

Final Project Proposal



Session 2023 - 2027

Submitted by:

Maryam Fatima	2023-CS-61
Humna Arshad	2023-CS-63
Saroosh Javed	2023-CS-67

Supervised by:

Sir Nazeef Ul Haq

Course:

CSC200L Data Structures and Algorithms

Department of Computer Science

University of Engineering and Technology

Lahore Pakistan

Table of Contents

Project Overview	5
▪ Description	5
▪ Motivation	5
▪ Objective	5
▪ Features.....	6
▪ Audience.....	6
▪ Technology	7
Use Cases	8
▪ Sign Up.....	8
▪ Login.....	8
▪ Add Contact	8
▪ Create Group.....	9
▪ One-to-one Messaging	9
▪ Group Chatting.....	9
▪ Search Contact.....	10
▪ View Group Description	10
▪ Accept a Request from an Unknown User	10
▪ Exit Group.....	11
▪ Read Message	11
▪ Reply to a Specific Message.....	11
▪ Logout.....	12
▪ Check Suggestions.....	12
▪ View Graph of all Users	12
Data Structures Used:	13
▪ LinkedList:	13
▪ Queue:.....	13
▪ Tree:	13
▪ Graph:	13

▪ Priority Queue:	13
▪ Trie:	13
▪ Hash Table:	13
Implementation	13
▪ Sign Up:	13
▪ Login:.....	14
▪ Adding a New Contact:.....	15
▪ Create Group;.....	16
▪ One-to-one Messaging	19
▪ Group Chatting.....	21
▪ View Group Description:	22
▪ Accept Request from an Unknown User:.....	24
▪ Exit Group:	24
▪ Read Message	25
▪ Replying to a Specific Message	26
▪ Logout.....	28
▪ Search Contact:	28
▪ Check Contact Suggestions:	30
▪ View Users Graph:	30
Classes:	33
▪ User	33
Attributes	38
Methods	39
▪ Message	40
Attributes	42
Methods	43
▪ Group.....	44
Attributes	45
Methods	46
Wireframes:	47

Table of Figure:

Figure 147

Figure 248

Figure 348

Figure 449

Figure 549

Figure 650

Figure 750

Figure 851

Project Overview

▪ Description

This project is a chat website that lets users communicate easily, with features like one-on-one messaging, group chats, and more. It provides an interactive space for chatting while using efficient data structures to manage everything. Instead of using a database, this app stores all data in memory, making it work like a database without actually having one. The project shows how data structures and algorithms can be applied in real-world apps. By creating a chat platform with only frontend technologies and in-memory storage, it demonstrates how data structures like queues, trees, graphs, priority queues, and hash tables can help build fast and scalable web applications.

▪ Motivation

The motivation behind this project is to build a lightweight and scalable chat application that can function effectively without the complexity of a database. By using efficient data structures like Hash Tables, Priority Queues, and Tries, the application aims to deliver high performance and meet modern communication demands while maintaining simplicity in design and implementation.

This chat application is perfect for situations that need quick, in-memory processing, such as small teams, academic projects, or temporary communication setups. It combines practical features with smart use of data structures.

▪ Objective

The main objective of this project is to create a lightweight, efficient, and interactive web-based messaging application that mimics the core functionality of WhatsApp using only HTML, CSS, and JavaScript. The goal is to demonstrate the practical application of data structures and algorithms in the development of communication platform. Key objectives include:

- Implementing features like messaging, group chats, and user presence etc.
- Implementing efficient data management techniques, using in-memory storage with data structures such as queues, trees, graphs, priority queues, and hash tables.
- Enhancing the scalability and performance of the application without using an external database.

■ Features

- **User Authentication:** Users can register by creating an account with a username and phone number. They can also log in with their credentials.
- **One-on-One Messaging:** The app allows users to send and receive direct messages with other users. Messages appear instantly in the chat window, and are delivered in the correct order. The user interface updates without the need for a page refresh.
- **Group Chats:** Users can create, leave or join group chats, where multiple participants can send and receive messages. The group chat feature supports user interaction, and messages sent in group chats are visible to all group members.
- **Message History:** Every conversation, whether it's one-on-one or in a group, maintains a history of messages. Users can scroll through past conversations, and the messages remain accessible as long as the app is running. This feature simulates a messaging history.
- **User Presence:** The application displays whether a user is online or offline, giving the impression of presence tracking. When users log in, they are marked as online, and their status is updated accordingly.
- **Search Functionality:** The search feature enables users to search for specific chats or contacts within their connections. This makes it easier to find past conversations or locate important connections quickly.
- **Scalability:** Although this project does not use a backend database, it is designed to handle a large number of users and messages by making use of efficient data structures. The app simulates scaling by storing all relevant data in memory and ensuring that the user interface responds efficiently.

■ Audience

Students studying data structures and algorithms can see how these concepts are applied in chat applications. This project can be used by instructors to teach how data structures are implemented in practical, scenarios, especially in the context of web development. It serves as an example for discussing the application of algorithms in system design.

■ Technology

The key technologies used in the project:

1. HTML (Hypertext Markup Language)

- HTML is used to structure the layout of the messaging app.
- It includes elements such as message bubbles, user chat lists, input forms, and buttons that allow users to send messages, create groups, and interact with the app.

2. CSS (Cascading Style Sheets)

- CSS is used to style the entire interface of the WhatsApp clone, including the chat window, message input fields, buttons, user lists, and group chat interfaces.
- It ensures the application is responsive and looks good on different screen sizes (desktop, tablet, mobile).

3. JavaScript

- It handles all the interactive features, such as sending/receiving messages, updating user presence (online/offline status).
- It manages all data manipulation, such as storing and retrieving user data and messages in memory using data structures (queues, trees, graphs, priority queues, and hashtables).
- It updates the user interface dynamically without refreshing the page, making the app feel like a live, interactive service.

4. Data Structures and Algorithms

- **Queue:** Used to manage the order of user connections and ensure that they are showed in the correct sequence.
- **Tree:** Used for managing chats and the hierarchy of messages within those chats. Each message is represented as node, having replies as its children.
- **Graph:** Helps represent relationships between users and group chats. Each user is a node, and edges represent connections (such as direct messaging or group membership). The graph structure helps in managing these connections.
- **Priority Queue:** Helps prioritize important messages, ensuring that recent messages are processed first.

- **Hash table:** Efficiently stores groups information (like members, creator, and settings)

5. Local Storage (Browser Memory)

- Local storage is used in this project to store user data, including usernames, chat history, and group memberships.
- It simulates the database functionality, allowing the data to persist even when the page is refreshed.

6. Event-Driven Programming (JavaScript)

- JavaScript in this project listens for user interactions like sending messages, creating groups and more.
- When a user interacts with the app, such as clicking a button or typing a message, the event is captured, and the relative action is performed.

Use Cases

▪ Sign Up

Actor	User
Goal	Register to the system
Steps	<ul style="list-style-type: none"> • User enters phone number and select a name • System verifies that phone number is unique • User is registered
Outcome	User is signed up successfully and registered to the system or an error message is shown if credentials are not unique.

▪ Login

Actor	User
Goal	Access the system by logging in
Steps	<ul style="list-style-type: none"> • User enters username and phone number • System verifies credentials • User gains access
Outcome	User is logged in successfully or an error message is shown if credentials are incorrect.

▪ Add Contact

Actor	User
Goal	Make connection with other users on system

Steps	<ul style="list-style-type: none"> • User enters phone number of another user • Set a name for contact • Contact is added
Outcome	Contact is added successfully to the contacts queue of user or an error message is shown if other user is not registered.

▪ Create Group

Actor	User
Goal	Make connection with multiple users at once
Steps	<ul style="list-style-type: none"> • User selects other users from contacts as group members • User enters name of group.
Outcome	Group is created successfully and shown to all members

▪ One-to-one Messaging

Actor	User (Sender or Receiver)
Goal	To send and receive messages between two users in a one-on-one chat.
Steps	<ul style="list-style-type: none"> • User opens the messaging app. • User selects a contact from their contact list. • User types a message. • User clicks "Send." • The system sends the message to User 2. • User 2 opens the app. • User 2 sees the message from User and replies.
Outcome	The message is successfully sent and received by both users, enabling continuous communication.

▪ Group Chatting

Actor	Multiple Users (Group Members)
Goal	To send and receive messages within a group chat.
Steps	<ul style="list-style-type: none"> • User 1 opens the messaging app. • User 1 selects the group from their chat list. • User 1 types a message. • User 1 clicks "Send." • The system sends the message to all group members. • Each group member receives the message. • Group members reply with their messages. • All replies are displayed for all group members to see.
Outcome	Messages are successfully sent to and received by all members of the group, enabling group communication.

▪ Search Contact

Actor	User
Goal	To search for a contact in the contact list.
Steps	<ul style="list-style-type: none">• User clicks on the search bar.• User types the name or part of the name of the contact they want to find.• The system filters and displays matching contacts.• User selects the desired contact from the search results.• The contact's chat is displayed.
Outcome	The user successfully finds and selects the desired contact, allowing them to continue a conversation.

▪ View Group Description

Actor	Multiple Users (Group Members)
Goal	To view the description of a group chat.
Steps	<ul style="list-style-type: none">• User opens the group chat.• User clicks on the group name or settings icon to access group information.
Outcome	The group description is successfully visible to the user.

▪ Accept a Request from an Unknown User

Actor	User
Goal	To accept a request or connection from an unknown user (e.g., friend request or message request).
Steps	<ul style="list-style-type: none">• The user receives a request from an unknown user (e.g., a message request, friend request, or group join request).• User views the request in the app.• User has the option to either accept or ignore the request.• User clicks on the "Add Contact" button to accept the request.• User sets a name for new contact.• The system verifies the action and adds the unknown user to the user's contact list and grants access to the conversation/group.• The user can now send messages or interact with the new contact as usual.
Outcome	The user successfully accepts the request, and the unknown user is added to their contacts, allowing for future interactions.

▪ Exit Group

Actor	User (Group Member)
Goal	To exit or leave a group chat.
Steps	<ul style="list-style-type: none">• User opens the group chat they want to exit.• User clicks on the group name or settings icon to access group options.• User selects the option to "Exit Group."• The system removes the user from the group.• The user is no longer able to send messages or receive from that group.
Outcome	The user successfully exits the group, and they are removed from the group chat.

▪ Read Message

Actor	User
Goal	To read a received message in a one-on-one or group chat.
Steps	<ul style="list-style-type: none">• User selects the desired conversation or group chat to open.• The system loads the chat history, displaying all previous messages.• The system marks the message as "seen" and the status is updated for both the sender and receiver.
Outcome	The user successfully reads message, and the message status is updated to confirm that it has been read.

▪ Reply to a Specific Message

Actor	User
Goal	To reply to a specific message in a one-on-one or group chat.
Steps	<ul style="list-style-type: none">• User opens the messaging app and selects the desired conversation or group chat.• User locates the specific message they want to reply to.• User clicks on the three dots (options menu) next to the message.• User selects the "Reply" option or icon from the dropdown menu.• The system opens the reply input field.• User types their reply in the input field.• User clicks "Send" to send the reply.• The system displays the reply in the chat, with the selected message visible above to the reply.

Outcome	The user successfully replies to a specific message, and the reply is displayed in the chat with the referenced message.
---------	--

▪ Logout

Actor	User
Goal	To log out of the system.
Steps	<ul style="list-style-type: none"> • User navigates to the profile (by clicking on their profile picture). • User selects the "Log Out" option from the settings menu. • The system logs the user out and redirects them to the login screen. • The user is no longer able to access their messages or account without logging in again.
Outcome	The user successfully logs out of the app.

▪ Check Suggestions

Actor	User
Goal	To view suggestions of users who are connected to their contacts.
Steps	<ul style="list-style-type: none"> • User navigates to the "Suggestions" tab (by clicking on suggestion button) • Suggestions will be displayed. • On clicking on any user add to contact page will be displayed with user phone number filled.
Outcome	The user is provided with meaningful suggestions of people connected to their network.

▪ View Graph of all Users

Actor	User
Goal	User can view graph of all users and how they are connected to each other
Steps	<ul style="list-style-type: none"> • User navigates to the Graph visualization page (by clicking on graph display button) • Full user connected graph will be displayed.
Outcome	The user is provided with the connection between all users.

Data Structures Used

- **LinkedList:**

Used to make dynamic custom data structures like queue and stack.

- **Queue:**

Used to store and maintain a contact list to preserve the order in which they were added.

- **Tree:**

Used to manage message threads. Messages are represented as nodes, and replies are child nodes, forming a tree-like structure that helps in finding out that a message is a reply of another specific message.

- **Graph:**

Used to represent relationships between users (e.g., friendships, group memberships). Adjacency matrix is used to recommend friends and explore connections.

- **Priority Queue:**

Used to store messages with the time as their priority that helps to display messages in order relative to the time at which they were sent.

- **Trie:**

Used to efficiently store and retrieve contact names or group names for the search functionality.

- **Hash Table:**

Used to store group information, making it easy and fast to retrieve details about any group. Hash table assigned a unique ID to each group. This key helps in quickly finding the associated data without having to search through all the groups one by one.

Implementation

- **Sign Up:**

To sign up on the system, user will open the signup form, he will enter his phone number and set a name for his personal account. By clicking the signup button, all fields will be saved and validation (e.g., Phone number must contain exactly 11 digits) will be checked by following function. The appropriate message will be shown indicating that if the user has currently registered or not.

```
function signup() {
    const username = document.getElementById('signup-username').value;
    const phone = document.getElementById('signup-phone').value;
    if (!/^\d{11}$/.test(phone)) {
        alert('Phone number must contain exactly 11 digits.');
```

```
        return;
    }

    const user = users.find(u => u.phoneNumber === phone);
    if(user) {
        alert('Phone number already exists.');
```

```
        return;
    }

    createUser(username, phone);
}
```

After validation, createUser() will create a new object of user and it will be save to the list and local storage.

■ **Login:**

To log in and access the system, user will open login form and enter his credentials that will be verified from saved data. After verification, the user object will be found from the list of all users and marked as currentUser in storage (this is done to perform the future operations on current user). The user’s online status is set to “Online” indicating that user is currently logged in. After that user granted access to the system by redirecting him to a new page “index”;

```
function login() {

    const username = document.getElementById('login-username').value;
    const phone = document.getElementById('login-phone').value;

    if (username && phone) {
```

```

const user = users.find(u => u && u.username === username && u.phoneNumber ===
phone);

if (user) {

    currentUser = FindUser(user.phoneNumber);

    localStorage.setItem('currentUser', JSON.stringify(currentUser));

    alert('Login successful!');

    FindUser(phone).updateOnlineStatus(true);

    window.location.href = 'index.html';

} else {

    alert('User not found or incorrect credentials.');
```

```

}

} else {

    alert('Please fill in all fields.');
```

```

}

}

```

▪ Adding a New Contact:

To add a new contact, user navigates to a “New Contact” option that displays a form for adding the credentials (phone number and the name for new contact). Then it is checked that contact does not exist already in contact list, it is added to the contact list of current user. For other user, it is checked that if connection is not from both sides, a request is sent to that user. If number entered by user is not registered on the system, an error message is displayed.

```

function AddContactToAUser() {

    const name = document.getElementById('contact-name').value;

    const phone = document.getElementById('contact-phone').value;

    let count = 0;

    for (let i = 0; i < users.length; i++) {

        if (users[i].phoneNumber === phone) {

```

```

    currentUser = FindUser(currentUser.phoneNumber);

    users[i].nickname = name;

    if(currentUser.addContact(users[i])) {
        saveUsersToStorage();
        if (currentUser.checkContactExistence(users[i].phoneNumber)) {
            if (!(users[i].checkContactExistence(currentUser.phoneNumber))) {
                users[i].addRequest(currentUser.phoneNumber);
                saveUsersToStorage();
            }
        }
    }
    count++;
}

}

if(count === 0)
{
    alert('not found');
}
}

```

■ **Create Group:**

To create a new group, user navigates to a “New Group” option that displays a form for adding the group name and selecting users from contact list as group members. There should be at least one member for creating a group. After selecting contacts user clicks on “Create Group” button that adds selected users to group. The new group is added to the contact list of all members.

```

function renderContactsForGroupCreation() {
    let group = new Group();
    const contactsSection = document.getElementById('contacts-section');

```



```

contactsSection.innerHTML = "";

const size = FindUser(currentUser.phoneNumber).getContacts().size;

let current = FindUser(currentUser.phoneNumber).getContacts().peek();

for (let i = 0; i < size; i++) {
  if(current)
  {
    if (current.data.username) {
      const contact = current.data;

      console.log(contact)

      const contactDiv = document.createElement('div');

      contactDiv.classList.add('contact');

      contactDiv.innerHTML = `

        <div class="content-message-image" style="background-color: var(--slate-100);
border: 2px solid var(--emerald-600); border-radius: 50%; display: flex; justify-content:
center; align-items: center; font-size: 20px; color: var(--emerald-600); font-weight: bold; text-
transform: uppercase;">

          ${contact.username.charAt(0).toUpperCase()}

        </div>

        <div class="contact-details">

          <span>${contact.username}</span>

          <small>${contact.phoneNumber}</small>

        </div>

        <input type="checkbox" data-phone-number="${contact.phoneNumber}" />

      `;

      const checkbox = contactDiv.querySelector('input[type="checkbox"]');

      checkbox.addEventListener('change', (e) => {
        if (e.target.checked) {

```

```

        group.addMember(contact.phoneNumber);
    }
    // } else {
    //     group.delete(contact.phoneNumber);
    // }
});

    contactsSection.appendChild(contactDiv);
}
    current = current.next;
}
}

document.getElementById('createGroupBtn').addEventListener('click', () => {
    const groupName = document.getElementById('groupName').value.trim();
    if (!groupName) {
        alert('Please enter a valid group name');
        return;
    }
    if (group.size === 0) {
        alert('Please select at least one contact');
        return;
    }
    group.setName(groupName);
    group.creator = currentUser.username;
    group.addMember(currentUser.phoneNumber);

```

```

FindUser(currentUser.phoneNumber).addContact(group);
FindUser(currentUser.phoneNumber).addGroup(group);

console.log(group.members[0]);
for (let i = 0; i < group.members.length; i++) {
    console.log(group.members[i] instanceof User);

    if (group.members[i] !== currentUser.phoneNumber) {
        FindUser(group.members[i]).addGroup(group);
        FindUser(group.members[i]).addContact(group);
    }

}

saveUsersToStorage();

console.log('Group Name:', groupName);
console.log('Group:', group);
console.log(group.members[0])
console.log(FindUser(currentUser.phoneNumber).getContacts());
alert("Group Created Successfully");
window.location.href = "index.html";

});
}

```

■ One-to-one Messaging:

For one-to-one messaging, user clicks a contact from contact list and chat box is visible. Chat box contains a form for typing a message followed by a 'Send' button. After typing the message, user clicks that button and sendMessage() function is called. This function gets the receiver information from chat box and message typed by user. For one-to-one messaging the if block with receiverPhone executes. Both current user and receivers are added to receivers

list (this list is used later to find out in which chat message will be visible). Then a new message object is created and added to a root of tree having no children because a message that has been sent currently does not have any child message or a reply. This messageTree is added to the messages of both users.

```
function sendMessage()
{
    const text = document.getElementById('message-text').value;

    let newMsg = null;

    let group = null;

    let receivers = [];

    const receiverPhone = document.getElementById('conversation-
template').dataset.phoneNumber;

    if(receiverPhone)
    {
        receivers.push(receiverPhone);

        receivers.push(currentUser.phoneNumber);

        newMsg = new Message(text, currentUser.phoneNumber, receiverPhone);

        let msgTree = new TreeNode(newMsg);

        currentUser.addMsg(msgTree);

        FindUser(receiverPhone).addMsg(msgTree);
    }

    saveUsersToStorage();

    initializeDropdown();

    if(receiverPhone)
    {
        if(newMsg)
        {
```

```

        addMessageReply(newMsg, receivers, receiverPhone, null);
    }
    loadMessages(receiverPhone, null);
}
document.getElementById('message-text').value="";
}

```

■ **Group Chatting:**

For group chatting, user selects a contact from contact list and chat box is visible. After typing the message, user clicks 'Send' button and `sendMessage()` function is called. This function gets the group information from chat box and message typed by user. For group messaging the `if(group)` block with `if(group)` executes which means that a group object is passed to the function. All group members are added to receivers list because message sent in a group is visible to all members. Then a new messageTree is added to the messages of currently opened group and messages are reloaded to show new message.

```

receivers = getGroupMembersPhoneNumber(group);
newMsg = new Message(text, currentUser.phoneNumber, null);
let msgTree = new TreeNode(newMsg);

if(group)
{
    receivers = getGroupMembersPhoneNumber(group);
    newMsg = new Message(text, currentUser.phoneNumber, null);
    let msgTree = new TreeNode(newMsg);

    for(let i=0; i < receivers.length; i++)
    {
        let user = FindUser(receivers[i]);
    }
}

```

```

        group = user.getGroups().getGroup(groupId);
        Object.setPrototypeOf(group, Group.prototype);
        Object.setPrototypeOf(group.messages, PriorityQueue.prototype);
        console.log('is pq.prototype?', Object.getPrototypeOf(group.messages) ===
PriorityQueue.prototype);

        console.log(group === user.getGroups().getGroup(groupId));
        group.addMsg(msgTree);
        console.log("Current Group:", typeof group.messages);
    }

    if(newMsg)
    {
        addMessageReply(newMsg, receivers, null, group);
    }

    loadMessages(null, group);
}

```

▪ **View Group Description:**

To view the group description, user selects a group and click on the information/settings

Button and following function display the relevant information:

```

function showGroupDescription(group) {
    const groupTopDiv = document.getElementById('conversationTop');
    const groupMainDiv = document.getElementById('conversationMain');
    const groupFormDiv = document.getElementById('conversationForm');
    groupTopDiv.style.display = 'none';
    groupMainDiv.style.display = 'none';
}

```

```
groupFormDiv.style.display = 'none';
```

```
const groupHeaderDiv = document.getElementById('headerGrp');
```

```
const groupContentDiv = document.getElementById('grpContainer');
```

```
groupHeaderDiv.style.display = 'flex';
```

```
groupContentDiv.style.display = 'block';
```

```
const grpName = document.getElementById('grpName');
```

```
grpName.innerHTML = "";
```

```
grpName.textContent = group.groupName;
```

```
const grpDate = document.getElementById('date');
```

```
const grpCreator = document.getElementById('creator');
```

```
grpDate.innerHTML = "";
```

```
grpCreator.innerHTML = "";
```

```
grpDate.textContent = "Created on: " + group.date;
```

```
grpCreator.textContent = "Created By: " + group.creator;
```

```
const listItem = document.getElementById('list');
```

```
listItem.innerHTML = "";
```

```
for (let i = 0; i < group.members.length; i++) {
```

```
    if (group.members[i] === currentUser.phoneNumber) {
```

```
        listItem.innerHTML += `
```

```
            <li>You</li>
```

```
        `;
```

```
    }
```

```
    else if (currentUser.checkContactExistence(group.members[i])) {
```

```

        console.log(FindUser(group.members[i]).username);

        listItem.innerHTML += `
            <li>${FindUser(group.members[i]).username}</li>
        `;
    }
    else {
        const uniqueID = `btn-${i}`;
        listItem.innerHTML += `
            <li>${group.members[i]}<button id="${uniqueID}" class="add-contact-
button">Add to Contacts</button></li>
        `;
    }
}
}
}

```

▪ **Accept Request from an Unknown User:**

To accept a request from an unknown user, user navigates to ‘Contact Requests’ option where all requests are visible. By clicking on a request, user gets the option to add them as a contact or ignore. If user selects to add, he is directed to add contacts form, user sets a name for new contact and clicks on add contact button that successfully adds the request to contact list of users.

▪ **Exit Group:**

For leaving a group, there is a button ‘Exit Group’ in the description of every group. When user clicks to that button and following function removes him from group members, and also removes the group from contact list of users.

```

function exitGroup(group) {
    for (let i = 0; i < group.members.length; i++) {
        if (group.members[i]) {

```



```

FindUser(group.members[i]).getGroups().getGroup(group.grpID).removeMember(currentUser.phoneNumber);

    saveUsersToStorage();

}

}

currentUser.getGroups().removeGroup(group.grpID);

currentUser.removeContact(group.grpID);

saveUsersToStorage();

renderUsers();

}

```

▪ **Read Message:**

To read a message, user selects the desired conversation or group chat. On selecting, readMessages() is called and traverse through messages, for one-to-one messaging, it marks all messages of that conversation (that are received by currentUser) as read and update their 'isSeen' status to True. In case of group chat, it marks those messages that are received by currentUser and sent by any of the group members. The status is updated in message list of all users included in the conversation.

```

function markMessageAsSeen(user)
{
    let currentNode = currentUser.messages.front;
    while (currentNode) {
        let messageObject = currentNode.data.root;
        Object.setPrototypeOf(messageObject, Message.prototype);
        if (
            (messageObject.receiver === currentUser.phoneNumber && messageObject.sender
            === user.phoneNumber)
        ) {
            messageObject.updateStatus(true);

```

```

    }
    currentNode = currentNode.next;
}

currentNode = user.messages.front;
while (currentNode) {
    let messageObject = currentNode.data.root;

    Object.setPrototypeOf(messageObject, Message.prototype);
    if (
        (messageObject.receiver == currentUser.phoneNumber && messageObject.sender ==
user.phoneNumber)
    ) {
        messageObject.updateStatus(true);
    }
    currentNode = currentNode.next;
}
saveUsersToStorage();
}

```

▪ **Replying to a Specific Message:**

To reply to a specific message, user locates a specific message in a chat and selects the ‘reply’ icon from options of that message. On clicking this message id is saved in document as ‘data-saved-message-id’. When user types a reply the following function adds the message to the child of selected message as a reply. After adding reply, saved message id is set to null.

```

function addMessageReply(newMsg, receivers, person, group)
{
    let replybutton = document.querySelector('.reply-button');
    let parentMessageId = null;

```

```

if(replybutton)
{
    parentMessageId = replybutton.getAttribute('data-saved-message-id');
    document.querySelector('.reply-button').setAttribute('data-saved-message-id', null);
}
if(parentMessageId)
{
    for(let i=0; i < receivers.length; i++)
    {
        let originalMessageForReceiver = null;
        if(person)
        {
            originalMessageForReceiver = findMessageByIdForReceiver(parentMessageId,
FindUser(receivers[i]), null);
        }
        else if(group)
        {
            originalMessageForReceiver = findMessageByIdForReceiver(parentMessageId, null,
group);
        }
        Object.setPrototypeOf(originalMessageForReceiver, TreeNode.prototype);
        if(originalMessageForReceiver)
        {
            originalMessageForReceiver.addChild(newMsg);
        }
        else
        {
            console.log("Message ID not found.");
        }
    }
}

```

```

        return;
    }
}
}
}
}

```

■ Logout:

To log out of the system, user clicks on profile image and selects ‘Logout’ option. On clicking the following piece of code redirects user to login page.

```

<li><a href="login.html" id="logout-link"><i class="ri-logout-box-line"></i>
Logout</a></li>

```

■ Search Contact:

For searching a contact or a group, user types the name or character of name for that contact or group and it is filtered from contact list. Prefix Trie is used in the following function to handle the string search as follows:

```

function ContactSearch()
{
    let input = document.getElementById('search-input');
    let query = input.value.trim();
    let tier = QueueToTier();
    let result = tier.SearchWithStart(query);
    let SearchedObjects=[];
    result.display();
    while(!result.isEmpty())
    {
        let user = FindContactsByName(result.dequeue());
        SearchedObjects.push(user);
    }
}

```

```

    console.log(SearchedObjects);
    renderSearchResults(SearchedObjects, query);
}
function QueueToTier()
{
    let tier = new Tier();
    let contacts = currentUser.contacts;
    for (let contact of contacts) {
        if(contact.data.nickname)
        {
            tier.Insert(contact.data.nickname);
        }
        else if(contact.data.groupName){
            tier.Insert(contact.data.groupName);
        }

    }
    return tier;

}
function FindContactsByName(name)
{
    let matchingContacts = [];
    let contacts = currentUser.contacts;
    for (let contact of contacts){
        if (contact.data.nickname === name || contact.data.groupName === name)
        {

```

```

        matchingContacts.push(contact);
    }
}
console.log(matchingContacts);
return matchingContacts;
}

```

▪ **Check Contact Suggestions:**

For checking suggestion user click on suggestions button and following function give user suggestions of users who are connected to their contacts.

```

function FindSuggestion()
{
    let graph = UserGraph();
    let suggestions = graph.suggestConnections(currentUser.phoneNumber);
    let suggestedContacts = [];
    for(let suggestion of suggestions)
    {
        let user = FindUser(suggestion)
        suggestedContacts.push(user);
    }
    console.log("suggestions ",suggestedContacts);
    return suggestedContacts;
}

```

▪ **View Users Graph:**

Users can visually view a graph representing all users and their connections, allowing them to explore relationships between individuals.

```
function UserGraph()
{
    let nodes = GetAllNodes();
    let adjacencyMatrix = GetAdjacencyMatrix();
    let graph = new Graph(nodes,adjacencyMatrix);
    return graph;
}

function GetAllNodes()
{
    let nodes = [];
    for(i=0;i<users.length;i++)
    {
        if (users[i] && users[i].phoneNumber) {
            let node = new GraphNode(users[i].phoneNumber, users[i].username);
            nodes.push(node);
        } else {
            console.error("Invalid user data for index " + i);
        }
    }
    return nodes;
}

function GetAdjacencyMatrix() {
    const size = users.length;
    const adjacencyMatrix = Array.from({ length: size }, () => Array(size).fill(0));
```

```

// Map usernames to indices
const usernameToIndex = {};
for (let i = 0; i < size; i++) {
    usernameToIndex[users[i].username] = i;
}

// adjacency matrix
for (let i = 0; i < size; i++) {
    const user = users[i];
    const contacts = user.contacts;
    for(let contact of contacts) {
        if(contact.data.username)
        {
            const contactIndex = usernameToIndex[contact.data.username];
            if (contactIndex !== undefined) {
                //after finding mark connection as 1
                adjacencyMatrix[i][contactIndex] = 1;
            }
        }
    }
}

return adjacencyMatrix;
}

```


Classes

▪ User

The User class represents a user in a chat application. It has attributes like username, phoneNumber, nickname and methods like sending messages, managing contacts, handling group chats, and more.

```
class User{

    constructor(username, phoneNumber, nickname, lastSeen = new Date().toLocaleString(),
contacts = new Queue(), messages = new PriorityQueue(), groups = new HashTable(), requests
= new Queue())

    {

        this.username = username;

        this.nickname = nickname;

        this.phoneNumber = phoneNumber;

        this.lastSeen = lastSeen;

        this.contacts = contacts;

        this.isOnline = false;

        this.messages = messages;

        this.groups = groups;

        this.requests = requests;

    }

    addMsg(newMsg)

    {

        //console.log('time int: ', newMsg.root.getTimeInt());

        this.messages.enqueue(newMsg, newMsg.root.getTimeInt());

    }

    updateOnlineStatus(status)
```

```
{
  this.isOnline = status;
}
addContact(newContact)
{
  let count = 0;
  let current = this.contacts.peek();
  console.log("hillo");

  if (newContact.phoneNumber === this.phoneNumber) {
    alert("Cannot add yourself to the Contacts");
    count++;
  }

  while(current) {
    console.log("hillo");
    if (newContact instanceof User) {
      if (newContact.phoneNumber === current.data.phoneNumber) {
        alert("Contact already exists.");
        count++;
        break;
      }
    }

    current = current.next;
  }
}
```

```

    if(count === 0) {
        this.contacts.enqueue(newContact);
        alert ("Contact successfully.");
        return true;
    }
    return false;
}

removeContact(id) {
    if (!this.contacts.peek()) {
        alert("No contacts to remove.");
        return;
    }

    let prev = null;
    let current = this.contacts.peek();

    while (current) {
        if (current.data.grpID === id) {
            if (prev === null) {
                // If it's the first node, adjust the head of the list
                this.contacts.dequeue();
            } else {
                // Remove the current node by skipping it in the chain
                prev.next = current.next;
            }

            alert("Contact removed successfully.");
        }
    }
}

```

```

        return;
    }

    // Move to the next node
    prev = current;
    current = current.next;
}

alert("Contact not found.");
}

addRequest(newRequest)
{
    let count = 0;
    let current = this.requests.front;
    if (newRequest === this.phoneNumber) {
        alert("Cannot add yourself to the Requests");
        count++;
    }
    while(current) {
        console.log(newRequest)
        console.log(current.data)
        if (newRequest == current.data) {
            alert("Request already exists.");
            count++;
            break;
        }
    }
}

```

```
        current = current.next;
    }

    if(count === 0) {
        this.requests.enqueue(newRequest);
        alert ("Request Added successfully.");
        return true;
    }
    return false;
}
updateLastSeen(seen)
{
    this.lastSeen = seen;
}
getContacts() {
    return this.contacts;
}
getGroups() {
    return this.groups;
}
addGroup(newGroup) {
    this.groups.insert(newGroup);
    alert ("Group added successfully.");
}
getRequests() {
    return this.requests;
}
```

```
checkContactExistence(contact) {  
    let current = this.getContacts().peek();  
  
    let count = 0;  
    while (current != null) {  
        if (current.data.phoneNumber === contact) {  
            count ++;  
            break;  
        }  
        current = current.next;  
    }  
  
    return count !== 0;  
}  
}
```

Attributes

username

The name used by the user to log in and identify themselves.

phoneNumber

The user's phone number, which is unique to each user.

nickname

A display name or alternative name for the user.

lastSeen

Tracks the last time the user was online. Default is the current date and time.

contacts

A queue that stores the user's contacts (other users).

isOnline

A status flag to indicate if the user is currently online or logged in.

messages

A priority queue that stores the user's messages that are ordered by time (priority).

groups

A hash table that stores the groups the user is part of.

requests

A queue that stores connection or friend requests sent to the user.

Methods**1. addMsg(newMsg)**

- Adds a new message to the user's messages queue.
- The messages are prioritized based on their timestamp (time of creation).

2. updateOnlineStatus(status)

- Updates the user's online status (true for online, false for offline).

3. addContact(newContact)

- Adds a new contact to the user's contacts queue.
- It Ensures that:
 - The user cannot add themselves as a contact.
 - Duplicate contacts are not allowed.
- Alerts the user if the contact is successfully added or already exists.

4. removeContact(id)

- Removes a contact from the user's contacts by matching a unique ID (like grpID).
- Alerts the user if the contact is removed or not found.

5. addRequest(newRequest)

- Adds a new friend request to the user's requests queue.
- Ensures that:
 - The user cannot send a request to themselves.
 - Duplicate requests are not allowed.

- Alerts the user if the request is successfully added or already exists.

6. updateLastSeen(seen)

- Updates the lastSeen property with a given time.

7. getContacts()

- Retrieves the user's list of contacts from the contacts queue.

8. getGroups()

- Retrieves the user's list of groups from the groups hash table.

9. addGroup(newGroup)

- Adds a new group to the user's groups hash table.
- Alerts the user when the group is successfully added.

10. getRequests()

- Retrieves the list of friend requests from the requests queue.

11. checkContactExistence(contact)

- Checks if a specific contact (by phone number) already exists in the user's contact list.
- Returns true if the contact exists, otherwise false.

■ **Message**

The Message class represents a message in chat application, with attributes such as the text, sender, receiver, timestamp, and replies.

```
class Message{
```

```
    constructor(text, sender, receiver, isSeen = false, time = new Date().toLocaleString())
```

```
{
```



```

    this.id = Math.floor(Math.random() * 1000);

    this.text = text;

    this.sender = sender;

    this.receiver = receiver;

    this.isSeen = isSeen;

    this.time = time;

    this.replies = [];
}

getTimeInt()
{

    // First, extract the parts of the time using a regular expression
    let regex = /(\d{1,2}):(\d{2}):(\d{2})\s(AM|PM)/;

    let matches = this.time.match(regex);

    if (matches) {
        let hour = parseInt(matches[1], 10);

        let minute = parseInt(matches[2], 10);

        let second = parseInt(matches[3], 10);

        let period = matches[4];

        // Convert the hour to 24-hour format if it's PM
        if (period === "PM" && hour !== 12) {
            hour += 12; // Convert PM hour to 24-hour format
        } else if (period === "AM" && hour === 12) {
            hour = 0; // 12 AM is 00 hours in 24-hour format
        }
    }
}

```

```

        // Now combine hour, minute, and second into a single integer
        let timeInt = hour * 10000 + minute * 100 + second;

        return timeInt; // For "10:30:45 PM", this will print 223045
    } else {
        console.log("Invalid time format");
        return -1;
    }

}

updateStatus(status)
{
    this.isSeen = status;
}

addReply(replyMessage) {
    this.replies.push(replyMessage);
}

// Checking if the message has any replies
hasReplies() {
    return this.replies.length > 0;
}
}

```

Attributes

id

A unique identifier for each message, randomly generated between 0 and 999.

Text

The content of the message (the actual message text).

sender

The user who sent the message. It is an instance of the User class.

receiver

The user who is receiving the message. Also, an instance of the User class.

isSeen

A boolean variable indicating whether the message has been seen by the receiver. Default is false.

time

The timestamp of when the message was sent.

Methods

1. getTimeInt()

- This method converts the time into a numerical representation of the time, making it easier to compare and sort messages based on their timestamps.
- The time format is in **12-hour format (HH:MM:SS AM/PM)**. The method:
 - Extracts the hour, minute, second, and AM/PM period from the time.
 - Converts the time to 24-hour format if necessary.
 - Combines the hours, minutes, and seconds into a single integer, which helps in sorting messages by time.
- Example output: For "10:30:45 PM", it returns 223045.

2. updateStatus(status)

- This method updates the isSeen attribute of the message. If the message is read by the receiver, this method is called with parameter true to mark it as seen.

3. addReply(replyMessage)

- This method adds a reply to the replies. When a user replies to a message, this function is used to store the response as a reply as a child of the original message.

4. hasReplies()

- This method checks if the message has any replies by returning a Boolean based on whether the replies array has any items.

■ Group

The Group class represents a group chat.

```
class Group {  
  
    constructor(groupName = "Group", creator, date = new Date()) {  
  
        this.grpID = Math.floor(Math.random() * 1000);  
  
        this.groupName = groupName;  
  
        this.members = [];  
  
        this.messages = new PriorityQueue();  
  
        this.size = 0;  
  
        this.creator = creator;  
  
        this.date = date;  
  
    }  
  
    addMember(user) {  
  
        if (!this.members.includes(user)) {  
  
            this.members.push(user);  
  
            this.size++;  
  
        }  
  
    }  
  
    addMsg(newMsg)
```

```

    {
        console.log('is      pq.prototype?',      Object.getPrototypeOf(this.messages)      ===
PriorityQueue.prototype);

        this.messages.enqueue(newMsg, newMsg.root.getTimeInt());
    }

    setName(name) {
        this.groupName = name;
    }

    removeMember(user) {
        const index = this.members.indexOf(user);
        console.log(index);
        if (index !== -1) {
            this.members.splice(index, 1);
            this.size--;
            console.log(this.members);
            console.log(`${user} has been removed from the group.`);
            return true;
        } else {
            console.log(`${user} is not a member of the group.`);
            return false;
        }
    }
}

```

Attributes

grpID

A unique identifier for the group.

groupName

The name of the group (e.g., "Friends Group").

members

An array that stores the users who are part of the group.

messages

A priority queue that stores the messages sent within the group

size

The total number of members in the group.

creator

The user who created the group.

date

The date when the group was created.

Methods**1. addMember(user)**

- This method adds a new member to the group if they are not already a member. It checks if the user is already in the members and adds them if not. The size of the group is updated when a new member is added.

2. addMsg(newMsg)

- This method adds a new message to the group's messages queue. The message is enqueued with a priority based on its timestamp (`newMsg.root.getTimeInt()`), which helps in sorting the messages in the group by time.

3. setName(name)

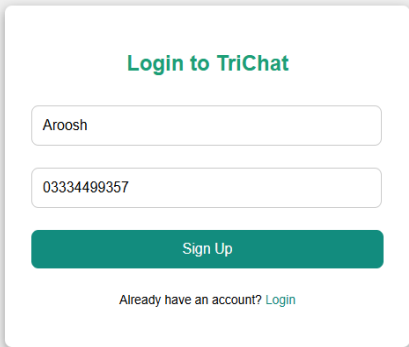
- This method allows the group name to be changed. It updates the `groupName` property to the new value.

4. removeMember(user)

- This method removes a user from the group. It searches for the user in the members array. If found, the user is removed and the size is decremented. If the user is not a member, the function logs an error message.

Wireframes

▪ Sign Up:



The wireframe shows a 'Sign Up' form for 'TriChat'. It features a title 'Login to TriChat' in teal. Below the title are two input fields: the first contains the text 'Aroosh' and the second contains '03334499357'. A teal 'Sign Up' button is positioned below the inputs. At the bottom, there is a link that reads 'Already have an account? Login'.

Figure 1

- **Login:**

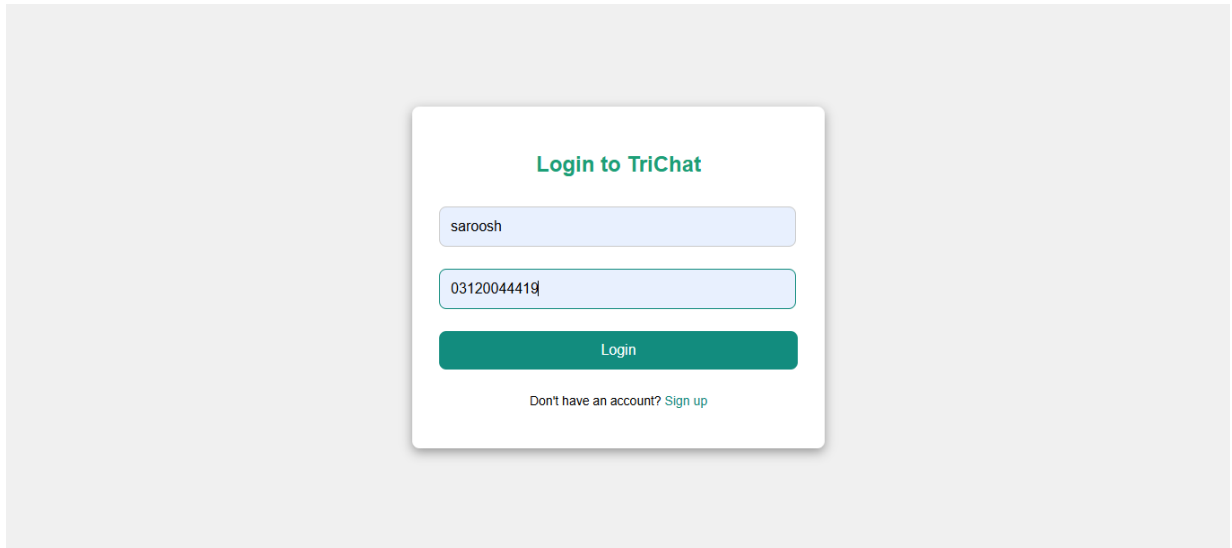


Figure 2

- **Add new contact:**

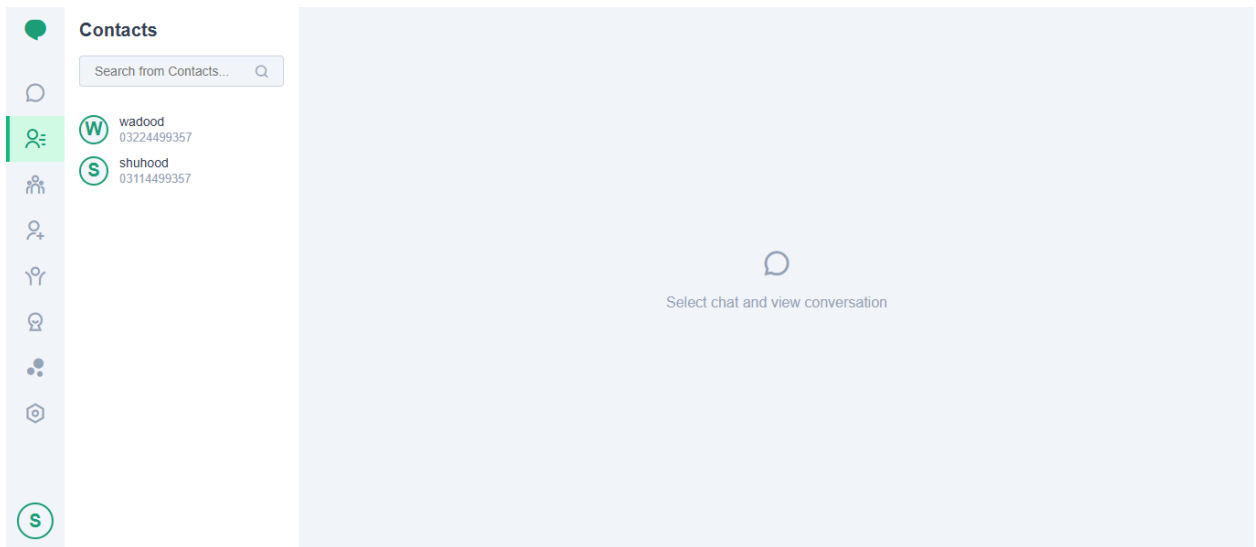


Figure 3

- **Create Group:**

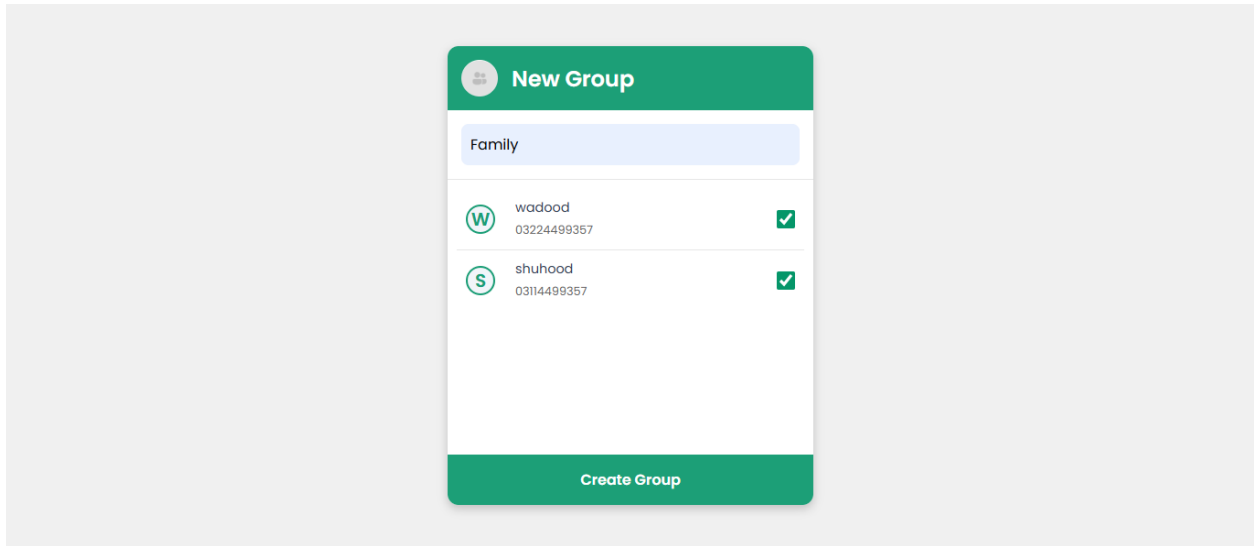


Figure 4

- **Contacts:**

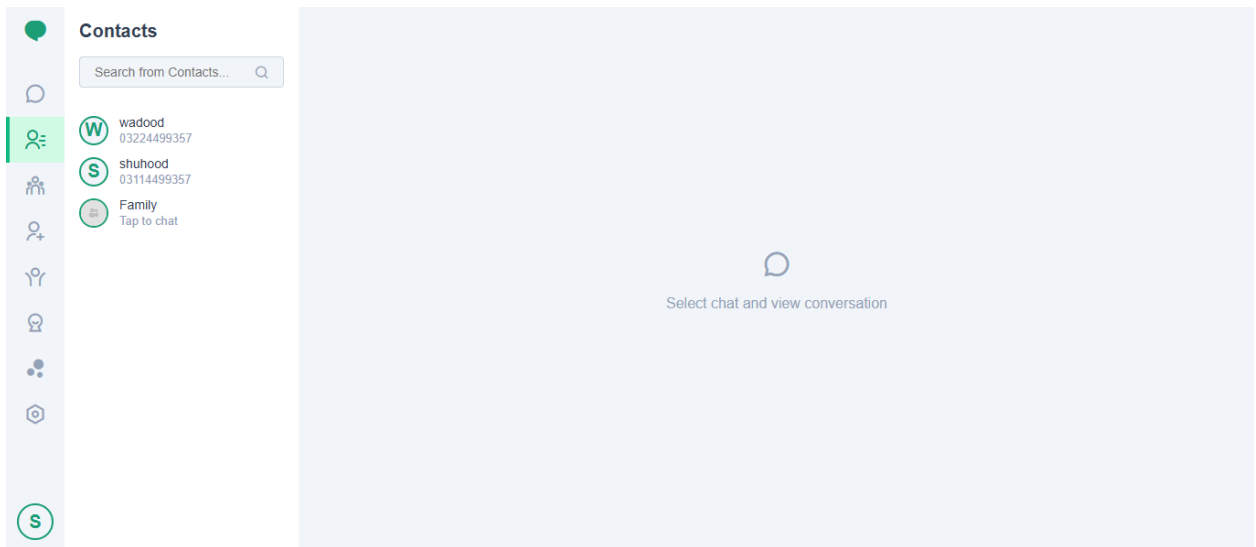


Figure 5

▪ Chat:

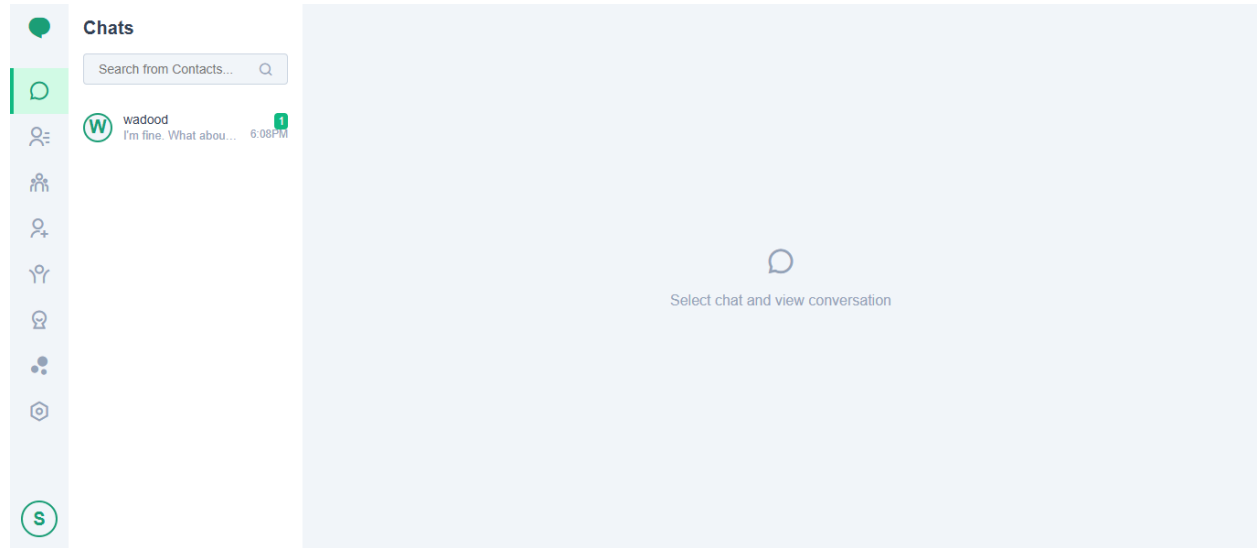


Figure 6

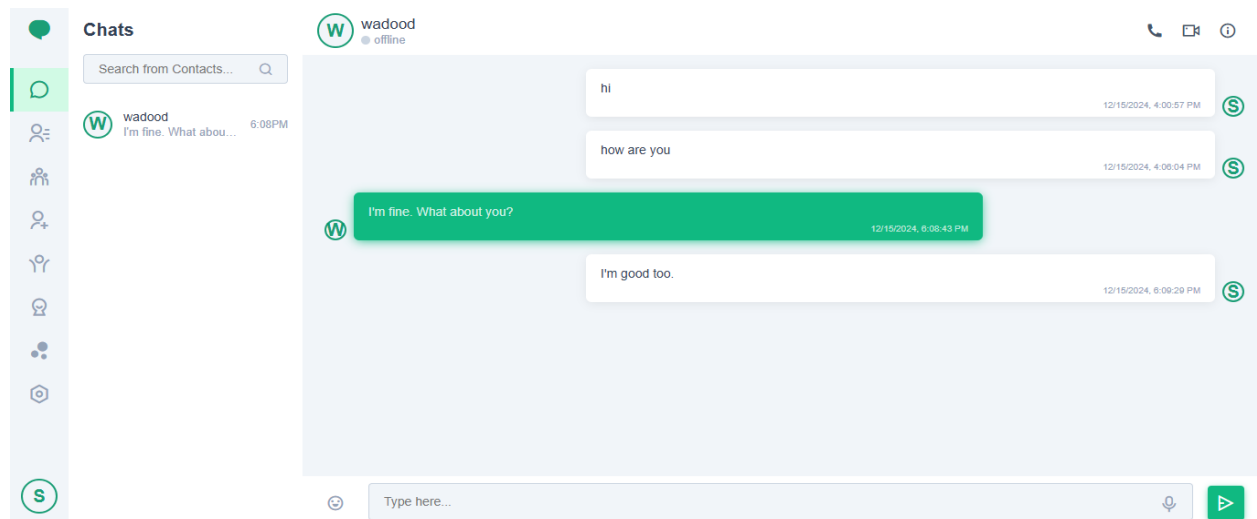


Figure 7

▪ Group Chat:

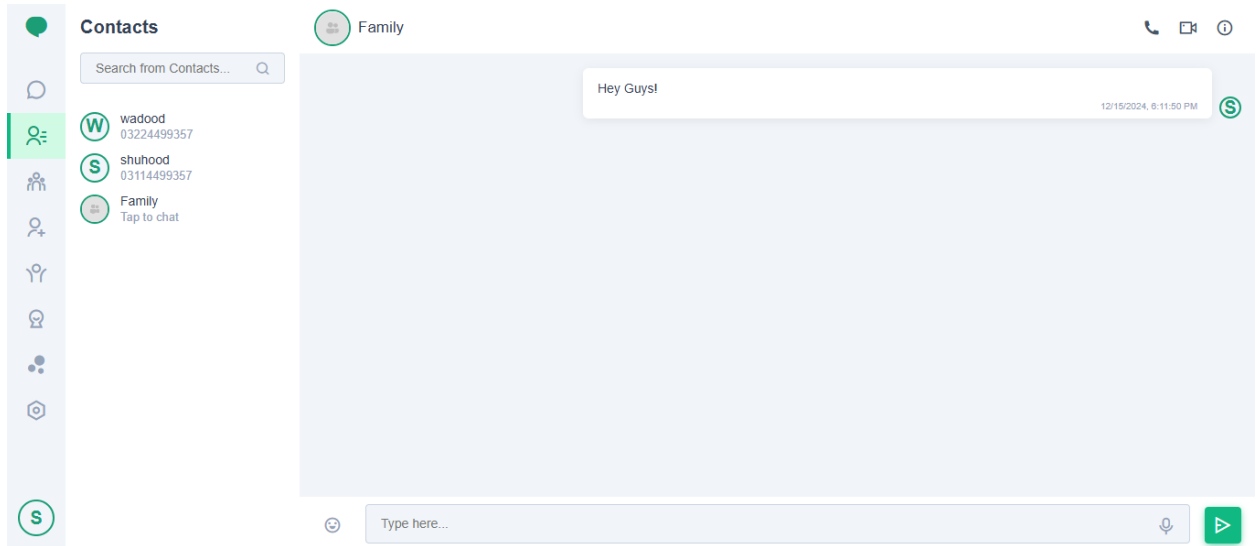


Figure 8

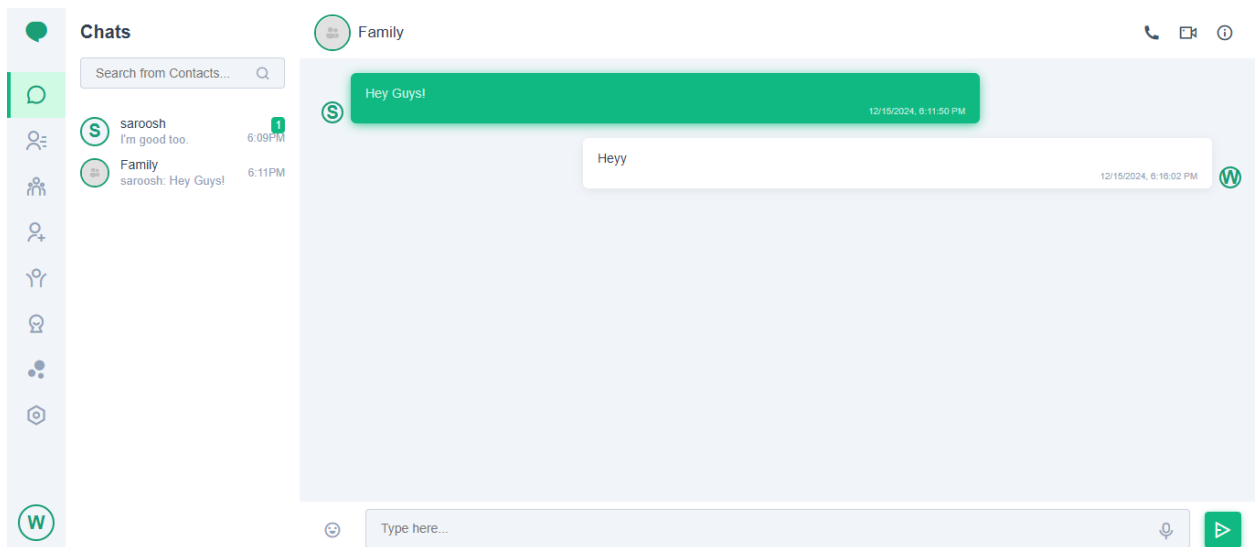


Figure 9

▪ Searched Contacts:

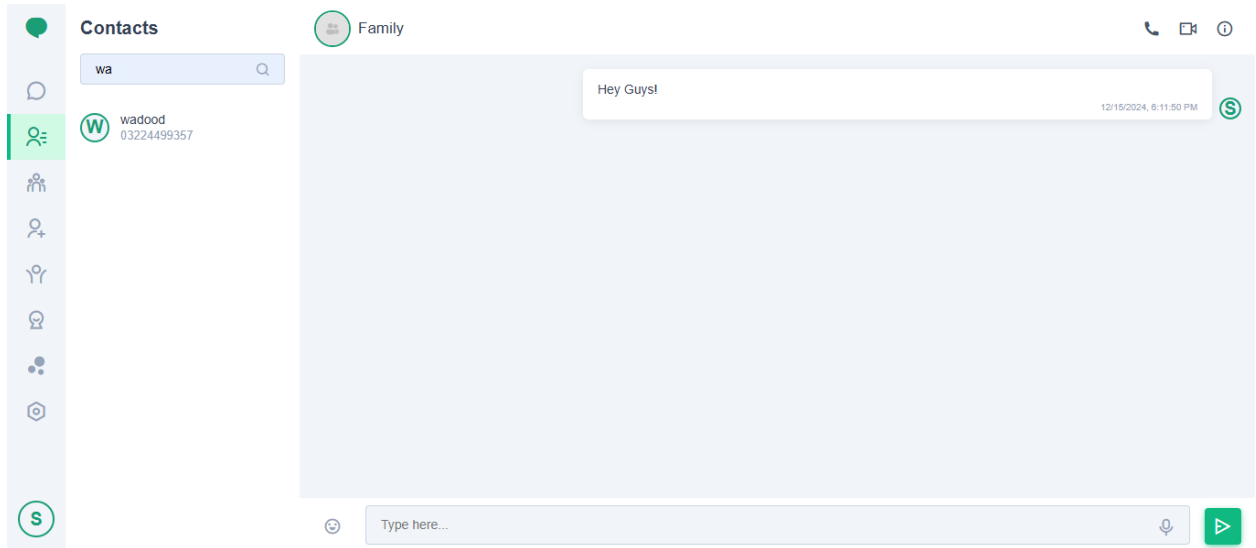


Figure 10

▪ Graph Visualization:

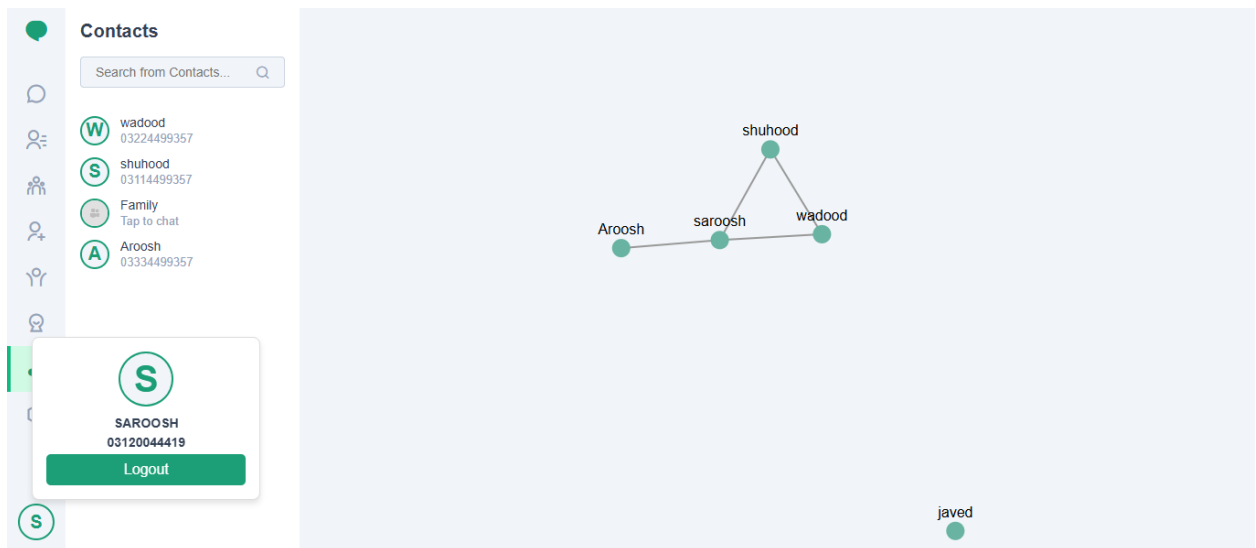


Figure 11

▪ Requests:

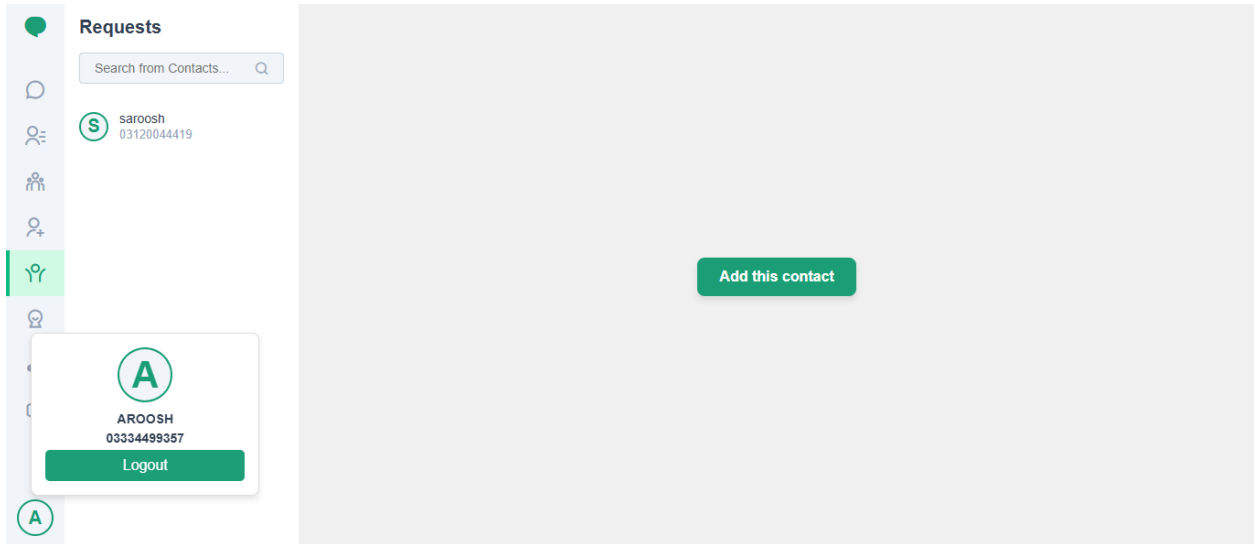


Figure 12

▪ Suggestions:

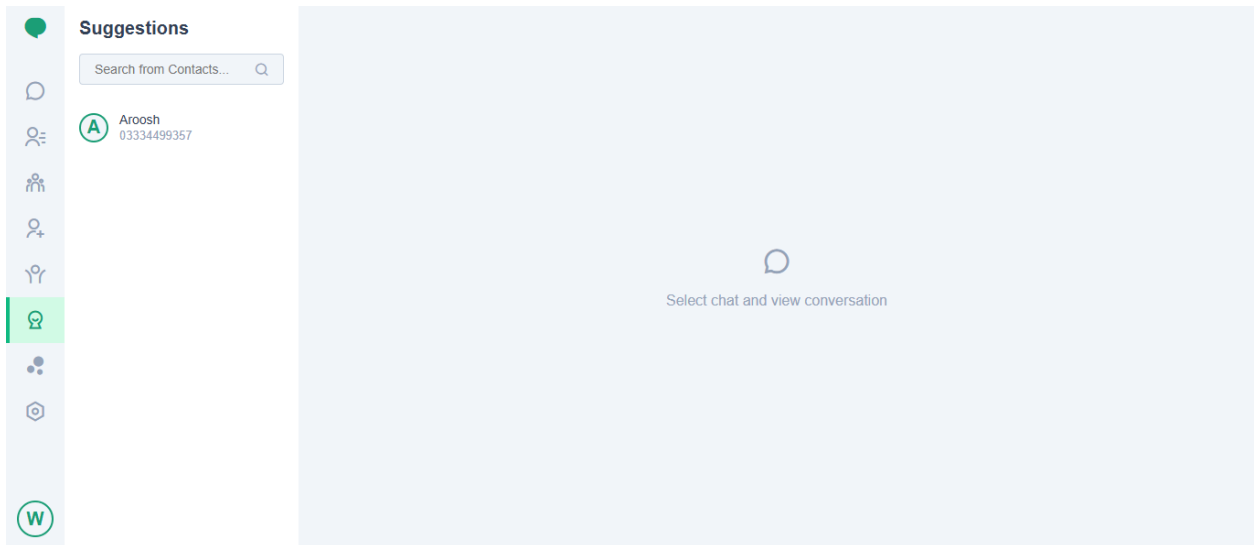


Figure 13

- **On clicking suggested user:**

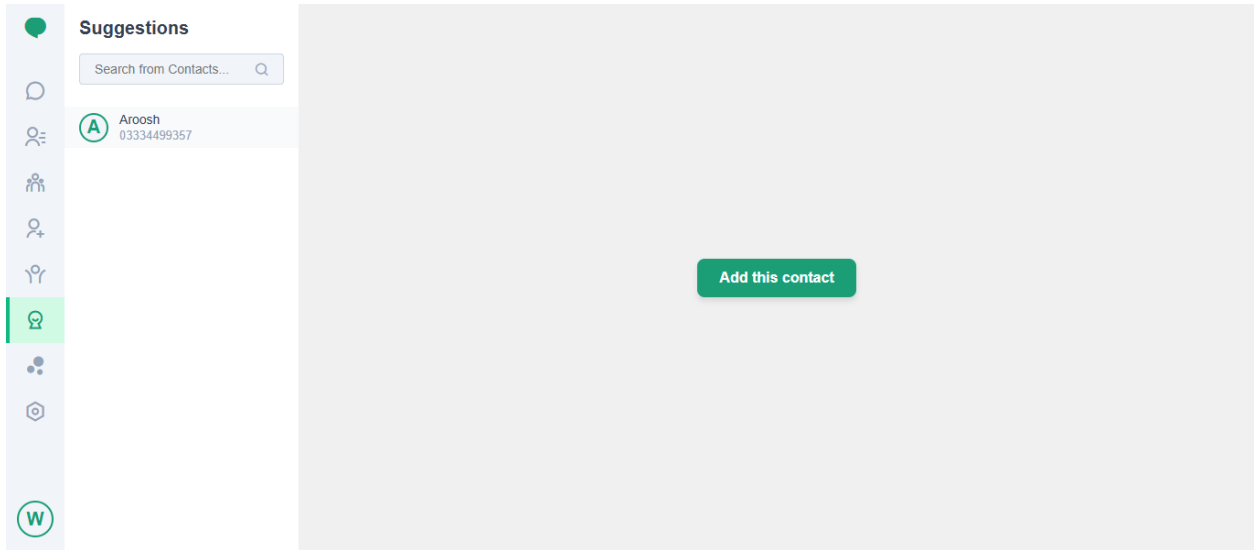


Figure 14