Chatting Engine

Introduction:: Server Side Events

We are going to make a chatting engine.

Server-Sided Events - A server-sent event is when a web page automatically gets updates from a server. With server-sided events, the updates come automatically.

The technology used behind this is called web sockets.
We will be using web sockets to create the chatting engine where one user will send a message and the other user will see a notification immediately in the chatbox that a new message has been received.

EXTRA:
A small mini assignment to create a chatbox layout using basic styling that is fixed over the bottom of the page. Add some dummy messages in it initially.

Reaching Sockets

- The term **observer subscriber software design pattern** - There is an observer and there are two subscribers { user1 and user2} who have actually subscribed to the server.
- The server is watching over these two users. Whenever a user sends something to the other user, the server pushes all the updates to every other user or the designated user.
- The server acts as the observer and users are the subscribers to this service.

- Let us assume there is a user X and user Y and that there is a server in between them. Both of them are online. In the database, there is a table that stores the messages and timestamp at which the message was delivered or sent by the user.
- User Y is repeatedly asking the server if there is a new message in the database that is directed to him. If there is, then our job is to display it. This process is called **Polling,** and it is performed by every user.
- Polling puts a lot of load on the server as the server is repeatedly answering back with empty requests.
- **Long Polling** - User X sends the request that whenever another message comes in tell the user. The server holds the request instead of giving the response. Similarly, user Y will also send the same request, which will be held by the server. If a third user, say user Z, sends a message to the other users that there is a message directed towards user X and user Y, the server returns the **long-held request**. After the **long-held request** is returned to them**,** user X and Y both will make another request, repeating the entire cycle. This is known as long polling and also requires a lot of server resources.
- **Web Sockets** - Currently the communication is one way in terms of making the request. But with a **web socket**, there is the creation of a two-way communication channel. This two-way communication channel is not running on normal HTTP protocol. A different protocol is defined for these sockets.
- Earlier there was only request and response between the user and the server. Using web sockets, there is an event detection mentioned below.

There are four main Web Socket API events −

- Open - when the connection has been established between the client and the server, the open event is fired from the Web Socket instance
- Message - Message event happens usually when the server sends some data.
- Close - Close event marks the end of the communication between the server and the client

- Error - Error marks for some mistake, which happens during the communication. It is marked with the help of an error event

Understanding the Chat Box

{ Views / chat_box.ejs }

```
<!-- CHANGE :: Creat the code for chat box -->
<% if (locals.user){ %>
        <div id="user-chat-box">
                <ul id="chat-messages-list">
                        <li class="other-message">
                                <span>Other Message</span>
                        </li>
                        <li class="self-message">
                                <span>
                                        Self Message
                                </span>

                        </li>

                </ul>
                <div id="chat-message-input-container">
                        <input id="chat-message-input" placeholder="Type message here">
                        <button id="send-message">Send</button>
                </div>

        </div>
<% } %>
```

***EXTRA*:**
Style the chatbox according to your preference

Setting Up Socket.io

- Install socket.io using the command { **npm install socket.io** } in the terminal.

- In the config folder create a new file { **chat_sockets.js** }.
- Include the socket.io library inside the { **index.js** } file.

{ **chat_sockets.js** }

```
module.exports.chatSockets = function(socketServer){

}
```

{ index.js }

```
// setup the chat server to be used with socket.io
const chatServer = require('http').Server(app);
const chatSockets = require('./config/chat_sockets').chatSockets(chatServer);
chatServer.listen(5000);
console.log('chat server is listening on port 5000');
```

- We have to create one file for the frontend as well.
- Inside the **JS** folder create a new file **{ chat_engine.js }.**
- We need to include the chat engine and socket.io library inside { **home. ejs** } file.

{ **home. ejs** }

```
    <%- include('_chat_box') -%>
</div>

<!-- importing this script for creating the comments -->
<script src="/js/home_post_comments.js" ></script>
<script src="/js/home_posts.js"></script>

<!-- CHANGE :: import the script file for toggle likes, also run it over for the already present posts and comments on the page -->
<script src="js/toggle_likes.js"></script>
<script>
    $('.toggle-like-button').each(function(){
        let self = this;
        let toggleLike = new ToggleLike(self);
    });
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.2.0/socket.io.js"></script>
<script src="/js/chat_engine.js"></script>
```

***EXTRA*: You can look at the link below to know about JQuery**

https://socket.io/

Beginning the Initial Connection

- We have to create a connection between the user and the server { observer and subscriber }.
- The user always initiates the connection then the observer detects it. The server then acknowledges that the connection has been established.
- We will begin with the front end by creating a class to enable the functionalities.
- The class will have a constructor that will take two parameters - { id of the chatbox and the email id of the user who is initiating the connection to know who is sending the message }.
- We need to initiate the connection on port 5000.
- IO is a global variable that is available as soon we include the file { **socket_io.js** } on CDN.js.

{ chat_sockets.js }

```
module.exports.chatSockets = function(socketServer){
    let io = require('socket.io')(socketServer);

    io.sockets.on('connection', function(socket){
        console.log('new connection received', socket.id);

        socket.on('disconnect', function(){
            console.log('socket disconnected!');
        });

    });

}
```

- We will create a connection, Handler, that has to and fro interaction between the server and user.

{ chat_engine.js }

```
class ChatEngine{
    constructor(chatBoxId, userEmail){
        this.chatBox = $(`#${chatBoxId}`);
        this.userEmail = userEmail;

        this.socket = io.connect('http://localhost:5000');

        if (this.userEmail){
            this.connectionHandler();
        }

    }


    connectionHandler(){
        this.socket.on('connect', function(){
            console.log('connection established using sockets...!');
        });
    }
}
```

- We need to initialize the class inside the { **home.ejs** } file by passing the id of the chatbox {user-chat-box}.

{ **home.ejs** }

```
    <%- include('_chat_box') -%>
</div>

<!-- importing this script for creating the comments -->
<script src="/js/home_post_comments.js" ></script>
<script src="/js/home_posts.js"></script>

<!-- CHANGE :: import the script file for toggle likes, also run it over for the already present posts and comments on the page -->
<script src="js/toggle_likes.js"></script>
<script>
    $('.toggle-like-button').each(function(){
        let self = this;
        let toggleLike = new ToggleLike(self);
    });
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.2.0/socket.io.js"></script>
<script src="/js/chat_engine.js"></script>
<% if (locals.user){ %>
<script>
    new ChatEngine('user-chat-box', '<%= locals.user._id %>')
</script>
<% } %>
```

- To receive the request for establishing a connection, we need a { **chat_socket.js** } file.
- Whenever a connection request is received, the server automatically will send back the acknowledgment to the frontend.
- Whenever the client disconnects from the server, an automatic disconnect event is fired.

{ **chat_socket.js** }

```
module.exports.chatSockets = function(socketServer){
    let io = require('socket.io')(socketServer);

    io.sockets.on('connection', function(socket){
        console.log('new connection received', socket.id);

        socket.on('disconnect', function(){
            console.log('socket disconnected!');
        });

    });

}
```

Time to Create and Join Chat Rooms

- Whenever two or more users are chatting with each other, we create a virtual environment that is called a Chat Room.
- There is an array of id's and emails wherein a user's email and unique ids are stored for a particular chatbox.
- One socket connection can hold multiple sockets.
- Whenever a user is connected or clicks a specific name, the user will ask to start a chat with that specific user.
- We will send a request to the other user and we receive an acknowledgment of that sent-in request. Whenever the other user also comes online or joins the room we will receive a notification.

{ chat_engine.js }

```
class ChatEngine{
    constructor(chatBoxId, userEmail){
        this.chatBox = $(`#${chatBoxId}`);
        this.userEmail = userEmail;

        this.socket = io.connect('http://localhost:5000');

        if (this.userEmail){
            this.connectionHandler();
        }

    }


    connectionHandler(){
        let self = this;

        this.socket.on('connect', function(){
            console.log('connection established using sockets...!');


            self.socket.emit('join_room', {
                user_email: self.userEmail,
                chatroom: 'codeial'
            });

            self.socket.on('user_joined', function(data){
                console.log('a user joined!', data);
            })


        });
    }
}
```

- We will send the join room request to other users.
- Whenever the joining request has been received, we just want that socket to be joined to that particular room.
- Once the chat room is joined, all the other users present in the chat room should receive a notification that the user has joined that chat room.

{ **chat_socket.js** }

```
module.exports.chatSockets = function(socketServer){
    let io = require('socket.io')(socketServer);

    io.sockets.on('connection', function(socket){
        console.log('new connection received', socket.id);

        socket.on('disconnect', function(){
            console.log('socket disconnected!');
        });


        socket.on('join_room', function(data){
            console.log('joining request rec.', data);

            socket.join(data.chatroom);

            io.in(data.chatroom).emit('user_joined', data);
        })

    });

}
```

Finally, Let's Send and Receive Messages.

- We have created a brief part of sending the message code.
- Whenever we type something and press the send button it should emit an event within the same chat room.
- Whenever the emit event is detected on the server-side, the server will broadcast it to everyone in the chat room.
- We have sent the event to the server with the chatroom name, the server receives the chat room name and broadcasts it to other users in the chat room.
- If the message is not empty, we emit it on the socket with the chat room name.

{ **chat_engine.js** }

```
// CHANGE :: send a message on clicking the send message button
$('#send-message').click(function(){
    let msg = $('#chat-message-input').val();
    if (msg != ''){
        self.socket.emit('send_message', {
            message: msg,
            user_email: self.userEmail,
            chatroom: 'codeial'
        });
    }
});
```

- We have to detect the event on the server.

{ **chat_socket.js** }

```
        // CHANGE :: detect send_message and broadcast to everyone in the room
        socket.on('send_message', function(data){
            io.in(data.chatroom).emit('receive_message', data);
        });

    });

}
```

- We have to create the message box whenever we receive the message, and then display it.

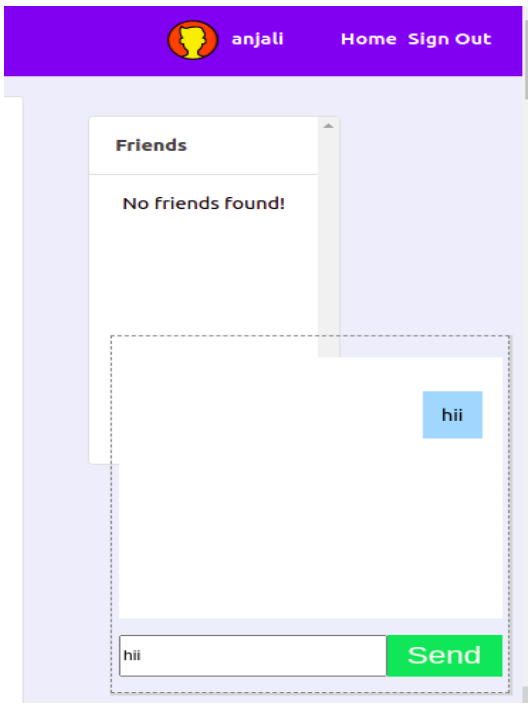
{ chat_engine.js }

```
        self.socket.on('receive_message', function(data){
            console.log('message received', data.message);

            let newMessage = $('<li>');

            let messageType = 'other-message';

            if (data.user_email == self.userEmail){
                messageType = 'self-message';
            }
           newMessage.append($('<span>', {
                'html': data.message
            }));

            newMessage.append($('<sub>', {
                'html': data.user_email
            }));

            newMessage.addClass(messageType);

            $('#chat-messages-list').append(newMessage);
        })
    }
}
```

- We are not storing any messages in the database right now. To do so, we need to create a schema for the same and store a message each time it is sent from one user to the other.

***EXTRA**: You can refer to the below page also.*

{ Chat engine }



***EXTRA**: You can look at the link below to know about JQuery*

http://websocket.org/demos.html