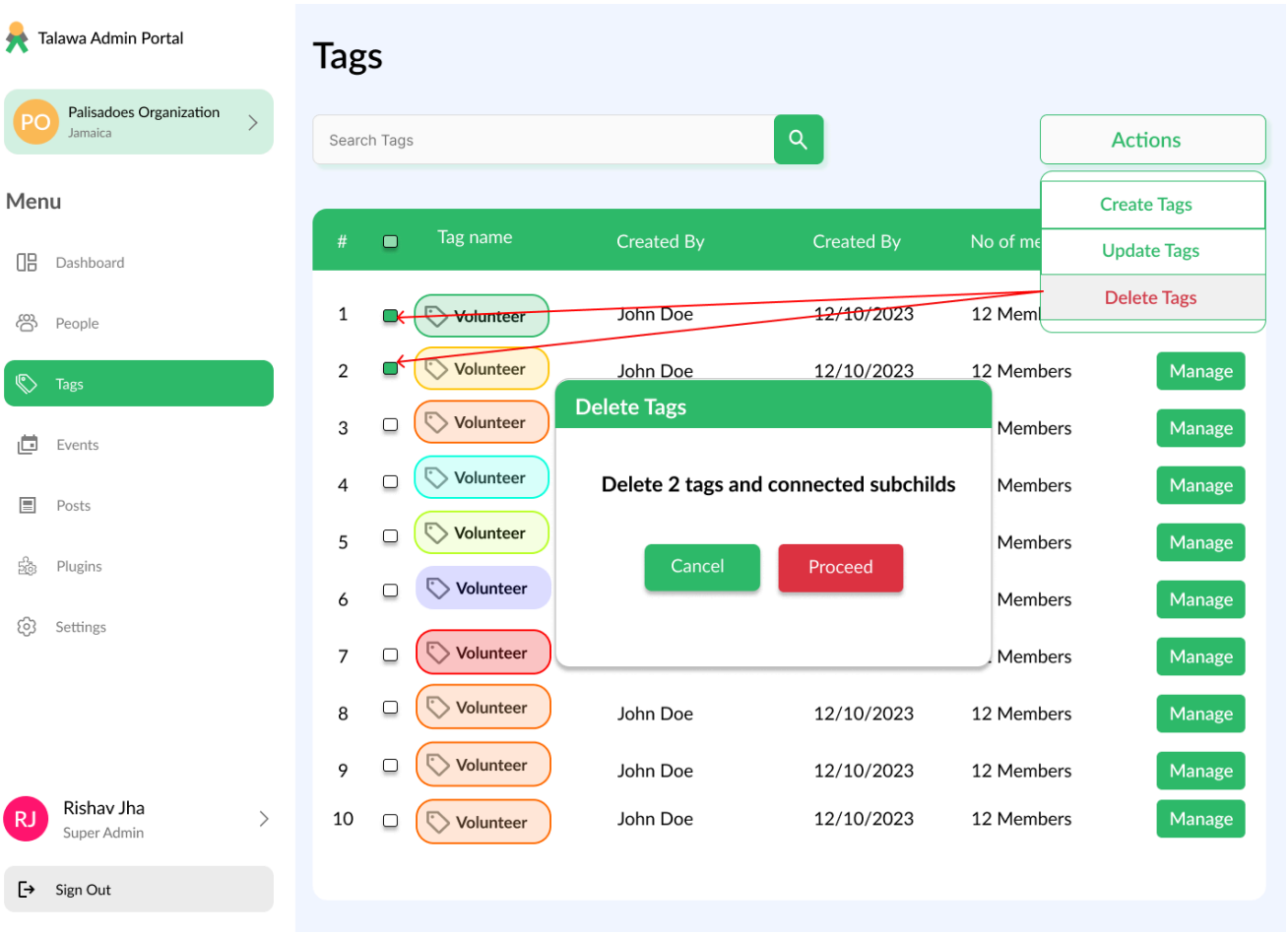


Figure 11 : UI for remove user tag

★Broadcasting Emails/Messages to a Group of Users with Certain Tags

Objective

Develop a system for sending targeted email/message broadcasts to users based on the tags they are associated with in Talawa.

Proposed Solution

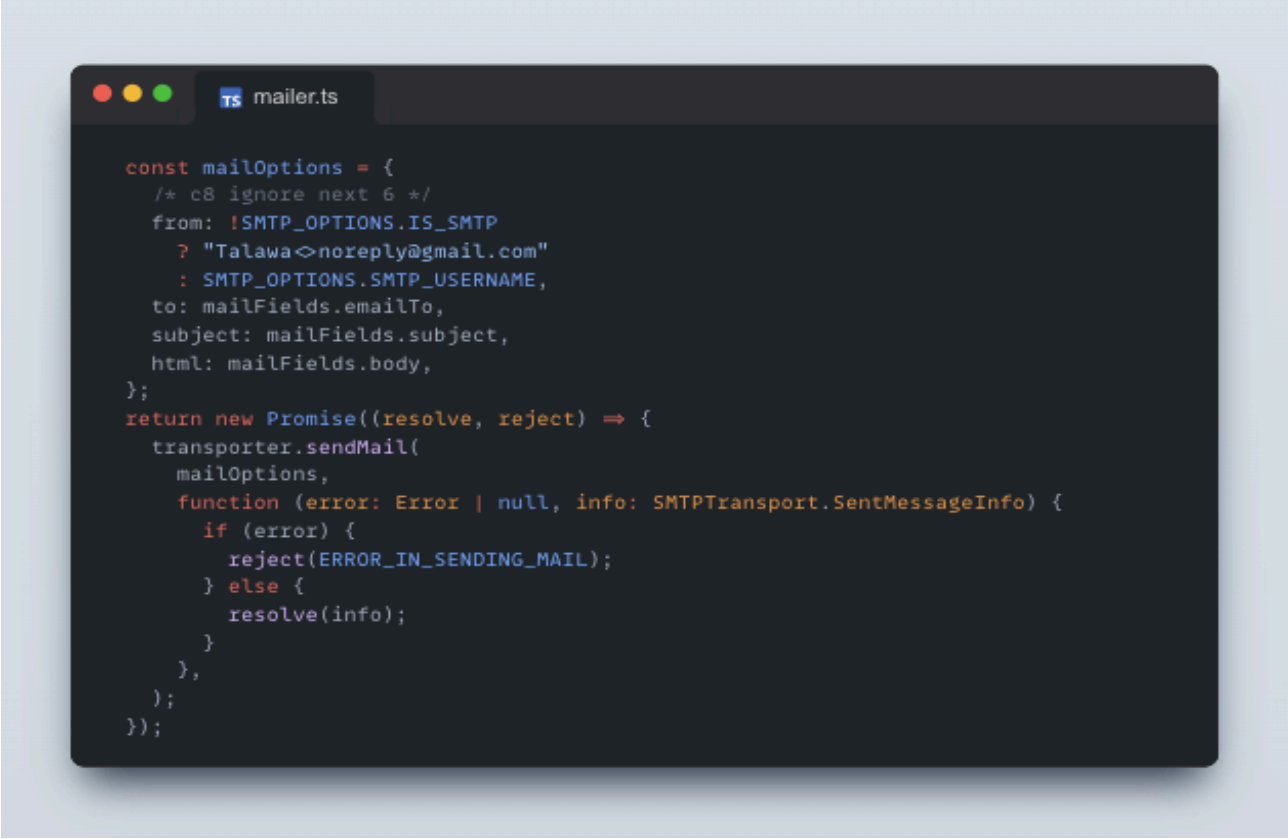
We can draw inspiration from Breeze CHMS for both mailing and messaging functionalities.

➔ Email:

Utilize the existing `mailer.ts` function with appropriate modifications for broadcasting:

- Batch emails to avoid overwhelming servers (e.g., send 100 at a time).

We will continue to use **Nodemailer** for sending emails to groups of users based on specific tags.



➔ **Send Email Modal:**

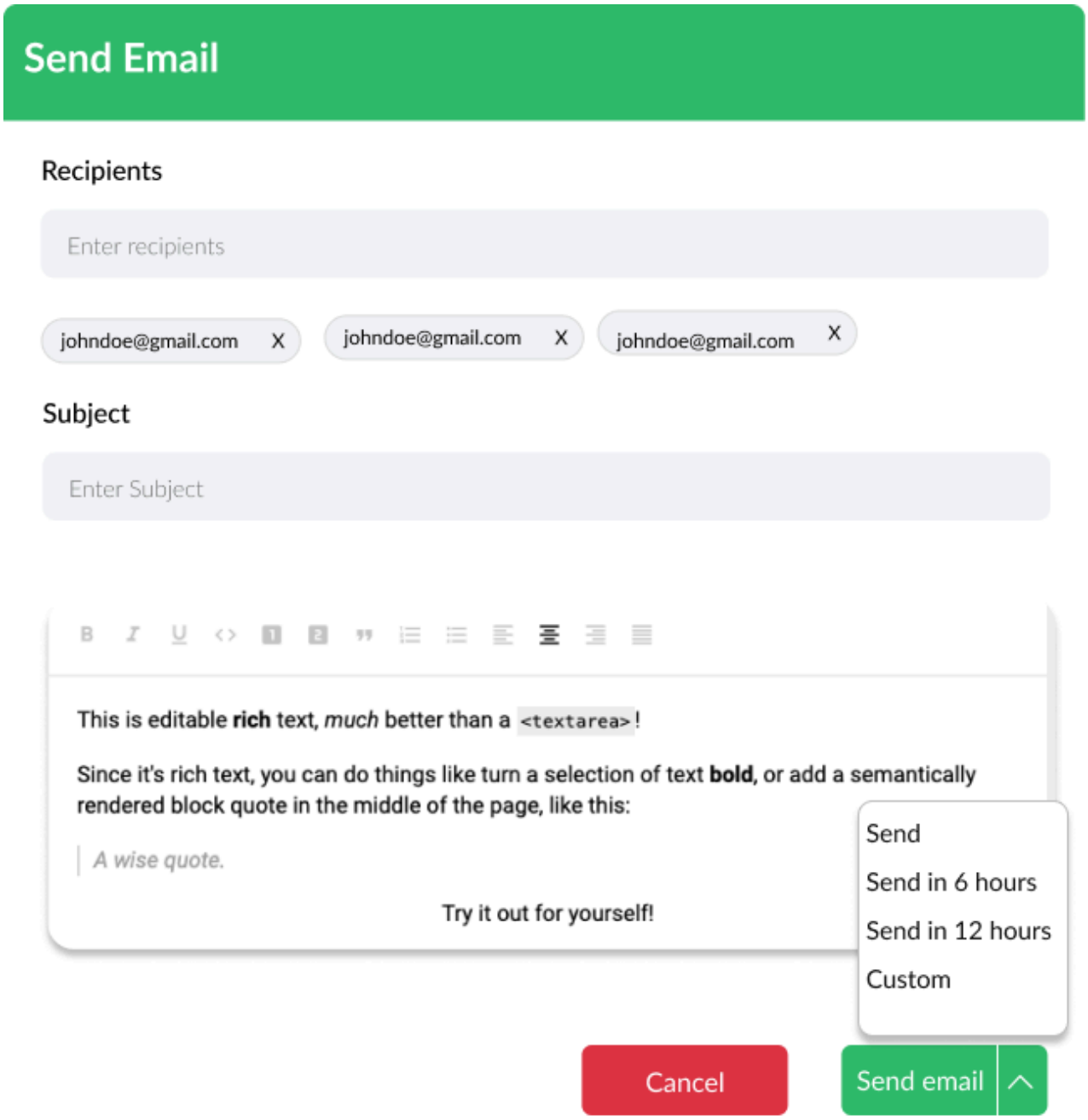


Figure 12 : Proposed modal UI for send Email

➔ **Implementing Time-Bound Broadcasts with Background Jobs**

Functionality:

- **Broadcast Scheduling:**

- Allow administrators to define a specific time for sending broadcasts (e.g., "send in 2 hours before the event") or a time offset (e.g., "send in 12 hours").

- **Background Job Scheduler:**
  - We will use a background job scheduler like Bull.
  - When a broadcast is scheduled:
    - We will calculate the actual trigger time based on the selected event and time offset.
    - A background job with the following information:
      - Broadcast ID
      - Trigger time
      - Broadcast content (email or message)
      - User selection criteria (tags)

Code Example:

```typescript
import { Queue } from 'bull';

interface BroadcastData {
  eventId: string;
  timeOffset: number;
  content: string;
  userSelection: any;
}

const queue = new Queue('broadcastQueue');

async function scheduleBroadcast(broadcastData: BroadcastData): Promise<void> {
  const { eventId, timeOffset, content, userSelection } = broadcastData;
  try {
    const event = await findEventById(eventId);
    const triggerTime = event.startTime.subtract(timeOffset, 'hours');

    // Create a job with broadcast details
    await queue.add({
      broadcastId: broadcastData.id,
      triggerTime,
      content,
      userSelection,
    });
  } catch (error) {
    console.error("Error scheduling broadcast:", error);
  }
}
```

scheduleBroadcast.ts

```ts
async function processBroadcastJob(job: Queue.Job): Promise<void> {
  const { broadcastId, content, userSelection } = job.data;

  try {
    const users = await findUsersWithTags(userSelection);

    // Send email/message to each user with the content
    for (const user of users) {
      await sendEmail(user.email, content);
    }

    // Mark the job as completed
    await job.complete();
  } catch (error) {
    console.error("Error processing broadcast job:", error);
    // Handle job processing errors (e.g., sending email failures)
  }
}

queue.process(processBroadcastJob);
```

Explanation:

- The `scheduleBroadcast` function calculates the trigger time and creates a Bull job with relevant data.
- The `processBroadcastJob` worker function is triggered at the scheduled time:
    - Fetches users based on the broadcast's tag criteria.
    - Sends email/message content to each user using your existing logic (e.g., `mailer.ts`).

Benefits:

- Enables sending broadcasts at specific times relative to events.
- Improves scheduling flexibility and user engagement.
- Leverages background jobs for efficient processing.

➔ Proposed UI for Email Members for specific tags

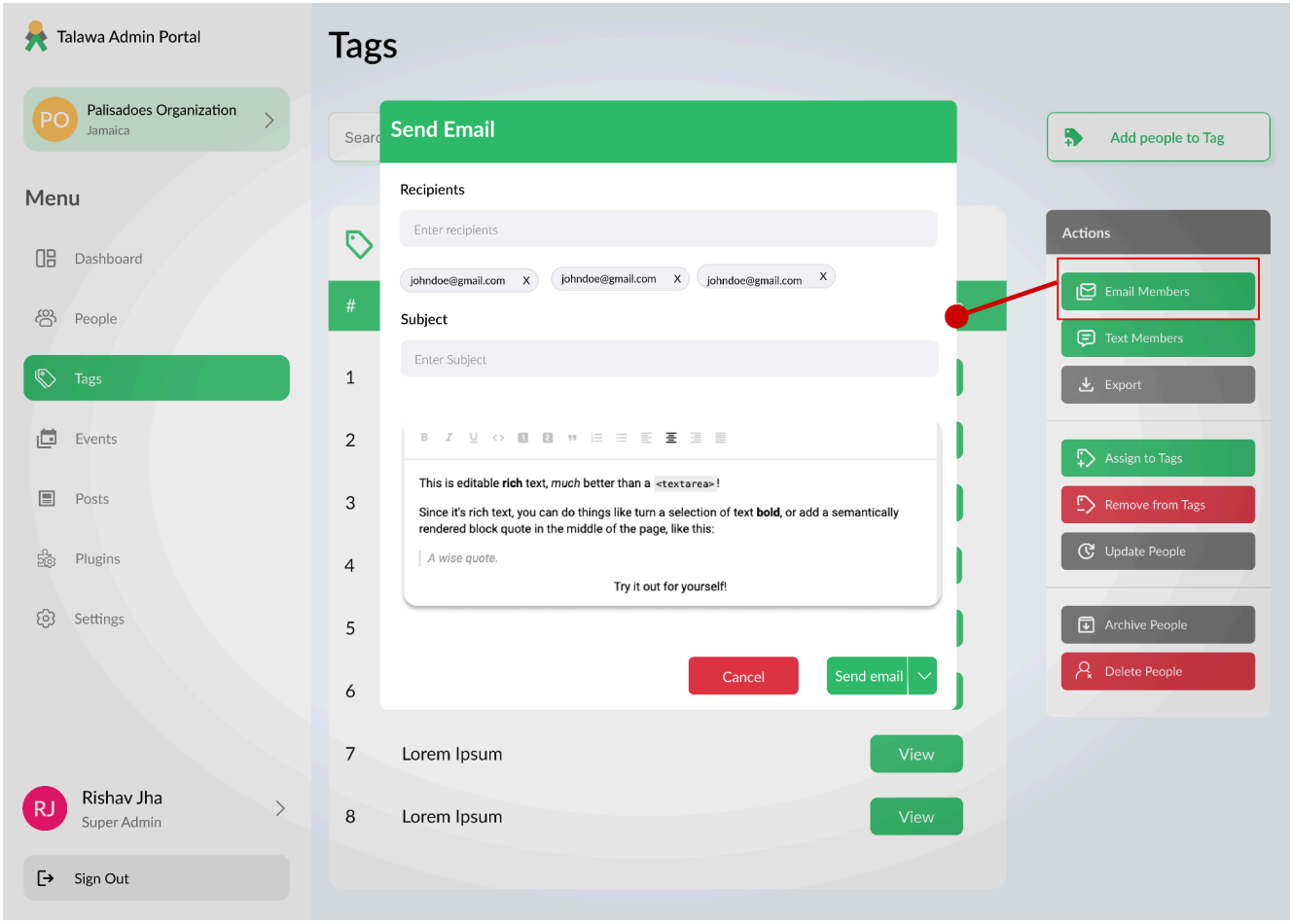


Figure 13 : UI for send Email

➔ Leveraging Talawa Admin's Chat Functionality for User Messaging

While the overall structure of message sending might resemble email, Talawa Admin's existing chat functionality offers a more suitable and integrated approach for sending messages to users within the platform. Here's a breakdown of key aspects and considerations:

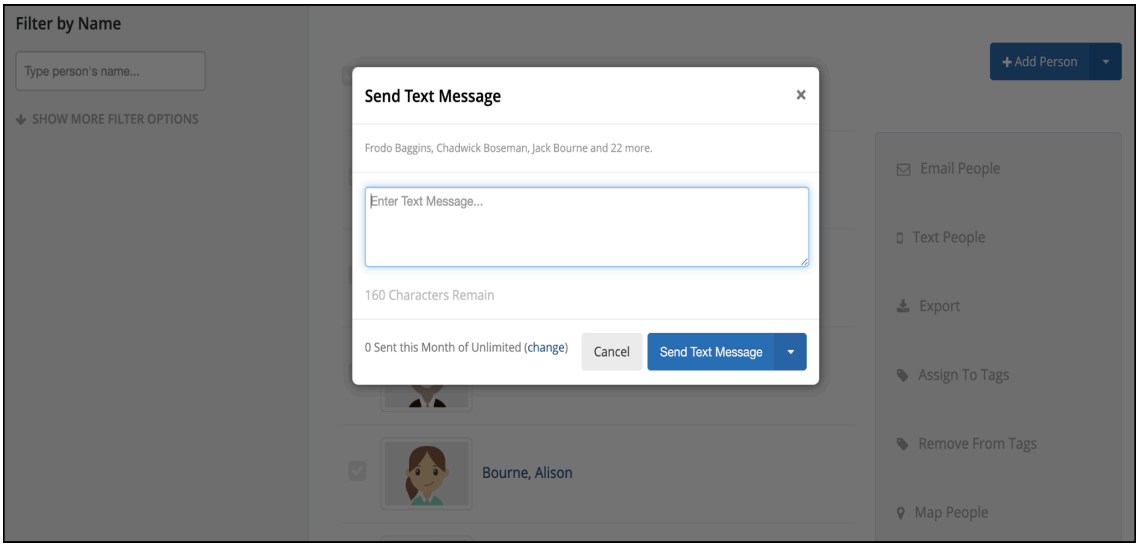
Core Functionality:

- **Utilize Direct Chat mutation:** Employ the existing direct chat mutation as the foundation for sending messages. This ensures a unified communication

experience within Talawa Admin.

- **Targeted Communication:** Allow sending messages user groups defined by relevant tags.
- **Content Composition:**
    - Offer a rich text editor or similar interface for composing messages.
    - Enable admins/superadmins to format text, include links, or attach files if supported by the chat system.

➔ **Breeze CHMS UI For sending Message:**



Proposed design for Broadcast message:

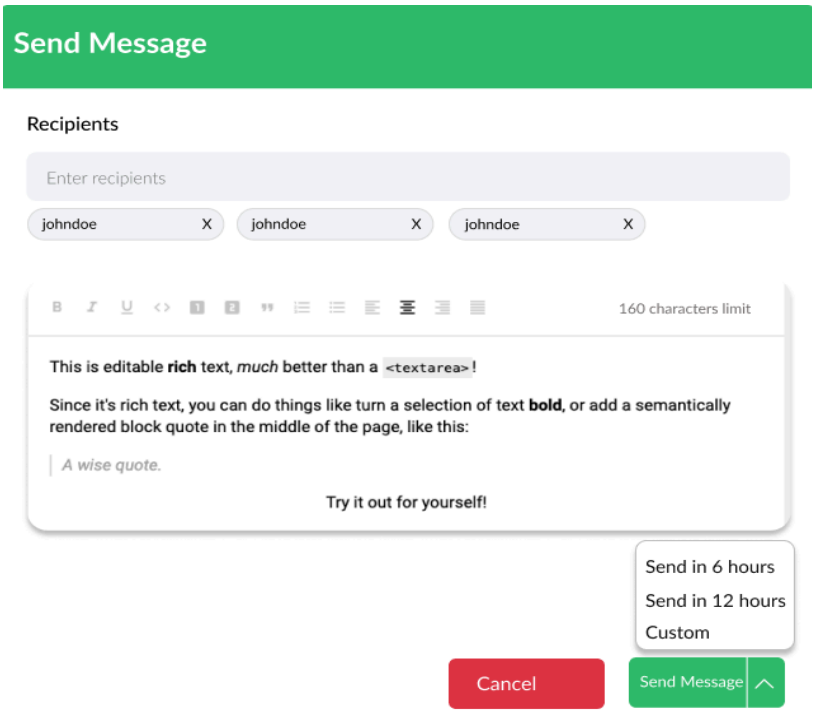


Figure 14 : Proposed modal UI for sending message

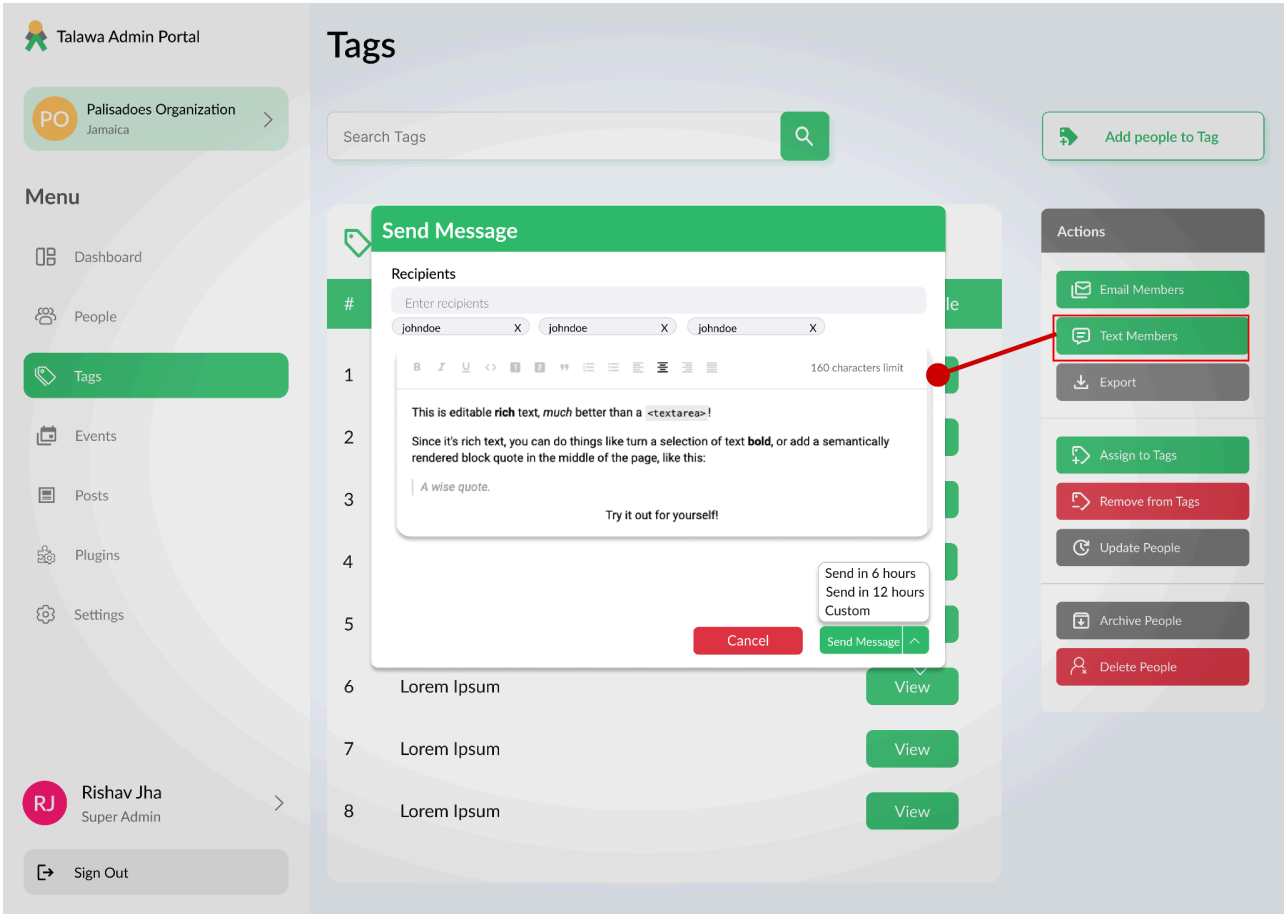


Figure 15 : Proposed UI for sending message

Here's a comprehensive schema for Talawa Admin**:**

- **Message Type:** Captures message details.
  - `sender`: Reference to the user who sent the message.
  - `recipients`: Non-empty array of `User` objects who received the message.
  - `message`: Content of the message.
  - `attachments`: Non-empty list of attachment URLs.
  - `metadata`: Optional data associated with the message.
  - `createdAt`: The createdAt field captures when the message object itself was created and doesn't directly control email delivery timing.

types.ts

```
type SendMessageInput {
  recipients: [String!]!  # Required array of recipient IDs (non-empty)
  message: String!  # Required message content (string)
  attachments: [String!]!  # Optional list of attachment URLs (non-empty)
  metadata: JSON!  # Optional JSON object for additional data
}

type SendMessageResponse {
  success: Boolean!  # Indicates message sending success
  error: String  # Optional error message if unsuccessful
}

type Message {
  id: ID!
  sender: User!  # Reference to the message sender
  recipients: [User!]!  # Array of message recipients
  message: String!  # Message content
  attachments: [String!]!  # List of attachment URLs (if any)
  metadata: JSON!  # Optional JSON object containing additional data
  createdAt: DateTime!  # Timestamp of message creation
}
```

- **SendMessageInput:** Captures data required to send a message.
    - `recipients`: Non-nullable list of non-nullable `String` values representing the recipient user IDs (ensures to provide at least one recipient).
    - `message`: Non-nullable `String` containing the message content.
    - `attachments`: Optional list of non-nullable `String` URLs holding the attachment file locations (if any).
    - `metadata`: Optional `JSON` object allowing you to store additional data associated with the message.

broadcastMessage.ts

```ts
const messageData: SendMessageInput = {
    recipients: ["user123", "user456"], // Array of recipient user IDs
    message: "This is a test message from GraphQL mutation.",
    attachments: ["https://example.com/image.jpg"],
    metadata: { priority: "high" },

};
```

- **SendMessageResponse:** Represents the response after attempting to send a message.
    - `success`: Non-nullable `Boolean` indicating whether the message sending was successful.
    - `error`: Optional `String` containing an error message if the message sending failed (provides details about the issue).

➔ GraphQL mutation for the Broadcasting message and function logic:

mutation.ts

```ts
const BROADCAST_MESSAGE = gql`
  mutation sendMessage($input: SendMessageInput!) {
    sendMessage(input: $input) {
      success
      error
    }
  }
`;
```

broadcastMessage.ts

```ts
async function sendMessage(messageData: SendMessageInput): Promise<void> {
  const { recipient, message,attachments, metadata } = messageData;

  // Send message using Talawa Api
  const chatResponse = await DirectChat({ recipient, message, attachments, metadata });

  if (chatResponse.success) {
    console.log("Message sent successfully via chat API.");
  } else {
    console.error("Error sending message via chat API:", chatResponse.error);
  }
}

async function DirectChat(data: any) {
  // Replace with actual Talawa Admin chat API call logic
  return { success: true };
}
```

★ Filtering of Users based on Tags for Targeted Communication.

Current Scenario

The current Tags screen in Talawa-Admin lacks filtering functionality, making it difficult to navigate and manage a potentially large volume of members. This hinders efficient identification users associated to the specific tag.

Proposed Solution

I propose the addition of a dedicated screen to list all members associated with a particular tag.

- In the Tags screen, I suggest including a "manage" button in the “View Tag” column.
- When clicked, this button will navigate to a manage screen, which will list all the members assigned to the respective tag.
- The manage screen will consist of two columns: "Member Name" and "View Profile."
- The "Member Name" column will display the member's name, while the "View Profile" column will navigate to the person's profile to view more detail about the person.

Furthermore, I recommend incorporating a dedicated search bar in the view tag screen. This search bar will allow Admins/Superadmins to search for specific members by name, enhancing the efficiency of communication efforts.

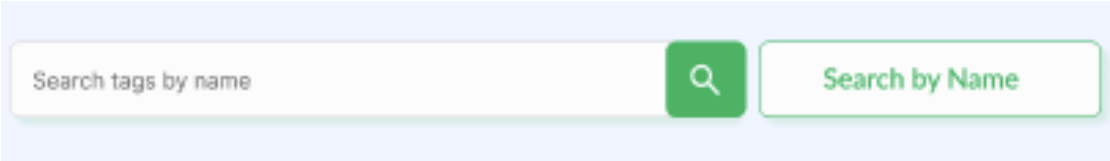


Figure 16 : Search bar in Tag's Screen

➔ Here's the sample code for filtering members using the search bar:

```
const searchMembersInTags = (value: string): void => {
  setSearchByNameInTags(value);
  if (value === '') {
    setMembersInTags(allMembers); // Assuming allMembers is the array containing all members
  } else {
    const filteredMembers = allMembers.filter((member) =>
      member.tags.some((tag) =>
        tag.name.toLowerCase().includes(value.toLowerCase()),
      ),
    );
    setMembersInTags(filteredMembers);
  }
};
```

➔ Proposed UI for View Tag Column:

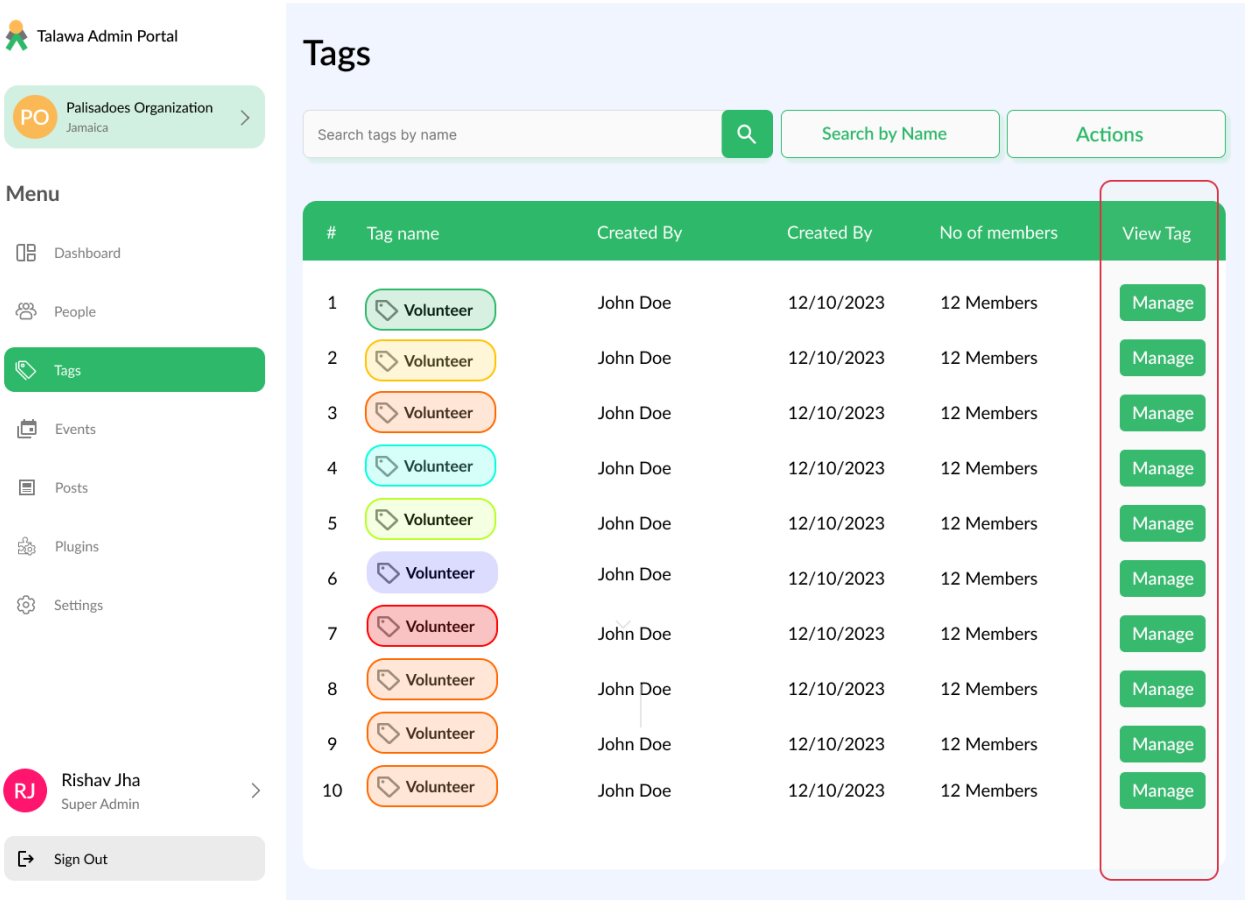


Figure 17 : View tag column UI

Additionally, I would like to propose tag filtering on the Tags screen, which facilitates smoother filtering of tags from a long list of tags.

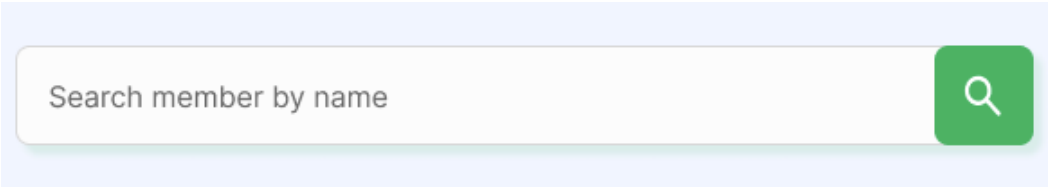


Figure 18 : Search bar in Tag's members

➔ Proposed UI for Filtering of members based on tags:

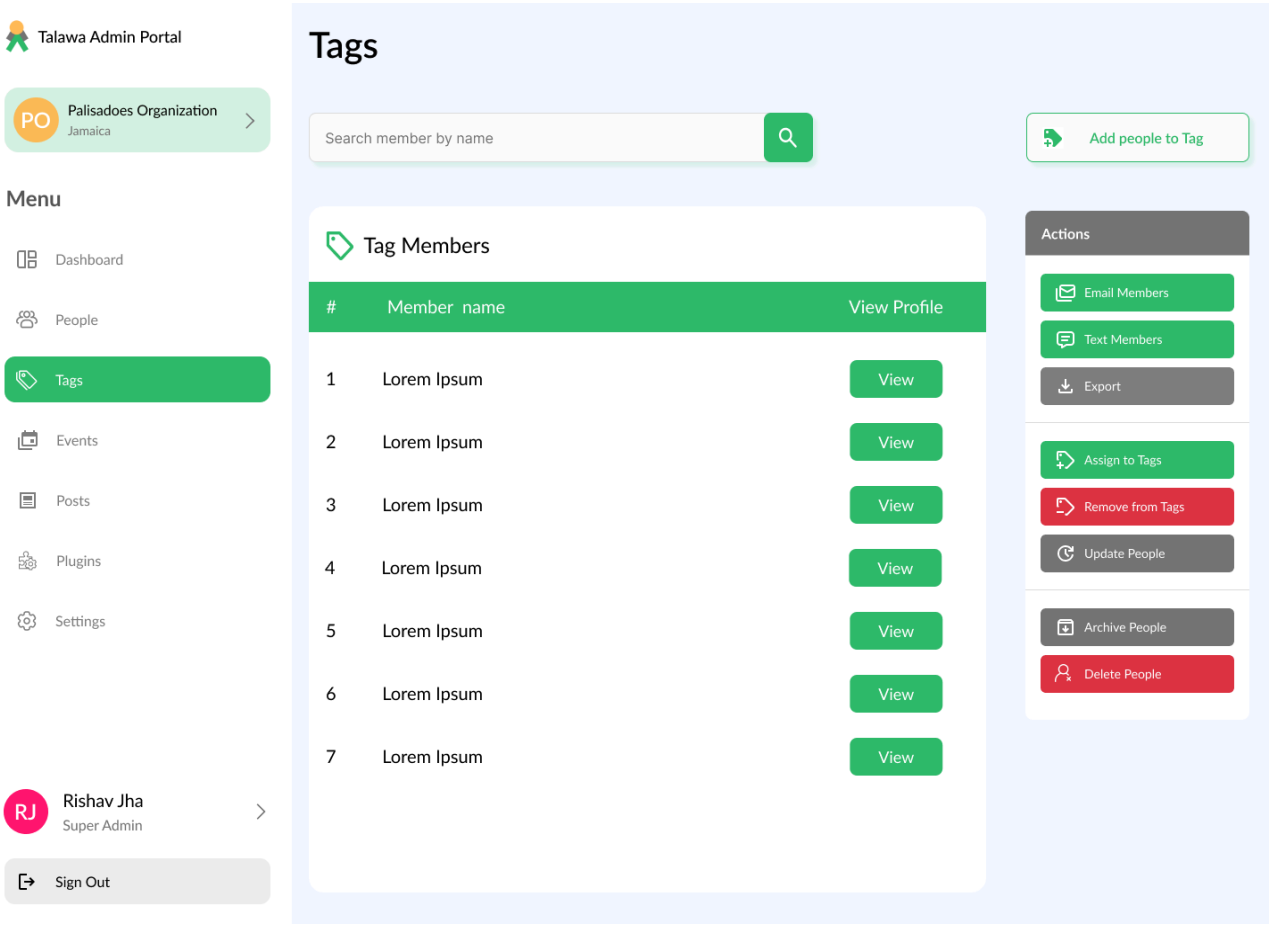


Figure 19 : Proposed UI for tag members

★ Assign/Unassign User Tag from Members:

Current Scenario

In Talawa-API, we already have backend functionalities called assignUserTag.ts and unassignUserTag.ts for assigning and unassigning members from tags. However, the design and support for admins haven't been developed yet.

Proposed Changes

I will implement the functionality on the front end to allow admins/superadmins to assign and unassign tags to members.

➔ GraphQL mutation for assigning user tags:

mutation.graphql

```
mutation Mutation($input: ToggleUserTagAssignInput!) {
  assignUserTag(input: $input) {
    firstName
    email
    _id
  }
}
```

➔ Input from Superadmin/admin will be:

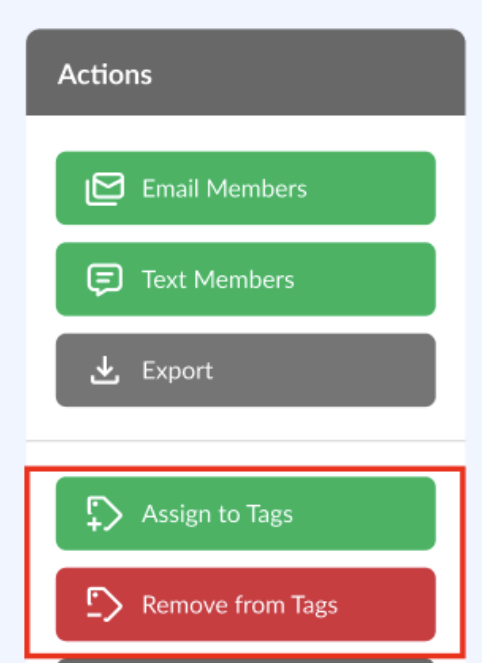
```
"input": {
    "userId": null,
    "tagId": null
  }
```

➔ Similarly, for unassigning the user tag we use the unassignUserTag mutation, which takes input as tagId and userId

mutation.graphql

```
mutation UnassignUserTag($input: ToggleUserTagAssignInput!) {
  unassignUserTag(input: $input) {
    _id
    email
    firstName
  }
}
```

UI for Assigning and unassigning tags to members:



★ Tags in Talawa-User Portal:

The Talawa User Portal is a web application focused on users who do not have access to fully-featured smartphones. We aim to utilize tags within the user portal to enhance communication among Talawa users.

Current Scenario

Currently, Talawa users lack the functionality of user tags.

Proposed Approach

I propose implementing tags in the Talawa User Portal. These tags will be visible on the People screen, Post Cards and User settings, enhancing the discoverability of individuals with similar interests within the organization.

User Journey:

- **Sign-in:** Member sign in through the User Login Portal.
- **Select Organization:** Member choose their respective organization.
- **Feeds Display:** Posts appear on the organization dashboard. On the feeds, members can view tags alongside other member's names, enabling them to find individuals with similar interests.
- **People Screen:** In the People screen, members can view a specific tags column in

the people list. This enhances the visibility and discoverability of individuals with particular interests and responsibilities within the organization.

➔ **Sample Interface for People Screen with tags**

people.tsx

```tsx
interface InterfaceOrganizationCardProps {
  id: string;
  name: string;
  image: string;
  email: string;
  userType:string;
  tags: [string]
}
```

➔ **Proposed UI for people screen with tags column:**

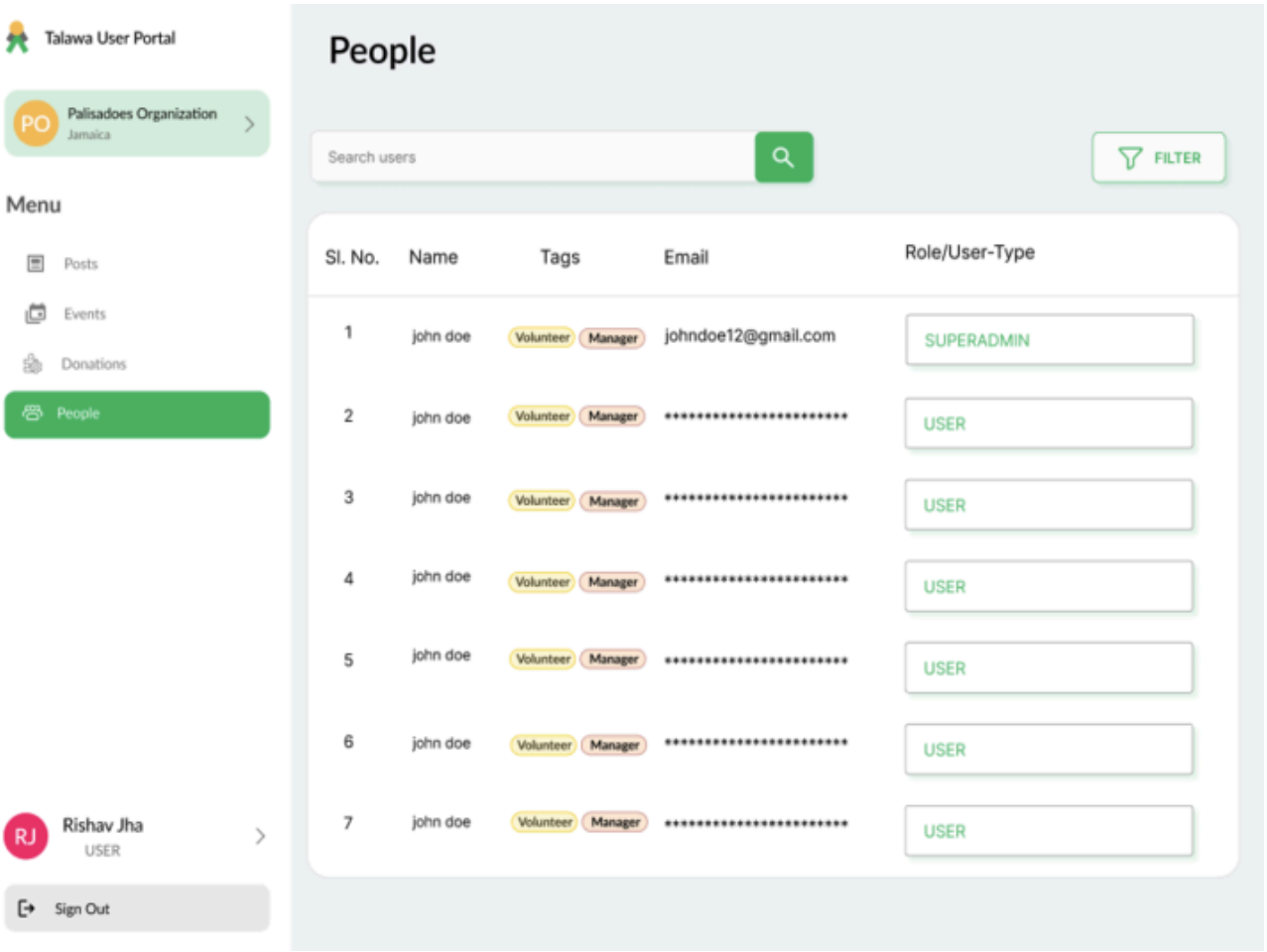


Figure 20 : Proposed UI for People screen

User Settings Tags:

➔ Proposed approach

We'll populate the assigned tags in the user profile by iterating through the organizations the user has joined and incorporating the tags assigned by those organizations.

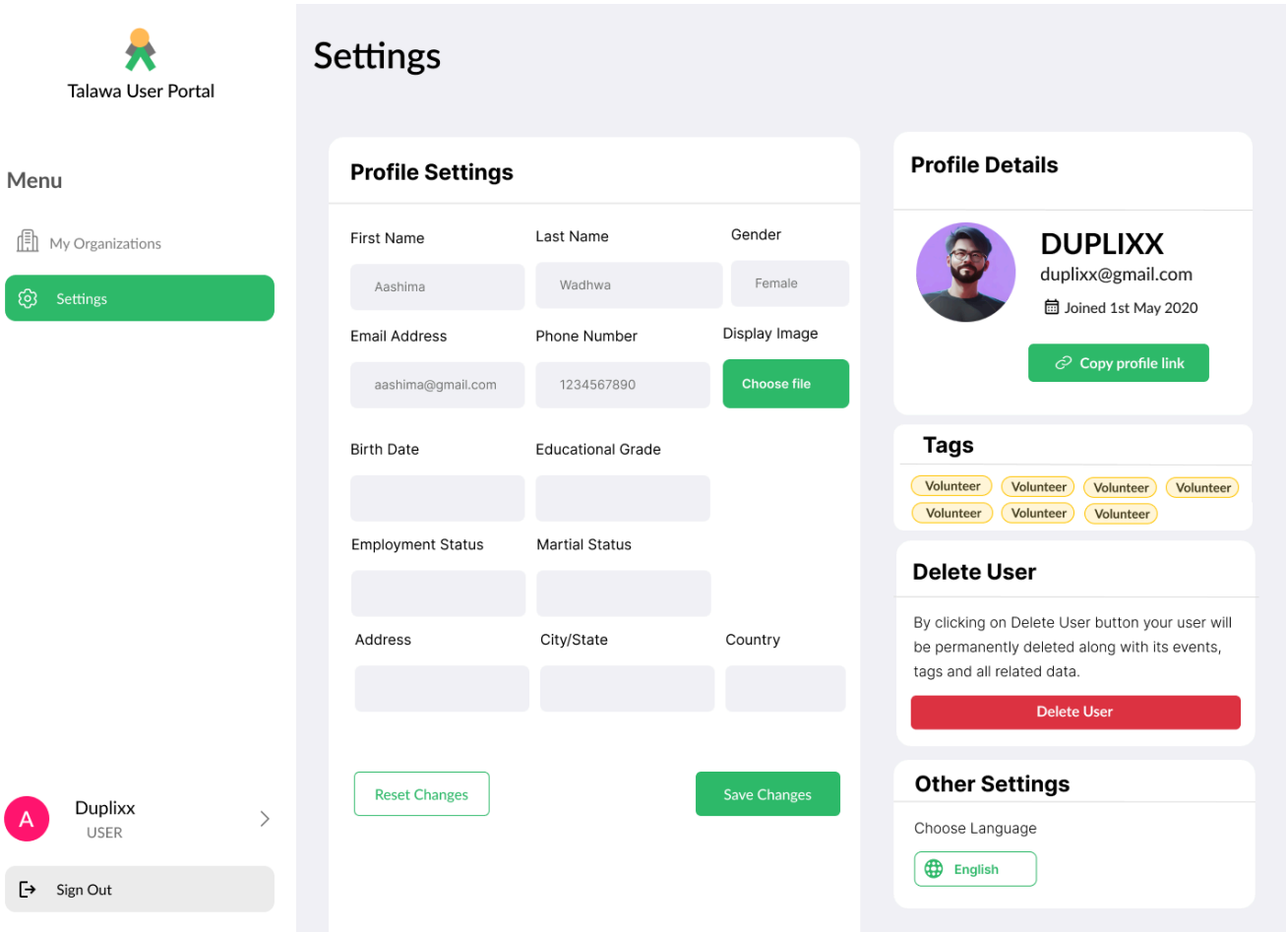


Figure 21 : Proposed UI for user settings

➔ **Sample Interface for feed post**:

OrgPostCard.tsx

```tsx
interface InterfaceOrgPostCardProps {
  key: string;
  id: string;
  postTitle: string;
  postInfo: string;
  postAuthor: string;
  postPhoto: string | null;
  postVideo: string | null;
  pinned: boolean;
  userTags: string[];
}
```

➔ **Proposed UI for feed post (optional)**:

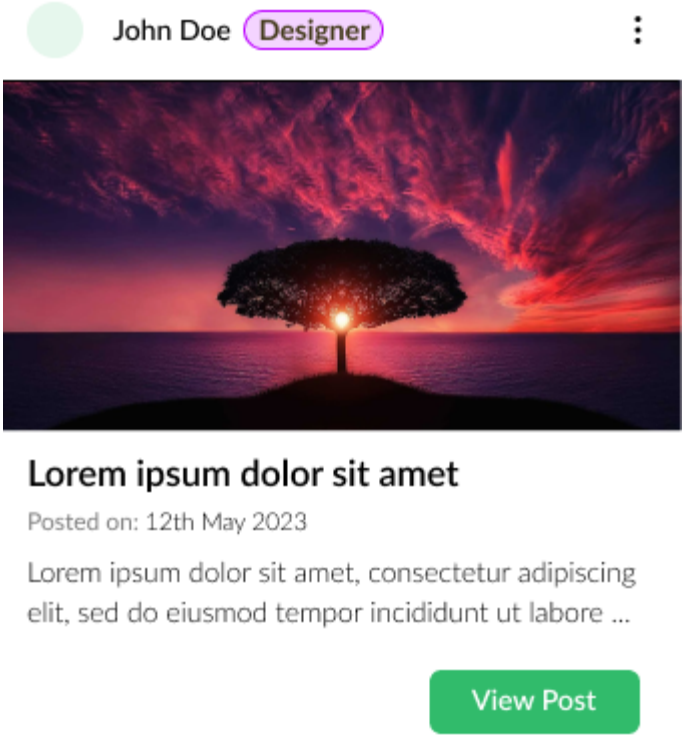


Figure 22 : Proposed UI for feed cards

```jsx
{/* Display user tags */}
<div className={styles.userTagsContainer}>
  {props.userTags && props.userTags.length > 0 && (
    <div className={styles.userTags}>
      {props.userTags.map((tag, index) => (
        <span key={index} className={styles.tag}>
          {tag}
        </span>
      ))}
    </div>
  )}
</div>
```

Figure 23 : Sample code for mapping tags

★ Tags in Talawa:

Overview:

Talawa is a comprehensive platform that aims to revolutionize the way organizations manage and interact with their data and content. Talawa empowers administrators to access and manage content with ease through the Talawa Admin interface, ensuring that information remains up-to-date and accessible to the platform's members. With the addition of tags in the Talawa mobile app, it will be easier to navigate and filter organization members, providing a sense of personalization to the users.

Proposed Approach:

I propose integrating user tags in mobile screens, particularly focusing on the *organization members* and *member profile* screens. This integration will benefit users by enhancing navigation and personalization features.

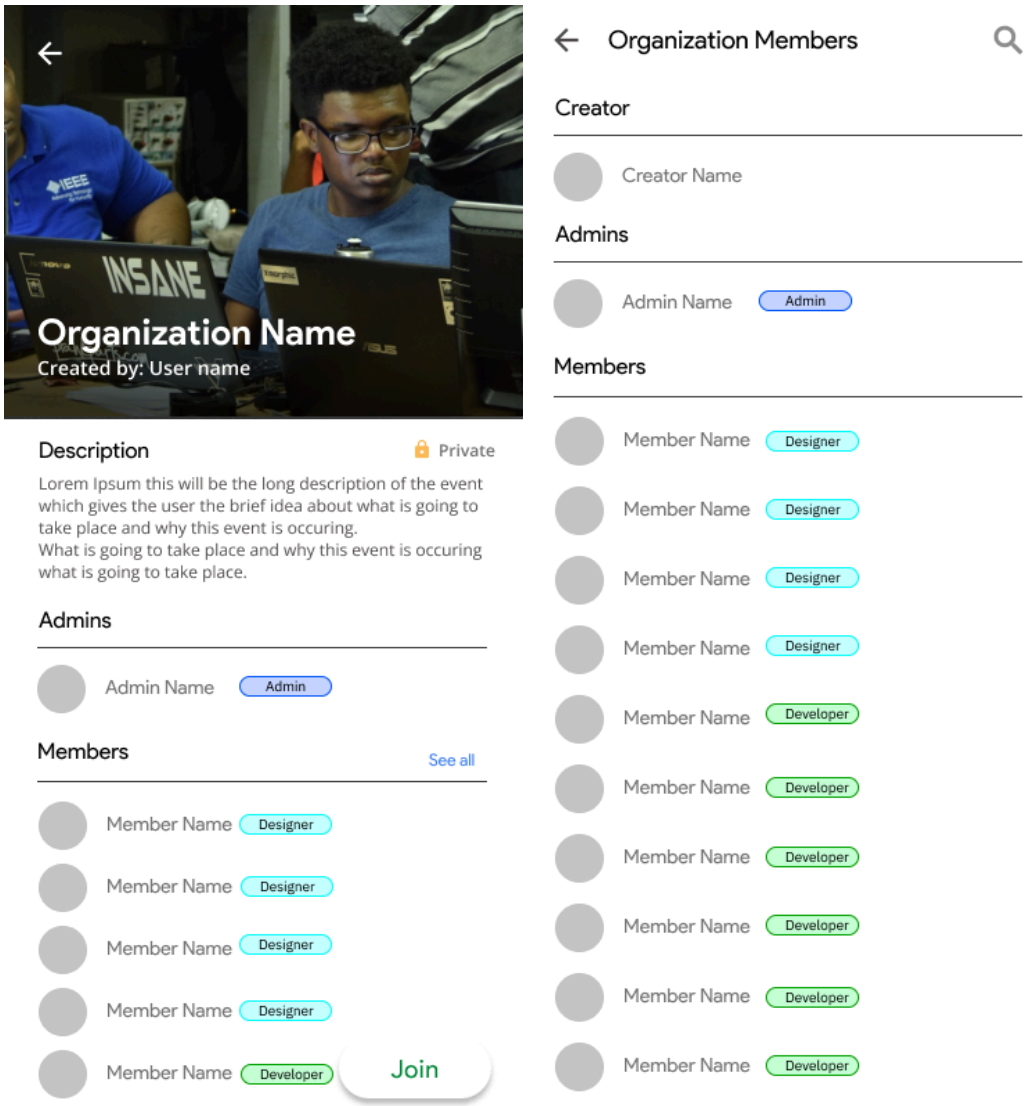


Figure 24 : Tags in Organization member screen

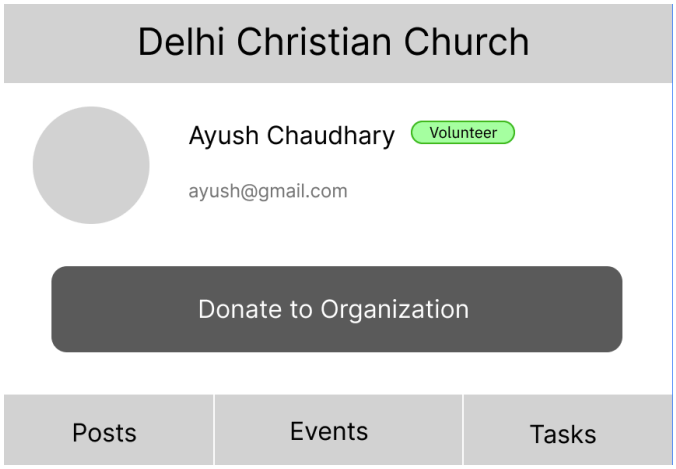


Figure 25 : Tags in member profile screen

Deliverables

This project requires expertise in both backend and front-end development. I'll be dividing my time strategically across these areas:

Backend (30%) :

- **API Optimization**
    - I'll redesign the existing APIs to improve their functionality.
    - I'll optimize the data models to add tag descriptions and validations
    - I'll write test cases and maintain a code coverage of 100%
    - Code refactoring and optimizing existing tag's logics

Frontend (70%) :

- **Designing User Interfaces for Tags in Talawa-Admin and the User Portal:**
    - I will develop comprehensive and user-centric interfaces along with components to facilitate tag functionalities in Talawa application suites.
- **Talawa-Admin Integration**
    - I'll develop the frontend components for Talawa-Admin.
    - I'll integrate the newly designed APIs into the Talawa-Admin frontend.
    - Hierarchical tag structures for improved organization.
    - Unit testing and code coverage above 100%
    - I will diligently update documentation for every feature implemented

Detailed Project Timeline

Pre GSoC	● Work on the existing PRs and issues ● Discuss the deliverables and approach with the mentors ● Work on new features/bug fixes assigned to me
Community bonding period (May 1 - May 26)	● Engage in community bonding activities to foster understanding and alignment on project goals ● Review documentation thoroughly to comprehend project requirements ● Discuss with mentors to clarify doubts and uncertainties about the codebase ● Gain a deep understanding of the internal architecture of the project ● Collaborate with mentors to determine the features to be implemented
Coding Period Starts	
Week 1-3 (May 27 - June 22)	**UI Screen and API design development** ● Restructuring of all pre-existing tables in models ● Add proposed fields in the tags schema ● UI screen development for Tags ● Integrating Tags in admin panel
Week 4-5 (June 23 - July 6)	**Implementing CRUD in tags** ● Developing Tags Screen in admin panel ● Implement CRUD operation of tags admin panel ● Add relevant queries and mutation that would be required ● Write unit tests for functionalities

Week 6-7 (July 7 - July 20)	**Add Child-tags support** ● Develop a design for viewing and creating child-tags ● Add support of child tags in admin-panel ● Create functionality of creating child-tags ● Implement the mutations and queries related to the same in the API with tests
Week 8-9 (July 21 - August 3)	**Implementing automatic tag allotment** ● Add support for automatic tag allotment in Talawa-API ● Integrate automatic tag allotment into the admin panel ● write tests in admin and api for ensuring its working correctly
Week 10-12 (August 4 - Aug 31)	**Implementing manual tag allotment**: ● Develop a modal component for manual tag allotment. ● Write relevant mutations for the allotment. ● Write unit test cases for the same.
Phase-1 Evaluation	
Week 13-15 (September 1 - September 21)	**Tag actions** ● Implement tag actions: assign/unassign tags, archive, and remove people from tags. ● Write tests to ensure the functionality of these tag actions. **Tags in People user portal** ● Implement support for tags on the people screen. ● Revise the types and properties for presenting tags in the table on the people screen. **Tags in People Screen admin portal** ● Will implement tags in People screen of admin panel with required testing

Week 16-17 (September 22 - October 5)	**Filtering in Tags:** ● Develop filtering features for streamlined tag management ● Enable admins/superadmins to filter by criteria like name and date ● Implement filtering options on the Tags screen for efficient tag searching ● Integrate filtering functionalities into the Tag Members screen **Tags in User Settings:** ● Develop the functionality for displaying tags in user settings ● Add necessary mutation and queries for tags display ● Implement tags in user-settings screen
Week 18-20 (October 6 - October 26)	**Email and message broadcasting** ● Work in implementing broadcasting emails and background scheduling configuring **bull** into the project ● Implement the broadcasting emails and message functionality ● Integrate the email and message functionality with the proposed design ● Implement the mutations and queries related to the same in the API with tests ● Write unit tests in admin-panel and api
Week 21 (October 27 - November 2)	● Documentation, Translation and Unit Testing ● Will write documentation for all the implemented features ● Add comments to the codebase for ease of contributions for future contributors ● Managing translation for all the implemented features ● Ensuring 100% test coverages for any feature if left
Final week (November 2 - November 9)	● If required for some adjustments or improvements or to complete any missed feature ● Completing blogs for the whole GSOC journey
Final Evaluation	

References:

This proposal is based upon the existing work done on the following issues in Talawa-API and designs provided by the Talawa-Admin design System:

- Breeze CHMS tags implementation
- Talawa-Admin Design System Figma Link
- Talawa-API Pull Request (for backend context):

  https://github.com/PalisadoesFoundation/talawa-api/pull/1121

  https://github.com/PalisadoesFoundation/talawa-api/pull/1187

  https://github.com/PalisadoesFoundation/talawa-admin/issues/496

  https://github.com/PalisadoesFoundation/talawa-api/issues/1126

  https://github.com/PalisadoesFoundation/talawa-api/pull/1192

  https://github.com/PalisadoesFoundation/talawa-api/pull/1191

Commitments

I plan to devote around 40-45 hours (Or more) per week on the project and can work full time on the weekdays. I'm usually available between **10 AM IST (4:30 AM UTC / 1:30 AM BRT)** to **2 AM IST (8:30 PM UTC /5:30 PM BRT)** for any type of communication/meetings/reports. I have no other commitments during the GSoC and I'm applying **only to Palisadoes** for this Google summer of code 2024.

Post GSoC

I became acquainted with Palisadoes during my first year of college, as some of my peers were selflessly contributing to this remarkable organization. Inspired by their dedication, I too aspire to embark on my own journey with Palisadoes, driven by my passion for contributing to open-source projects.

Thank you