A Dissertation Report

on

Realtime Web Chat App Project

Submitted in partial fulfilment of the

requirement for the award of the degree of

Bachelor of Technology

Under The Supervision of

Dr. Amit Kumar Goel

Assistant Professor

Submitted By

Shivang Gupta (18SCSE1010039)

Chandan Kumar Singh (18SCSE1010652

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GALGOTIAS UNIVERSITY, GREATER NOIDA

INDIA

December, 2021

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING

GALGOTIAS UNIVERSITY, GREATER NOIDA

CANDIDATE'S DECLARATION

I/We hereby certify that the work which is being presented in the thesis/project/dissertation, entitled **"WEB CHAT APP"** in partial fulfilment of the requirements for the award of the **Bachelor of Technology** submitted in the School of Computing Science and Engineering of Galgotias University, Greater Noida, is an original work carried out during the period of **September,2021 to December, 2021**, under the supervision of **Dr. Amit Kumar Goel Assistant Professor**, Department of Computer Science and Engineering/Computer Application and Information and Science, of School of Computing Science and Engineering , Galgotias University, Greater Noida

The matter presented in the thesis/project/dissertation has not been submitted by me/us for the award of any other degree of this or any other places.

Shivang Gupta, 18SCSE1010039

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

Supervisor Name

Designation

CERTIFICATE

The Final Thesis/Project/ Dissertation Viva-Voce examination of Shivang Gupta, 18SCSE1010039 has been held on _____ and his/her work is recommended for the award of Bachelor of Technology.

Signature of Examiner(s) **Signature of Supervisor(s)**

Signature of Project Coordinator **Signature of Dean**

Date: December, 2021 Place: Greater Noida

Table of Contents

Title	Page No
Candidates Declaration	2
Acknowledgement	4
Abstract	5
Introduction	6
Literature Survey	7
UML Diagram	10
Source Code	11
Conclusion	49
Reference	50

ACKNOWLEDGEMENT

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my

sincere thanks to all of them.

I am highly indebted to Dr. Amit Kumar Goel for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

I would like to express my gratitude towards my parents & member of Galgotias University for their kind co-operation and encouragement which help me in completion of this project.

I would like to express my special gratitude and thanks to industry persons for giving me such attention and time.

My thanks and appreciations also go to my colleague Chandan Singh in developing the project and people who have willingly helped me out with their abilities.

Abstract

Web Chat application is an element or a program on the Internet to impart straightforwardly among Internet clients who are on the web or who were similarly utilizing the web. Talk applications permit clients to convey despite the fact that from a significant stretch. Thusly, this talk application should be ongoing and multi-stage to be utilized by numerous clients. The improvement of data and correspondence innovations are quickly made through in-memory database. This visit application in the assembling starts with the assortment of significant information that will be shown in the web and versatile adaptations. The programming language used to construct worker is Node.js with express structure. Users can chat in real time in a room by simply signing in and choosing which room they want to join. Client can share messages and files in **real-time** within theroom. The in-memory database will store data that rely primarily on memory for data storage, in contrast to databases that store data on disk or SSDs. In-memory data stores are designed to enable minimal response times by eliminating the need to access disks. Because all data is stored and managed exclusively in main memory, in-memory databases risk losing data upon a process or server failure. In-memory databases can persist data on disks by storing each operation in a log or bytaking snapshot

Introduction

Web Chat App is a social-networking tool that leverages on technology advancement thereby allowing its users communicate and share media. It can be used for messaging, placing voice messages, making voice and video calls, share updates and photos.

BACKGROUND AND MOTIVATION

For example- WhatsApp has the potential to become a widely used socializing app in all over world.

STATEMENT OF PROBLEMS

Starting any Web App or service has many problems but one of the main problems is which tool, language, stack or framework to build one's service or application on. As building a real time application has to do with slow latency message delivery which in turn means latency, data transfer size over the network must be as low as possible. Other problems include SMS messaging, cross platform permissions for android and IOS.

AIMS AND OBJECTIVE-

MESSAGING

One of the primary use of Web Chat App is messaging. Just like other social apps, you have a list of conversations that you're engaged in. This feature is pivotal as you can add people in a variety of ways aside the conventional way of details collection. When fully operational, you

will be amazed how individuals will have to scan their phones during details collection. Users of Web Chat App can also use a phone number to add a person to their contact list and even searchfor people nearby.

Literature Survey

Introduction

Messaging apps now have more global users than traditional social networks—which mean they will play an increasingly important role in the distribution of digital information in the future. In 2016, over 2.5 billion people used at least one messaging app. That's one-third of theworld's entire population, with users ranging from various age grades. Today, it's common place for offices to use a messaging app for internal communication in order to coordinate meetings, share pitch decks, and plan happy hours. And with the latest bot technology, chat apps are becoming a hub for employees to do work in their apps without leaving the chat console. For many people, chat apps are a given part of their workday. But how did these chat apps become so popular?

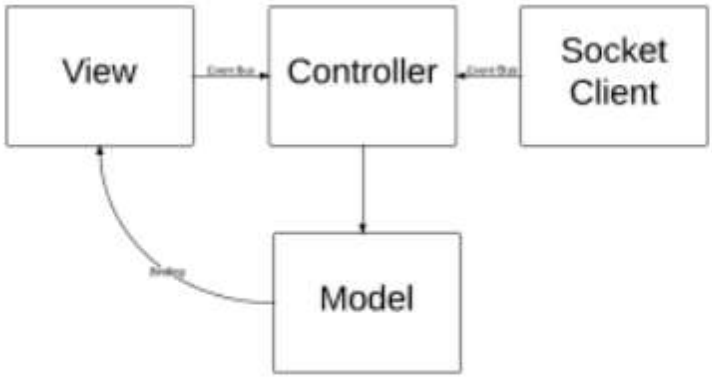
Historical Overview

INSTANT MESSAGING: CHILD OF THE 90'S

Chat apps (and their subsets, chat rooms) bring to remembrance images of the 1990s, with its dial-up internet and classic sitcoms, however, commercial chat apps date back to the 1980s. CompuServe released CB Simulator in 1980, and 1985 brought the launch of Commodore's Quantum Link (also known as Q-Link). An online service, it allowed multi- user chat, email, file sharing, and games.

If Q-Link sounds familiar, that's because it is: in 1991, the company changed its name to America Online (AOL). But AOL wouldn't launch its signature product, AOL Instant Messenger (AIM), until 1997. In the meantime, the Vodafone GSM network enabled the first SMS in 1992. And in 1996, ICQ launched as the first widely-adopted instant messaging platform.

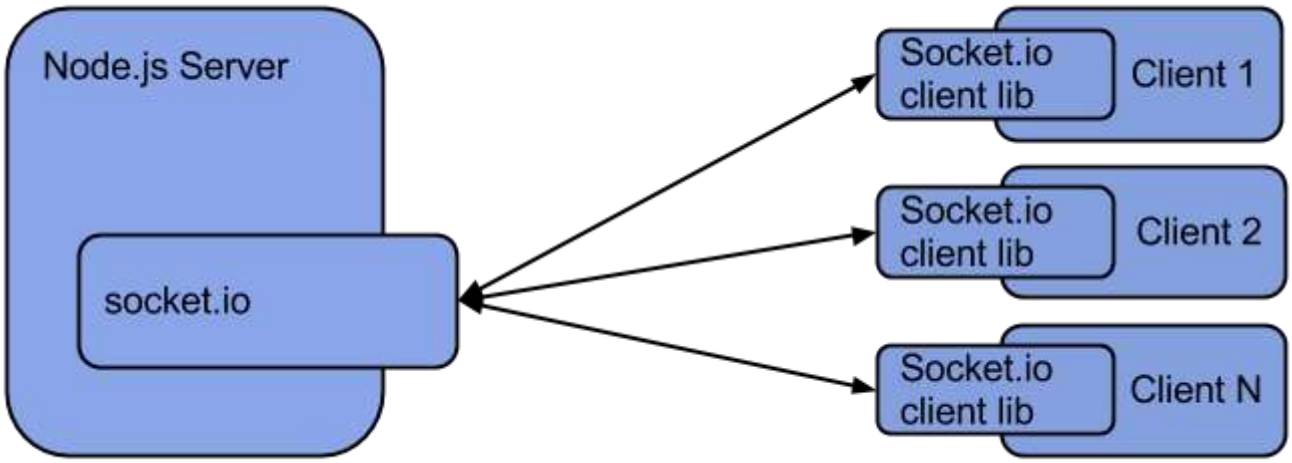
Advances which can be utilized for realtime correspondence are:



Short Polling: AJAX, makes substantial traffic.

Long Polling: Like AJAX, yet the server hangs on the reaction until it has an update. Subsequent to getting it, the customer sends another solicitation, and requirements extra header to be navigated to and fro causing extra overhead.

Web Sockets: make it conceivable to open intelligent correspondence between the customer and server. One can send a solicitation to the server and get occasion driven reactions without Polling the server for an answer, settling on web attachments a most ideal decision for our utilization case.



The late 1990s brought dramatic changes in the chat app market. Both Yahoo! And MSN launched their own instant messengers (in 1998 and 1999, respectively), and AOL bought out competitor ICQ's parent company, Mirabilis, for a fee around $287 million upfront, withan additional $120 million paid out later. That's about $612.7 million today!

OPPORTUNITIES

1. **HIGHER ENGAGEMENT:** Since many chat apps provide publishers with push notifications or chatbot experiences (programmable robots that converse with users), they can deliver significantly higher engagement rates.

2. **AUDIENCE DEVELOPMENT:** With billions of active users across multiple major chat apps, there is the opportunity in building large audiences fairly quickly on several platforms.

Advancement people work indefatigably to make building programs as simple as could be expected. The JavaScript, Web, and Mobile application engineers networks have expanded radically since Node and Cordova were presented. Engineers who had website composition abilities could, with less exertion, carry out a server utilizing JavaScript for their applications, through the assistance of Node.js.

Portable darlings can with the assistance of Cordova presently fabricate rich cross breed applications utilizing just JavaScript. Today, in spite of the fact that it is old information, I am eager to share the capacity to utilize JavaScript to fabricate work area independent applications.

Hub WebKit ordinarily expressed: "hub webkit" or "NW.js" is an application runtime dependent on Node.js and Chromium and empowers us to foster OS local applications utilizing just HTML, CSS, and JavaScript.

Basically, Node WebKit simply assists you with using your ability as a web engineer to fabricate a local application that runs easily on Mac, Windows, and Linux with a snort/swallow (whenever liked) form order.

This article focuses significantly more on utilizing Node WebKit, however to make things seriously fascinating, we will incorporate other astounding arrangements and they will include:

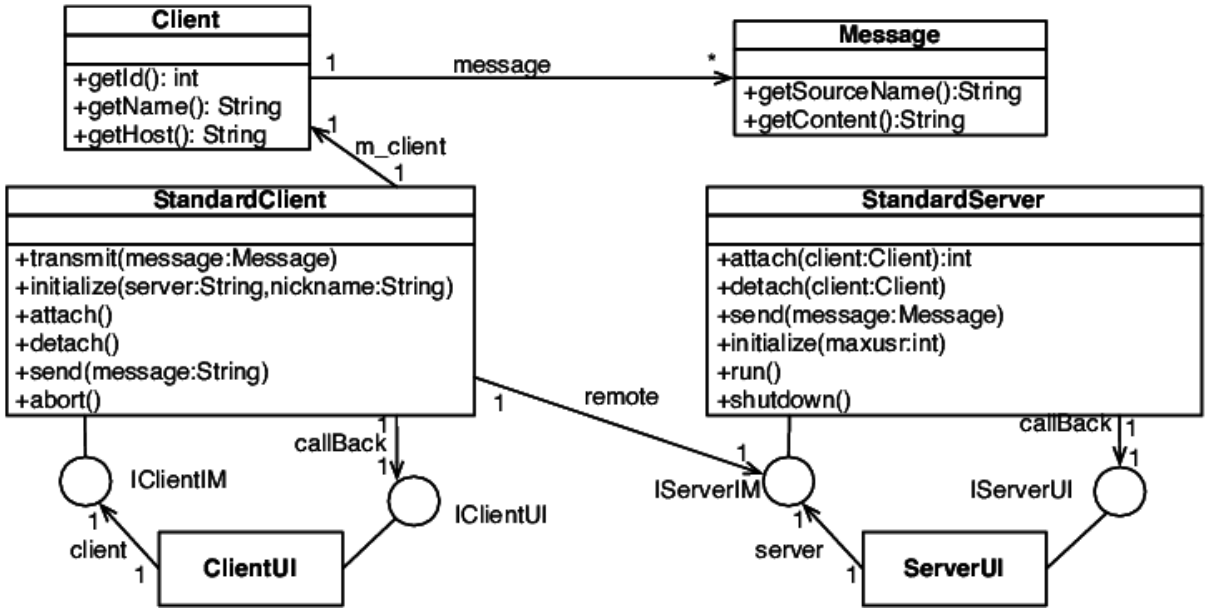
- Socket.io A realtime library for Node.js
- Rakish Material: Angular's execution of Google's Material Design
- MEAN: MEAN is only an idea of consolidating the highlights of Mongo, Express, Angular, and Node to fabricate incredible applications

Besides, the application has three areas:

- The server
- The work area (customer)
- The web (customer)

The web area won't be covered here, however it will fill in as a test stage yet relax, the code will be given

UML DIAGRAM -



.

Installation

We need to grab node-webkit and other dependencies for our application. Fortunately, there are frameworks that make workflow easy and we will be using one of them to scaffold our application and concentrate more on the implementation.

Yo and Slush are popular generators and any of these will work. I am going to be using Slush, but feel free to use Yo if you prefer to. To install Slush make sure you have node and npm installed and run

```
npm install -g slush gulp bower slush-wean
```

The command will install the following globally on our system.

- slush: a scaffolding tool
- slush-wean: the generator for Node WebKit
- gulp: our task runner
- bower: for frontend dependencies

Just like YO, make your directory and scaffold your app using:

```
mkdir scotch-chat

cd scotch-chat

slush wean
```

Running the below command will give us a glance of what we have been waiting for:

```
gulp run
```

The image shows our app loading. The author of the generator was generous enough to provide a nice template with a simple loading animation. To look cooler, I replaced the loading text with Scotch's logo.

If you are not comfortable with Slush automating things you can head right to Node WebKit on GitHub.

Now that we have set up our app, though empty, we will give it a break and prepare our server now.

The Server

The server fundamentally comprises of our model, courses, and attachment occasions. We will keep it as basic as could be expected and you can go ahead and broaden the application as taught toward the finish of the article.

Catalog Structure

Arrangement an envelope in your PC at your cherished catalog, yet ensure the organizer content resembles the beneath:

```
|- public
    |- index.html
|- server.js
|- package.json
```

Conditions

In the package.json document situated on your root catalog, make a JSON record to portray your application and incorporate the application's conditions.

```
{
  "name": "scotch-talk",
  "primary": "server.js",
  "conditions": {
    "mongoose": "most recent",
    "morgan": "most recent",
    "socket.io": "most recent"
  }
}
```

That will do. It is only a negligible arrangement and we are keeping things straightforward and short. Run npm introduce on the catalog root to introduce the predetermined conditions.

- npm introduce
-

Beginning Our Server Setup

The time has come to take care of business! The principal thing is to set up worldwide factors in server.js which will hold the application's conditions that are now introduced.

server.js

```
// Import every one of our conditions
var express = require('express');
var mongoose = require('mongoose');
var application = express();
var server = require('http').Server(app);
var io = require('socket.io')(server);
```

Alright, I didn't keep to my promise. The factors are holding the conditions, however some are designing it for use.
To serve static documents, express opens a technique to assist with designing the static records envelope. It is basic:
server.js

```
...

// advise express where to serve static records from
app.use(express.static(__dirname + '/public'));
```

Next up is to make an association with our information base. I'm working with a nearby MongoDB which clearly is discretionary as you can think that it is' facilitated by Mongo information bases. Mongoose is a hub module that uncovered astounding API which makes working with MongoDB a great deal a lot more straightforward.
server.js

```
...

mongoose.connect("mongodb://127.0.0.1:27017/scotch-visit");
```

With Mongoose we would now be able to make our information base composition and model. We additionally need to permit CORS in the application as we will get to it from an alternate area.
server.js

```
...

// make a pattern for visit
var ChatSchema = mongoose.Schema({
```

```
  made: Date,
  content: String,
  username: String,
  room: String
});

// make a model from the talk pattern
var Chat = mongoose.model('Chat', ChatSchema);

// permit CORS
app.all('*', function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE,OPTIONS');
  res.header('Access-Control-Allow-Headers', 'Content-type,Accept,X-Access-Token,X-
Key');
  in the event that (req.method == 'Choices') {
    res.status(200).end();
  } else {
    next();
  }
});
```

Our server will have three courses in it. A course to serve the record document, one more to set up talk information, and the last to serve visit messages separated by room names:

server.js

```
/*||||||||||||||||||||ROUTES||||||||||||||||||||||*/
// course for our list document
app.get('/', function(req, res) {
  //send the index.html in our public catalog
  res.sendfile('index.html');
});

//This course is essentially run distinctly on first dispatch just to create some visit history
app.post('/arrangement', function(req, res) {
```

```
  //Exhibit of visit information. Each article properties should match the mapping object
properties
  var chatData = [{
    made: new Date(),
    content: 'Hello',
    username: 'Chris',
    room: 'php'
  }, {
    made: new Date(),
    content: 'Hi',
    username: 'Obinna',
    room: 'laravel'
  }, {
    made: new Date(),
    content: 'Ait',
    username: 'Bill',
    room: 'precise'
  }, {
    made: new Date(),
    content: 'Stunning room',
    username: 'Tolerance',
    room: 'socet.io'
  }];

  //Circle through every one of the visit information and addition into the data set
  for (var c = 0; c < chatData.length; c++) {
    //Make an occasion of the talk model
    var newChat = new Chat(chatData[c]);
    //Call save to embed the talk
    newChat.save(function(err, savedChat) {
      console.log(savedChat);
    });
  }
  //Send a resoponse so the serve would not stall out
```

```
  res.send('created');
});

//This course delivers a rundown of talk as filterd by 'room' inquiry
app.get('/msg', function(req, res) {
  //Find
  Chat.find({
    'room': req.query.room.toLowerCase()
  }).exec(function(err, msgs) {
    //Send
    res.json(msgs);
  });
});

/*||||||||||||||||||END ROUTES||||||||||||||||||||*/
```

The main course I accept is sufficiently simple. It will simply send our index.html document to our clients.

The second/arrangement is intended to be hit only a single time and at the underlying dispatch of the application. It is discretionary assuming you needn't bother with some test information. It essentially makes a variety of talk messages (which matches the mapping), circles through them, and additions them into the data set.

The third course/msg is answerable for bringing visit history separated with room names and returned as a variety of JSON objects.

The main piece of our server is the continuous rationale. Remembering that we are running after delivering a straightforward application, our rationale will be completely insignificant. Successively, we want to:

- Know when our application is dispatched
- Send every one of the accessible rooms on association
- Tune in for a client to associate and dole out them to a default room
- Tune in for when they switch room
- Furthermore, at last, tune in for another message and just send the message to those in the room at which it was made

Thusly:

server.js

```
/*||||||||||||||||SOCKET||||||||||||||||||||||*/
//Tune in for association
io.on('connection', function(socket) {
  //Globals
  var defaultRoom = 'general';
  var rooms = ["General", "precise", "socket.io", "express", "hub", "mongo", "PHP",
"laravel"];

  //Produce the rooms cluster
  socket.emit('setup', {
    rooms: rooms
  });

  //Tunes in for new client
  socket.on('new client', function(data) {
    data.room = defaultRoom;
    //New client joins the default room
    socket.join(defaultRoom);
    //Tell every one of those in the room that another client joined
    io.in(defaultRoom).emit('user joined', information);
  });

  //Tunes in for switch room
  socket.on('switch room', function(data) {
    //Handles joining and leaving rooms
    //console.log(data);
    socket.leave(data.oldRoom);
    socket.join(data.newRoom);
    io.in(data.oldRoom).emit('user left', information);
    io.in(data.newRoom).emit('user joined', information);

  });
```

```
  //Tunes in for another visit message
  socket.on('new message', function(data) {
    //Make message
    var newMsg = new Chat({
      username: data.username,
      content: data.message,
      room: data.room.toLowerCase(),
      made: new Date()
    });
    //Save it to data set
    newMsg.save(function(err, msg){
      //Send message to those associated in the room
      io.in(msg.room).emit('message made', msg);
    });
  });
});
/*||||||||||||||||||END SOCKETS||||||||||||||||||*/
```

Then, at that point, the conventional server start:

server.js

```
server.listen(2015);
console.log('It\'s going down in 2015');
```

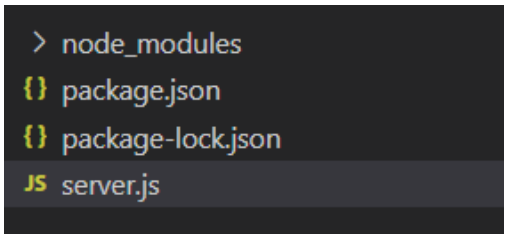
Fill the index.html with any HTML that suits you and run hub server.js. localhost:2015 will provide you with the substance of your HTML.

Installing Node.js dependencies

First, we're going to use Express as the API that will handle every request made by the front-end:

```
npm --save express
```

Now that we have installed our core library we can proceed to create the Node file that will execute all the logic of our application. Create a server.js File in the root directory of your back-end project:



Inside server.js place the following code:

```
var app = require('express')();var http = require('http').createServer(app);const
PORT = 8080;

http.listen(PORT, () => {
    console.log(`listening on *:${PORT}`);
});
```

Now that you have placed the code inside server.js, let's quickly make sure everything is working by running the following command:

```
node server.js
```

If everything is in order, the message "listening on *:8080" should appear in the console.

Socket.IO

Now we can proceed to install the library which can handle the web sockets connections:

```
save socket.io
```

Let's create our first socket listener in the server.js file:

```
var app = require('express')();var http =
require('http').createServer(app);const PORT = 8080;var io =
require('socket.io')(http);
http.listen(PORT, () => {
    console.log(`listening on *:${PORT}`);
```

```
});
io.on('connection', (socket) => { /* socket object may be used to send
specific messages to the new connected client */

    console.log('new client connected');
});
```

Creating the Front End

The first step to create the front end of our applications will be to initialize the React application. You can easily do this with the command:

```
npx create-react-app my-app
```

Now that the code base is initialized, we can proceed to install the Socket.IO library for our front end. Just use the client library of socket.io with:

```
npm i socket.io-client
```

Connecting the client with the server

If this is your first time using Socket.IO, this part will be exciting since we are enabling real-time communication between a single client and our back end using web sockets. In the `~/App.js` include the Socket.IO client and create a variable to store the socket object like this:

```
import React from 'react';import logo from './logo.svg';import
'./App.scss';import socketClient  from "socket.io-client";
const SERVER = "http://127.0.0.1:8080";
function App() {
    var socket = socketClient (SERVER);

    return (
        <div classname="App"></div>
```

```
            <header classname="App-header"></header>

                <img src="{logo}" classname="App-logo" alt="logo">
                <p></p>
                Edit <code>src/App.js</code> and save to reload.
               <p></p>

                   className="App-link"
                   href="https://reactjs.org"
                   target="_blank"
                   rel="noopener noreferrer"
                >

                   Learn React
    );

}

export default App;
```

Now that we have our socket variable, we can start listening to events emitted by our back end. In order to be notified when the client is connected we have to tweak our previous code in the `server.js` file:

```
var app = require('express')();
var http = require('http').createServer(app);
const PORT = 8080;
var io = require('socket.io')(http);
const STATIC_CHANNELS = ['global_notifications', 'global_chat'];

http.listen(PORT, () => {
    console.log(`listening on *:${PORT}`);
});

io.on('connection', (socket) => { /* socket object may be used to send
```

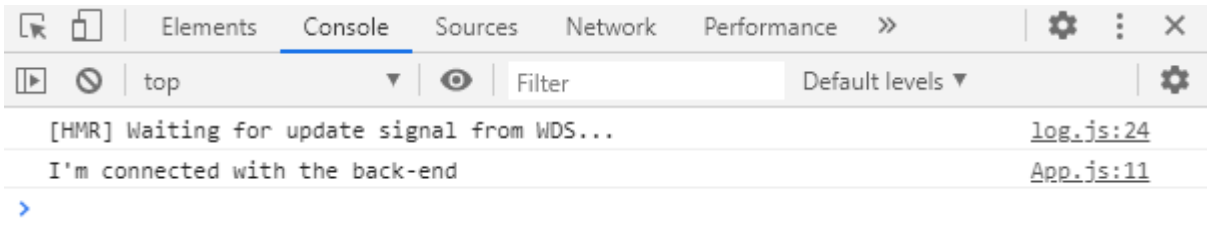
```
specific messages to the new connected client */
    console.log('new client connected');
    socket.emit('connection', null);
});
```

With the `socket.emit` function, custom events can be sent from the back end to the front end through the newly established socket connection. This method will only send messages between the specific client that was recently connected. Sending messages to all the clients connected will be explained later.

In order to receive those notifications from the back end, we need to listen for the events created there. For example, we are emitting the `connection` event to the client as soon it opens a new connection, so we have to put the same label in our front to execute some code when this happens:

```
var socket = socketClient (SERVER);
    socket.on('connection', () => {
        console.log(`I'm connected with the back-end`);
});
```

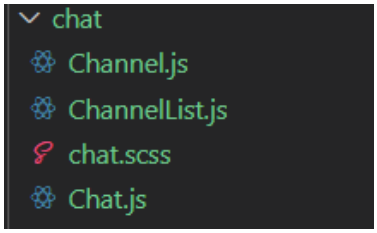
In the browser it should look like this:



Building the chat

With the installation complete, we can focus on building the UI of our application and the logic behind it.

In order to create a standalone components, we have to create a folder called “chat” with the following structure:



Chat.js

```
import React from 'react';import { ChannelList } from
'./ChannelList';import './chat.scss';
export class Chat extends React.Component {

        state = {
            channels: [{ id: 1, name: 'first', participants: 10 }]
        }
        render() {
            return (
                <div classname="chat-app"></div>

                    <channellist
channels="{this.state.channels}"></channellist>



            );
        }
}
```

chat.scss

```
.chat-app {
    width: 100%;
    height: 100%;
    display: flex;
    .channel-list {
        width: 20%;
        border: 1px solid rgb(224, 224, 224);
        margin: 10px;
        height: calc(100% - 22px);
```

```
    }

    .channel-item {
        border-bottom: 1px solid rgb(224, 224, 224);
        padding: 10px;

        div {
            font-weight: bold;
        }
        span {
            font-size: 10px;
        }
        &:hover {
            background-color: rgb(224, 224, 224);
        }
    }
}
```

To see the chat as a full window component, you might want to stylize the root element to fill the whole screen. To do so, you will have to modify the index.css file and add this:

```
html, body {
    margin: 0;
    height: 100%;
}
#root {
    width: 100%;
    height: 100%;
}
```

Message.js

```
import React from 'react';
export class Message extends React.Component {
```

```
    render() {
        return (
            <div classname="message-item"></div>

                <div><b>{this.props.senderName}</b></div>

                <span>{this.props.text}</span>


        )
    }
}
```

MeesagesPanel.js

```
import React from 'react';import { Message } from './Message';
export class MessagesPanel extends React.Component {
    render() {

        let list = <div classname="no-content-message">There is no
messages to show</div>;
        if (this.props.channel && this.props.channel.messages) {
            list = this.props.channel.messages.map(m => <message
key="{m.id}" id="{m.id}" sendername="{m.senderName}"
text="{m.text}">);</message>
        }
        return (
            <div classname="messages-panel"></div>

                <div classname="meesages-list">{list}</div>

                <div classname="messages-input"></div>

                    <input type="text">
                    <button>Send</button>
```

```
            );
        }

}
```

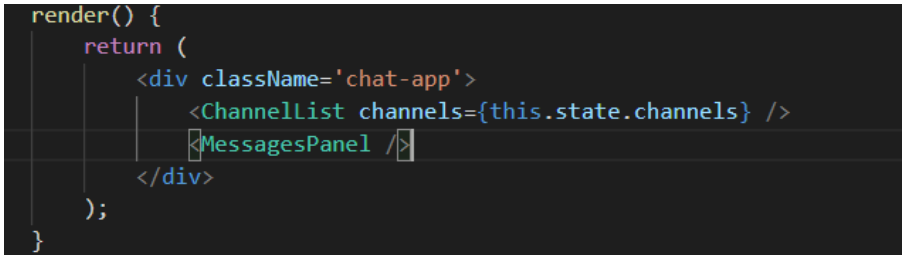
In order to adjust the styles of the application, we have to make some changes to our `chat.scss` file. It should look like this:

```
.chat-app {
    width: 100%;
    height: 100%;
    display: flex;

    .no-content-message {
        color: #cccccc;
        font-style: italic;
        font-size: 20px;
        text-align: center;
        margin: 20px;
    }

    .channel-list {
        width: calc(20% - 12px);
        border: 1px solid rgb(224, 224, 224);
        margin: 10px;
        margin-right: 0;
        border-right: none;
        height: calc(100% - 22px);
    }

    .channel-item {
        border-bottom: 1px solid rgb(224, 224, 224);
        padding: 10px;
```

```
        div {
            font-weight: bold;
        }

        span {
            font-size: 10px;
        }


        &:hover {
            background-color: rgb(224, 224, 224);
            cursor: pointer;
        }
    }

    .messages-panel {
        width: calc(80% - 12px);
        border: 1px solid rgb(224, 224, 224);
        margin: 10px;
        margin-left: 0;
        height: calc(100% - 22px);
        display: flex;
        flex-direction: column;
        align-items: flex-start;

        .meesages-list {
            align-self: stretch;
            height: 100%;
        }

        .messages-input {
            width: 100%;
            height: 40px;
            border-top: 1px solid rgb(224, 224, 224);
            background-color: #f0f0f0;
            display: flex;
```

```
            input {
                margin: auto;
                height: 20px;
                width: 100%;
                margin-left: 15px;
                border-radius: 15px;
                border: 1px solid rgb(224, 224, 224);

                &:focus {
                    border-radius: 15px;
                    border: 2px solid #66a6ff;
                    outline: none;
                 }
            }

            button {
                width: 60px;
                margin: auto 10px;
                background-color: #0e62da;
                color: white;
                border: 1px solid;
                border-radius: 10px;
                padding: 5px 13px;

                &:hover {
                    cursor: pointer;
                    background-color: #66a6ff;
                }
            }
        }
    }
}
```

After these changes are made, we need to introduce the new panel into the `Chat.js` file:



Developing the logic

The interface has its basic form. Now it's time to start developing some logic to send and receive messages.

Chat.js

```
handleChannelSelect = id => {
        this.socket.emit('channel-join', id, ack => {
        });
    }
    render() {

        return (
            <div classname="chat-app"></div>

                <channellist channels="{this.state.channels}"
onselectchannel="{this.handleChannelSelect}"></channellist>

                <messagespanel></messagespanel>


        );
}
```

server.js

```
io.on('connection', (socket) => { // socket object may be used to
send specific messages to the new connected client
    console.log('new client connected');
    socket.emit('connection', null);
    socket.on('channel-join', id => {
        console.log('channel join', id);
        STATIC_CHANNELS.forEach(c => {
            if (c.id === id) {
                if (c.sockets.indexOf(socket.id) == (-1)) {
                    c.sockets.push(socket.id);
                    c.participants++;
                    io.emit('channel', c);
                }
            } else {
                let index = c.sockets.indexOf(socket.id);
                if (index != (-1)) {
                    c.sockets.splice(index, 1);
                    c.participants--;
                    io.emit('channel', c);
                }
            }
        });

        return id;
    })
});
```

Sending Messages

It's time to start sending some specific messages over the `websocket`. To accomplish this, we have to capture the information in our `textbox` and then submit it by clicking the send button. When the click event is captured, we have to make sure that we can send all the information related to the message. In this case the information will be:

- senderName: The id of the socket sending the message.
- id: The id of the message which is going to be the current timestamp.
- text: The text captured in the input.
- channel_id: The id of the channel where the message was sent to.

Chat.js

```
configureSocket = () => {

  var socket = socketClient(SERVER);
  socket.on('connection', () => {
    if (this.state.channel) {
      this.handleChannelSelect(this.state.channel.id);
    }
  });

  socket.on('channel', channel => {

    let channels = this.state.channels;
      channels.forEach(c => {
        if (c.id === channel.id) {
          c.participants = channel.participants;
```

```
        }
  });

  this.setState({ channels });});

socket.on('message', message => {
  let channels = this.state.channels
    channels.forEach(c => {
      if (c.id === message.channel_id) {
        if (!c.messages) {
          c.messages = [message];
        } else {
          c.messages.push(message);
        }
      }
    });
    this.setState({ channels });
  });
  this.socket = socket;}
handleSendMessage = (channel_id, text) => {
  this.socket.emit('send-message', { channel_id, text, senderName:
this.socket.id, id: Date.now() });}
render() {
  return (
    <div classname="chat-app"></div>

      <channellist channels="{this.state.channels}"
onselectchannel="{this.handleChannelSelect}"></channellist>

      <messagespanel onsendmessage="{this.handleSendMessage}"
channel="{this.state.channel}"></messagespanel>


  );
}
```

For the back the integration is simple, as we only have to broadcast the messages received.

```
socket.on('send-message', message => {
    io.emit('message', message);
});
```

The Node WebKit Client

Time to uncover what we left to make our server which is running as of now. This part is very simple as it requires your regular information on HTML, CSS, JS, and Angular.

Registry Structure

We don't have to make any! I surmise that was the motivation for generators. The primary document you should assess is the package.json.

Hub WebKit requires, essentially, two significant documents to run:

1. an section point (index.html)

2. a package.json to tell it where the section point is found

package.json has the essential substance we are utilized to, then again, actually its fundamental is the area of the index.html, and it has a bunch of setups under "window": from which we characterize every one of the properties of the application's window including symbols, sizes, toolbar, outline, and so on

Dependencies

Not at all like the server, we will utilize grove to stack our conditions as it is a customer application. Update your bower.json conditions to:

"conditions": {

"rakish": "^1.3.13",

"rakish material" : "^0.10.0",

"rakish attachment io" : "^0.7.0",

"rakish material-icons":"^0.5.0",

"animate.css":"^3.0.0"

}

For an alternate route, just run the accompanying order:

- nook introduce - - save precise rakish material rakish attachment io rakish material-symbols animate.css

Since we have our frontend conditions, we can refresh our perspectives/index.ejs to:

index.ejs

```
<html><head>
    <title>scotch-chat</title>
    <connect rel="stylesheet" href="css/app.css">
    <connect rel="stylesheet" href="css/animate.css">
    <connect rel="stylesheet" href="libs/rakish material/precise material.css">

    <script src="libs/rakish/angular.js"></script>
    <script src="http://localhost:2015/socket.io/socket.io.js"></script>
    <script type="text/javascript" src="libs/rakish enliven/precise animate.js"></script>
    <script type="text/javascript" src="libs/rakish aria/precise aria.js"></script>
    <script type="text/javascript" src="libs/rakish material/precise material.js"></script>
    <script type="text/javascript" src="libs/rakish attachment io/socket.js"></script>
    <script type="text/javascript" src="libs/rakish material-symbols/precise material-icons.js"></script>

    <script src="js/app.js"></script>
```

```
</head>

<body ng-controller="MainCtrl" ng-init="usernameModal()">

  <md-content>

    <section>

      <md-list>

        <md-subheader class="md-essential header">Room: {{room}} <span
align="right">Userame: {{username}} </span> </md-subheader>



        <md-whiteframe ng-repeat="m in messages" class="md-whiteframe-z2 message"
format design align="center center">

          <md-list-thing class="md-3-line">

            <img ng-src="img/user.png" class="md-symbol" alt="User"/>

            <div class="md-list-thing text">

              <h3>{{ m.username }}</h3>

              <p>{{m.content}}</p>

            </div>

          </md-list-item>

        </md-whiteframe>



      </md-list>

    </section>



    <div class="footer">
```

```
        <md-input-container>

            <label>Message</label>

            <textarea ng-model="message" columns="1" md-maxlength="100" ng-
enter="send(message)"></textarea>

        </md-input-container>



    </div>



  </md-content>

</body>

</html>
```

We incorporated every one of our conditions and custom documents (app.css and app.js). Things to note:

- We are utilizing rakish material and its mandates are making our code resemble "HTML 6".

- We are circling through our messages scope utilizing ng-rehash and delivering its qualities to the program

- A mandate which we will see later assists us with sending the message when the ENTER key is squeezed

- On init, the client is requested a favored username

- There is an Angular library that is incorporated to assist with working with Socket.io in Angular simpler.

The Application

The primary piece of this segment is the app.js record. It makes administrations to associate with the Node WebKit GUI, a mandate to deal with the ENTER keypress and the regulators (fundamental and discourse).

app.js

```
//Load precise

var application = angular.module('scotch-visit', ['ngMaterial', 'ngAnimate', 'ngMdIcons', 'btford.socket-io']);



//Set our server url

var serverBaseUrl = 'http://localhost:2015';



//Administrations to cooperate with nodewebkit GUI and Window

app.factory('GUI', work () {

    //Return nw.gui

    return require('nw.gui');

});

app.factory('Window', work (GUI) {

    return GUI.Window.get();

});



//Administration to associate with the attachment library

app.factory('socket', work (socketFactory) {

    var myIoSocket = io.connect(serverBaseUrl);
```

```
    var attachment = socketFactory({
        ioSocket: myIoSocket
    });

    bring attachment back;
});
```

Following up, we make three Angular administrations. The primary assistance assists us with getting that Node WebKit GUI object, the second returns its Window property, and the third bootstraps Socket.io with the base URL.

app.js

```
//ng-enter order
app.directive('ngEnter', work () {
    return work (scope, component, attrs) {
        element.bind("keydown keypress", work (occasion) {
            on the off chance that (event.which === 13) {
                scope.$apply(function () {
                    scope.$eval(attrs.ngEnter);
                });

                event.preventDefault();
            }
```

```
        });

    };

});
```

The above scrap is one of my top choices since the time I have been utilizing Angular. It ties an occasion to the ENTER key, which accordingly an occasion can be set off when the key is squeezed.

At last, with the app.js is the all-powerful regulator. We want to separate things to ease understanding as we did in our server.js. The regulator is relied upon to:

1. Create a rundown of window menus from utilizing the rooms radiated from the server.

2. The client on joining is relied upon to give their username.

3. Listen for another message from the server.

4. Notify the server of new messages when they are made by composing and hitting the ENTER key.

Make a List of Rooms

With our targets characterized let us code:

app.js

```
//Our Controller

app.controller('MainCtrl', work ($scope, Window, GUI, $mdDialog, attachment, $http){



  //Menu arrangement



  //Modular arrangement
```

```
  //tune in for new message



  //Tell server of the new message



});
```

That is our regulator's skeleton with its conditions as a whole. As may be obvious, it has four inward remarks which is filling in as a placeholder for our codes as characterized in the targets. So how about we single out the menu.

app.js

```
//Worldwide Scope

$scope.messages = [];

$scope.room = "";



//Assemble the window menu for our application utilizing the GUI and Window administration

var windowMenu = new GUI.Menu({

    type: 'menubar'

});

var roomsMenu = new GUI.Menu();



windowMenu.append(new GUI.MenuItem({

    name: 'Rooms',
```

```
    submenu: roomsMenu
}));

windowMenu.append(new GUI.MenuItem({
    name: 'Exit',
    click: work () {
        Window.close()
    }
}));
```

We basically made occasions of the menu and added some menu (Rooms and Exit) to it. The rooms menu is relied upon to fill in as a drop-down thus we need to ask the server for accessible rooms and annex it to the rooms menu:

app.js

```
//Tune in for the arrangement occasion and make rooms
socket.on('setup', work (information) {
    var rooms = data.rooms;

    for (var r = 0; r < rooms.length; r++) {
        //Circle and add space to the window room menu
        handleRoomSubMenu(r);
    }
```

```
//Handle formation of room
work handleRoomSubMenu(r) {
    var clickedRoom = rooms[r];
    //Add each space to the menu
    roomsMenu.append(new GUI.MenuItem({
        mark: clickedRoom.toUpperCase(),
        click: work () {
            //What occurs on tapping the rooms? Swtich room.
            $scope.room = clickedRoom.toUpperCase();
            //Tell the server that the client changed his room
            socket.emit('switch room', {
                newRoom: clickedRoom,
                username: $scope.username
            });
            //Get the new rooms messages
            $http.get(serverBaseUrl + '/msg?room=' + clickedRoom).success(function (msgs) {
                $scope.messages = msgs;
            });
        }
    }));
}
//Append menu
```

```
    GUI.Window.get().menu = windowMenu;

});
```

The above code with the assistance of a capacity, circles through a variety of rooms when they are free from the server and afterward add them to the rooms menu. With that, Objective #1 is finished.

Requesting a Username

Our subsequent goal is to ask the client for username utilizing precise material modular.

app.js

```
$scope.usernameModal = work (ev) {
    //Dispatch Modal to get username
    $mdDialog.show({
        regulator: UsernameDialogController,
        templateUrl: 'partials/username.tmpl.html',
        parent: angular.element(document.body),
        targetEvent: ev,
    })
    .then(function (reply) {
        //Set username with the worth got back from the modular
        $scope.username = reply;
        //Tell the server there is another client
        socket.emit('new client', {
            username: reply
        });
        //Set space to general;
        $scope.room = 'GENERAL';
        //Get visit messages in GENERAL
        $http.get(serverBaseUrl + '/msg?room=' + $scope.room).success(function (msgs) {
            $scope.messages = msgs;
        });
    }, work () {
```

```
        Window.close();
    });
};
```

As determined in the HTML, on init, the usernameModal is called. It utilizes the mdDialog administration to get username of a joining client and assuming that is fruitful it will dole out the username entered to a limiting extension, advise the server concerning that movement and afterward push the client to the default (GENERAL) room. On the off chance that it isn't effective we close the application. Objective #2 finished!

```
//Tune in for new messages (Objective 3)
    socket.on('message made', work (information) {
        //Push to new message to our $scope.messages
        $scope.messages.push(data);
```

Tuning in For Messages

The third, and the last, objective is basic. #3 simply tunes in for messages and assuming any push it to the variety of existing messages and #4 informs the server of new messages when they are made. Toward the finish of app.js, we make a capacity to fill in as the regulator for the Modal:

app.js

```
//Exchange regulator
work UsernameDialogController($scope, $mdDialog) {
    $scope.answer = work (reply) {
        $mdDialog.hide(answer);
    };
}
```

CSS and Animation

To fix some appalling looks, update the app.css.

```
body {
    foundation: #fafafa !significant;
}
```

```
.footer {
    foundation: #fff;
    position: fixed;
    left: 0px;
    base: 0px;
    width: 100%;
}

.message.ng-enter {
    -webkit-movement: zoomIn 1s;
    -ms-movement: zoomIn 1s;
    movement: zoomIn 1s;
}
```

Note the last style. We are utilizing ngAnimate and animate.css to make a beautiful liveliness for our messages.

I previously composed on how you can play with this idea here.

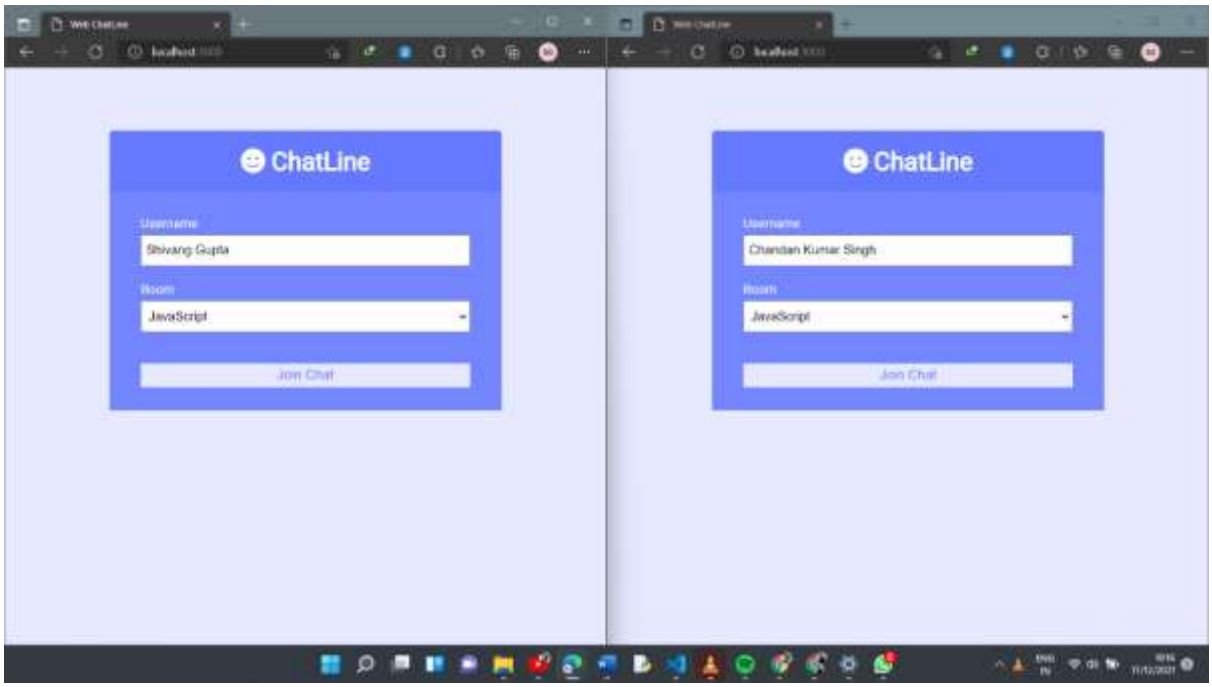
Quitting for the day

I can think about what you are stressed over subsequent to checking out the picture! The location bar, isn't that so? This is the place where the window design in the package.json comes in. Simply change "toolbar": consistent with "toolbar": bogus.

I likewise set my symbol to "symbol": "application/public/img/scotch.png" to change the window symbol to the Scotch logo. We can likewise add notice once there is another message:

```
var choices = {
    body: data.content
};
var warning = new Notification("Message from: "+data.username, choices);
```

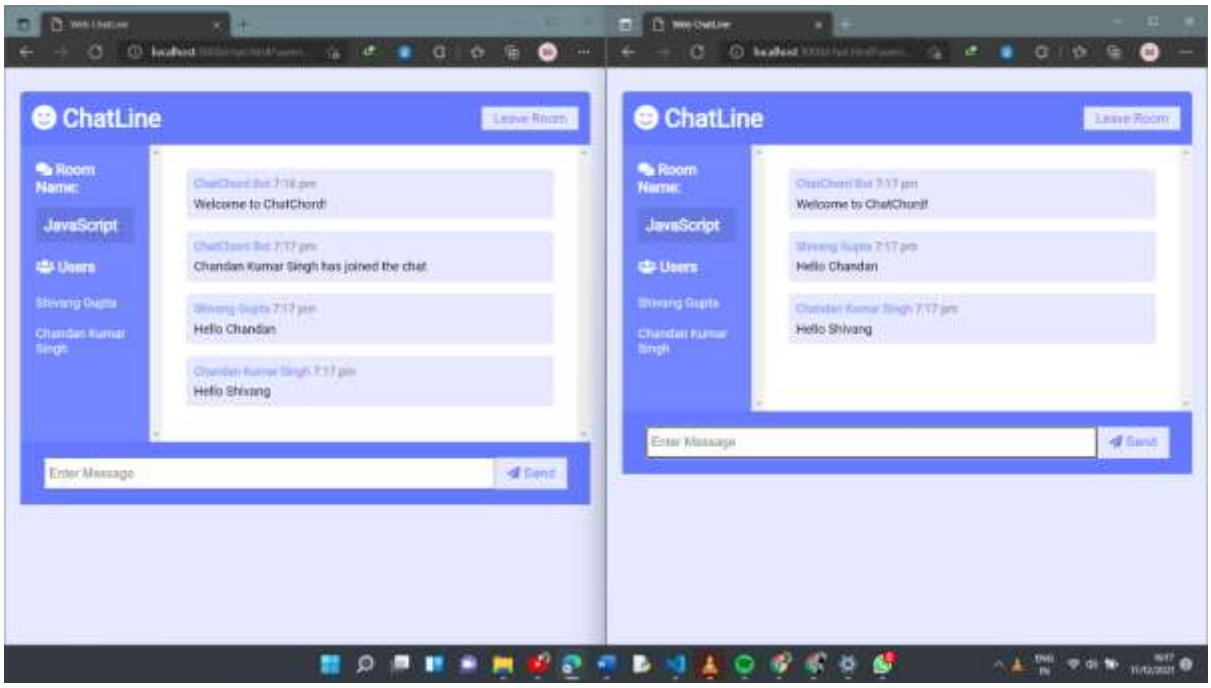
```
notification.onshow = work () {

  // auto close following 1 second
  setTimeout(function () {
     notification.close();
  }, 2000);
}
```



Testing

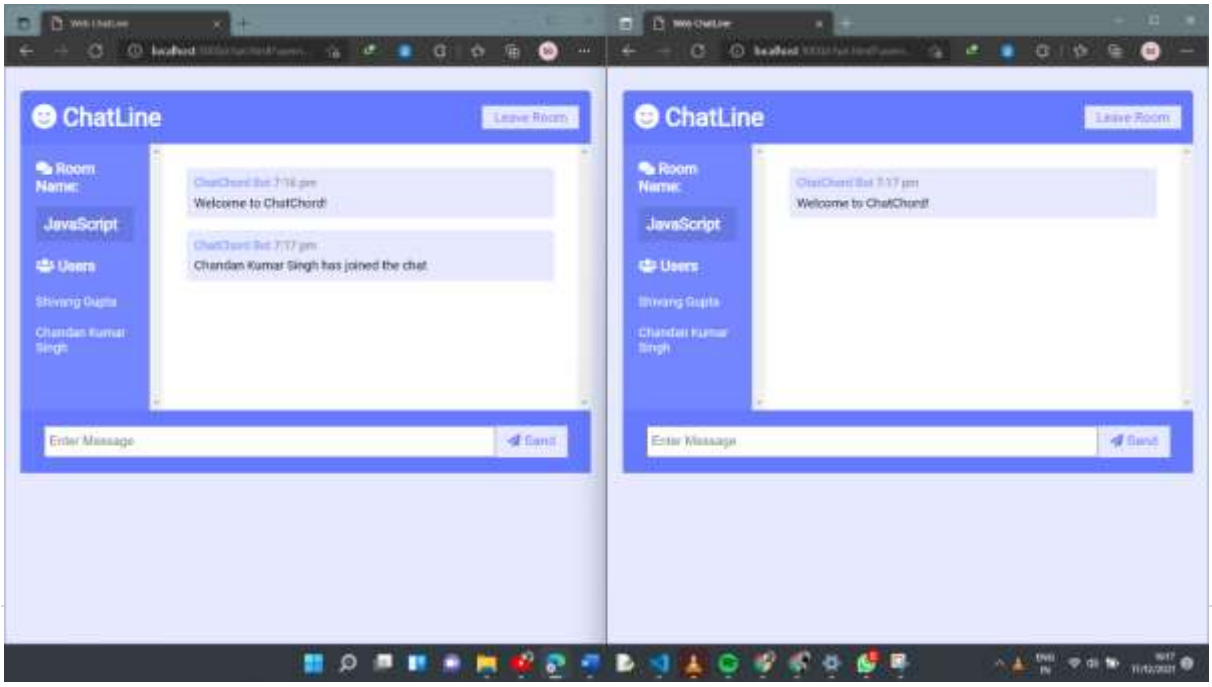
I recommend you test the application by downloading the web customer from Git Hub. Run the server, then, at that point, the web customer, and afterward the application. Begin sending messages from both the application and the web customer and watch them show up continuously assuming you are sending them in a similar room.

The final result should look like this:



CONCLUSION: -

Run the application with **npm start** and low and behold, your chat application is complete. At least, the basic functionality is in place. With ChatChord, you could easily expand the app



to include a “who’s online list”, direct messages, media messages, and a bunch of other features.

REFERENCES:

- Aaron, P., (2020) “Webrtc vs websockets.” http://stackoverflow.com/a/18825175, sep 2013. Accessed: 2017-04-19.

- Ackley, B., (2020) “Webrtc samples.” https://webrtc.github.io/samples/, sep 2013. Accessed: 2017-04-19.

- Abby, T., (2020) “User stories: An agile introduction.” http://www.agilemodeling.com/ artifacts/userStory.htm. Accessed: 2016-10-20.

- Bairam, W., (2020) “Making the switch from making the switch from node.js to golang.” http://blog.digg.com/post/141552444676/ making-the-switch-from-nodejs-to-golang, mar 2016. Accessed: 2016-10- 19.

- Callum, S., (2020) “Sinon - best practices for spies, stubs and mocks.” https://semaphoreci.com/community/tutorials/best-practices-for-spies-stubsand-mocks-in- sinon-js, 2016. Accessed: 2016-10-03