SOCKET.IO - Live Chat Communication

domenica 3 dicembre 2023 12:42

What is Socket.io?

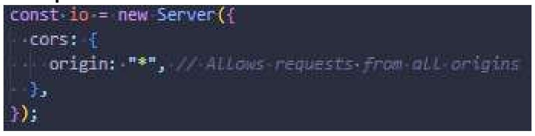
Socket.IO is a library that enables real-time, bidirectional, and event-based communication between web clients and server.
It uses WebSocket as its primary transport but will seamlessly switch to HTTP long polling as a fallback when necessary.
So we can say:

1. **Real-time communication:** Socket.IO allows for the exchange of data in real-time between clients and servers, which is ideal for applications alike online chat, multiplayer games, and real-time trading.
2. **Bidirectionality:** Unlike traditional HTTP requests, which are unidirectional, Socket.IO enables bidirectional communication, allowing both the server and the client to initiate communication.
3. **Event-based:** Communication occurs through events, which can be emitted by both the client and the server, and to which both can listen and respond.
4. **Fallbacks:** If WebSocket is not available or is blocked by firewalls or proxies, Socket.IO will automatically attempt to establish a connection using HTTP long polling.

Server-Side Socket.io Methods:

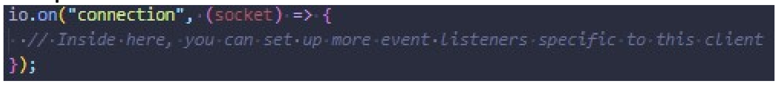
Let's now take a look at Socket.IO implementation in the code:

1. **Initialization with** Server
   - **Purpose**: To create a new Socket.io server instance.
   - **Usage**: Import the Server class from socket.io and initialize it with configurations like CORS to allow cross-origin requests.
   - **Example**:



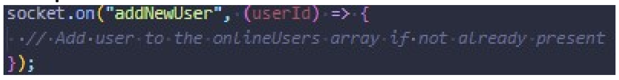
```
const io = new Server({
  cors: {
    origin: "*", // Allows requests from all origins
  },
});
```

2. **Connection Listener with** io.on('connection', callback)
   - **Purpose**: To listen for new client connections to the server.
   - **Usage**: Use this method to set up a connection event listener. When a client connects, the callback function is called with a socket object representing the client.
   - **Example**:



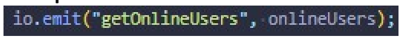
```
io.on("connection", (socket) => {
  // Inside here, you can set up more event listeners specific to this client
});
```

3. **Event Listener with** socket.on('event', callback)
   - **Purpose**: To listen for custom events emitted by the client.
   - **Usage**: Inside the connection listener, use socket.on to react to specific events like 'addNewUser' or 'sendMessage'.
   - **Example**:



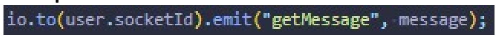
```
socket.on("addNewUser", (userId) => {
  // Add user to the onlineUsers array if not already present
});
```

4. **Broadcasting with** io.emit('event', data)
   - **Purpose**: To send an event to all connected clients.
   - **Usage**: Use io.emit to broadcast data, such as notifying all clients of the updated list of online users.
   - **Example**:



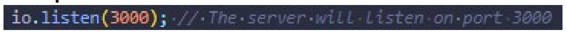
```
 io.emit("getOnlineUsers", onlineUsers);
```

5. **Private Messaging with** io.to(socketId).emit('event', data)
   - **Purpose**: To send an event to a specific client using their socket ID.
   - **Usage**: Use io.to(socketId).emit to send private messages or notifications to a particular client.
   - **Example**:



```
io.to(user.socketId).emit("getMessage", message);
```

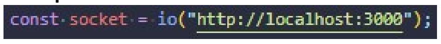
6. **Starting the Server with** io.listen(port)
   - **Purpose**: To start listening for connections on a specified port.
   - **Usage**: Use io.listen to make the server listen on the desired port for incoming connections.
   - **Example**:



```
 io.listen(3000); // The server will listen on port 3000
```

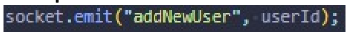
Client-Side Socket.io Methods:

1. **Establishing Connection with** io('URL')
   - **Purpose**: To connect to the Socket.io server from the client.
   - **Usage**: Use io with the server's URL to start the connection process.
   - **Example**:



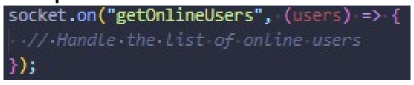
```
const socket = io("http://localhost:3000");
```

2. **Emitting Events with** socket.emit('event', data)
   - **Purpose**: To send events from the client to the server.
   - **Usage**: Use socket.emit to send data to the server, such as a user's ID or a chat message.
   - **Example**:



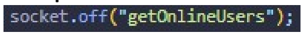
```
socket.emit("addNewUser", userId);
```

3. **Listening for Events with** socket.on('event', callback)
   - **Purpose**: To set up event listeners for events emitted by the server.
   - **Usage**: Use socket.on to handle incoming events, like receiving messages or user status updates.
   - **Example**:



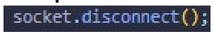
```
socket.on("getOnlineUsers", (users) => {
  // Handle the list of online users
});
```

4. **Removing Event Listeners with** socket.off('event')
   - **Purpose**: To remove event listeners from the client.
   - **Usage**: Use socket.off to clean up event listeners when they're no longer needed, such as when a component unmounts.
   - **Example**:



```
socket.off("getOnlineUsers");
```

5. **Disconnecting with** socket.disconnect()
   - **Purpose**: To disconnect the client from the server.
   - **Usage**: Use socket.disconnect to end the connection, typically when the user logs out or closes the application.
   - **Example**:



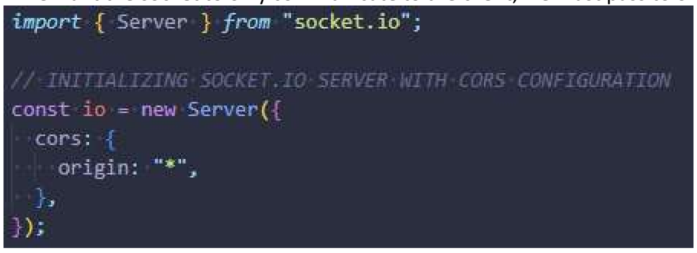
```
socket.disconnect();
```

First of all, let's go to the **socket** backend folder and install it with this command: **npm install socket.io**
We also need to install it in our client with this command: **npm i socket.io-client**

HOW TO MAKE A CHAT WITH SOCKET.IO GUIDE

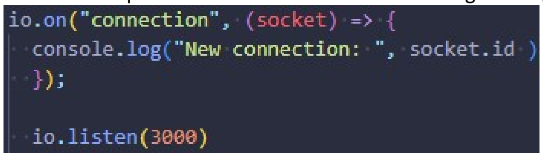
1. **SERVER - INITIALIZE SERVER**

Let's initialize the Socket.io server
If we want the socket to only communicate to the client, we must pass to origin the address of the client server.



```
import { Server } from "socket.io";

// INITIALIZING SOCKET.IO SERVER WITH CORS CONFIGURATION
const io = new Server({
  cors: {
    origin: "*",
  },
});
```

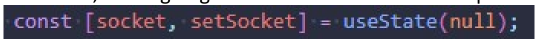
io.on listens to an event called **"connection".**
Every time the front-end connects to the server, a new connection **(socket.id)** will be found.
Doing it this way, every time we refresh the page on the front-end, the socket.id of the current logged-in user will change. We'll see later on how to fix this issue and keep track of the socket.id without letting it change (so we can have a private chat).



```
io.on("connection", (socket) => {
  console.log("New connection: ", socket.id )
  });

  io.listen(3000)
```

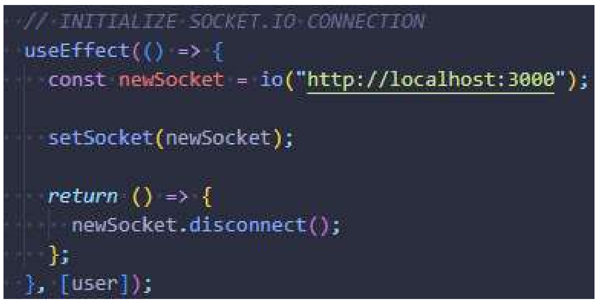
2. **CLIENT - INITIALIZE CLIENT**

Let's initialize the Socket.io client
First of all, we're going to create a state that will help us to set the



```
const [socket, setSocket] = useState(null);
```

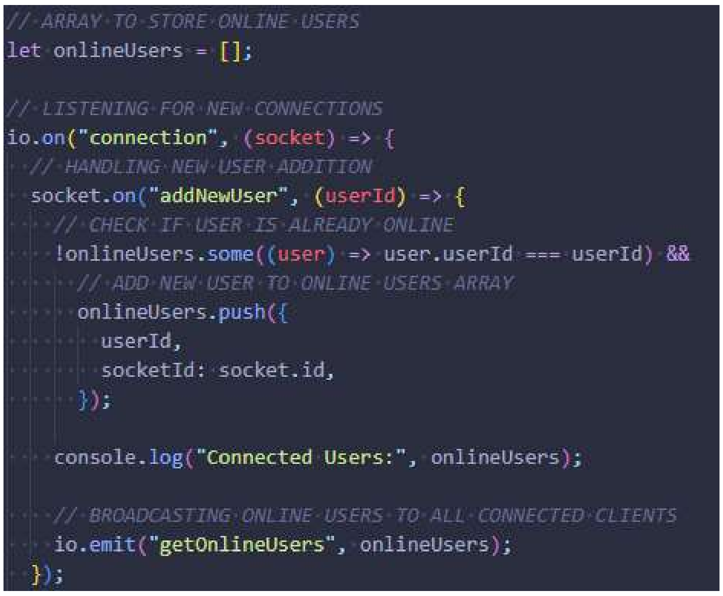
Let's now connect the Socket.io **client** to the Socket.io **server** by passing the **address** of the server as a parameter.
Then we will set the connection we enstablished with the server in our state called **socket.**
From now on, we'll use the **socket** state to do all the operations we need.
For last, we're going to return a clean-up function so that we can disconnect the socket if we're no longer using it or if we're trying to connect again.
So, if we're trying to connect again, the clean-up function will disconnect and then connect again to the socket.
Also, we're added the current logged-in user to the dependecy array so that whenever we have a new user, we restart the connection.



```
// INITIALIZE SOCKET.IO CONNECTION
useEffect(() => {
  const newSocket = io("http://localhost:3000");

  setSocket(newSocket);

  return () => {
    newSocket.disconnect();
  };
}, [user]);
```

3. SERVER - FIND ONLINE USERS

The code we're about to write:



```
// ARRAY TO STORE ONLINE USERS
let onlineUsers = [];

// LISTENING FOR NEW CONNECTIONS
io.on("connection", (socket) => {
  // HANDLING NEW USER ADDITION
  socket.on("addNewUser", (userId) => {
    // CHECK IF USER IS ALREADY ONLINE
    !onlineUsers.some((user) => user.userId === userId) &&
      // ADD NEW USER TO ONLINE USERS ARRAY
      onlineUsers.push({
        userId,
        socketId: socket.id,
      });

    console.log("Connected Users:", onlineUsers);

    // BROADCASTING ONLINE USERS TO ALL CONNECTED CLIENTS
    io.emit("getOnlineUsers", onlineUsers);
  });
```

Let's try to fix that issue we were talking about before.

Now, the code listens for an **"addNewUser"** event from the client, which passes a **userId** as an argument.
Upon receiving this event, the server will attempt to add the user to the **onlineUsers** array, which tracks all currently connected users.
The user is represented as an object containing both their **userId** and the **socket.id** associated with their connection.

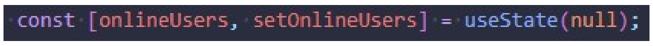
Before adding the user to the array, the code checks if the user is already marked as online by searching the **onlineUsers** array for an existing entry with the same **userId** (we're passing it from the client)
This is done using the **some** method, which returns **true** if a matching user is found.
If the user is not already in the array, (meaning the **some** method returns **false)**, the user's object is pushed into the **onlineUsers** array, effectively marking them as online.

For last, we create an event that can be triggered, and each time it is triggered, we send the list of the onlineUsers from the server. The event is called **"getOnlineUsers"** which refers for all the online users.

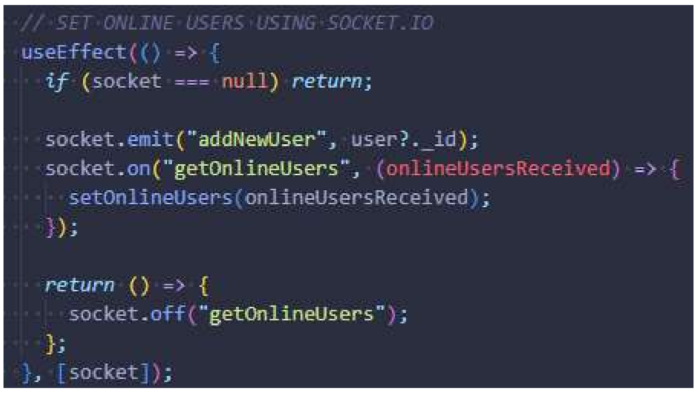
This process ensures that each user is only added once to the **onlineUsers** array, preventing duplicates and allowing accurate tracking of who is currently connected. The updated list of online users is then broadcasted to all connected clients, keeping everyone informed of the active users.
This method works because even if the **socketId** changes, the **userId** will remain the same.

4. CLIENT - ADD GREED DO TO ONLINE USERS

Let's now create a state where to save the current online users and a useEffect function for the purpose:



```
const [onlineUsers, setOnlineUsers] = useState(null);
```



```
// SET ONLINE USERS USING SOCKET.IO
useEffect(() => {
  if (socket === null) return;

  socket.emit("addNewUser", user?._id);
  socket.on("getOnlineUsers", (onlineUsersReceived) => {
    setOnlineUsers(onlineUsersReceived);
  });

  return () => {
    socket.off("getOnlineUsers");
  };
}, [socket]);
```

This is what happens:

Check if we have a connection: Upon mounting the component, we stablish a connection to the Socket.IO server. If the **socket** state is null, indicating no active connection, the function returns early to prevent further execution.

Emitting the "addNewUser" Event: Once a connection is established, we emit the **"addNewUser"** event using **socket.emit.** This event is accompanied by the unique identifier of the currently logged-in user, **user?._id**. This identifier is conditionally accessed to ensure it exists before attempting to emit the event.

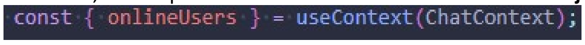
Backend Processing: The backend server listens for the **"addNewUser"** event. Upon receiving a **userId**, it checks whether the user is already present in the **onlineUsers** array. If not, the server adds the user to this array, effectively marking them as online.

Receiving Online Users List: The server then emits the **"getOnlineUsers"** event, broadcasting the updated list of online users to all connected clients. On the frontend, we listen for this event with **socket.on** and update our local state with the received list of online users using the **setOnlineUsers** function.

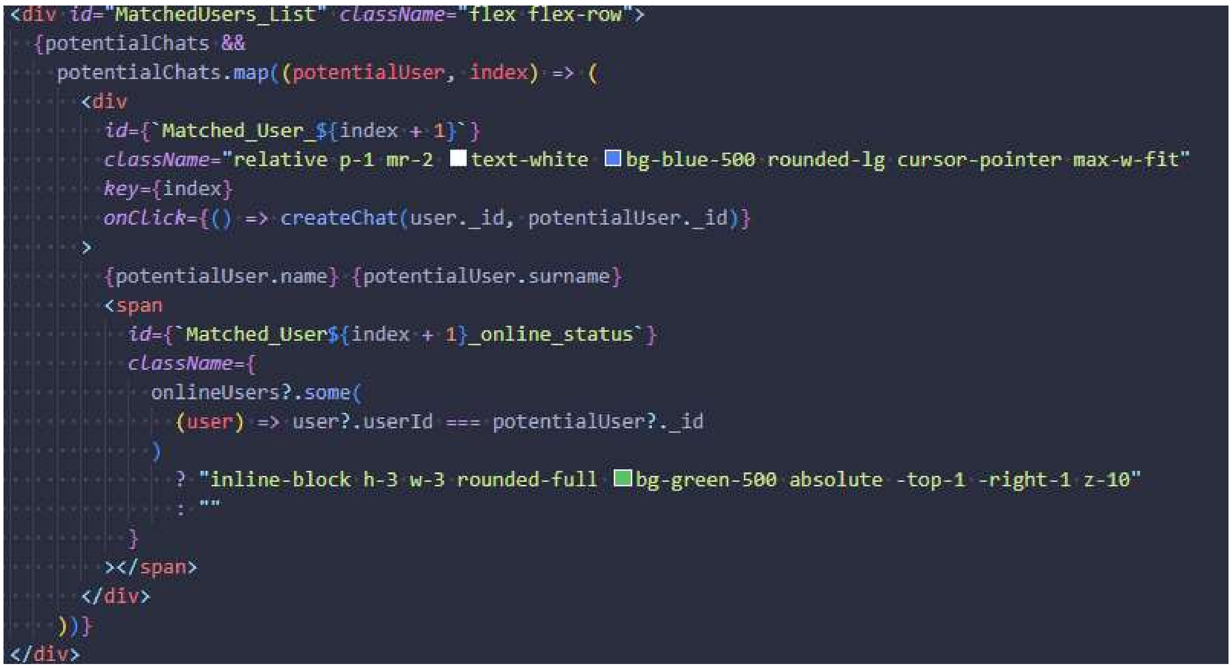
Cleaning Up: To maintain clean and efficient resource usage, we define a cleanup function within the **useEffect** hook. This function is executed when the component unmounts. It removes the event listener for **"getOnlineUsers"**, ensuring that we do not receive updates after the component is no longer active.

Dependency Array: The **socket** state is included in the dependency array of the **useEffect** hook. This means that if the **socket** state changes, indicating a new connection, the hook will re-run, re-establishing the event listeners and ensuring that our application remains in sync with the server.

Now that we have populated the onlineUsers state, we can now use it to create a feature: give the green dot to the current online user.
Let's go in **MatchedUsers.jsx** component and add a condition to the **<span>** which will be our green dot next to the user.
First of all, let's import the onlineUser state from **ChatContext.jsx**



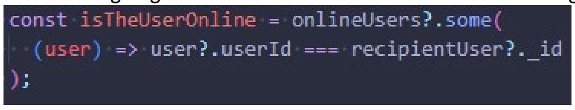
```
const { onlineUsers } = useContext(ChatContext);
```



```
<div id="MatchedUsers_List" className="flex flex-row">
  {potentialChats &&
    potentialChats.map((potentialUser, index) => (
      <div
        id={`Matched_User_${index + 1}`}
        className="relative p-1 mr-2 text-white bg-blue-500 rounded-lg cursor-pointer max-w-fit"
        key={index}
        onClick={() => createChat(user._id, potentialUser._id)}
      >
        {potentialUser.name} {potentialUser.surname}
        <span
          id={`Matched_User${index + 1}_online_status`}
          className={
            onlineUsers?.some(
              (user) => user?.userId === potentialUser?._id
            )
              ? "inline-block h-3 w-3 rounded-full bg-green-500 absolute -top-1 -right-1 z-10"
              : ""
          }
        ></span>
      </div>
    ))}
</div>
```

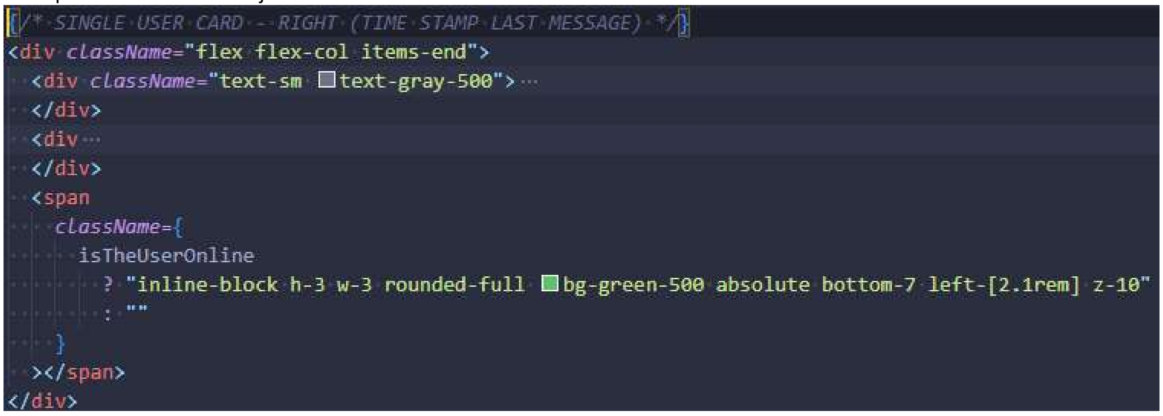
We're going to use the **some method** on the onlineUsers array to check if it contains an ID the is equal the current mapped potential user id.
If it is true, it'll add a green dot next to its icon.

We can do the same in our **UserCard** component.
We'll import the onlineUsers array and use it to add the green dot if the user is online.
Then we're going to create a variable which will contain the login to check if the recipient user is online. Afterward, we'll pass this condition where needed.



```
const isTheUserOnline = onlineUsers?.some(
  (user) => user?.userId === recipientUser?._id
);
```

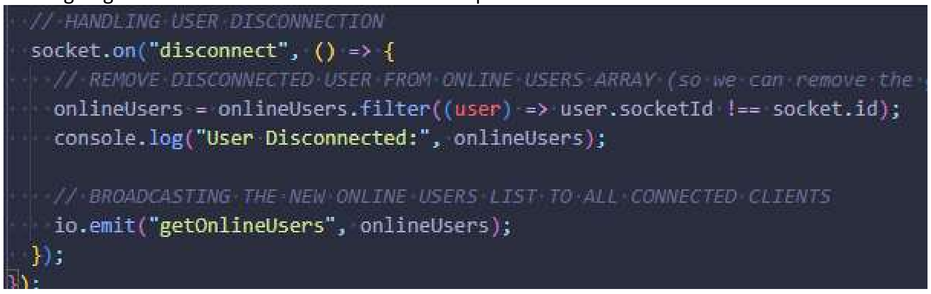
We'll pass the condition we just created here:



```
{/* SINGLE USER CARD - RIGHT (TIME STAMP LAST MESSAGE) */}
<div className="flex flex-col items-end">
  <div className="text-sm text-gray-500">…
  </div>
  <div…
  </div>
  <span
    className={
      isTheUserOnline
        ? "inline-block h-3 w-3 rounded-full bg-green-500 absolute bottom-7 left-[2.1rem] z-10"
        : ""
    }
  ></span>
</div>
```

4. SERVER - REMOVE USER FROM ONLINE LIST (IF HE DISCONNECTS)

Now that found a way to add users to the online list, we also need to remove them.
We're going to use the function below to make this possible:

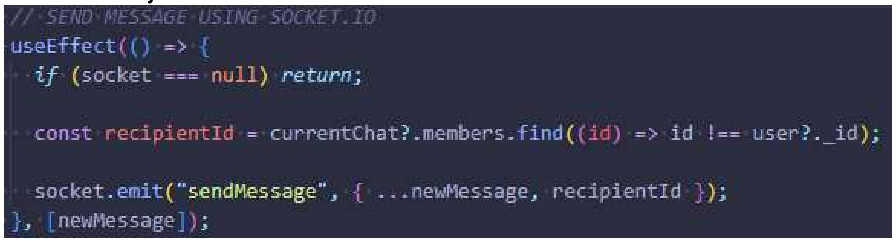


```
  // HANDLING USER DISCONNECTION
  socket.on("disconnect", () => {
    // REMOVE DISCONNECTED USER FROM ONLINE USERS ARRAY (so we can remove the
    onlineUsers = onlineUsers.filter((user) => user.socketId !== socket.id);
    console.log("User Disconnected:", onlineUsers);

    // BROADCASTING THE NEW ONLINE USERS LIST TO ALL CONNECTED CLIENTS
    io.emit("getOnlineUsers", onlineUsers);
  });
});
```

When a user disconnects from the front-end, a **"disconnect"** event is automatically triggered by Socket.IO.
The server side **socket.on('disconnect', ...)** listener function detects the event.
The code inside the listener function then filters out the disconnected user from the onlineUsers array by matching **socket.id**
This ensures that only the remaining connected users are in the array.
Finally, the updated list of online users is broadcasted to all connected clients using **io.emit('getOnlineUsers', onlineUsers)**, which can be used to update the user status (like removing the green dot next to their icon) on the client side.

5. CLIENT - SEND A MESSAGE IN REAL TIME

Let's now see how we can update our messages in real time.
In **ChatContext.jsx** let's add this useEffect function:



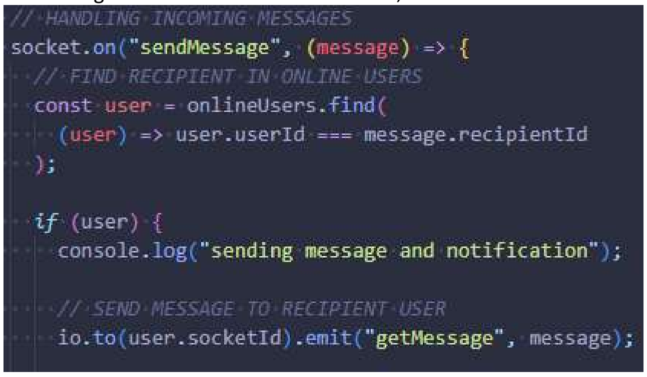
```
// SEND MESSAGE USING SOCKET.IO
useEffect(() => {
  if (socket === null) return;

  const recipientId = currentChat?.members.find((id) => id !== user?._id);

  socket.emit("sendMessage", { ...newMessage, recipientId });
}, [newMessage]);
```

1. **Socket Check:** Initially, the function checks if the **socket** object is not **null.** If it is **null**, this means that there is no active socket connection, so the function will return early and not proceed further.
2. **Recipient Identification:** The function then determines the **recipientId** by searching through the **currentChat.Members** array. It finds the member's ID that is not equal to the **user._id**, ensuring that the message is not sent to the sender themselves.
3. **Message Emission:** With the **recipientId** identified, the function emits a **sendMessage** event via the socket. This event includes an object that contains the **newMessage** (the last message we sent to the database)**,** and the **recipientId**, specifying the intended recipient of the message.
4. **Dependency Array:** The **newMessage** state variable is included in the dependency array of the **useEffect** hook. This means that the function will re-execute whenever the **newMessage** state changes, which typically occurs when a new message is sent and received from the server.

RECAP: This **useEffect** hook is triggered when there is a change in the **newMessage state.** If the socket connection is active, it identifies the recipient user of the current chat, and emits the message to that specific user through the socket connection.

6. SERVER - TAKE THE SENT MESSAGE AND DELIVER IT TO THE RIGHT RECIPIENT USER

Now let's go back in our socket.io index file, and add this:



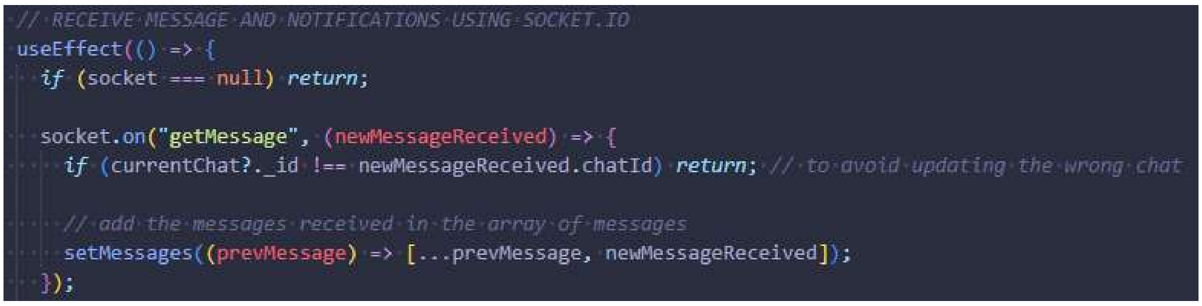
```
// HANDLING INCOMING MESSAGES
socket.on("sendMessage", (message) => {
  // FIND RECIPIENT IN ONLINE USERS
  const user = onlineUsers.find(
    (user) => user.userId === message.recipientId
  );

  if (user) {
    console.log("sending message and notification");

    // SEND MESSAGE TO RECIPIENT USER
    io.to(user.socketId).emit("getMessage", message);
```

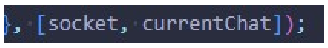
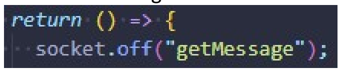
1. **Event Listening:** the **socket.on("sendMessage", ...)** listener is set up to handle the **"sendMessage"** event. It receives a **message** object that contains details of the message, including the user we want to send this message to. (This is what we were passing from current logged in user in the front-end: {...newMessage, recipientId}, just for reminder)
2. **Recipient user Lookup:** The code then searches the **onlineUsers** array to find the user where the **userId** matches the **message.recipientId**. This ensures that the message is directed to the correct recipient who is currently online.
3. **Message delivery:** If the recipient user is found (indicating that he is currently online), the server then emits a **"getMessage"** event to the recipient's socket ID **(io.to(user.socketId))** , along with the message object (which we know contains the message and the recipient id). This action allows the recipient's client to receive the new message and update the chat interface accordingly.

RECAP: The **"getMessage"** event is a crucial part of the real-time communication process in the chat application. It ensures that when a user sends a message, the intended recipient, if online, will receive it.
The logic behind this is: the client sends the new message, socket.io server receives and sends it to the right recipient user (front-end) of the current chat.

7. CLIENT - THE RECIPIENT USER RECEIVES THE MESSAGE

Now it's time for the current user online to receive the new message that the person sent.


```
// RECEIVE MESSAGE AND NOTIFICATIONS USING SOCKET.IO
useEffect(() => {
  if (socket === null) return;

  socket.on("getMessage", (newMessageReceived) => {
    if (currentChat?._id !== newMessageReceived.chatId) return; // to avoid updating the wrong chat

    // add the messages received in the array of messages
    setMessages((prevMessage) => [...prevMessage, newMessageReceived]);
  });
```

1. **Socket check:** The function first checks if the **socket** object is **null.** If it is, indicating there is no active socket connection, the function will return early and not proceed further.
2. **Chat validation:** The function listens for the **"getMessage"** event. When a new message is received, it checks if the **chatId** of the received message matches the **chatId** of the current chat. This step ensures that the message belongs to the active chat session. If the **chatId** does not match, the function returns early to prevent updating the wrong chat.
3. **Message update:** if the **chatId** matches, the function updates the state by adding the new message to the existing array of messages. This is done using the **setMessages** function, which appends the new message to the previous messages.
4. **Dependency Array:** the **socket** and **currentChat** variables are included in the dependency array of the **useEffect** hook. This means that the function will re-execute whenever there is a change in the socket connection or the current chat context.
   
   ```
   }, [socket, currentChat]);
   ```
5. **Cleanup function:** A clean up function is provided to remove the event listeners for **"getMessage"** when the component unmounts or before the **useEffect** runs again. This ensures that updates are not receive after the user logs our or switches chats.
   
   ```
   return () => {
     socket.off("getMessage");
   ```

RECAP: This **useEffect** hook ensures that the user's chat interface is updated with new messages as they are received, if part of the current chat session.