

RV COLLEGE OF ENGINEERING<sup>®</sup>  
BENGALURU-560059  
(Autonomous Institution Affiliated to VTU, Belagavi)



**“Live Chatroom”**

**Report**

**Web Technology Experiential Learning**

**(18IS6D1)**

*Submitted By*

Nehal N Shet (1RV18IS026)

Dheeraj Shenoy N (1RV18IS012)



**Under the Guidance of**

**Dr. Padmashree T,**

**Dept. of ISE, RVCE**

*in partial fulfillment for the award of degree of*

***Bachelor of Engineering***

*in*

**INFORMATION SCIENCE AND ENGINEERING**

**2020-21**

**RV COLLEGE OF ENGINEERING<sup>®</sup>, BENGALURU - 560059**  
**(Autonomous Institution Affiliated to VTU, Belagavi)**

**DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING**



**CERTIFICATE**

Certified that the Experiential Learning entitled '**Live Chatroom**' has been carried out as a part of Web Technology (18IS6D1) in partial fulfillment for the award of degree of **Bachelor of Engineering in Information Science and Engineering** of the Visvesvaraya Technological University, Belagavi during the year **2020-2021** by **Nehal N Shet (1RV18IS026)**, **Dheeraj Shenoy N (1RV18IS012)**, who are bonafide students of **RV College of Engineering<sup>®</sup>**, Bengaluru. It is certified that all the corrections/suggestions indicated for the internal assessment have been incorporated in the report deposited in the departmental library. The report has been approved as it satisfies the academic requirements in respect of work prescribed by the institution for the said degree.

**Dr. Padmashree T**

Department of ISE,  
RVCE, Bengaluru-59

## TABLE OF CONTENTS

<b>CHAPTER 1</b>	<b>4</b>
INTRODUCTION	4
<b>CHAPTER 2</b>	<b>5</b>
WEB PROGRAMMING TECHNIQUES	5
2.1 Specific Technology Used	5
2.2 Concept Description	7
<b>CHAPTER 3</b>	<b>9</b>
IMPLEMENTATION DETAILS	9
<b>CHAPTER 4</b>	<b>16</b>
EXPERIENTIAL LEARNING OUTCOMES AND CONCLUSION	16
REFERENCES	20

## 1.INTRODUCTION

With the rapid development of mobile phones, mobile devices have become one of the integral parts of daily activities. In recent years, chat applications have evolved and made a major change in social media because of their distinctive features that attract audiences. It provides real-time messaging and offers different services including, exchange text messages, images, files etc. Moreover, it supports cross platforms such as Android and iOS. There are currently millions of smartphone users who are using chat applications.

In a world where the value of time is steadily increasing, building applications that users can interact with in real-time has become a norm for most of the developers. Most of the applications we see today, whether they are mobile, desktop, or web applications, have at least a single real-time feature included. As an example, real-time messaging and notifications are two of the most commonly used real-time features used in applications. Most of us are familiar with the use of real-time messaging applications, especially in mobile devices, in the form of Whatsapp, Facebook Messenger, and numerous other messaging applications. However, real-time messaging usage is not limited to purely messaging applications. We can also use web based consoles for implementing these things in a more interactive way. These provides us with a more realistic approach by making two sided communication more effective.

In this project we are going to build a web based realtime chat room using reactjs and nodejs frameworks. It will allow only authenticated users to send and read messages and users can sign up by providing their email and creating a password, or by authenticating through a Google account. A chat room is a Web site, part of a Web site, or part of an online service, that provides a venue for communities of users with a common interest to communicate in real time. Forums and discussion groups, in comparison, allow users to post messages but don't have the capacity for interactive messaging. Most chat rooms don't require users to have any special software; those that do, such as Internet Relay Chat (IRC) allow users to download it from the Internet.

## 2.WEB PROGRAMMING TECHNIQUES

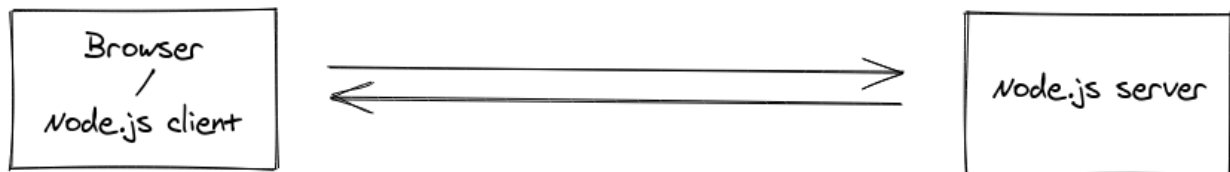
### 2.1 Specific Technology Used

#### I) Socket.IO

Socket.IO enables real-time, bidirectional and event-based communication. It works on every platform, browser or device, focusing equally on reliability and speed.

It consists of:

- Node.js server
- Javascript client library for the browser



#### II) Node Express: Back-End Framework:

Express is a Node.js framework. Rather than writing the code using Node.js and creating loads of Node modules, Express makes it simpler and easier to write the back-end code. Express helps in designing great web applications and APIs. Express supports many middlewares which makes the code shorter and easier to write.

### III) React: Front-End Library

React is a JavaScript library that is used for building user interfaces. React is used for the development of single-page applications and mobile applications because of its ability to handle rapidly changing data. React allows users to code in JavaScript and create UI components.

### IV) Node.js: JS Runtime Environment

Node.js provides a JavaScript Environment which allows the user to run their code on the server (outside the browser). Node pack manager i.e. npm allows the user to choose from thousands of free packages (node modules) to download.

## 2.2 Concept Description

### 1. Socket.IO

Reasons for using Socket.IO:

- reliability (fallback to HTTP long-polling in case the WebSocket connection cannot be established)
- automatic reconnection
- packet buffering
- acknowledgements
- broadcasting to all clients or to a subset of clients (what we call “Room”)
- multiplexing (what we call “Namespace”)

Installation: *npm install socket.io express --save*

### 2. Express

Advantages of using Express:

- Asynchronous and Single-threaded.
- Efficient, fast & scalable

- Has the biggest community for Node.js
- Express promotes code reusability with its built-in router.
- Robust API
- Create a new folder to start your express project and type below command in the command prompt to initialize a package.json file. Accept the default settings and continue.

Installation:*npm install express --save*

### **3 . React JS**

Features of React:

- Virtual DOM – A virtual DOM object is a representation of a DOM object. Virtual DOM is actually a copy of the original DOM. Any modification in the web application causes the entire UI to re-render the virtual DOM. Then the difference between the original DOM and this virtual DOM is compared and the changes are made accordingly to the original DOM.
- JSX – Stands for JavaScript XML. It is an HTML/XML JavaScript Extension which is used in React. Makes it easier and simpler to write React components.
- Components – ReactJS supports Components. Components are the building blocks of UI wherein each component has a logic and contributes to the overall UI. These components also promote code reusability and make the overall web application easier to understand.
- High Performance – Features like Virtual DOM, JSX and Components makes it much faster than the rest of the frameworks out there.
- Developing Android/Ios Apps – With React Native you can easily code Android-based or IOS-Based apps with just the knowledge of JavaScript and ReactJS.
- You can start your react application by first installing “create-react-app” using npm or yarn.

Installation:*npm install --save react*

## 4.Node JS

Reasons for using Node.JS :

- Open source JavaScript Runtime Environment
- Single threading – Follows a single threaded model.
- Data Streaming
- Fast – Built on Google Chrome's JavaScript Engine, Node.js has a fast code execution.
- Highly Scalable
- Initialize a Node.js application by typing running the below command in the command window. Accept the standard settings.

Installation:*npm install --save node*



### 3. IMPLEMENTATION DETAILS

#### 1. Setting up client and server communication

This is done using socket.io at the backend and using express for retrieval of the information.

```
const http = require("http");
const express = require("express");
const cors = require("cors");
const socketIo = require("socket.io");
const app = express();
app.use(cors());
const PORT = 4000;
server.listen(PORT, () => {
  console.log(`Listening on port ${PORT}`);
});
```

#### 2. Listening messages at the backend

First the socket.io is added to the server project and then the index.js is updated as follows

```
const socketIo = require("socket.io");
const io = socketIo(server, {
  cors: {
    origin: "http://localhost:3000",
    methods: ["GET", "POST"],
    credentials: true,
  },
});
const NEW_CHAT_MESSAGE_EVENT = "NEW_CHAT_MESSAGE_EVENT";
io.on("connection", (socket) => {
  console.log(`${socket.id} connected`);
  // Join a conversation
  const { roomId } = socket.handshake.query;
  socket.join(roomId);
  // Listen for new messages
  socket.on(NEW_CHAT_MESSAGE_EVENT, (data) => {
    io.in(roomId).emit(NEW_CHAT_MESSAGE_EVENT, data);
  });
  // Leave the room if the user closes the socket
  socket.on("disconnect", () => {
    socket.leave(roomId);
  });
});
```

- We import the socket.io module and create a new socket object. the cors parameter is necessary to execute the code locally.
- We define the NEW\_CHAT\_MESSAGE\_EVENT constant. This is the event name the socket will listen to for new messages.
- Upon a new connection, the socket will join the room requested.
- When the socket receives a new message, it will simply emit it to the room, so all socket clients of the same room will receive it.
- When the socket is disconnected, it leaves the room.

### 3. Sending messages to the client:

Similar to the server side socket.io is added on the client and then the code for maintaining a chatroom is given below

```
import { useEffect, useRef, useState } from "react";
import socketIOClient from "socket.io-client";
const NEW_CHAT_MESSAGE_EVENT = "NEW_CHAT_MESSAGE_EVENT";
const SOCKET_SERVER_URL = "http://localhost:4000";
const useChat = (roomId) => {
  const [messages, setMessages] = useState([]);
  const socketRef = useRef();
  useEffect(() => {
    socketRef.current = socketIOClient(SOCKET_SERVER_URL, {
      query: { roomId },
    });
    socketRef.current.on("connect", () => {
      console.log(socketRef.current.id);
    });
    socketRef.current.on(NEW_CHAT_MESSAGE_EVENT, (message) => {
      const incomingMessage = {
        ...message,
        ownedByCurrentUser: message.senderId === socketRef.current.id,
      };
      setMessages((messages) => [...messages, incomingMessage]);
    });
    return () => {
      socketRef.current.disconnect();
    };
  }, [roomId]);
  const sendMessage = (messageBody) => {
    if (!socketRef.current) return;
    socketRef.current.emit(NEW_CHAT_MESSAGE_EVENT, {
      body: messageBody,
      senderId: socketRef.current.id,
    });
  };
}
```

```

    };
    return {
      messages,
      sendMessage
    };
  };
};
export default useChat;

```

#### 4. Keeping track of Users in the Server:

Creating memory to keep track of users in the chatroom is implemented in User.js

```

const users = [];
const addUser = (id, room, name, picture) => {
  const existingUser = users.find(
    (user) => user.room === room && user.name === name
  );
  const user = { id, name, picture, room };
  users.push(user);
  return { id, name: user.name, picture: user.picture };
};
const removeUser = (id) => {
  const index = users.findIndex((user) => user.id === id);
  if (index !== -1) return users.splice(index, 1)[0];
};
const getUser = (id) => users.find((user) => user.id === id);
const getUsersInRoom = (room) => users.filter((user) => user.room === room);
module.exports = { addUser, removeUser, getUser, getUsersInRoom };

```

Next we notify the users everytime when a new user joins or leaves the room:

```

const PORT = 4000;
const USER_JOIN_CHAT_EVENT = "USER_JOIN_CHAT_EVENT";
const USER_LEAVE_CHAT_EVENT = "USER_LEAVE_CHAT_EVENT";
const NEW_CHAT_MESSAGE_EVENT = "NEW_CHAT_MESSAGE_EVENT";
io.on("connection", (socket) => {
  console.log(` ${socket.id} connected`);
  // Join a conversation
  const { roomId, name, picture } = socket.handshake.query;
  socket.join(roomId);
  const user = addUser(socket.id, roomId, name, picture);
  io.in(roomId).emit(USER_JOIN_CHAT_EVENT, user);
  // Listen for new messages
  socket.on(NEW_CHAT_MESSAGE_EVENT, (data) => {
    io.in(roomId).emit(NEW_CHAT_MESSAGE_EVENT, data);
  });
  // Leave the room if the user closes the socket
  socket.on("disconnect", () => {

```

```

    removeUser(socket.id);
    io.in(roomId).emit(USER_LEAVE_CHAT_EVENT, user);
    socket.leave(roomId);
  });
});
server.listen(PORT, () => {
  console.log(`Listening on port ${PORT}`);
});

```

## 5. Creating new user on the client

First, we will create a new random generated user every time there is a new connection to the chat room.

```

import { useEffect, useRef, useState } from "react";
import socketIOClient from "socket.io-client";
import axios from "axios";
const USER_JOIN_CHAT_EVENT = "USER_JOIN_CHAT_EVENT";
const USER_LEAVE_CHAT_EVENT = "USER_LEAVE_CHAT_EVENT";
const NEW_CHAT_MESSAGE_EVENT = "NEW_CHAT_MESSAGE_EVENT";
const SOCKET_SERVER_URL = "http://localhost:4000";
const useChat = (roomId) => {
  const [messages, setMessages] = useState([]);
  const [users, setUsers] = useState([]);
  const [typingUsers, setTypingUsers] = useState([]);
  const [user, setUser] = useState();
  const socketRef = useRef();
  useEffect(() => {
    const fetchUser = async () => {
      const response = await axios.get("https://api.randomuser.me/");
      const result = response.data.results[0];
      setUser({
        name: result.name.first,
        picture: result.picture.thumbnail,
      });
    };
    fetchUser();
  }, []);
  useEffect(() => {
    const fetchUsers = async () => {
      const response = await axios.get(
        `${SOCKET_SERVER_URL}/rooms/${roomId}/users`
      );
      const result = response.data.users;
      setUsers(result);
    };
    fetchUsers();
  }, [roomId]);
  useEffect(() => {

```

```

if (!user) {
  return;
}
socketRef.current = socketIOClient(SOCKET_SERVER_URL, {
  query: { roomId, name: user.name, picture: user.picture },
});
socketRef.current.on("connect", () => {
  console.log(socketRef.current.id);
});
socketRef.current.on(USER_JOIN_CHAT_EVENT, (user) => {
  if (user.id === socketRef.current.id) return;
  setUsers((users) => [...users, user]);
});
socketRef.current.on(USER_LEAVE_CHAT_EVENT, (user) => {
  setUsers((users) => users.filter((u) => u.id !== user.id));
});
socketRef.current.on(NEW_CHAT_MESSAGE_EVENT, (message) => {
  const incomingMessage = {
    ...message,
    ownedByCurrentUser: message.senderId === socketRef.current.id,
  };
  setMessages((messages) => [...messages, incomingMessage]);
});
return () => {
  socketRef.current.disconnect();
};
}, [roomId, user]);
const sendMessage = (messageBody) => {
  if (!socketRef.current) return;
  socketRef.current.emit(NEW_CHAT_MESSAGE_EVENT, {
    body: messageBody,
    senderId: socketRef.current.id,
    user: user,
  });
};
return {
  messages,
  user,
  users,
  sendMessage
};
};

export default useChat;

```

- When the hook mounts we create a new user

- We keep track of all users in the chat room with a new useState hook.
- When joining a room, we fetch the list of all users in the chat room.
- The socket listens for when a user joins or leaves the chat room.
- When sending the message, we send the user that sent the message, not just the body text.

## 6.Fetching message history

```
const useChat = (roomId) => {
  ...
  useEffect(() => {
    const fetchMessages = async () => {
      const response = await axios.get(
        `${SOCKET_SERVER_URL}/rooms/${roomId}/messages`
      );
      const result = response.data.messages;
      setMessages(result);
    };
    fetchMessages();
  }, [roomId]);
  ...
  return {
    messages,
    user,
    users,
    sendMessage,
  };
};
export default useChat;
```

We will be modifying index.js to listen for events when a user starts or stops typing. Just like for new messages, the socket will emit an event to notify all clients.

```
const START_TYPING_MESSAGE_EVENT = "START_TYPING_MESSAGE_EVENT";
const STOP_TYPING_MESSAGE_EVENT = "STOP_TYPING_MESSAGE_EVENT";
io.on("connection", (socket) => {
  ...
  // Listen typing events
  socket.on(START_TYPING_MESSAGE_EVENT, (data) => {
    io.in(roomId).emit(START_TYPING_MESSAGE_EVENT, data);
  });
  socket.on(STOP_TYPING_MESSAGE_EVENT, (data) => {
    io.in(roomId).emit(STOP_TYPING_MESSAGE_EVENT, data);
  });
  ...
});
```

We will create a useTyping React hook to handle the number of users typing a message in the chatroom:

```
import { useEffect, useState } from "react";
const useTyping = () => {
  const [isTyping, setIsTyping] = useState(false);
  const [isKeyPressed, setIsKeyPressed] = useState(false);
  const [countdown, setCountdown] = useState(5);
  const startTyping = () => {
    setIsKeyPressed(true);
    setCountdown(5);
    setIsTyping(true);
  };
  const stopTyping = () => {
    setIsKeyPressed(false);
  };
  const cancelTyping = () => {
    setCountdown(0);
  };
  useEffect(() => {
    let interval;
    if (!isKeyPressed) {
      interval = setInterval(() => {
        setCountdown((c) => c - 1);
      }, 1000);
    } else if (isKeyPressed || countdown === 0) {
      clearInterval(interval);
    }
    if (countdown === 0) {
      setIsTyping(false);
    }
    return () => clearInterval(interval);
  }, [isKeyPressed, countdown]);
  return { isTyping, startTyping, stopTyping, cancelTyping };
};

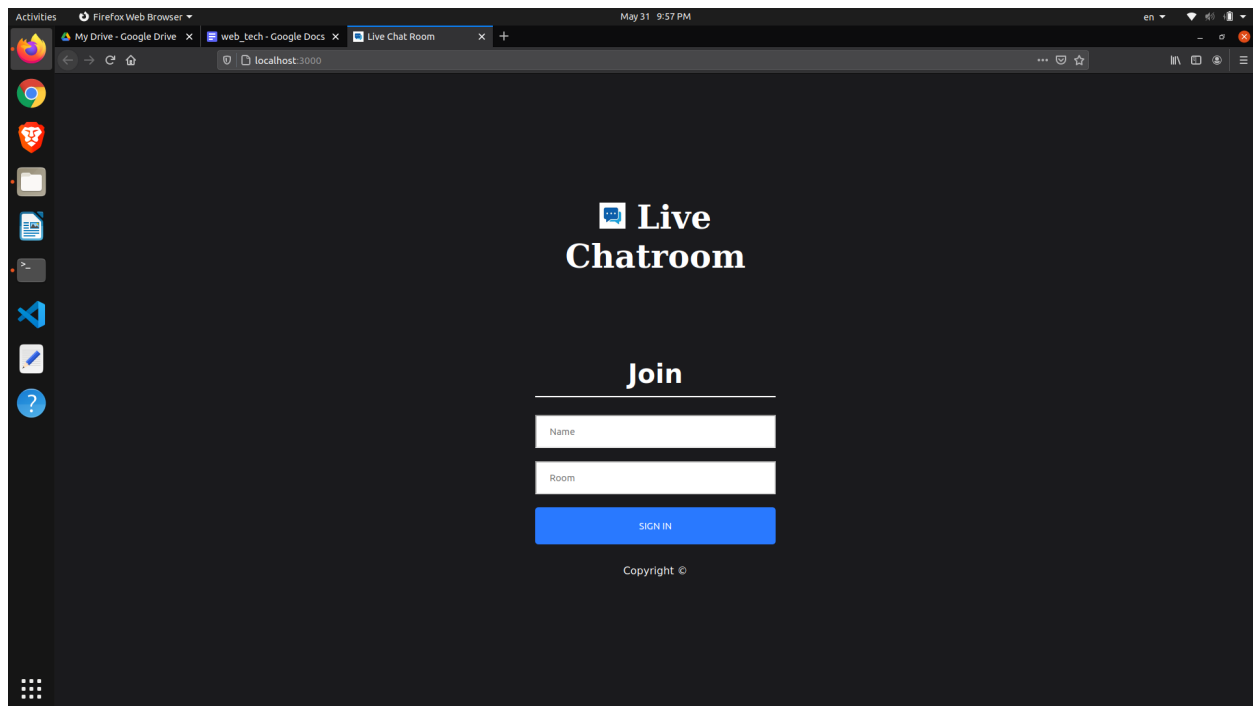
export default useTyping;
```

## 4.EXPERIENTIAL LEARNING OUTCOMES AND CONCLUSION

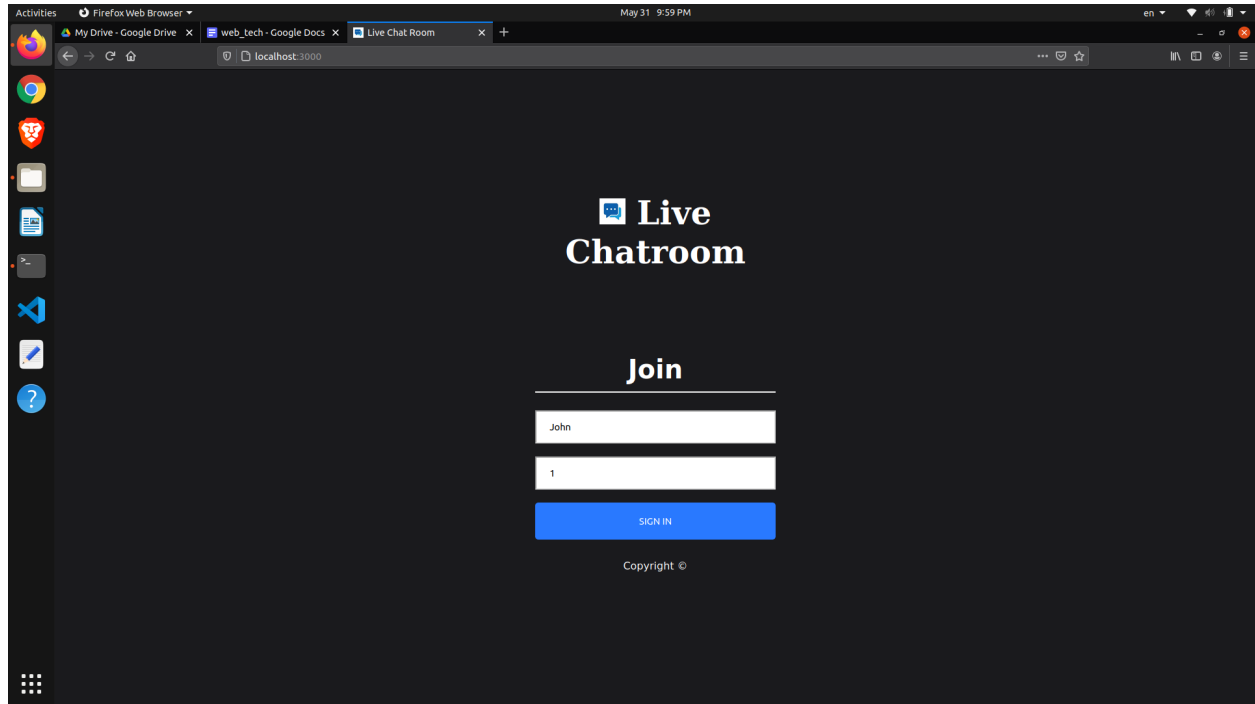
The following Experiential learning outcomes were achieved based on the project:

- Working on websites based on Node JS and React JS.
- Using attractive design techniques and strategies which are focused on customer centric design strategies.
- Using Node Express to simplify the amount of code allowing ease of maintenance.
- Working on real time bidirectional communication between client and server using socket.IO.
- Setting and maintaining dynamic page layout, fonts and colours for better response from the users.

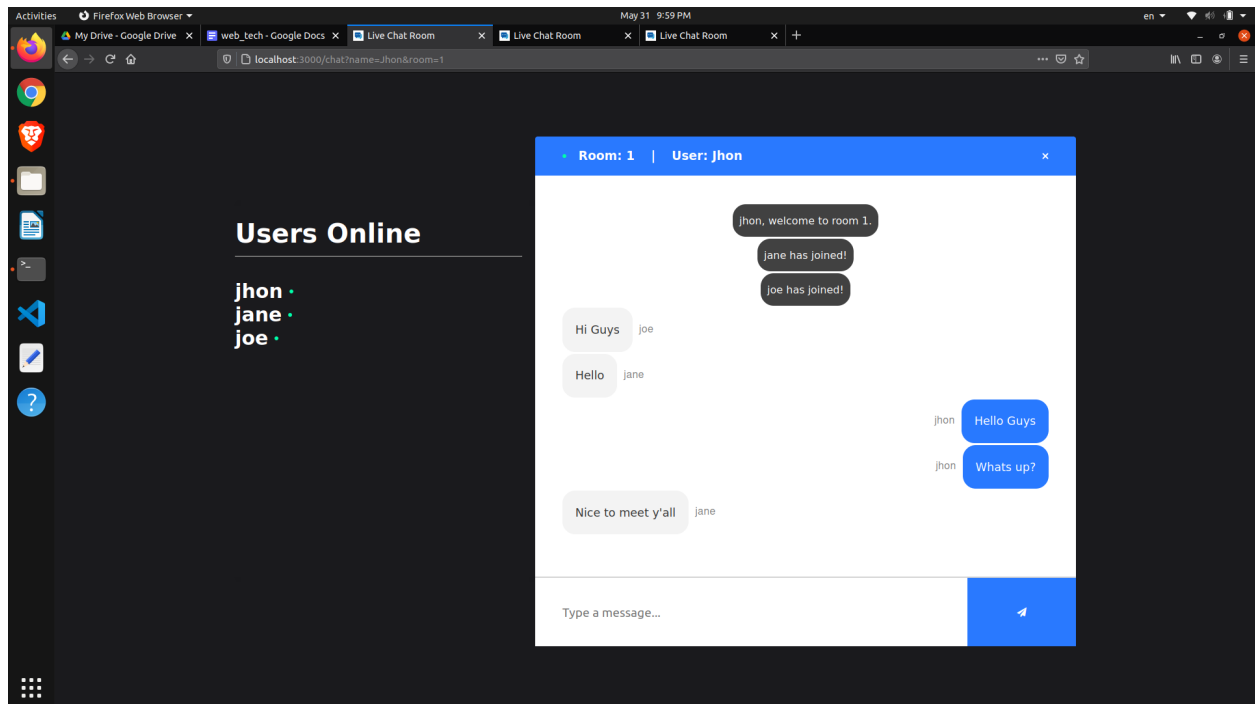
## SCREEN SHOTS

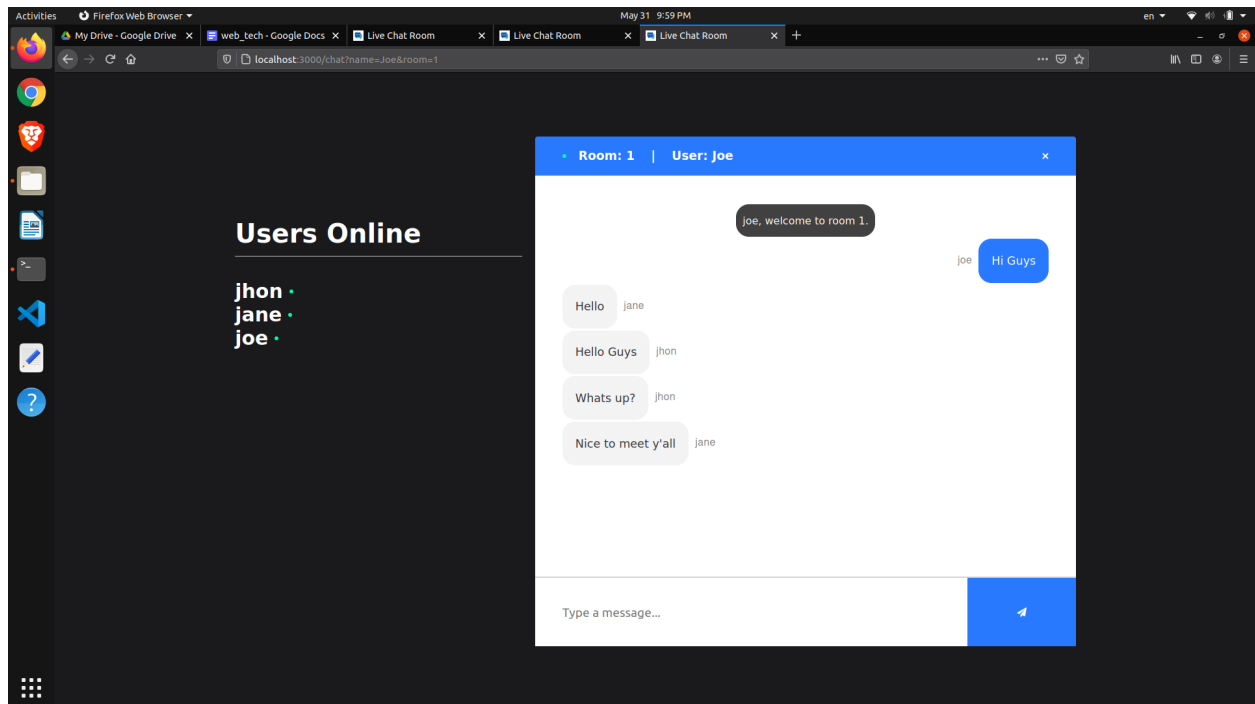
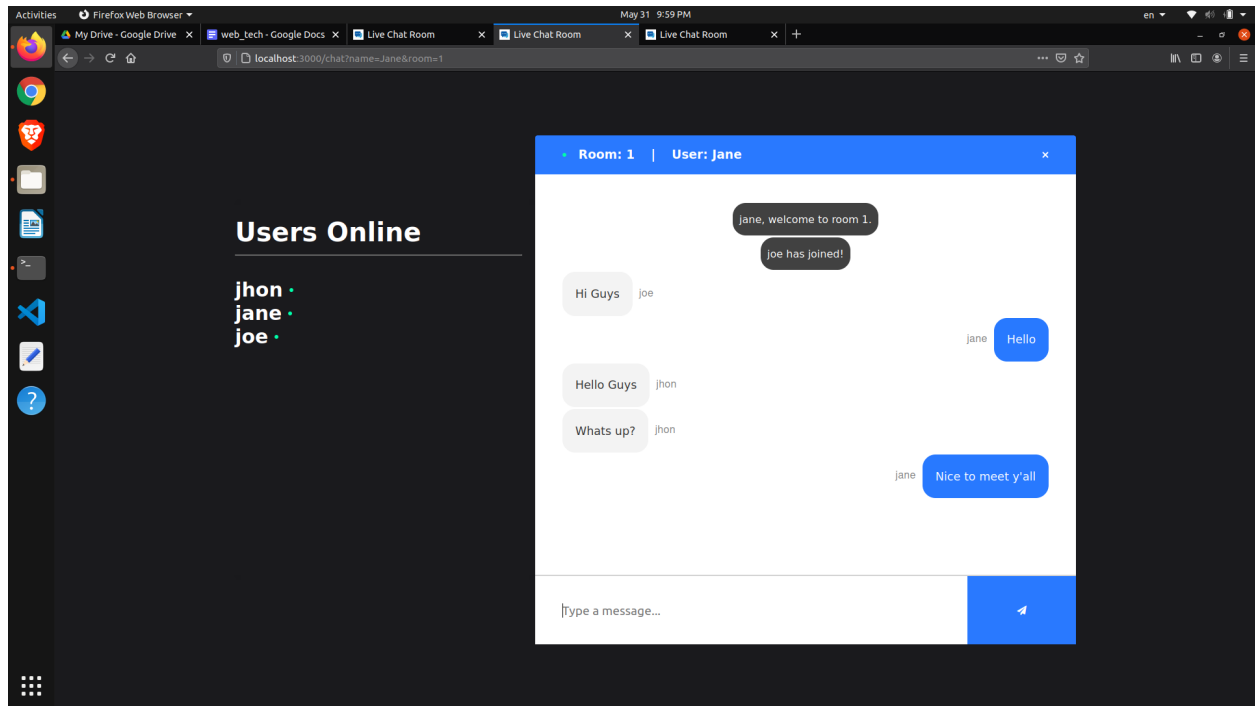




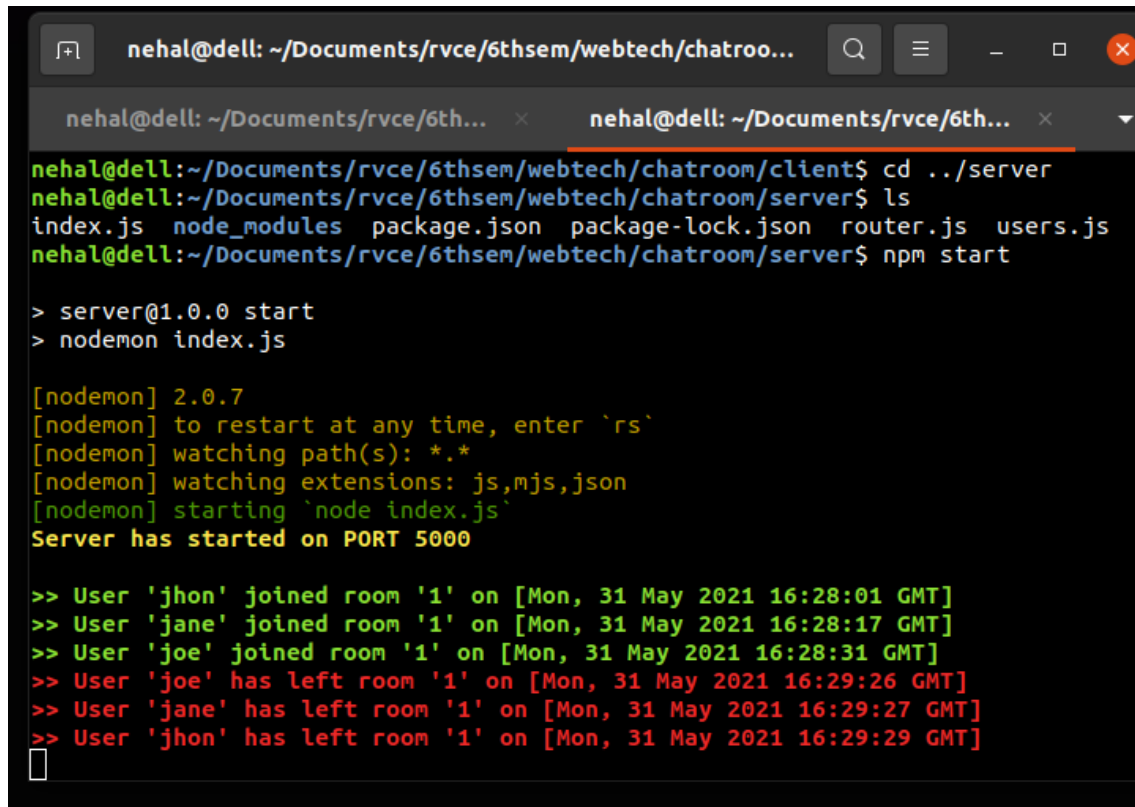


Login Page





User's page



```
nehal@dell: ~/Documents/rvce/6thsem/webtech/chatroom...
nehal@dell: ~/Documents/rvce/6thsem/webtech/chatroom/client$ cd ../server
nehal@dell:~/Documents/rvce/6thsem/webtech/chatroom/server$ ls
index.js  node_modules  package.json  package-lock.json  router.js  users.js
nehal@dell:~/Documents/rvce/6thsem/webtech/chatroom/server$ npm start

> server@1.0.0 start
> nodemon index.js

[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Server has started on PORT 5000

>> User 'jhon' joined room '1' on [Mon, 31 May 2021 16:28:01 GMT]
>> User 'jane' joined room '1' on [Mon, 31 May 2021 16:28:17 GMT]
>> User 'joe' joined room '1' on [Mon, 31 May 2021 16:28:31 GMT]
>> User 'joe' has left room '1' on [Mon, 31 May 2021 16:29:26 GMT]
>> User 'jane' has left room '1' on [Mon, 31 May 2021 16:29:27 GMT]
>> User 'jhon' has left room '1' on [Mon, 31 May 2021 16:29:29 GMT]
```

Server side logs

## REFERENCES

- <https://nodejs.org/en/docs/>  
Node.js Documentation
- <https://reactjs.org/docs/getting-started.html>  
ReactJS Documentation
- <https://stackoverflow.com/>  
StackOverflow
- <https://socket.io/docs/v4>  
Socket.IO
- <https://whatis.techtarget.com/definition/chat-room>  
Chatroom Blog