HERITAGE INSTITUTE OF TECHNOLOGY

Submitted under the banner of **CSEN3202**

COMPUTER NETWORK PROJECT REPORT ON

trueChat App: Instant Real Time Chat Messenger

[implementation featuring socket.io and Node.js for backend Server]

Subject Teacher: **Dr. M.B. Kar**

PREPARED BY-

NAME	COLLEGE ROLL	AUTONOMY ROLL
SUBHADEEP KUNDU	2051040	12620001155
RAMAYUDH MUKHERJEE	2051055	12620001095
SUPRIYO BANERJEA	2051064	12620001166
PRIYAM SAHA	2051267	12620017031

Requirement specification :

Server Hardware:

- **Processor:** A multi-core processor with a clock speed of at least 2.0 GHz is recommended.
- **RAM:** At least 8GB of RAM is recommended, but this can vary depending on the number of concurrent users and the size of the database.
- **Storage:** At least 100GB of storage is recommended for the server, but this can vary depending on the size of the database and the number of files being stored.
- **Network Interface:** A gigabit Ethernet card is recommended for faster data transfer rates.
- **Operating System:** The server should run on a stable and secure operating system, such as Linux or Windows Server.

Client Hardware:

- **Processor:** A multi-core processor with a clock speed of at least 1.6 GHz is recommended.
- **RAM:** At least 4GB of RAM is recommended, but this can vary depending on the number of open windows and the amount of data being processed.
- **Storage:** At least 256GB of storage is recommended for the client, but this can vary depending on the amount of data being stored.
- **Network Interface:** A fast internet connection is recommended for smoother communication and transfer of data.
- **Operating System:** The client can run on any operating system, such as Windows, Mac, or Linux.

Server Software:

- **Operating System:** The server should run on a stable and secure operating system, such as Linux or Windows Server.
- **Web Server:** A web server, such as Apache or NGINX, is required to handle HTTP and HTTPS requests.
- **Messaging Protocol:** A messaging protocol, such as XMPP or MQTT, is required to establish communication between the server and clients.
- **Security:** The server software should include security features such as SSL/TLS encryption, user authentication, and access control to protect user data.

Client Software:

- **Operating System:** The client software can run on any operating system, such as Windows, Mac, or Linux.
- **Messaging Protocol:** The client software should be compatible with the messaging protocol used by the server.
- **User Interface:** The client software should provide a user-friendly interface for users to send and receive messages, view contacts, and configure settings.
- **Security:** The client software should include security features such as SSL/TLS encryption, user authentication, and access control to protect user data.

Additional Features:

- **Notification System:** The messaging system includes a notification system to alert users of new messages, even if they are not actively using the client software.
- **Group Messaging:** The messaging system should support group messaging, allowing users to communicate with multiple contacts at once.
- **Notification sounds:** The chat app features 4 different notification sounds for 4 different events----- (i) User joining a chat, (ii) A message sent by an user, (iii) A message received by other users, (iv) A user leaving the chat.

About the Project :

- ## Server-side code:

This is a Node.js server-side code that uses the socket.io library to handle socket connections. The server listens on port 8000 for incoming connections and logs a message to the console when it starts.

The server keeps track of connected users in an object called users. Whenever a new user joins the chat, the server receives a new-user-joined event from the client with the user's name. The server adds the user's name to the users object and broadcasts a user-joined event to all connected clients except the new user.

Whenever a client sends a message, the server receives a send event with the message content. The server broadcasts a receive event to all connected clients except the sender with the message content and sender's name.

Whenever a client disconnects from the chat, the server receives a disconnect event and broadcasts a left event to all connected clients except the disconnected user with the disconnected user's name.

- ## Client-side code:

append (message, position, audio): This function appends a new message element to the main message container on the web page. It takes three arguments:

message (string): The text content of the message to be displayed.

position (string): The position of the message on the screen - left, center, or right.

audio (HTML audio element): The audio element to be played along with the message.

The function creates a new div element with the text content of the message, sets its class attribute to message and position (the value of the position argument), appends it to the message container, and plays the audio element (the value of the audio argument).

append2(message2, position, audio): This function is similar to the append() function, but it creates a new message element with a different class attribute (message2), which is used to style the message element differently from the other messages.

append3(message3, position, audio): This function is similar to the append() function, but it creates a new message element with a different class attribute (message3), which is used to style the message element differently from the other messages. This function is used specifically for displaying time-stamps.

append4(message4, position): This function is similar to the append() function, but it creates a new message element with a different class attribute (message4), which is used to style the welcome message element differently from the other messages. This function is used specifically for displaying the welcome message on the screen.

All of these functions work by creating a new div element, setting its class attribute to the appropriate value, setting its text content to the message content, appending it to the message container, and playing an audio element (if provided).

Each function is called at different times throughout the script, depending on the context of the message being displayed (e.g. a user joining or leaving the chat, sending or receiving a message, etc.).

- **Client-Server Communications**

1) Socket-establishment

(……..Server)

index.html | style.css | index.js | Client.js

ChatServer > index.js > ...

```
1   //Chat server which has to handle the socket.io connections
2
3   const io = require('socket.io')(8000);
4   console.log("Server has started.......")
5
```

(……..Client, line 6)

index.html | style.css | index.js | Client.js

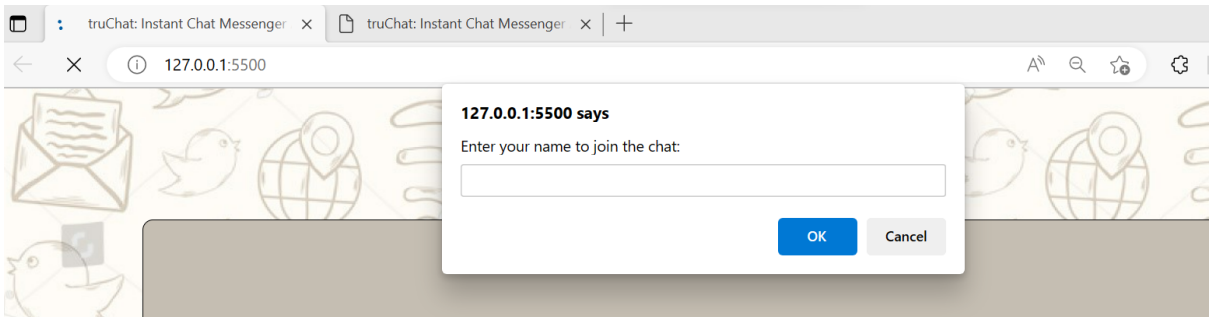
js > Client.js > ...

```
1   // giving variable names to respective DOM(Document Object Modelling) elements
2   const form = document.getElementById('send-container');
3   const messageInput = document.getElementById('messageInput');
4   const messageContainer = document.querySelector('.container');
5
6   const socket = io('http://localhost:8000');
7
```

2) Client triggering a prompt to ask the user to enter his/her name

(……..Client, line 78)

```
77  // Prompt to ask a user his or her name before he/she joins the chat
78  const name = prompt("Enter your name to join the chat: ");
79  console.log("user joined", `${name}`);
80  socket.emit('new-user-joined', name);
```



3) **Whenever a new user joins (new Client login), an event 'new-user-joined' is triggered from the client to the server.**

(……..Client, line 80)

```
socket.emit('new-user-joined', name);
```

4) **The event 'new-user-joined' is handled at the server side by an arrow function which adds this new user (identified by his/her unique socket id) to the users dictionary and thereafter broadcasts an event 'user-joined' to all other clients notifying them about the new user joined.**

(……..Server)

```
 9   io.on('connection', socket =>{
10
11       // whenever a new user joins
12       socket.on('new-user-joined', name => {
13               console.log("New user ", name);
14               users[socket.id] = name;
15               socket.broadcast.emit('user-joined', name);
16           });
17
```

5) **The event 'user-joined' is handled at the client side and notifies all other clients about the new user joined (along with timestamp). An if condition checks if the time of joining matches the time when the client joined the chatroom. If yes, then date need not be displayed once again!!**

(……..Client)

```
96  // receive the event from the Server whenever another user joins
97  socket.on('user-joined', name => {
98      console.log(typeof(name));
99
100
101     var today = new Date();
102     var date = today.getDate()+'/'+(today.getMonth()+1)+'/'+today.getFullYear();
103     var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();
104     var dateTime;
105     if (date == global_date)
106         dateTime = 'Time--> ' + time;
107     else
108         dateTime = 'Time--> ' + time + ', Date--> ' + date;
109
110
111     append2(`${dateTime}\n` + `${name} joined the chat!`, 'center', user_joined);
112 })
113
```

6) **The ‘submit’ is invoked when the client enters a message in the input box and clicks on Send button. The message is appended to the container element in viewport to the left side (for the clients who sent the message) . Thereafter , it triggers the ‘send’ event at the server.**

(…….Client)

```
// trigger an event to the server in response to the user clicking on the 'Send!' button for sending a message
form.addEventListener('submit', (e)=>{
    e.preventDefault();
    const message = messageInput.value;

    var today = new Date();
    var date = today.getDate()+'/'+(today.getMonth()+1)+'/'+today.getFullYear();
    var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();
    var dateTime;
    if (date == global_date)
        dateTime = 'Time--> ' + time;
    else
        dateTime = 'Time--> ' + time + ', Date--> ' + date;


    append3(`${dateTime}`, 'right', message_sent);
    append(`You: ${message}`, 'right', message_sent);
    socket.emit('send', message);
    messageInput.value = '';
})

```

7) **The ‘send’ event broadcasts the ‘receive’ event back to the client side so that the message can be received by all other users at the terminal.**

(…….Server)

```
// whenever a message is sent by any client, it must be broadcasted to all OTHER users
socket.on('send', message =>{
    socket.broadcast.emit('receive', {message: message, name: users[socket.id]});
});
```

8) **The 'receive' event is handled at the client side. The message is appended to the container element in viewport to the right side (for the clients who are to receive the message) along with timestamp. The same if condition has been incorporated here as well.**

(…….Client)

```
115
116  // receive messages sent by other users in the chat
117  socket.on('receive', data=> {
118      setTimeout(function() {
119          //your code to be executed after 1 second
120          var today = new Date();
121          var date = today.getDate()+'/'+(today.getMonth()+1)+'/'+today.getFullYear();
122          var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();
123          var dateTime;
124          if (date == global_date)
125              dateTime = 'Time--> ' + time;
126          else
127              dateTime = 'Time--> ' + time + ', Date--> ' + date;
128
129
130          append3(`${dateTime}`, 'left', message_sent);
131          append(`${data.name}: ${data.message}`, 'left', message_received);
132        }, delayInMilliseconds);
133  })
134
```

9) **The 'disconnect' event is fired at the server side whenever the socket gets disconnected at a particular client terminal. Now, whenever a client gets disconnected, the corresponding user is forced to leave the chat room, which implies that we must broadcast an event 'left' to all other clients stating that the aforementioned user has left the chat.**

(…….Server)

```
23      // whenever a user leaves the chat
24      socket.on('disconnect', message =>{
25          socket.broadcast.emit('left', users[socket.id]);
26          delete users[socket.id];
27      });
28  })
```

10) **The ‘left’ event at the client side creates a message element corresponding to the user who has left the chat along with the timestamp. The same if condition mentioned above has been included here as well.**

(…….Client)

```
// receive the event from the Server whenever another user leaves the chat
socket.on('left', name=> {

    var today = new Date();
    var date = today.getDate()+'/'+(today.getMonth()+1)+'/'+today.getFullYear();
    var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();
    var dateTime;
    if (date == global_date)
        dateTime = 'Time--> ' + time;
    else
        dateTime = 'Time--> ' + time + ', Date--> ' + date;


    append2(`${dateTime}\n` + `${name} left the chat!`, 'center', user_left);
})
```

Index.htms :

- **Source code:**

```
<!DOCTYPE html>
<html lang = "en">
<head>
   <meta charset = "UTF-8">
   <meta name = "viewport" content="width=device-width, initial-scale=1.0">
   <title> truChat: Instant Chat Messenger App </title>
   <script defer src="http://localhost:8000/socket.io/socket.io.js"></script>
   <script defer src="js/Client.js"></script>
   <link rel="stylesheet" href="css/style.css">
</head>
<body>
   <nav>
      <img class="logo" alt='image of app logo' src="temp.png">
   </nav>
   <div class = "container">
   </div>
   <div class="send-message">
      <form action="#" id="send-container">
         <label for="middle-initial">Type a message to chat!
         <input type="text" name="messageInput" id="messageInput"></label>
         <button id = 'send' class="button" type="submit"><font face = "Comic sans MS"
size="4">Send!</font></button>
      </form>
   </div>
</body>
</html>
```

Style.css :

- **Source code:**

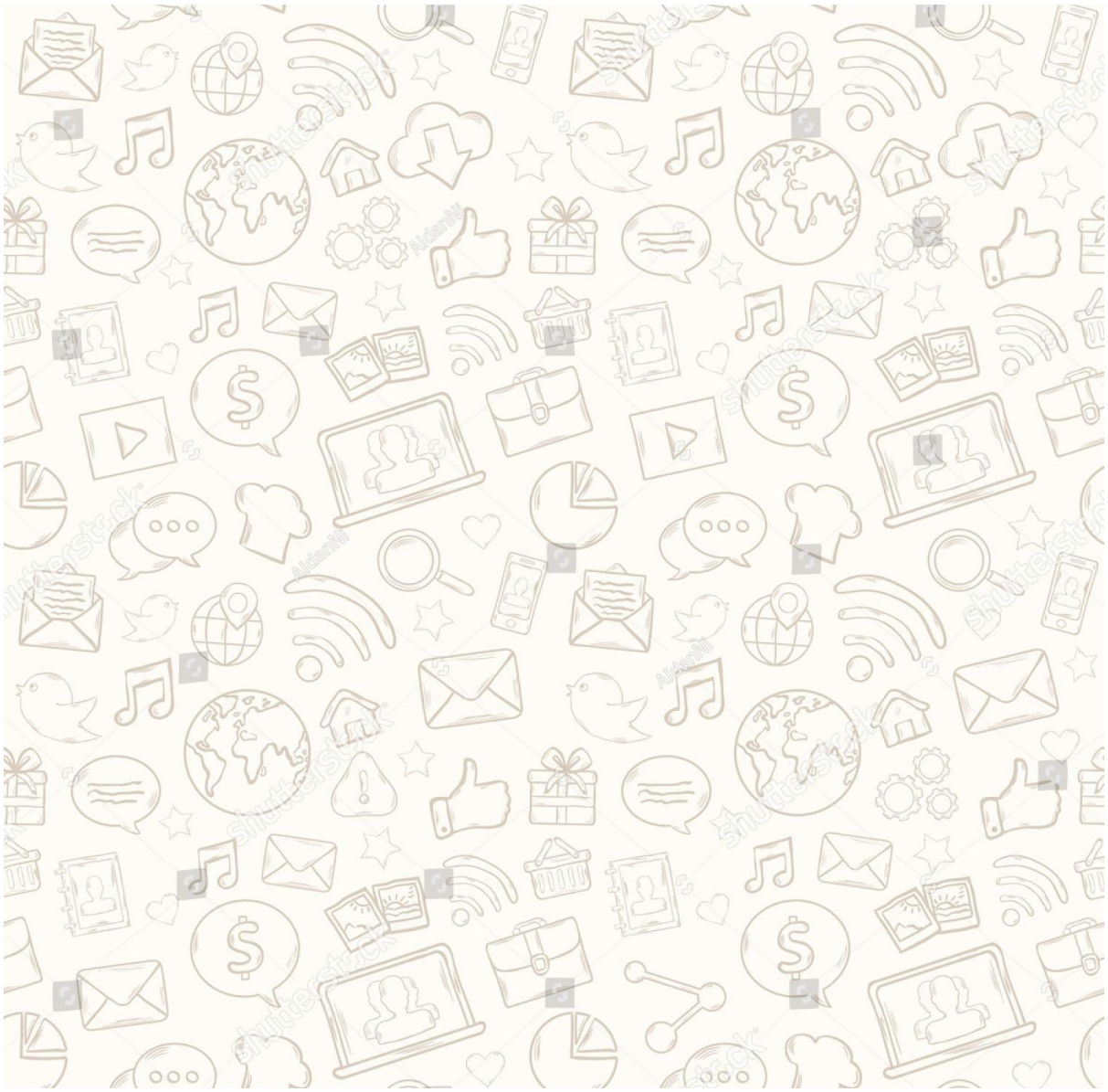
```
.logo
{
   display: block;
   margin: auto;
   width: 150px;
   height: 120px;
}



body{
   height: 100vh;
   background-image: url("bg.jpg");
}

.container{
   max-width: 985px;
   background-color: rgb(197, 190, 178);
   margin: auto;
   height: 55vh;
   padding: 33px;
   overflow-y: auto;
   margin-bottom: 13px;
   margin-top: 13px;
   border: 1px solid black;
   border-radius: 10px;
}
```

```
.message{
    background-color: rgb(223, 223, 203);
    width: 30%;
    padding: 10px;
    border: 1px solid black;
    border-radius: 12px;
    font-size: larger;
    font-family: 'Lucida Sans';
}

.message2{
    background-color: rgb(220, 217, 210);
    width: 30%;
    padding: 10px;
    margin: 17px 12px;
    font-style: italic;
    font-family: 'Courier New';
    /*border: 1px solid black;
    border-radius: 12px;*/
}

.message3{
    background-color: transparent;
    width: 30%;
    margin: 17px 12px;
    font-family: 'Courier';
    font-size: small;
    /*border: 1px solid black;
    border-radius: 12px;*/
}
```

```
.message4{
    background-color: rgba(255, 255, 255, 0.699);
    width: 97%;
    margin: 17px 12px;
    font-family: 'Arial Black';
    font-size: large;
    font-style: normal;
    /*border: 1px solid black;
    border-radius: 12px;*/
}



.left{
    float: left;
    clear: both;
}

.right{
    float: right;
    clear: both;
}

.center{
    margin-left: 320px;
    clear: both;
}

#send-container{
    display: block;
```

```
    margin: auto;
    text-align: center;
    max-width: 985px;
    width: 100%;
    height: 40px;
    /*display: flex;
    justify-content: center;
    align-items: center;*/
}

#messageInput{
    width: 92%;
    border: 1px solid black;
    border-radius: 6px;
    height: 41px;
    margin-bottom: 13px;
}

.button{
    cursor: pointer;
    border: 1px solid black;
    border-radius: 6px;
    height: 42px;
    margin-bottom: 13px;
}
```

- **Jpg files used for Frontend:**



Client.js :

- **Source code :**

```
// giving variable names to respective DOM(Document Object Modelling) elements
const form = document.getElementById('send-container');
const messageInput = document.getElementById('messageInput');
const messageContainer = document.querySelector('.container');


const socket = io('http://localhost:8000');


// for creating a time-delay
var delayInMilliseconds = 3000; //3 second


// load the audio files
var user_joined = new Audio('user_joined.mp3');
var user_left = new Audio('user_left.mp3');
var message_sent = new Audio('message_sent.mp3');
var message_received = new Audio('message_received.mp3');



// function to append a message element to the main message container
function append(message, position, audio) {
   const messageElement = document.createElement('div');
   messageElement.innerText = message;
   messageElement.classList.add('message');
   messageElement.classList.add(position);
   messageContainer.append(messageElement);
   audio.play();
}
```

```
function append2(message2, position, audio) {
    const messageElement = document.createElement('div');
    messageElement.innerText = message2;
    messageElement.classList.add('message2');
    messageElement.classList.add(position);
    messageContainer.append(messageElement);
    audio.play();
}

function append3(message3, position, audio) {
    const messageElement = document.createElement('div');
    messageElement.innerText = message3;
    messageElement.classList.add('message3');
    messageElement.classList.add(position);
    messageContainer.append(messageElement);
    //audio.play();
}

function append4(message4, position) {
    const messageElement = document.createElement('div');
    messageElement.innerText = message4;
    messageElement.classList.add('message4');
    messageElement.classList.add(position);
    messageContainer.append(messageElement);
    //audio.play();
}

// trigger an event to the server in response to the user clicking on the 'Send!' button for sending a message
form.addEventListener('submit', (e)=>{
```

```
    e.preventDefault();
    const message = messageInput.value;

    var today = new Date();
    var date = today.getDate()+'/'+(today.getMonth()+1)+'/'+today.getFullYear();
    var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();
    var dateTime;
    if (date == global_date)
        dateTime = 'Time--> ' + time;
    else
        dateTime = 'Time--> ' + time + ', Date--> ' + date;


    append3(`${dateTime}`, 'right', message_sent);
    append(`You: ${message}`, 'right', message_sent);
    socket.emit('send', message);
    messageInput.value = '';
})


// Prompt to ask a user his or her name before he/she joins the chat
const name = prompt("Enter your name to join the chat: ");
console.log("user joined", `${name}`);
socket.emit('new-user-joined', name);

var global_today = new Date();
var global_date =
global_today.getDate()+'/'+(global_today.getMonth()+1)+'/'+global_today.getFullYear();
var global_time = global_today.getHours() + ":" + global_today.getMinutes() + ":" +
global_today.getSeconds();
const weekday = ["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"];
```

```
var global_dateTime = global_time + ' hours on Date = ' + global_date + ' (' +
weekday[global_today.getDay()] + ')';

welcomeMessage = '******Welcome to truChat : Instant Chat Messenger App******\n\n' ;

welcomeMessage += `You joined the Chat Room at Time = ${global_dateTime}` ;

append4('\n'+welcomeMessage+'\n ', 'left');









// receive the event from the Server whenever another user joins

socket.on('user-joined', name => {

   console.log(typeof(name));



   var today = new Date();

   var date = today.getDate()+'/'+(today.getMonth()+1)+'/'+today.getFullYear();

   var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();

   var dateTime;

   if (date == global_date)

      dateTime = 'Time--> ' + time;

   else

      dateTime = 'Time--> ' + time + ', Date--> ' + date;



   append2(`${dateTime}\n` + `${name} joined the chat!`, 'center', user_joined);

})
```

```
// receive messages sent by other users in the chat
socket.on('receive', data=> {
   setTimeout(function() {
      //your code to be executed after 1 second
      var today = new Date();
      var date = today.getDate()+'/'+(today.getMonth()+1)+'/'+today.getFullYear();
      var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();
      var dateTime;
      if (date == global_date)
         dateTime = 'Time--> ' + time;
      else
         dateTime = 'Time--> ' + time + ', Date--> ' + date;


      append3(`${dateTime}`, 'left', message_sent);
      append(`${data.name}: ${data.message}`, 'left', message_received);
    }, delayInMilliseconds);
})


// receive the event from the Server whenever another user leaves the chat
socket.on('left', name=> {

   var today = new Date();
   var date = today.getDate()+'/'+(today.getMonth()+1)+'/'+today.getFullYear();
   var time = today.getHours() + ":" + today.getMinutes() + ":" + today.getSeconds();
   var dateTime;
   if (date == global_date)
      dateTime = 'Time--> ' + time;
   else
      dateTime = 'Time--> ' + time + ', Date--> ' + date;
```

```
    append2(`${dateTime}\n` + `${name} left the chat!`, 'center', user_left);
})
```

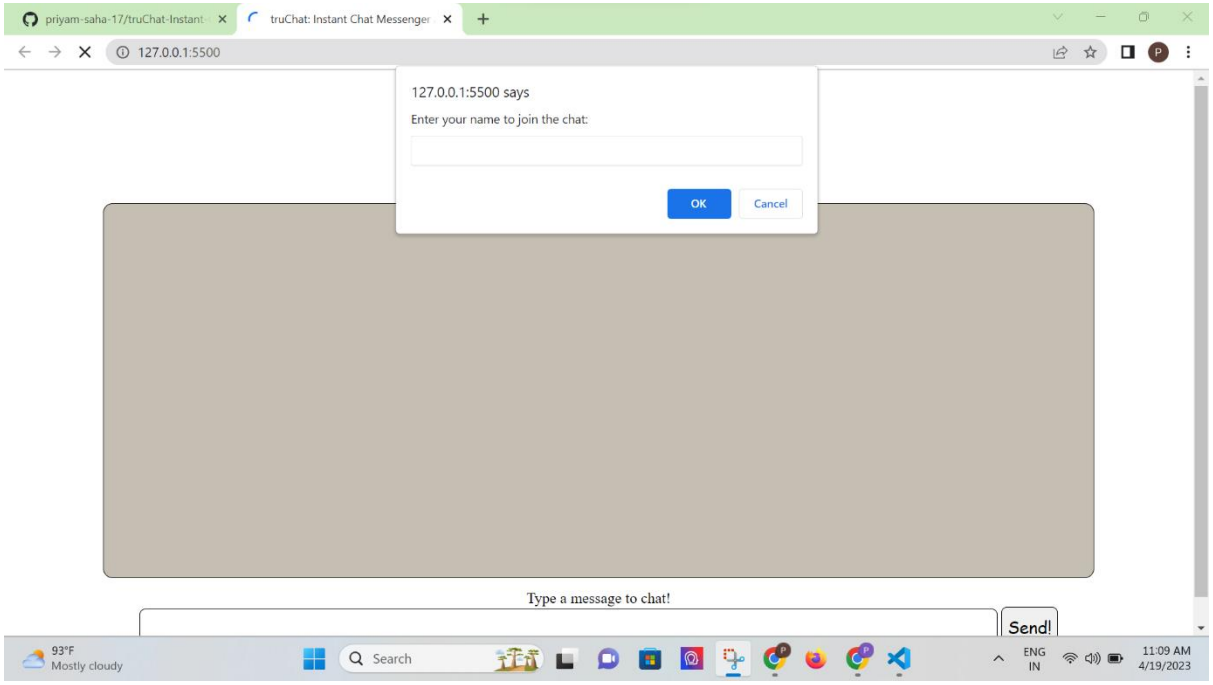
Index.js :

- **Source code :**

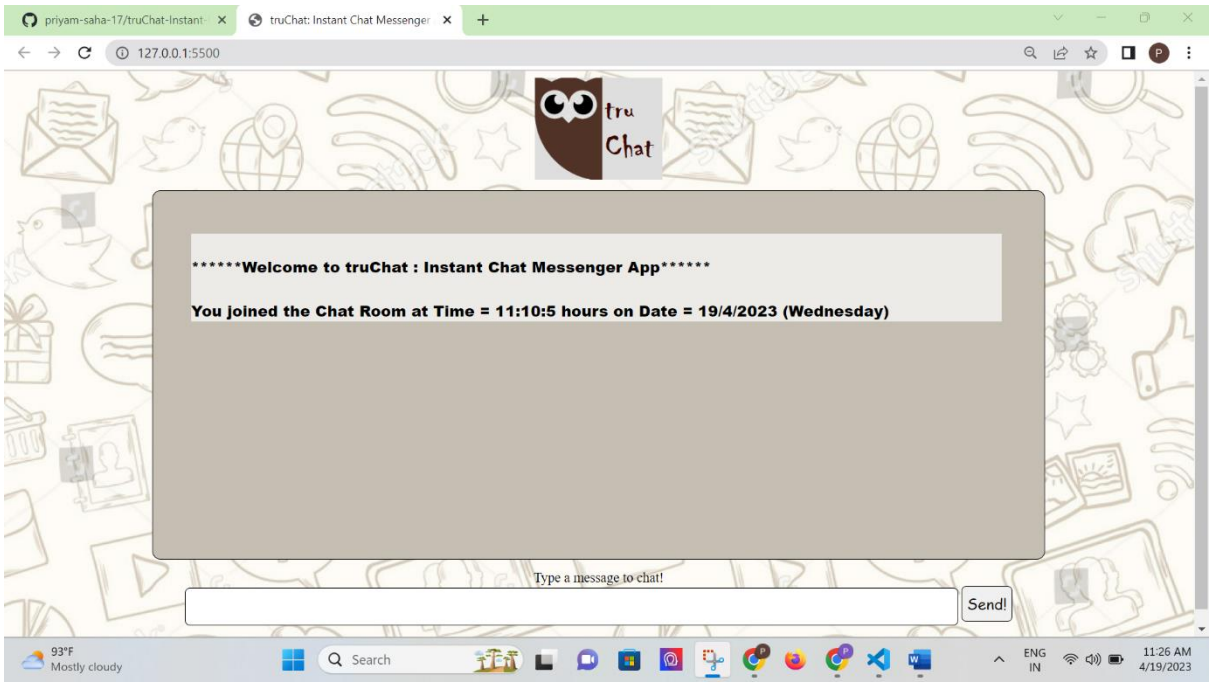
```
//Chat server which has to handle the socket.io connections
const io = require('socket.io')(8000);
console.log("Server has started.....")

const users = {}

// This initiates the io so as to listen to all client requests
io.on('connection', socket =>{

   // whenever a new user joins
   socket.on('new-user-joined', name=> {
         console.log("New user ", name);
         users[socket.id] = name;
         socket.broadcast.emit('user-joined', name);
      });

   // whenever a message is sent by any client, it must be broadcasted to all OTHER users
   socket.on('send', message =>{
      socket.broadcast.emit('receive', {message: message, name: users[socket.id]});
   });

   // whenever a user leaves the chat
   socket.on('disconnect', message =>{
      socket.broadcast.emit('left', users[socket.id]);
      delete users[socket.id];
   });
})
```

Output snapshots:

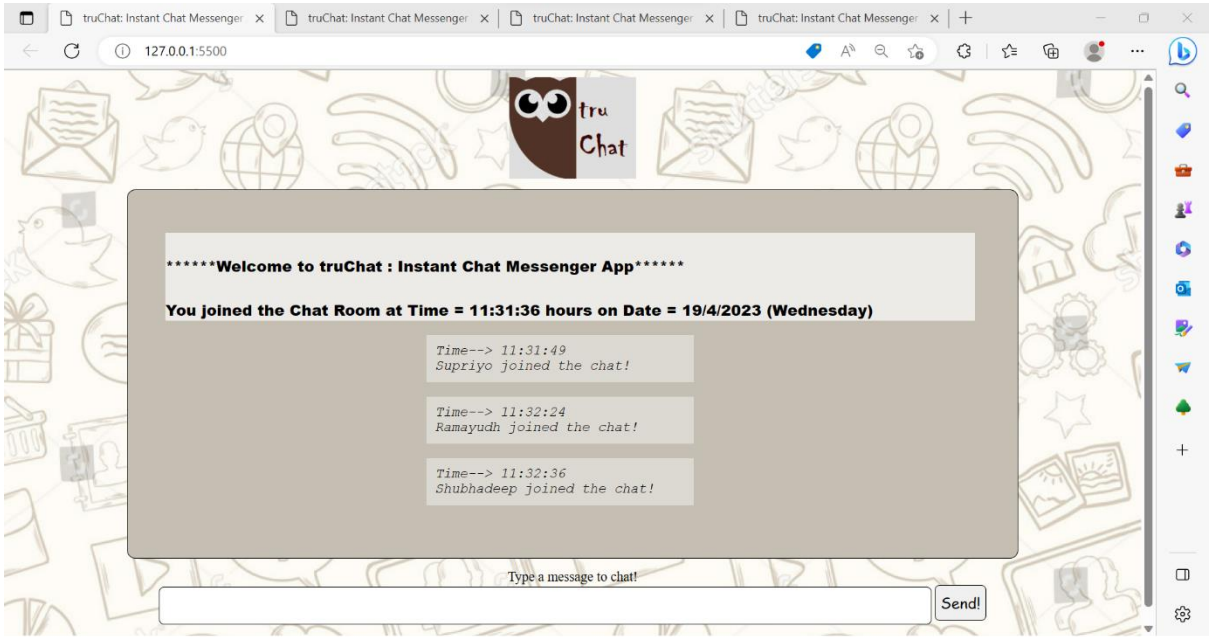
Joining Page:

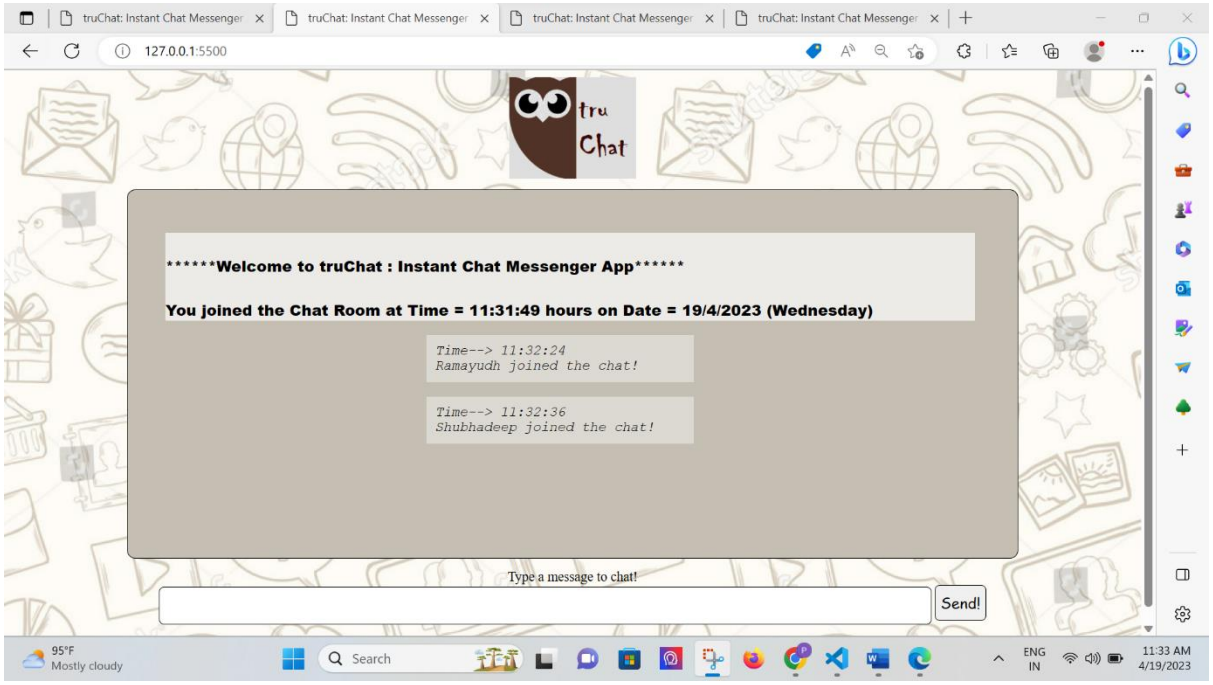


Welcome Message (Chat room created with time and date stamp):

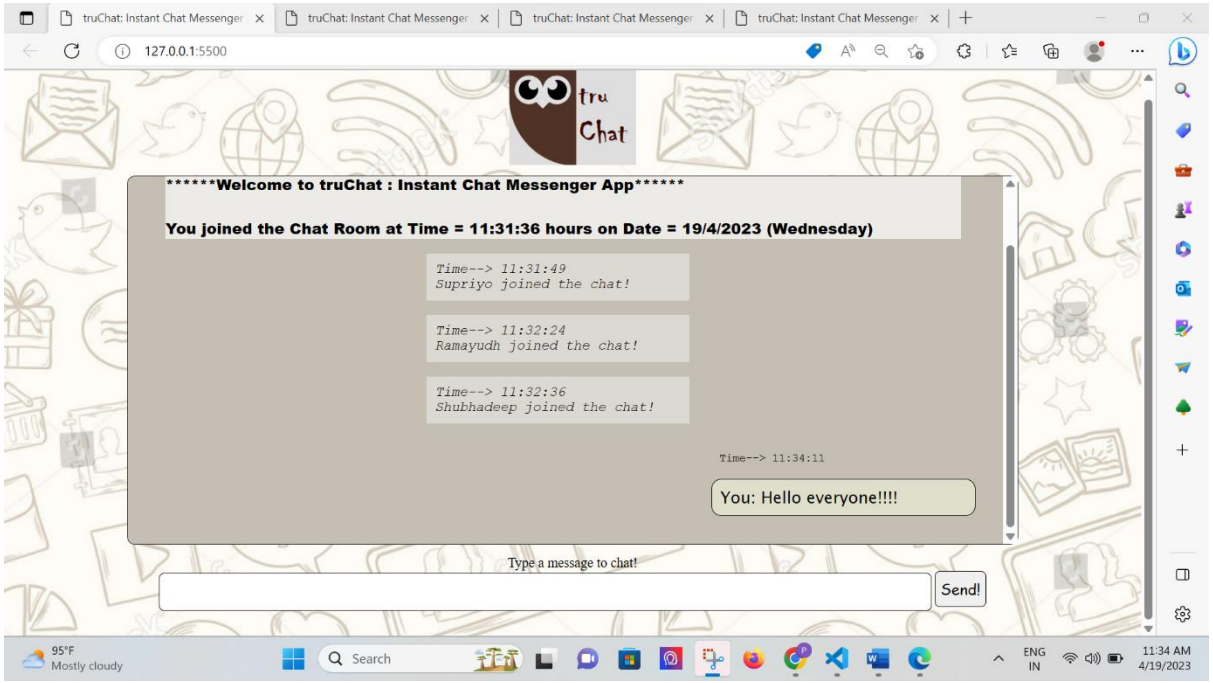


Joining notification when other clients have joined the chat room (along with the timestamp):

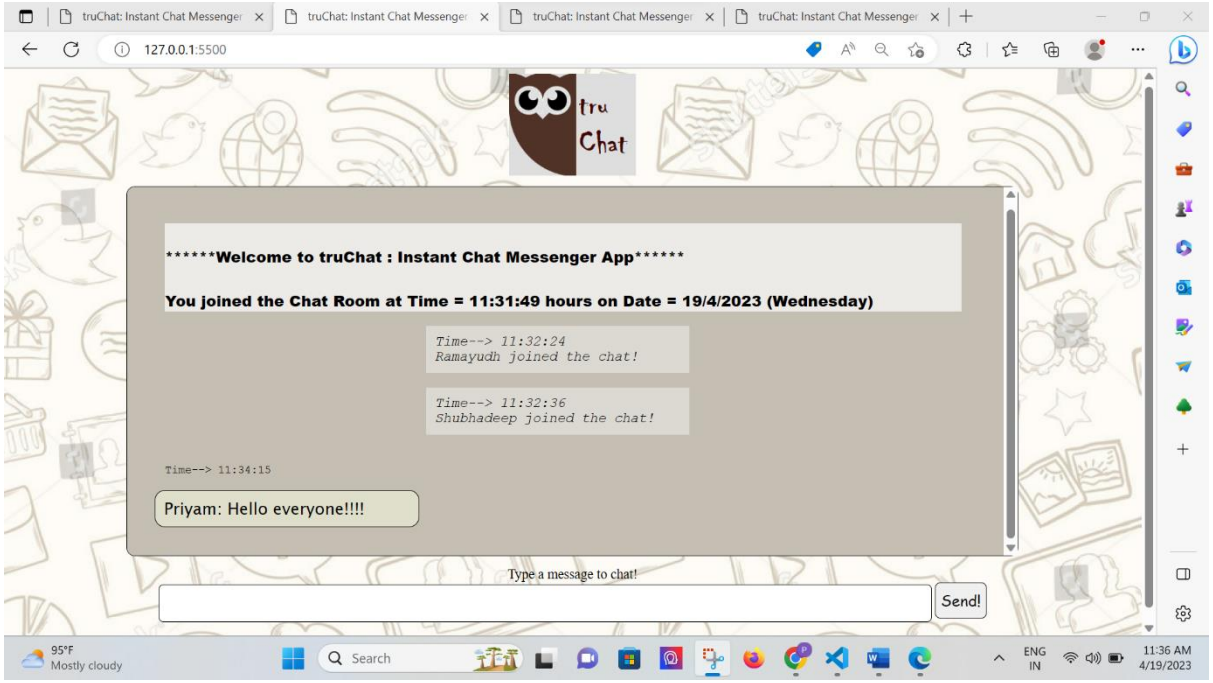




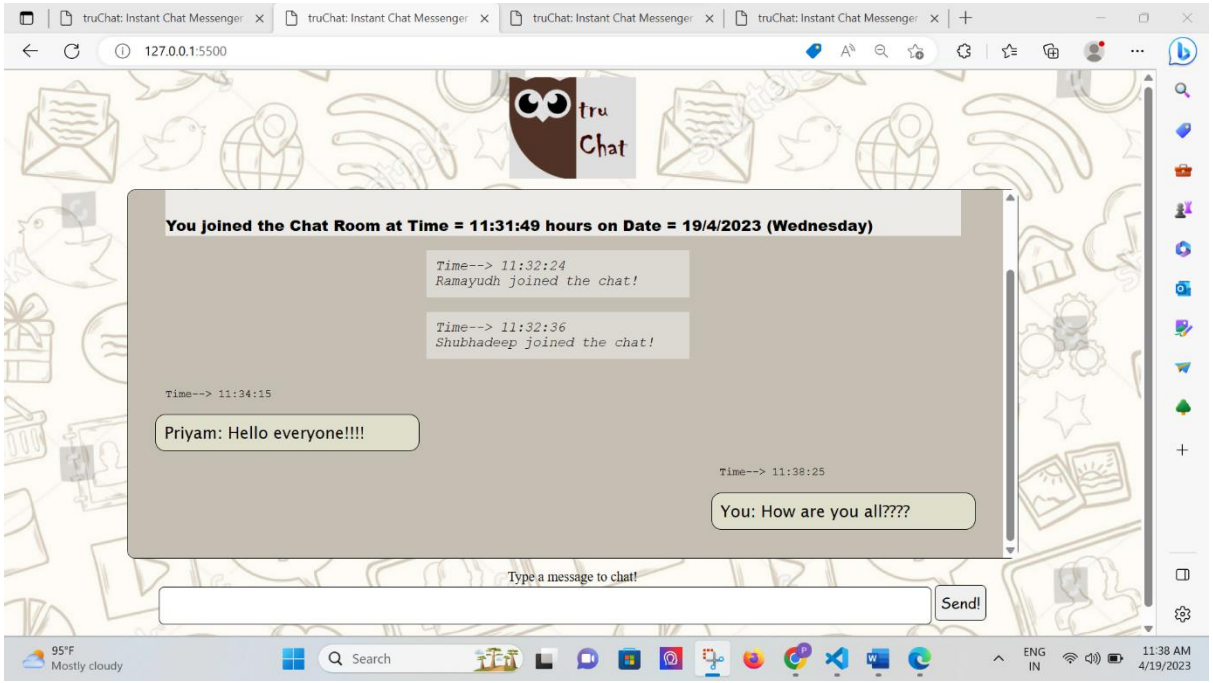
A client sending a message to other clients:



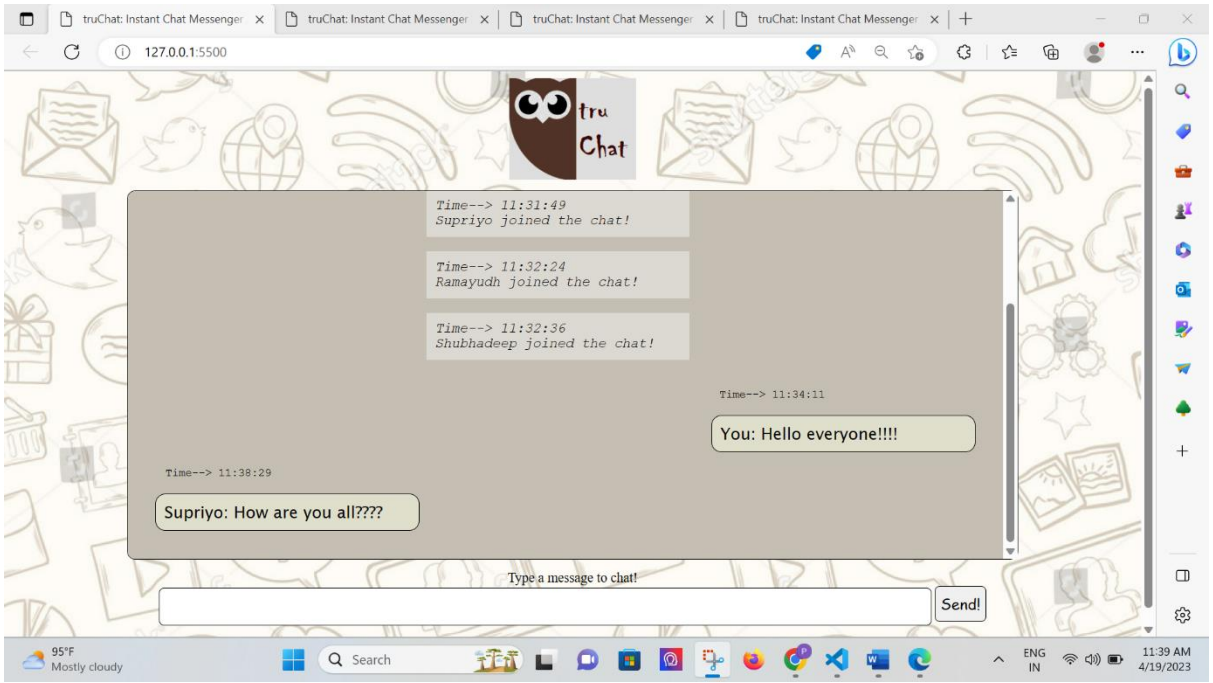
Other clients receiving the same message:



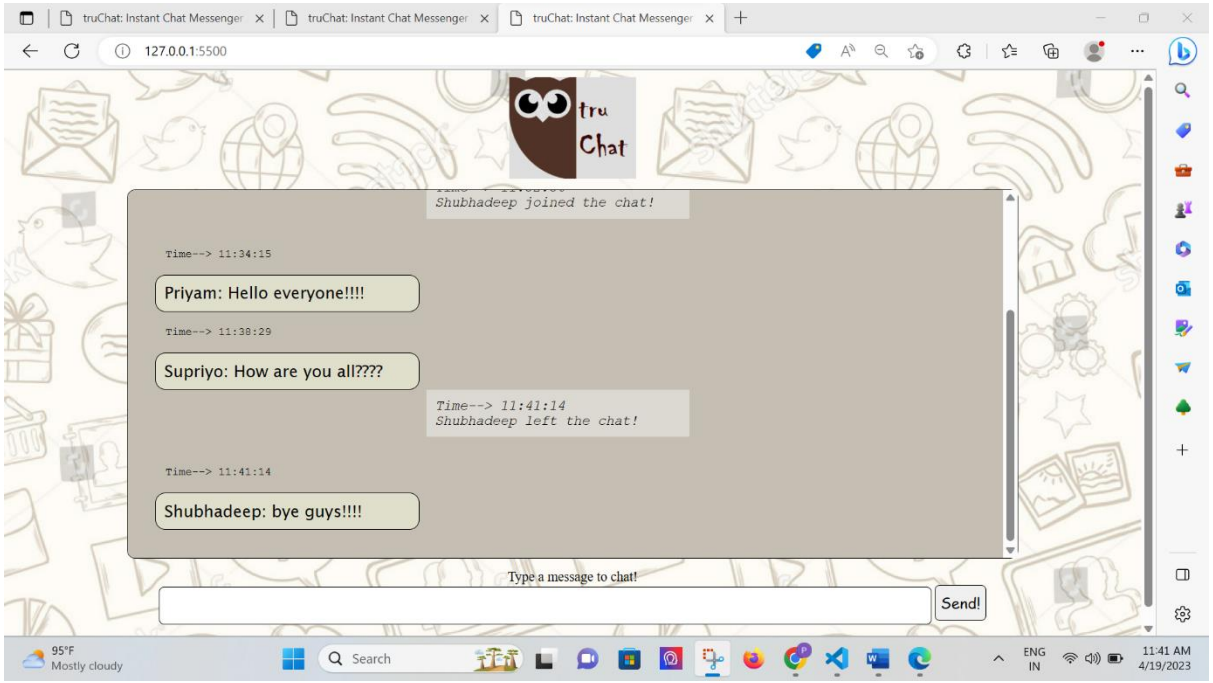
Replying to a hello message!



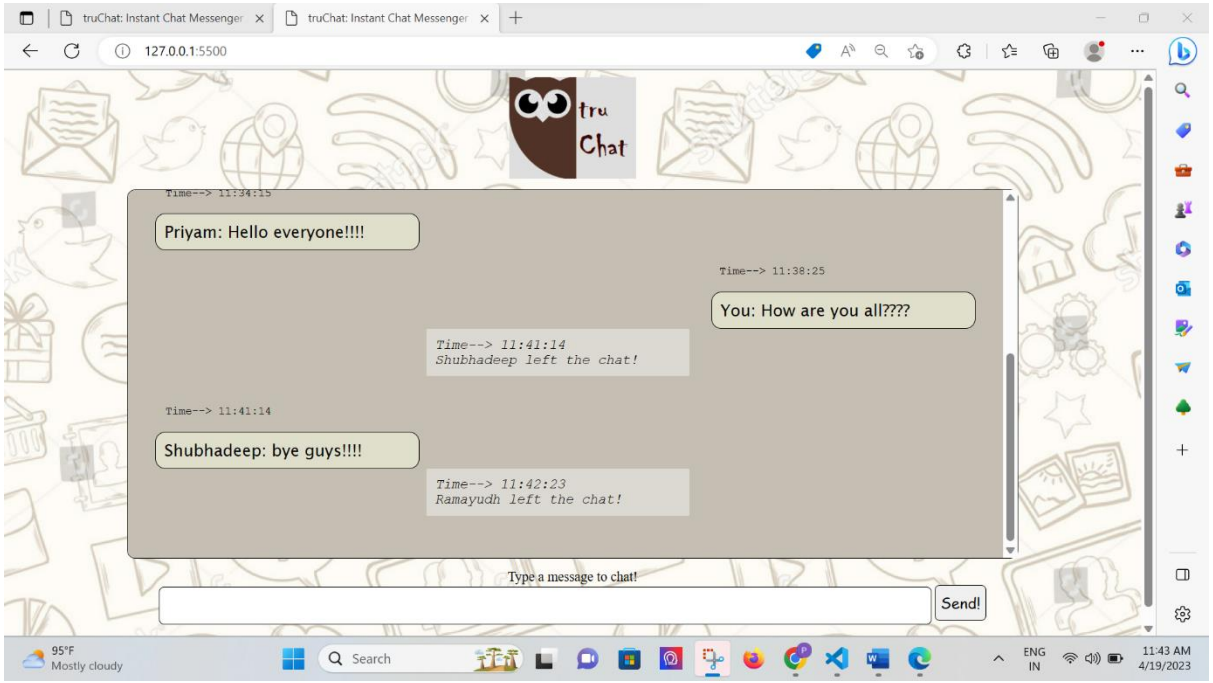
One can scroll down to see newer messages, scroll up to see the older messages:



A client leaving the chat room by closing the tab (or due to disconnection from the server port):



Whenever a user leaves the chat room, other clients get broadcasted with a notification along with a timestamp:



Future scope:

- **File Transfer:** The messaging system should allow users to send and receive files, such as images or documents, through the client software.
- **Authentication system.**
- **Scaling up to accommodate more users using techniques like load-balancing etc.**
- **Create more chat rooms.**
- **Send media, voice records etc.**
- **Better UI/UX**