International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Web Chat App

Garvit Chand[1], Sonam Ram[2]

[1]B Tech Student, Dept. of CSE, Raipur Institute of Technology, Raipur, Chhattisgarh, India chandgarvit@gmail.com
[2]Assistant Professor Dept. of CSE, Raipur Institute of Technology, Raipur, Chhattisgarh, India Sonam2911@gmail.com

ABSTRACT

Teleconferencing, sometimes known as chatting, is a technique for bringing people and ideas together over long distances through technology.

Despite the fact that the technology has been around for a long time, it was only recently accepted. A chat server is an example of our project.

Different users join a shared room in our web-based Chat App to talk with one another, and our web-based Chat App also offers a location-sharing feature.

Keywords : Teleconferencing; web-based Chat; location-sharing feature.

INTRODUCTION

For my project, I decided to use Node to build a real-time web application. We choose WebSockets because the WebSocket protocol allows for real-time bi-directional communication, which is ideal for the chat application. To make my project more efficient and lively, I made a few tweaks.

The chat software is designed in such a way that a user can join any room by typing the room id on the front screen, and if another user joins the room with the same code, the user will be able to join the same room as well.

People in rooms are immediately visible to everyone else in the room, and whenever a new person enters, a fresh broadcast message alerts everyone that a new person has entered the room.

Another thing I did was share the location with the individuals in the room, which I accomplished using Google API. It is completely open source.

Everyone has access to the source code and may understand what's going on behind the scenes as well as contribute to it.

As a result, we intended to build clean, scalable code that adhered to the most popular patterns and conventions for each of the languages and necessary libraries.

TECHNOLOGY USED

To create this program, we utilized Visual Studio Code as the editor. We built this Web Application with HTML and CSS for the front end of the Chat app and node.js for the backend. Aside from that, we used a slew of npm modules during the development process. We utilized "Heroku" to deploy our website and hosted our source code on GitHub. We attempted to use a considerable quantity of branched codes in order to make it easier for the readers to understand.

Node.js: It is a runtime environment that runs on a V8 engine. Outside the web browser, Node.js executes the javascript code, It is an open-source, cross-platform, back-end JavaScript.

Socket.io: It enables communication from both the client side and server side, it is event-based communication. It focuses on reliability and speed.

Express.js: Express.js, or simply Express, is a free and open-source Node.js back-end web application framework licensed under the MIT License. The fundamental goal of Express is to provide server-side logic for web applications.

Mustache: It is a web template system. The implementations of the Mustache are available for Actionscript.

Moment: Moment is a date package written in JavaScript that lets you parse, validate, manipulate, and format dates. In our web-based application, it was used to render the time stamp.

Npm package bad-words: It's a npm module for filtering profanity

DESIGN DESCRIPTION

WEBSOCKETS

The WebSocket protocol allows for real-time bi-directional communication, making it ideal for the chat application.

```
npm i socket.io@2.20
```

SOCKET.IO

Everything you need to get started with a Node-based WebSocket server is included in Socket.io. Socket.io can be used on its own or in tandem with Express. Express and Socket.io will be installed because the chat application will be serving client-side assets.

```
const path = require('path')
const http = require('http')
const express = require('express')
const socketio = require('socket.io')

// Create the Express application
const app = express()

// Create the HTTP server using the Express app
const server = http.createServer(app)

// Connect socket.io to the HTTP server
const io = socketio(server)

const port = process.env.PORT || 3000
const publicDirectoryPath = path.join(__dirname, '../public')

app.use(express.static(publicDirectoryPath))

// Listen for new connections to Socket.io
io.on('connection', () => {
    console.log('New WebSocket connection')
})

server.listen(port, () => {
    console.log(`Server is up on port ${port}!`)
```

The server in the preceding example employs io.on, which is offered by Socket.io. on enables the server to listen for and respond to events.

On the client, Socket.io is also used to connect to the server. By invoking io, your client-side JavaScript can then connect to the Socket.io server. The client-side Socket.io library provides io. Calling this function will establish the connection and activate the server's connection event handler.

SOCKET.IO EVENTS

Every event has two sides: the sender and the receiver. The client is the receiver if the server is the sender. The server is the receiver if the client is the sender.

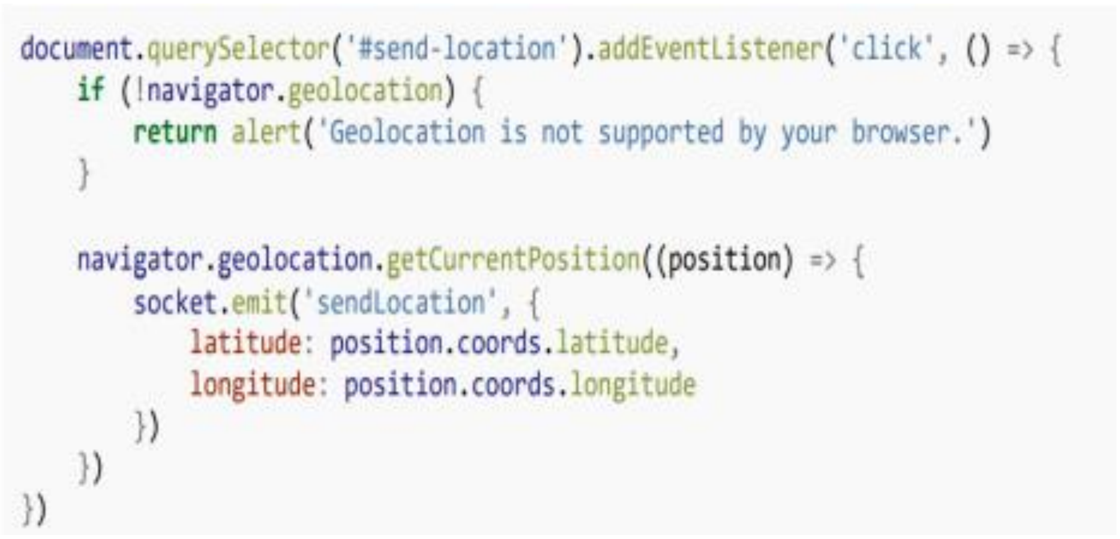
```
const socket = io()

// Listen for "countUpdated"
socket.on('countUpdated', (count) => {
    console.log('The count has been updated!', count)
})

document.querySelector('#increment').addEventListener('click', () => {
    // Emit "increment
    socket.emit('increment')
})
```

Using emit, events can be transmitted from the sender. The receiver can receive events by using on. The example below demonstrates how to utilize this pattern to develop a simple counter application.

On is used by the client-side code to listen for the countUpdated event. When that event is received, a message containing the current count will be logged. The emit function is also used by the client-side code to convey the increment event.

SHARING LOCATION

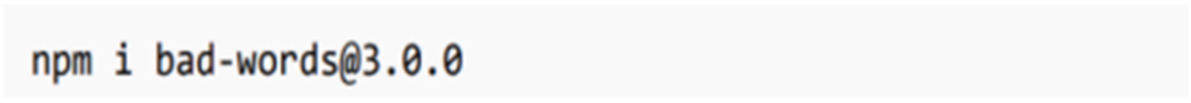
Using client-side JavaScript, you may retrieve a user's location using the Geolocation API. When a user grants you permission to view their location, you can share it with everyone else in the chat group.



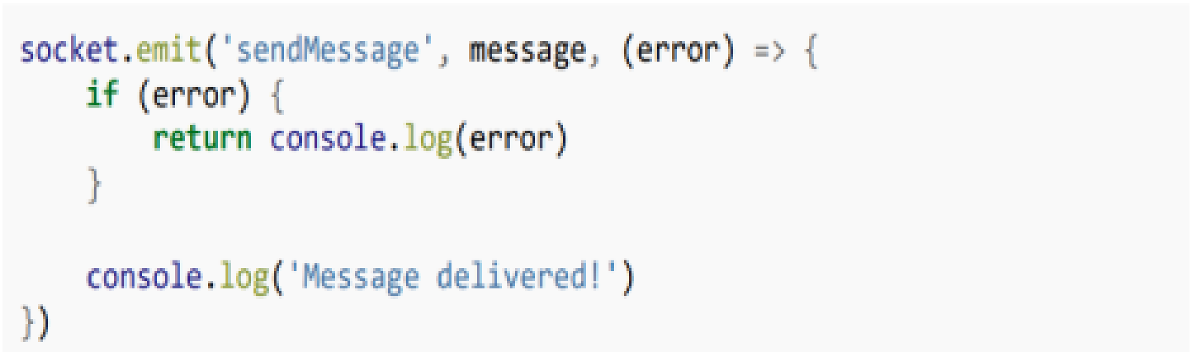
```
document.querySelector('#send-location').addEventListener('click', () => {
    if (!navigator.geolocation) {
        return alert('Geolocation is not supported by your browser.')
    }

    navigator.geolocation.getCurrentPosition((position) => {
        socket.emit('sendLocation', {
            latitude: position.coords.latitude,
            longitude: position.coords.longitude
        })
    })
})
```

ADDING BAD-WORD NPM LIBRARY

If profane language is recognized, the callback function is invoked with an error. The argument would be given to the client, who would see the error. If no foul language is detected, the callback is invoked without any arguments. This informs the client that the message has been successfully processed.



```
npm i bad-words@3.0.0
```



```
socket.emit('sendMessage', message, (error) => {
    if (error) {
        return console.log(error)
    }

    console.log('Message delivered!')
})
```

```
socket.on('sendMessage', (message, callback) => {
    const filter = new Filter()

    if (filter.isProfane(message)) {
        return callback('Profanity is not allowed!')
    }

    io.emit('message', message)
    callback()
})
```

RENDERING MESSAGES

The messages were rendered using Mustache.

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/mustache.js/3.0.1/mustache.min.js
"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.22.2/moment.min.js"><
/script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/qs/6.6.0/qs.min.js"></script>
```

ADDING TIME STAMPS

Every communication will have a timestamp. This timestamp represents the time when the server distributed the message to everyone in the chat application. The timestamp will be generated by the server, preventing the client from lying about when a message was sent. 13

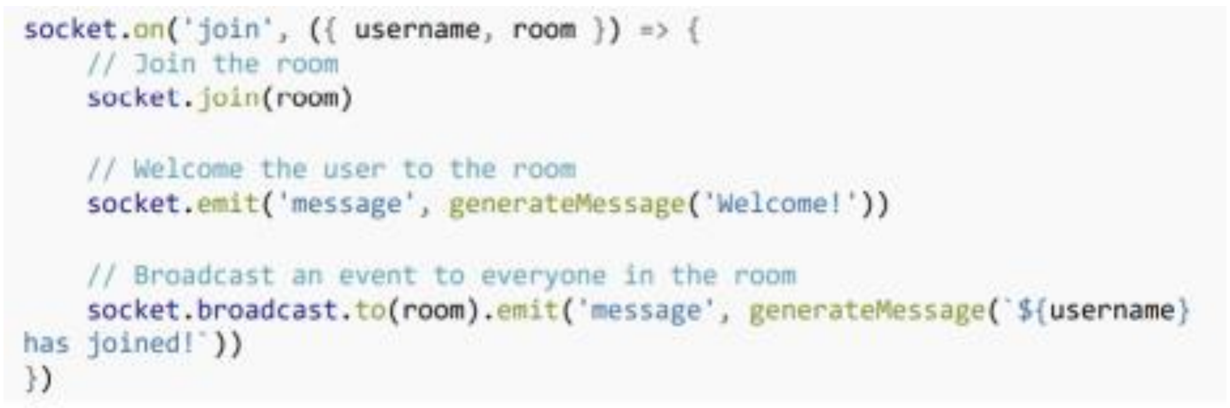
The client can format the timestamp before rendering it once the server transmits the message and timestamp to it. The Moment package makes it simple to format timestamps to our specifications.

```
// The getTime method is used to get the timestamp
const timestamp = new Date().getTime()

console.log(timestamp) // Will print: 1549481884646
```

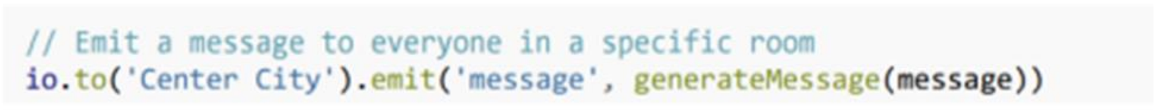
```
moment(message.createdAt).format('h:mm a')
```

SOCKET.IO ROOMS

It is the server's responsibility to add and remove users from rooms. By executing a socket, the server can add a user to a room. Join by entering the room name. The listener for join accepts the client's login and room name in the example below. socket. The function join(room) is then used to add the user to the room they wish to join.



```
socket.on('join', ({ username, room }) => {
    // Join the room
    socket.join(room)

    // Welcome the user to the room
    socket.emit('message', generateMessage('Welcome!'))

    // Broadcast an event to everyone in the room
    socket.broadcast.to(room).emit('message', generateMessage(`${username}
has joined!`))
})
```

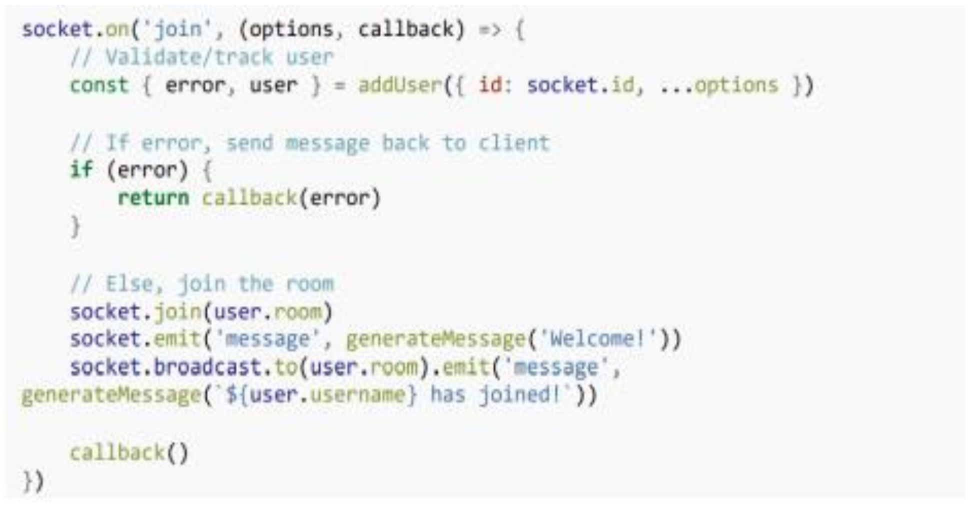


```
// Emit a message to everyone in a specific room
io.to('Center City').emit('message', generateMessage(message))
```

TRACKING USERS JOINING AND LEAVING

When a person tries to join a room, their username and the name of the room are supplied to addUser. The program will be able to track the user and validate the data as a result of this. If a problem occurs, the error message will be returned to the client, and the user will not be able to join the requests room. If a user is removed, a message is delivered to the entire chat room informing them that someone has gone.

To send errors back to the client, the server employs an acknowledgment. The client can configure the callback function and handle any issues that may arise. If an error occurs, the snippet below displays the error message before redirecting the user to the join page.



```
socket.on('join', (options, callback) => {
    // Validate/track user
    const { error, user } = addUser({ id: socket.id, ...options })

    // If error, send message back to client
    if (error) {
        return callback(error)
    }

    // Else, join the room
    socket.join(user.room)
    socket.emit('message', generateMessage('Welcome!'))
    socket.broadcast.to(user.room).emit('message',
generateMessage(`${user.username} has joined!`))

    callback()
})
```

RESULTS

- Learned frontend and backend web development.
- Learned Html,CSS,Javascript,Node.js etc.
- I Built a chat application as a project.

CONCLUSIONS AND DISCUSSIONS

In any app, there is always room for improvement. We are currently only working with text conversation and location sharing. There are other Android apps that fulfill similar functions as our project, but they are difficult to use and have complex interfaces.

This project aims to provide a web-based chat service with a high-quality user experience. We may be expanded in the future to incorporate features such as:

1. File Transfer
2. Voice Message
3. Video Message
4. Audio Call
5. Video Call
6. Connect it to a database, to save all the past messages.
7. Set up a login system and have the ability to have a friends list.
8. Add About and Profile sections.