# COMPUTER NETWORKS PROJECT

# REAL TIME CHAT APPLICATION
## (USING SOCKET.IO, EXPRESS, DJANGO)


**SUBMITTED BY :-**
**SONAL BERA - 2K18/IT/120**
**VIVEK YADAV - 2K18/IT/135**

**SUBMITTED TO :-**
**Prof. ANAMIKA CHOUHAN**

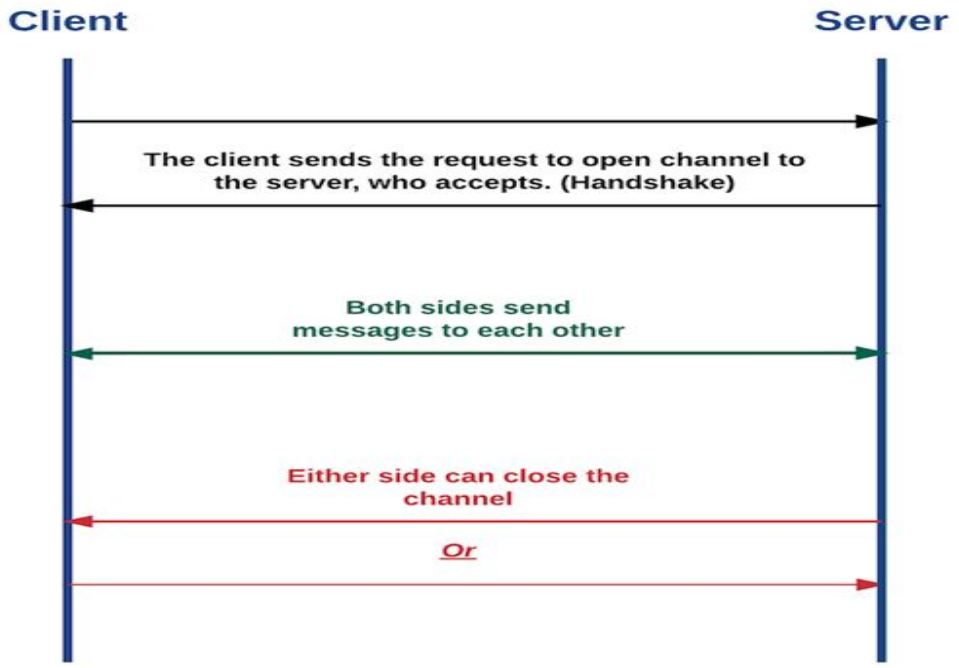
# INDEX

# INTRODUCTION

Chat Applications are primarily meant to support the one to one or a group exchange of messages, pictures, voice or even video. They help connect individuals from any corners of the world, without any barriers.

In today's digital age, the rise of messaging and chat apps like WhatsApp, Telegram, Messenger etc have further made communicating with anyone present at any part of the world feel like a child's play. Although these apps have high level code and functions, the basic underlying principle of any messaging chat app is based on its dependence on sockets for the transfer of data.

These apps are able to handle huge traffic simultaneously, and enables the end user to chat simultaneously with multiple users by making use of the availability of 'n' number of ports on a system. Each socket system involves a combination of an IP address and a port number of the user. Thus, the user doesn't have any such limit on the number of users he/she can chat with at a time.



# AIM

In this project, we aim to develop a chat application that enables a user to have a real time chat with another user over the network.This Chatterpro application is developed using Django channels as backend and Vanilla JS as frontend.
We have the project as a real time chat app and it can be hosted on the system's localhost. It can handle multi user chats simultaneously, i.e the user can chat with multiple people together at a time. The primary objective of this report is to present the principles behind websocket programming and the libraries available for websocket programming applications in python.

# TECH STACKS INVOLVED

## ▪ Socket

A socket is basically an endpoint of a two-way communication link between two programs running on a network. A socket is attached to a port number so that the TCP layer can identify the application that data is destined to be sent to. Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request. On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request.The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.

## ▪ Web Sockets

Web Sockets are similar to sockets and are also a way to communicate between a client, the browser and a server and this communication is bi-directional. That means that data can flow in both ways, it can flow from the client to the server and also from the server to the client. The difference is that, because these Web Sockets are always open it allows for real-time data flow in the chat applications.

## ▪ JavaScript

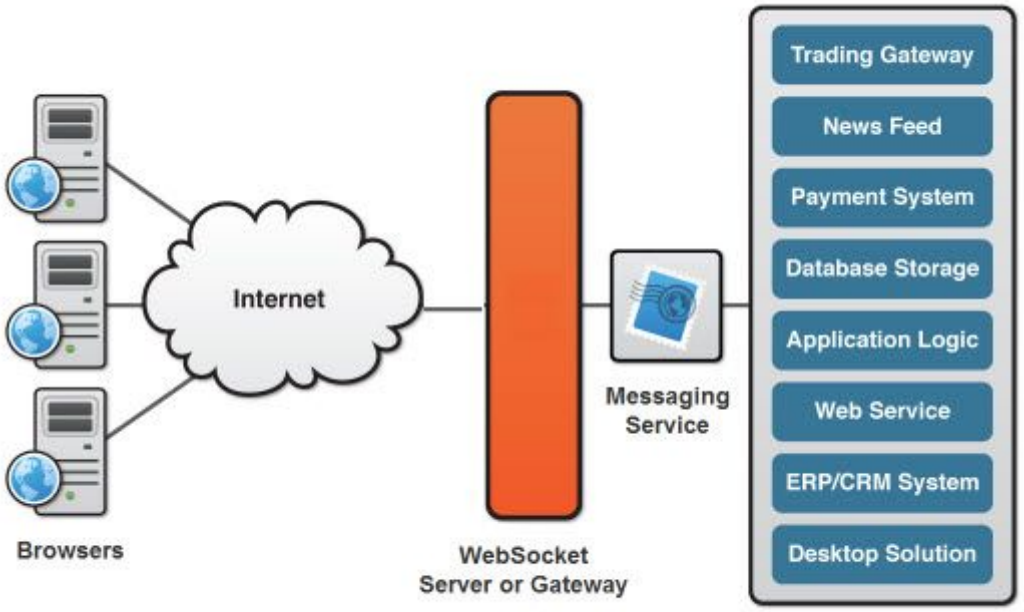
JavaScript is a text-based programming language used both on the client-side and server-side that allows you to make web pages interactive. Where HTML and CSS are languages that give structure and style to web pages, **JavaScript** gives web pages interactive elements that engage a user.

### ▪ Django Channels

Channels is a project that takes Django and extends its abilities beyond HTTP - to handle WebSockets, chat protocols, IoT protocols, and more. It’s built on a Python specification called ASGI.

# Sockets vs Web Sockets

The main difference is that even though they achieve similar things, Web Sockets typically run from browsers connecting to Application Server over a protocol similar to HTTP that runs over TCP/IP. So they are primarily for Web Applications that require a permanent connection to its server. On the other hand, plain sockets are more powerful and generic. They run over TCP/IP but they are not restricted to browsers or HTTP protocol. They could be used to implement any kind of communication.



# IMPLEMENTATION

We create a **web chat application** using Django channels and JS. A normal chat app in which people can communicate if they are in the same room. We have implemented Client/Server Protocol in this project. The purpose of the project is to communicate between a client and a server using a websocket via Django channel. We will be focused on TCP/IP socket connections which are a fundamental part of socket programming.

These include the scripts that will run on the user's device like a browser or the app and deals with the UI/UX and any other processing that can happen on the user's side such as dealing with cookies. We also need to have a server program that runs on a server dealing with the generation of content of a web page, deals with queries and manages the information flow over the sockets.

**This project can be mainly divided into three parts:**

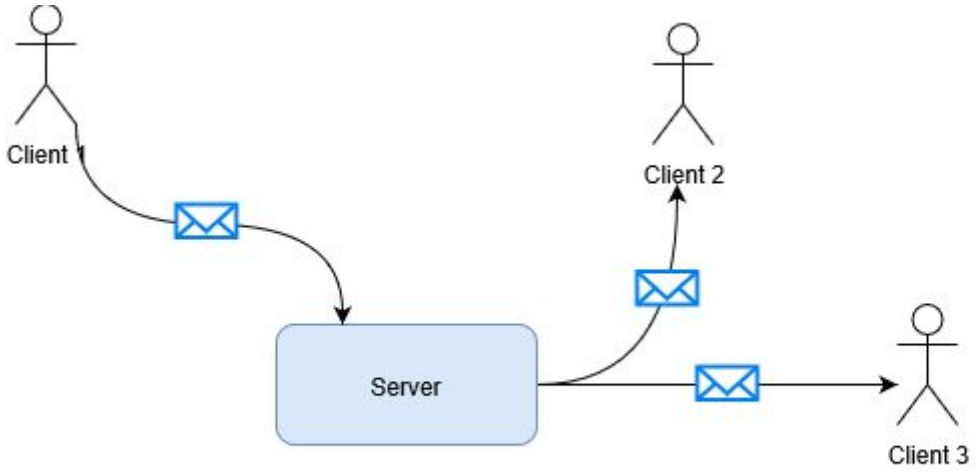
1. websocket from Django channels

2. Consumer and rooms

3. User authentication

## Creating the server

In order to build a chat application, we need a way to relay the messages sent by one user, to all the other users logged into the channel. The server acts as the message hub: accepting messages from the connected client applications, and sending them to all the other connected client applications.



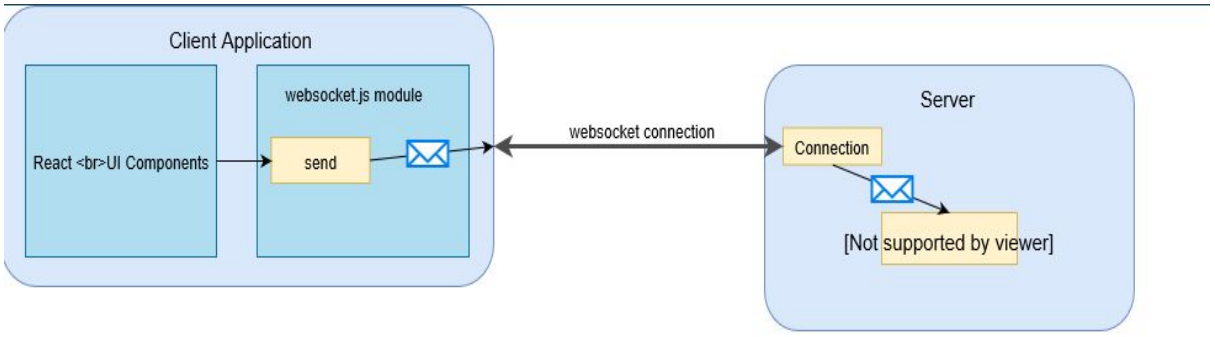

## Websocket connections(via Django Channels)

The majority of the websites we visit make HTTP API calls, which means the client sends a request to the server, and the server sends back a response. But, the communication can only be initiated by the client. This is a problem if the server ever wants to notify the client at any random time. Hence we use Websockets.

Unlike an HTTP call, a Websocket connection remains open as long as both the client and server choose not to close it. While the connection is open, messages can be exchanged both ways as:

## Sending messages

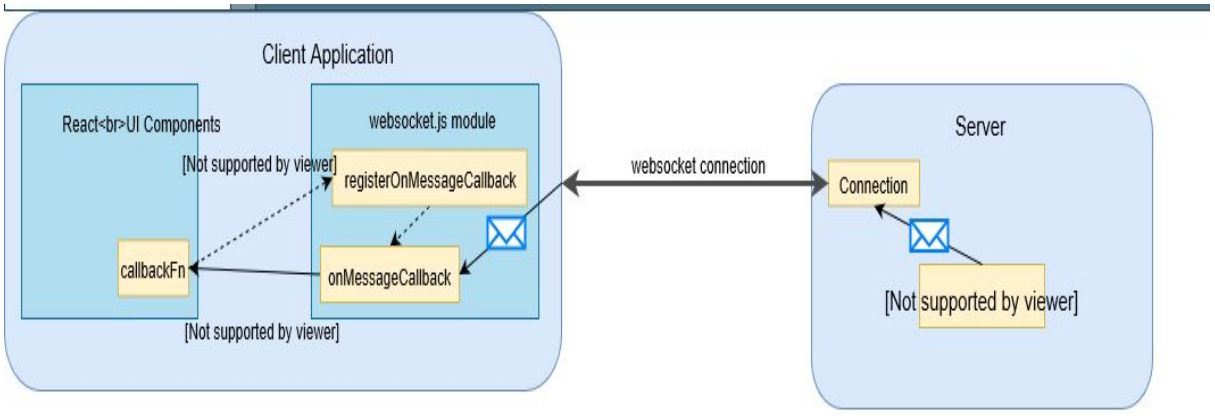
The `send` function is assigned only after the Websocket connection is established, and exported to allow the application code to call it, and in turn send messages.




## Receiving messages

Now to receive messages, the Websocket package will need to call some function that resides in the application code (the opposite direction as sending a message).

The `registerOnMessageCallback` function allows the application code to set the function that the websocket package will then call every time a new message is received. In this way, we have separated out the application code and the Websocket client interface



## Consumer and Rooms

**Consumer** is the one who handles connections between client and server. Consumers are the counterpart to Django views. Any user connecting to Chatterpro app, will be added to the ‘users’ group in that particular room and he will receive messages sent by the server. And when the user disconnects, the channel is removed from the group and the user will stop receiving messages.

If another user is connected to the same room and sends a message to the server, the server will check the members of that particular room and only send the message to users of that room. Server checks users and room via authentication before sending the messages.

Consumers will also display a list of users and send a message to the room when a user connects and disconnects. The message will include the user's username and connection status.

## User Authentication

Before establishing the connection to chatterpro app, the user has to choose the room name which he wants to connect. Then the server receives messages from the user and sends it to other users of that particular room. If a user is not connected to that particular room, then he will not receive messages sent by the server.

# RUNNING THE APPLICATION

When we run a server side application, we run it on a particular physical port e.g. 8080 or 3000. And to access the server side application, we use an IP. Similarly, when we log in to our browser and ask for a particular site, we send to the request our computer's IP as well as dynamically generated port number. So, we have four items which help us complete communication between our computer and the server and these four items are unique for every request.

1. **Server IP address** => It is hidden in the URL given to the client and known to the client.
2. **Server port** => It is also hidden in the URL given to the client.
3. **Client IP address** => Unique for every client
4. **Client port** => Unique for every client and is generated dynamically

When a client wants to connect to the server, a TCP socket is created to represent that connection at the server side. Then, the client sends a packet from the client IP address and from the unique client port number. When the server gets the packet on its own port number, it

stores the client IP address and the particular client port number. This separates that client’s traffic from all the other currently connected sockets. Server now triggers an event for that particular socket.

When we use WebSocket, it doesn't close that connection until the client says so. WebSocket connections start out with an HTTP connection and contain an “upgrade” header requesting the server to upgrade the protocol from HTTP to WebSocket. If the server agrees to the upgrade, then it returns a response that indicates that the protocol will be changed to the WebSocket protocol. In a way, both server and client agreed to change the way they were talking, from HTTP to WebSocket .
Meanwhile the same port is servicing other WebSocket as well as HTTP requests. After a WebSocket connection is established, server and client can talk bidirectionally and in any order; there is no concept of request and response.

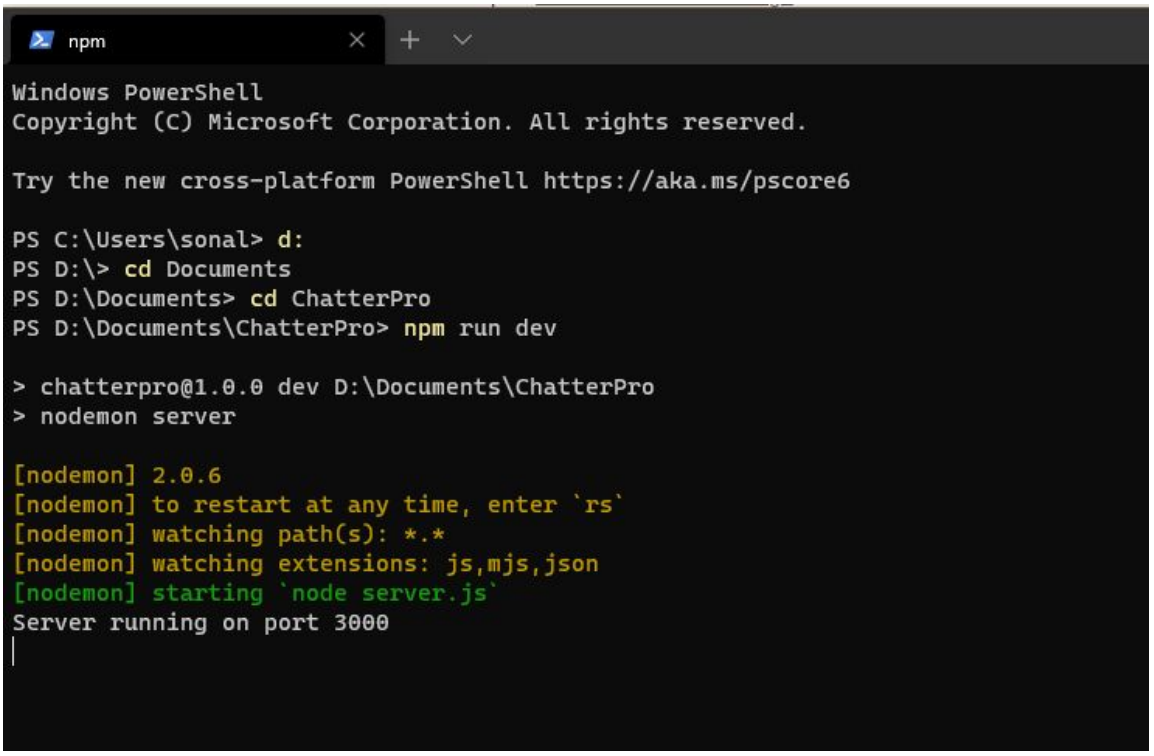
# PROJECT SOURCE CODE

The source code for the project has been uploaded on GitHub and is available here .To run the code on the local machine, one can download this project into their PC. Then, one can install the dependencies onto their repo, initialize the server and open localhost:3000 on their PC.

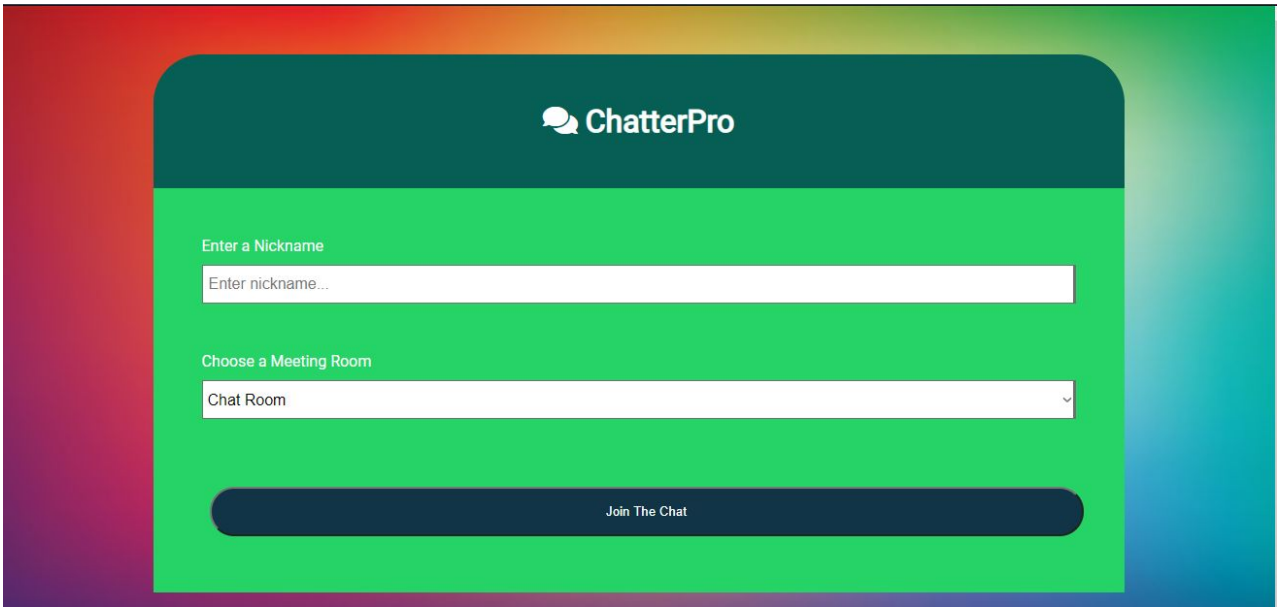
Alternatively, we have included some code snippets of the various files at Appendix - B.

# OUTPUT & RESULTS

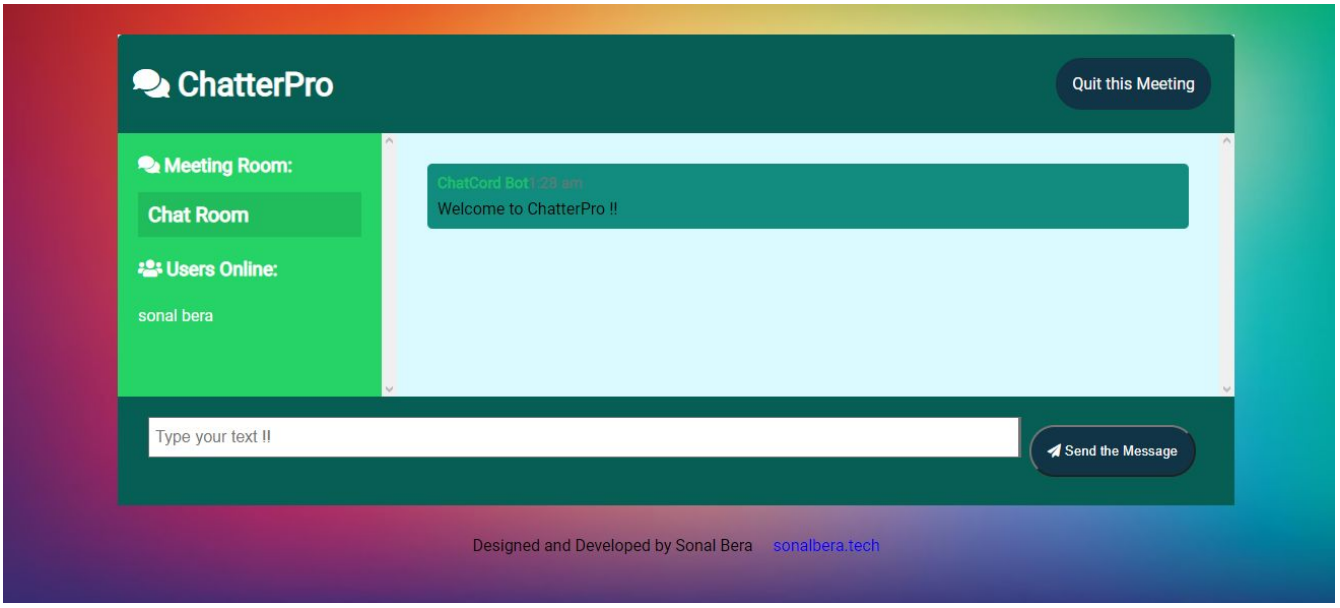
We start by starting the nodemon server from cmd



We then go to localhost:3000, as our port mentioned is 3000 and we are greeted with the login screen. Let us join with this username to the room ‘Chat Room’

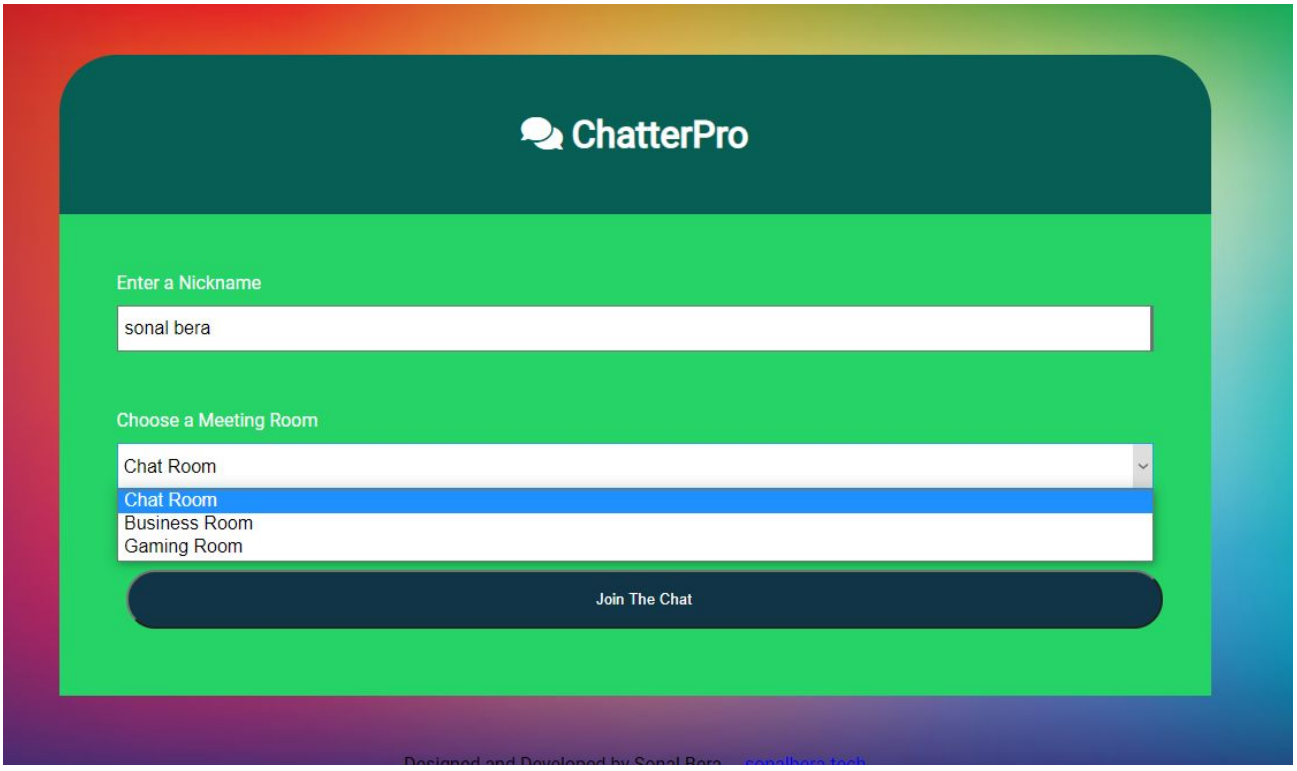



This is the chatroom, where only 1 user is online in this room.

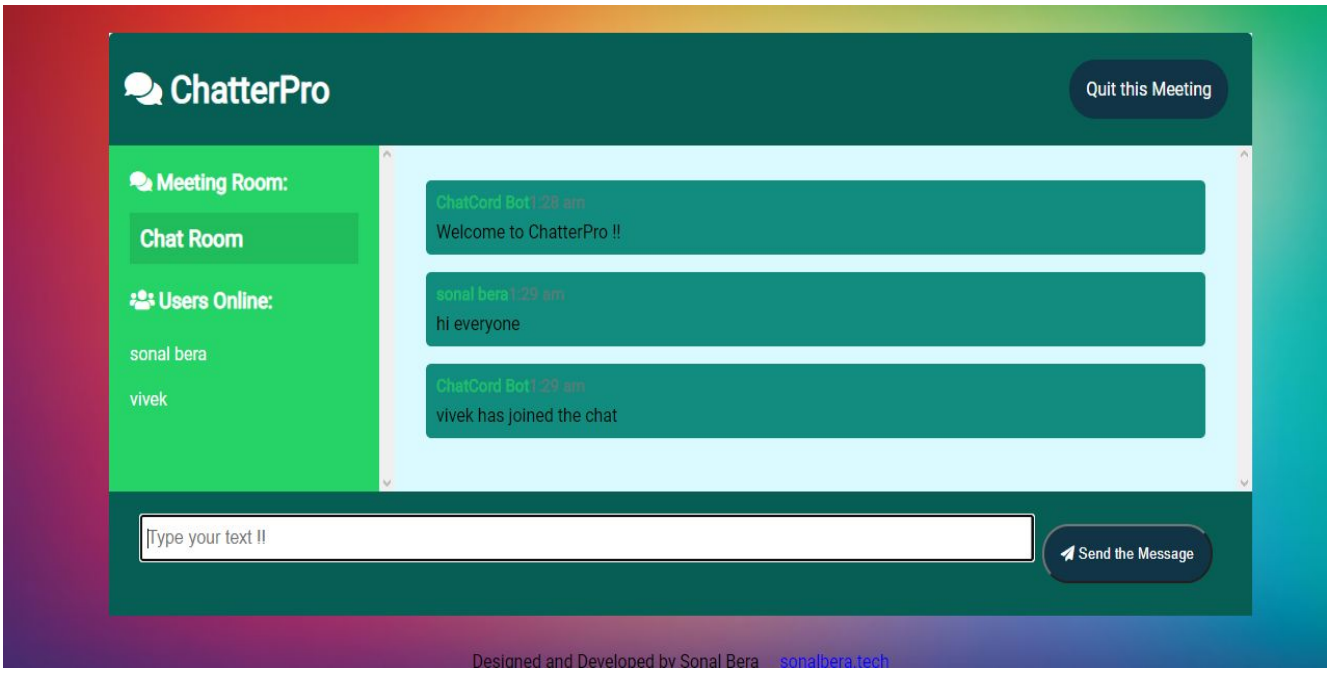


Let’s type a message and it gets broadcasted onto the window.

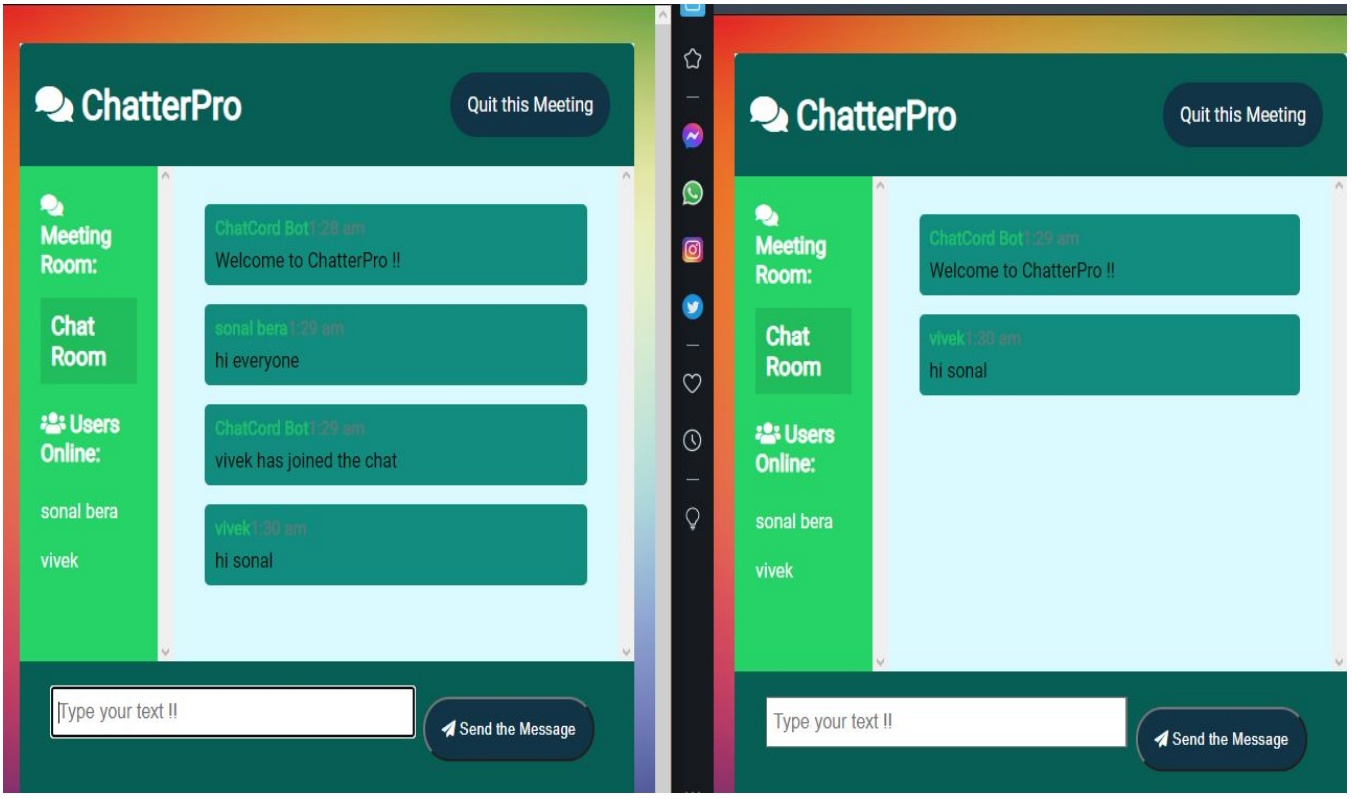

Let another user join the same room now. He will be now in the same room as the previous user.



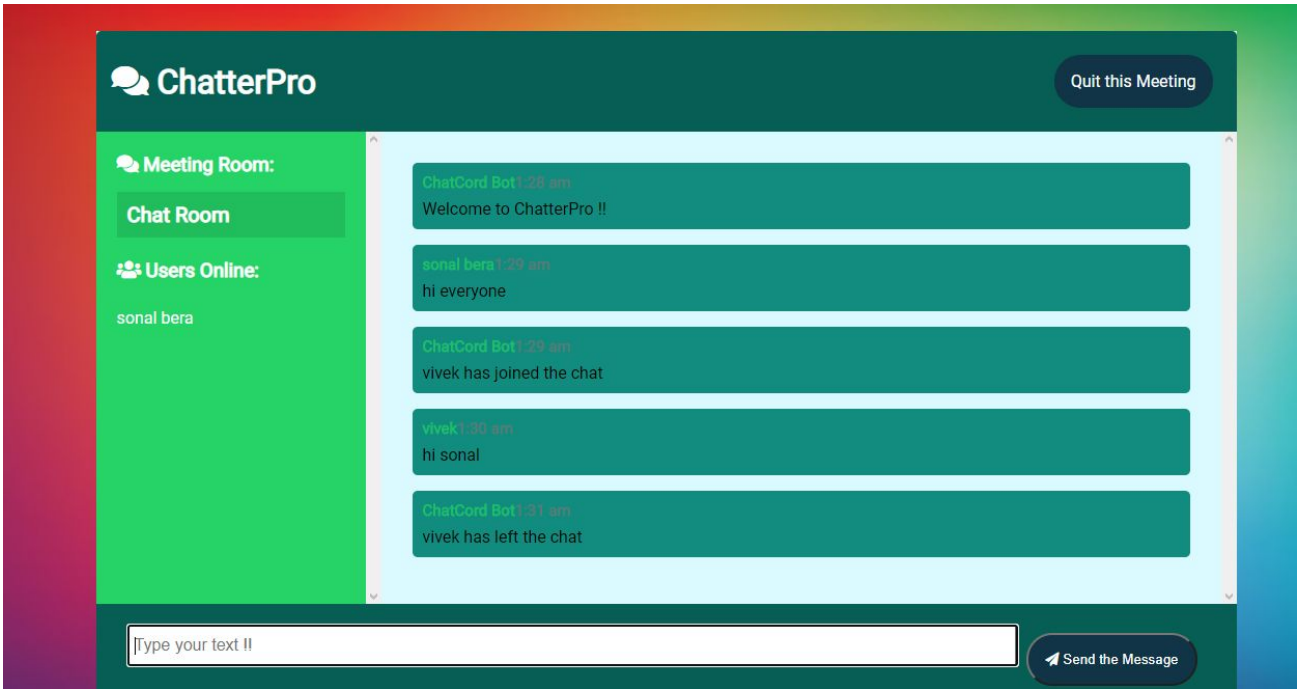
We get a notification in the chat box saying that this user has now joined the chat room.



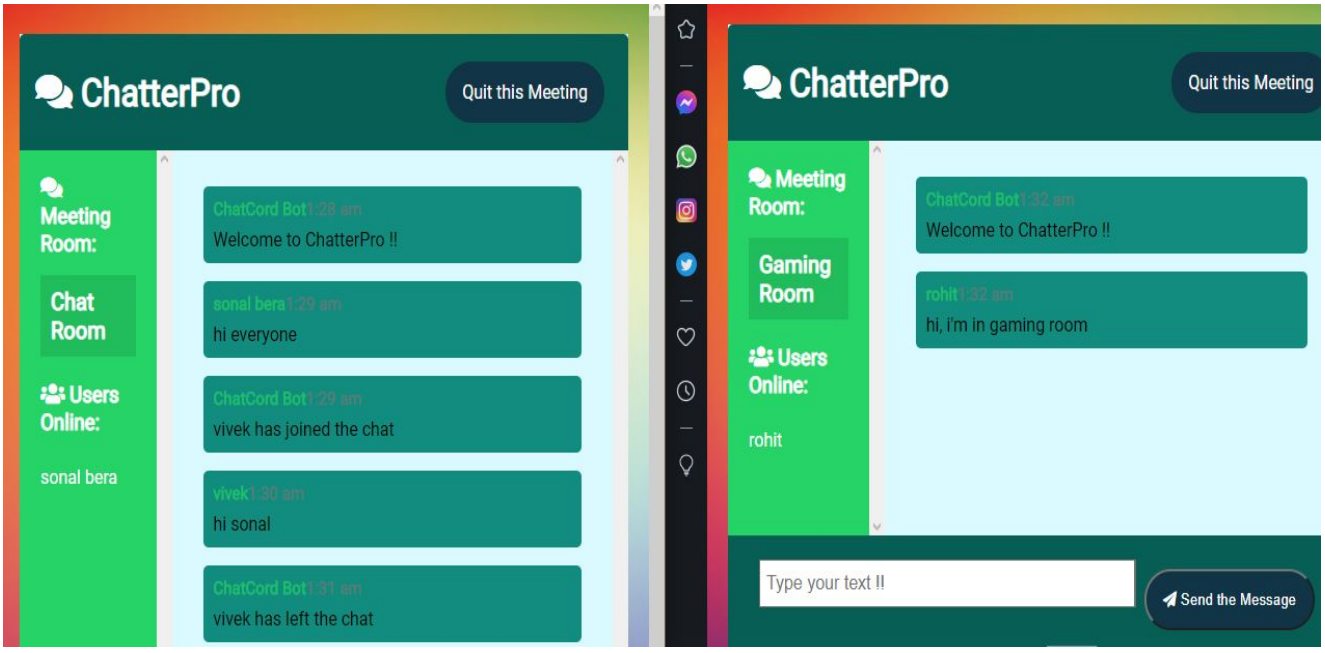

Below is an example of the chatting between the users of the same room.



Now, when a user leaves the room, we get another message saying that this user has left the chat.




Earlier, it was shown that this app has 3 rooms. So if a user joins another room, say ‘game’, then the user online in ‘chat’ room will not be greeted with any message or notifications as shown below.



# CONCLUSION

Our app has been created to perform the task as a chatting application. Based on the concepts of WebSockets, this app is a basic start into the creation of famous messaging apps of today such as WhatsApp, Messenger, Telegram and so on.

We have developed this chat app using websocket via Django channels and JS as a frontend. This web application is on localhost, efficient, and easily maintainable for a large number of clients and rooms. We have used Django as a backend. Its frontend features are easily customizable.

Django Channels is a project to handle WebSockets, chat protocols, IoT protocols, and more. It's built on a Python specification called ASGI (*Asynchronous Server Gateway Interface*). ASGI is a spiritual successor to WSGI, intended to provide a standard interface between async-capable Python web servers, frameworks, and applications.

As of now our app runs on localhost, but further enhancements can make it ready to be hosted online to be able to let users chat over the internet from any location.

# APPENDIX - A

## References


1. Client-server relation - techterms.com

2. javascript.com

3. Nodemon server - nodemon.io

4. Web socket API - Mozilla Developers https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

5. https://channels.readthedocs.io/en/stable/tutorial/index.html

6. https://blog.hubspot.com/marketing/web-design-html-css-javascript

7. Why Use Django - https://djangostars.com/blog/why-we-use-django-framework/

8. Real Time Django : https://blog.heroku.com/in_deep_with_django_channels_the_future_of_real_time_apps_in_django

9. Build your own real time app: https://www.freecodecamp.org/news/building-a-chat-application-with-mean-stack-637254d1136d/

10. WebSockets or HTTP/HTTPS: https://medium.com/platform-engineer/web-api-design-35df8167460

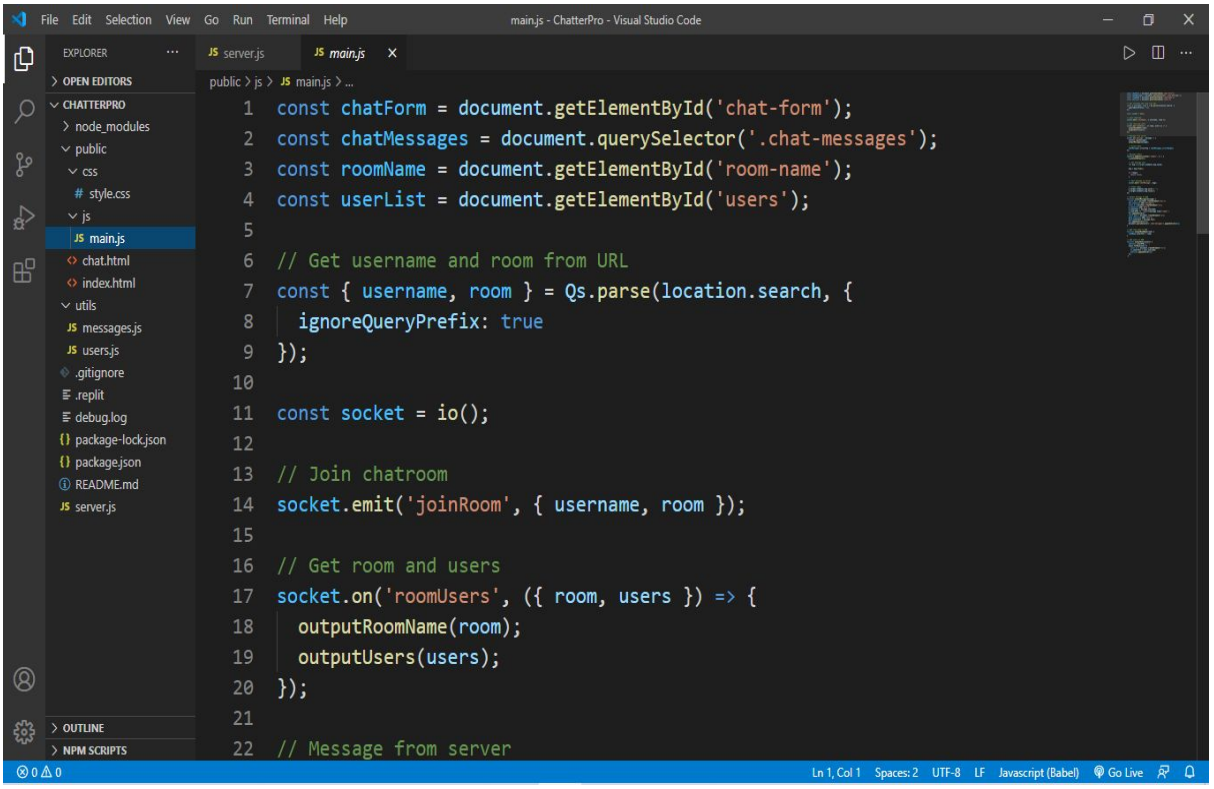
# APPENDIX - B

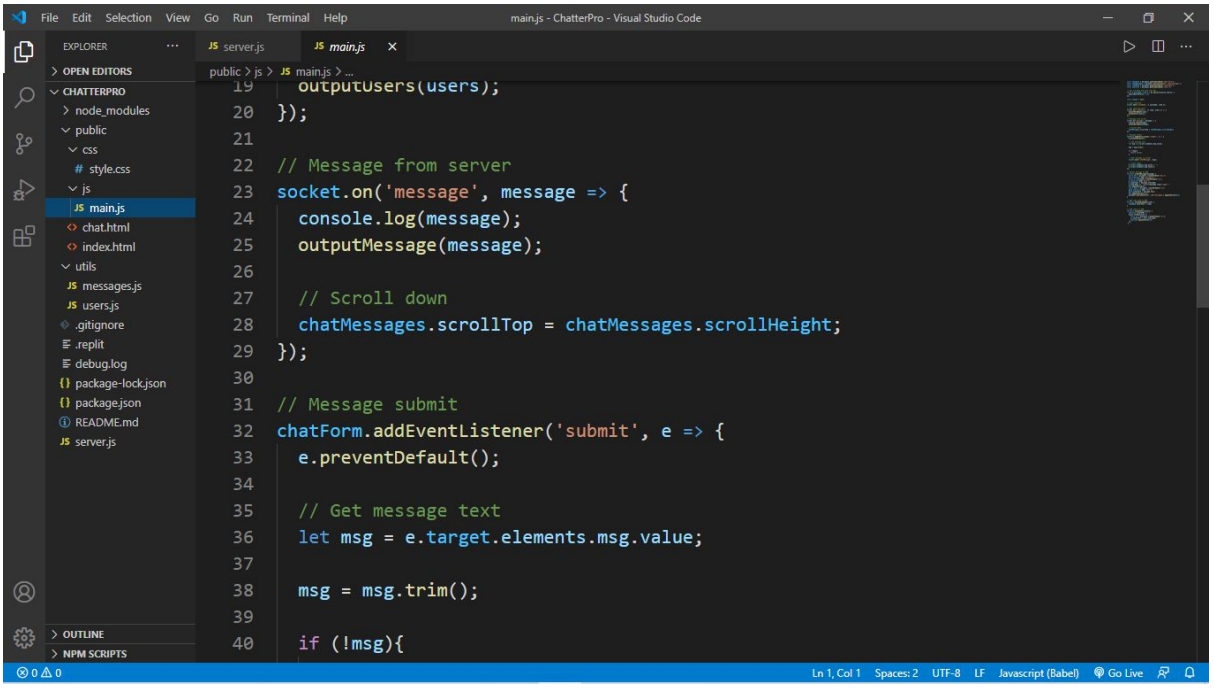
The source code for the project has been uploaded on GitHub and is available here .

## CODE SNIPPETS

```javascript
const path = require('path');
const http = require('http');
const express = require('express');
const socketio = require('socket.io');
const formatMessage = require('./utils/messages');
const {
  userJoin,
  getCurrentUser,
  userLeave,
  getRoomUsers
} = require('./utils/users');

const app = express();
const server = http.createServer(app);
const io = socketio(server);

// Set static folder
app.use(express.static(path.join(__dirname, 'public')));

const botName = 'ChatCord Bot';

// Run when client connects
```

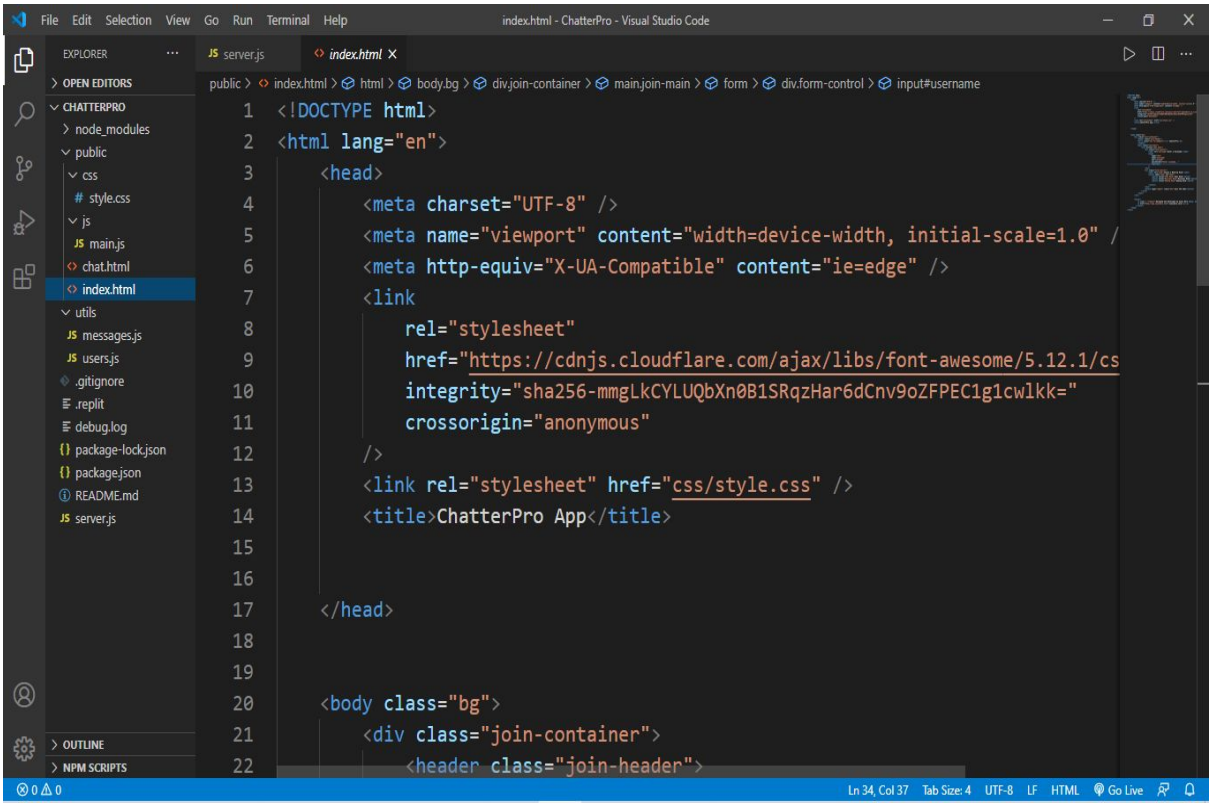
```javascript
app.use(express.static(path.join(__dirname, 'public')));

const botName = 'ChatCord Bot';

// Run when client connects
io.on('connection', socket => {
  socket.on('joinRoom', ({ username, room }) => {
    const user = userJoin(socket.id, username, room);

    socket.join(user.room);

    // Welcome current user
    socket.emit('message', formatMessage(botName, 'Welcome to ChatterPro !!'));

    // Broadcast when a user connects
    socket.broadcast
      .to(user.room)
      .emit(
        'message',
        formatMessage(botName, `${user.username} has joined the chat`)
      );
```


File Edit Selection View Go Run Terminal Help
main.js - ChatterPro - Visual Studio Code
EXPLORER
server.js
main.js
public > js > main.js > ...
OPEN EDITORS
CHATTERPRO
node_modules
public
css
style.css
js
main.js
chat.html
index.html
utils
messages.js
users.js
.gitignore
.replit
debug.log
package-lock.json
package.json
README.md
server.js
OUTLINE
NPM SCRIPTS
const chatForm = document.getElementById('chat-form');
const chatMessages = document.querySelector('.chat-messages');
const roomName = document.getElementById('room-name');
const userList = document.getElementById('users');
// Get username and room from URL
const { username, room } = Qs.parse(location.search, {
ignoreQueryPrefix: true
});
const socket = io();
// Join chatroom
socket.emit('joinRoom', { username, room });
// Get room and users
socket.on('roomUsers', ({ room, users }) => {
outputRoomName(room);
outputUsers(users);
});
// Message from server
Ln 1, Col 1 Spaces: 2 UTF-8 LF Javascript (Babel) Go Live

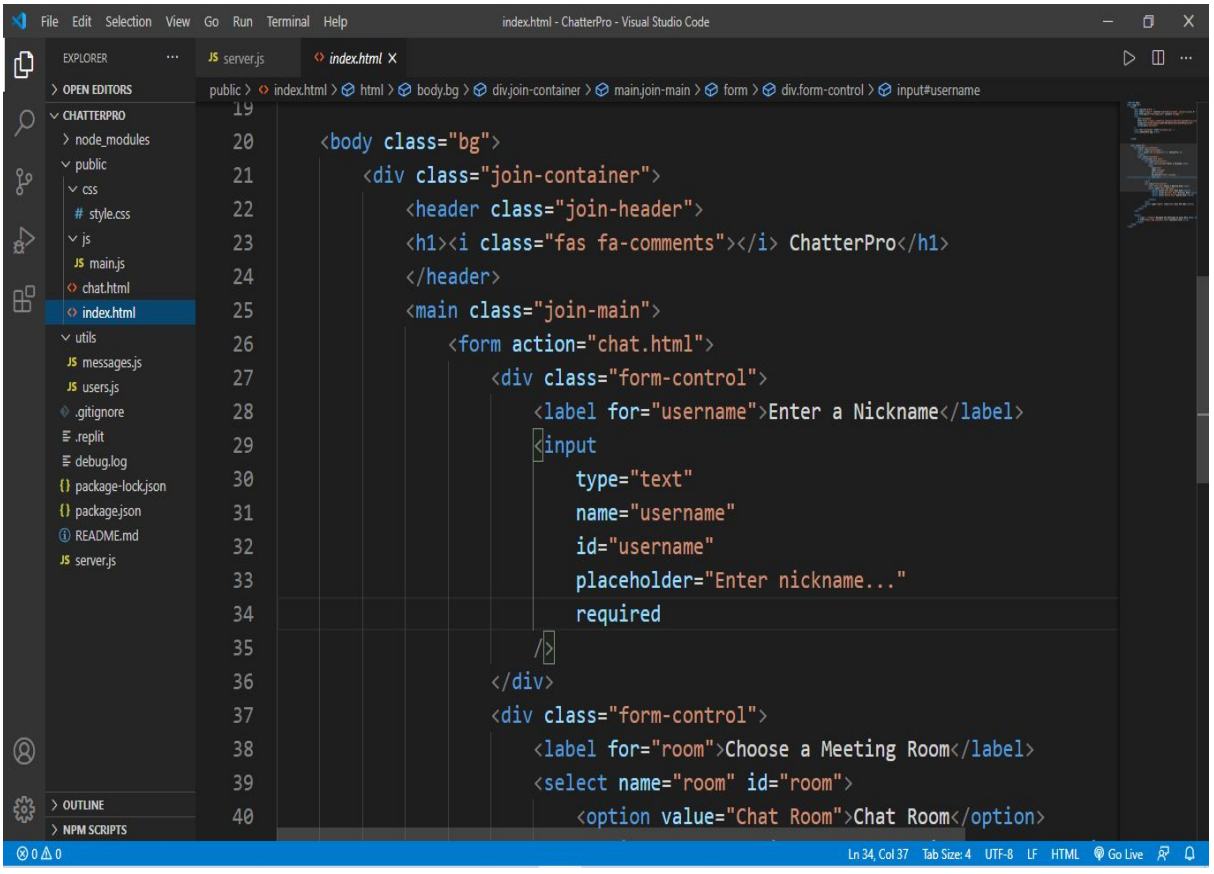

File Edit Selection View Go Run Terminal Help
main.js - ChatterPro - Visual Studio Code
EXPLORER
server.js
main.js
public > js > main.js > ...
OPEN EDITORS
CHATTERPRO
node_modules
public
css
style.css
js
main.js
chat.html
index.html
utils
messages.js
users.js
.gitignore
.replit
debug.log
package-lock.json
package.json
README.md
server.js
OUTLINE
NPM SCRIPTS
outputUsers(users);
});
// Message from server
socket.on('message', message => {
console.log(message);
outputMessage(message);
// Scroll down
chatMessages.scrollTop = chatMessages.scrollHeight;
});
// Message submit
chatForm.addEventListener('submit', e => {
e.preventDefault();
// Get message text
let msg = e.target.elements.msg.value;
msg = msg.trim();
if (!msg){
Ln 1, Col 1 Spaces: 2 UTF-8 LF Javascript (Babel) Go Live

```javascript

  // Clear input
  e.target.elements.msg.value = '';
  e.target.elements.msg.focus();
});

// Output message to DOM
function outputMessage(message) {
  const div = document.createElement('div');
  div.classList.add('message');
  const p = document.createElement('p');
  p.classList.add('meta');
  p.innerText = message.username;
  p.innerHTML += `<span>${message.time}</span>`;
  div.appendChild(p);
  const para = document.createElement('p');
  para.classList.add('text');
  para.innerText = message.text;
  div.appendChild(para);
  document.querySelector('.chat-messages').appendChild(div);
}

```

```javascript
  div.appendChild(para);
  document.querySelector('.chat-messages').appendChild(div);
}

// Add room name to DOM
function outputRoomName(room) {
  roomName.innerText = room;
}

// Add users to DOM
function outputUsers(users) {
  userList.innerHTML = '';
  users.forEach(user=>{
    const li = document.createElement('li');
    li.innerText = user.username;
    userList.appendChild(li);
  });
}

```


File Edit Selection View Go Run Terminal Help
index.html - ChatterPro - Visual Studio Code
EXPLORER
OPEN EDITORS
CHATTERPRO
node_modules
public
css
style.css
js
main.js
chat.html
index.html
utils
messages.js
users.js
.gitignore
.replit
debug.log
package-lock.json
package.json
README.md
server.js
OUTLINE
NPM SCRIPTS
server.js
index.html
public > index.html > html > body.bg > div.join-container > main.join-main > form > div.form-control > input#username
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" /
<meta http-equiv="X-UA-Compatible" content="ie=edge" />
<link
rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.12.1/cs
integrity="sha256-mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPEC1g1cwlkk="
crossorigin="anonymous"
/>
<link rel="stylesheet" href="css/style.css" />
<title>ChatterPro App</title>
</head>
<body class="bg">

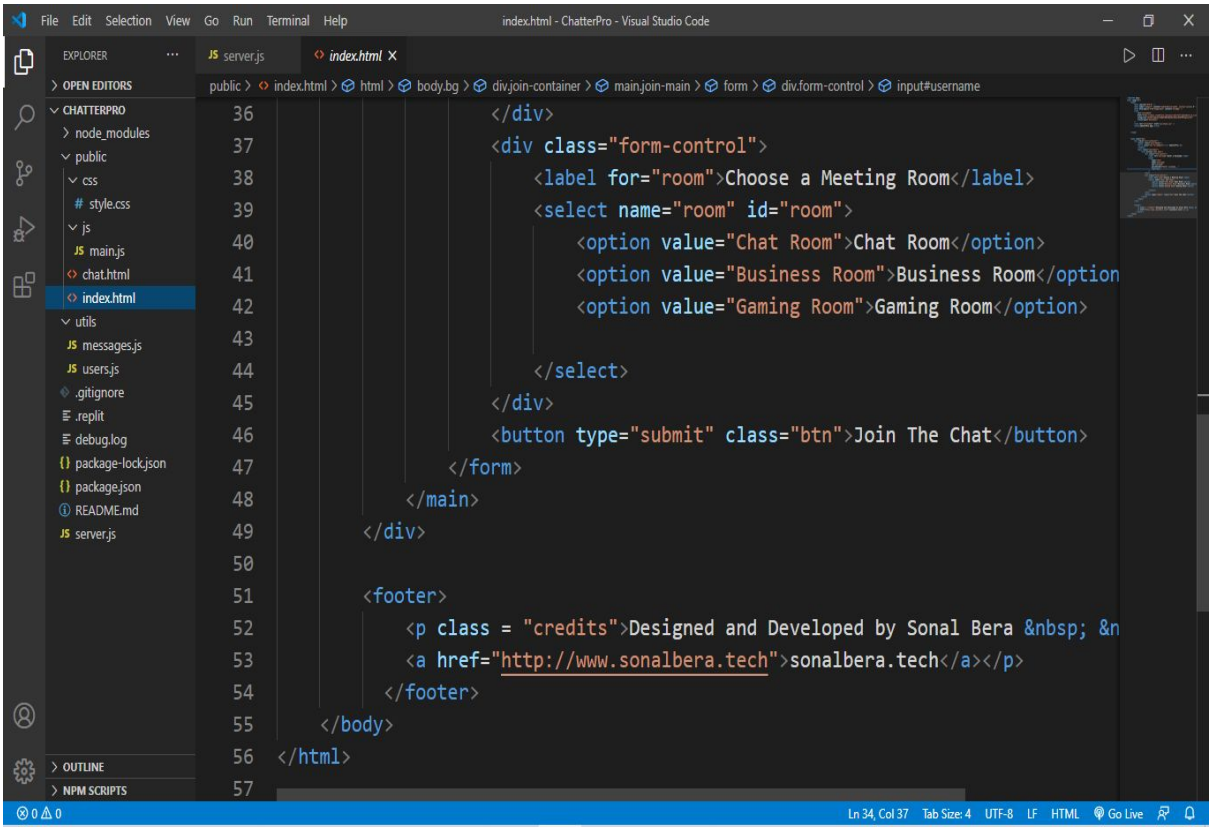
<header class="join-header">
Ln 34, Col 37 Tab Size: 4 UTF-8 LF HTML Go Live


File Edit Selection View Go Run Terminal Help
index.html - ChatterPro - Visual Studio Code
EXPLORER
OPEN EDITORS
CHATTERPRO
node_modules
public
css
style.css
js
main.js
chat.html
index.html
utils
messages.js
users.js
.gitignore
.replit
debug.log
package-lock.json
package.json
README.md
server.js
OUTLINE
NPM SCRIPTS
server.js
index.html
public > index.html > html > body.bg > div.join-container > main.join-main > form > div.form-control > input#username
<body class="bg">

<header class="join-header">
<h1> ChatterPro</h1>
</header>
<main class="join-main">
<form action="chat.html">

<label for="username">Enter a Nickname</label>
<input
type="text"
name="username"
id="username"
placeholder="Enter nickname..."
required
/>

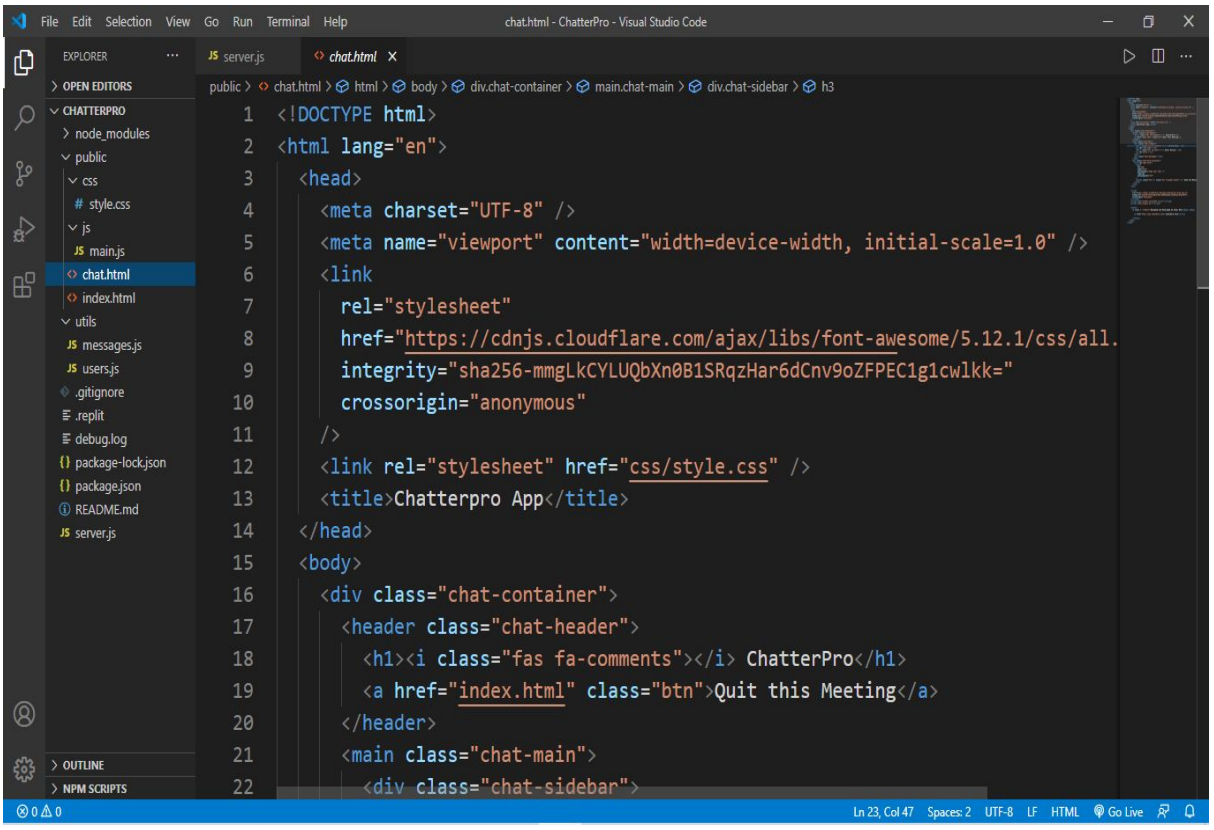
<label for="room">Choose a Meeting Room</label>
<select name="room" id="room">
<option value="Chat Room">Chat Room</option>
Ln 34, Col 37 Tab Size: 4 UTF-8 LF HTML Go Live


index.html - ChatterPro - Visual Studio Code
public > index.html > html > body.bg > div.join-container > main.join-main > form > div.form-control > input#username

<label for="room">Choose a Meeting Room</label>
<select name="room" id="room">
<option value="Chat Room">Chat Room</option>
<option value="Business Room">Business Room</option
<option value="Gaming Room">Gaming Room</option>
</select>

<button type="submit" class="btn">Join The Chat</button>
</form>
</main>

<footer>
Designed and Developed by Sonal Bera   &n
<a href="http://www.sonalbera.tech">sonalbera.tech</a>
</footer>
</body>
</html>
Ln 34, Col 37 Tab Size: 4 UTF-8 LF HTML Go Live


chat.html - ChatterPro - Visual Studio Code
public > chat.html > html > body > div.chat-container > main.chat-main > div.chat-sidebar > h3
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<link
rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.12.1/css/all.
integrity="sha256-mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPEC1g1cwlkk="
crossorigin="anonymous"
/>
<link rel="stylesheet" href="css/style.css" />
<title>Chatterpro App</title>
</head>
<body>

<header class="chat-header">
<h1> ChatterPro</h1>
<a href="index.html" class="btn">Quit this Meeting</a>
</header>
<main class="chat-main">

Ln 23, Col 47 Spaces: 2 UTF-8 LF HTML Go Live

```html
<div class="chat-container">
  <header class="chat-header">
    <h1><i class="fas fa-comments"></i> ChatterPro</h1>
    <a href="index.html" class="btn">Quit this Meeting</a>
  </header>
  <main class="chat-main">
    <div class="chat-sidebar">
      <h3><i class="fas fa-comments"></i> Meeting Room: </h3>
      <h2 id="room-name"></h2>
      <h3><i class="fas fa-users"></i> Users Online: </h3>
      <ul id="users"></ul>
    </div>
    <div class="chat-messages"></div>
  </main>
  <div class="chat-form-container">
    <form id="chat-form">
      <input
        id="msg"
        type="text"
        placeholder="Type your text !!"
        required
        autocomplete="off"
```

```html
        />
        <button class="btn"><i class="fas fa-paper-plane"></i> Send the Messa
    </form>
  </div>
</div>

<script
  src="https://cdnjs.cloudflare.com/ajax/libs/qs/6.9.2/qs.min.js"
  integrity="sha256-TDxXjkAUay70ae/QJBEpGKkpVslXaHHayklIVglFRT4="
  crossorigin="anonymous"
></script>
<script src="/socket.io/socket.io.js"></script>
<script src="js/main.js"></script>

<footer>
  <p class = "credits">Designed and Developed by Sonal Bera    

    <a href="http://www.sonalbera.tech">sonalbera.tech</a></p>

  </footer>
</body>
</html>
```

users.js - ChatterPro - Visual Studio Code

```javascript
const users = [];

// Join user to chat
function userJoin(id, username, room) {
  const user = { id, username, room };

  users.push(user);

  return user;
}

// Get current user
function getCurrentUser(id) {
  return users.find(user => user.id === id);
}

// User leaves chat
function userLeave(id) {
  const index = users.findIndex(user => user.id === id);

  if (index !== -1) {
    return users.splice(index, 1)[0];
```

users.js - ChatterPro - Visual Studio Code

```javascript
function userLeave(id) {
  const index = users.findIndex(user => user.id === id);

  if (index !== -1) {
    return users.splice(index, 1)[0];
  }
}

// Get room users
function getRoomUsers(room) {
  return users.filter(user => user.room === room);
}

module.exports = {
  userJoin,
  getCurrentUser,
  userLeave,
  getRoomUsers
};
```

messages.js - ChatterPro - Visual Studio Code

utils > messages.js > ...

```javascript
const moment = require('moment');

function formatMessage(username, text) {
  return {
    username,
    text,
    time: moment().format('h:mm a')
  };
}

module.exports = formatMessage;
```

package.json - ChatterPro - Visual Studio Code

package.json > name

```json
{
  "name": "chatterpro",
  "version": "1.0.0",
  "description": "ChatterPro- Realtime chatting application",
  "main": "server.js",
  "scripts": {
    "start": "node server",
    "dev": "nodemon server"
  },
  "author": "Sonal Bera",
  "license": "MIT",
  "dependencies": {
    "express": "^4.17.1",
    "moment": "^2.29.1",
    "socket.io": "^2.3.0"
  },
  "devDependencies": {
    "nodemon": "^2.0.6"
  }
}
```