SOCKET.IO -

Socket.IO is a JavaScript library that makes **real-time, two-way communication** between clients (like browsers) and servers easier.

HTTP Vs WebSocket

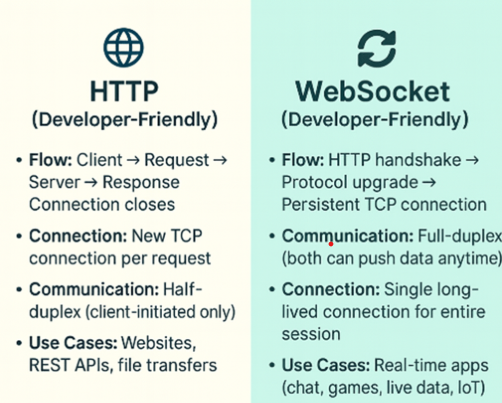
***1.HTTP -***

* **Flow:** Client sends a request → Server replies → Transaction ends
* **Connection:** A new TCP connection is established for each request
* **Communication:** Half-duplex (only client can initiate communication)
* **Use Cases:** Websites, REST APIs, file downloads/uploads

## *2.WebSocket -*

* **Flow:** HTTP handshake → Protocol upgrade → Persistent TCP connection
* **Connection:** A single long-lived connection is maintained for the entire session
* **Communication:** Full-duplex (both client and server can send data anytime)
* **Use Cases:** Real-time applications such as chat, multiplayer games, live data streaming, and IoT

HTTPis stateless, new connection per request, half-duplex whereasWebSocketis stateful, one persistent connection, full-duplex.



### ***1️. HTTP Handshake***

* Before a WebSocket can be established, the client and server first talk using normal HTTP.
* The browser sends a special **HTTP request** asking,  
  "Hey server, can we switch to WebSocket protocol?"

### ***2️. Protocol Upgrade***

* If the server agrees, it replies with a status like 101 Switching Protocols.
* At this point, the connection **changes** from HTTP rules to **WebSocket rules** — meaning it’s now **full-duplex** and persistent.
* This is called a **protocol upgrade**.

### ***3️. Persistent TCP Connection***

* Once upgraded, the **same TCP connection** stays open for the entire session.
* No reconnect for every message — data flows back and forth instantly.
* This saves time, reduces overhead, and is perfect for **real-time apps.**

## Socket.IO - What It Does

### ***1️. Real-time Communication***

* Server ↔ Client can send messages instantly (no page refresh).
* **Use cases:** Chat apps, multiplayer games, live dashboards, stock price tickers.

### ***2️. WebSocket + Fallback***

* **Tries WebSocket first** (fastest, full-duplex).
* If not supported, **falls back** to HTTP long-polling so it works everywhere.

### ***3️. Event-based Messaging***

You send and listen for custom events instead of raw data streams.



### ***4️. Auto Reconnect-***

### If the connection drops, it automatically tries to reconnect — no extra code needed.

### ***5️. Broadcasting***

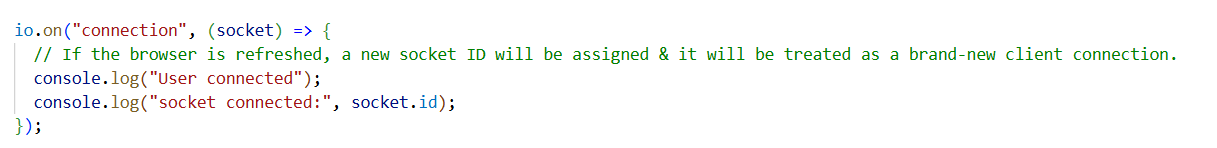
* Send data to **everyone** or only to specific groups (**rooms**).
* Example: Notify only users in "room1" when something changes

Socket.IO works like WhatsApp’s engine - once connected, the client and server can exchange messages instantly, reliably, and even recover smoothly after network interruptions.

*To use Socket.IO, you must* ***create the HTTP server yourself*** *so you can pass it to Socket.IO* ***and*** *still handle Express routes.*



***Backend -***

**

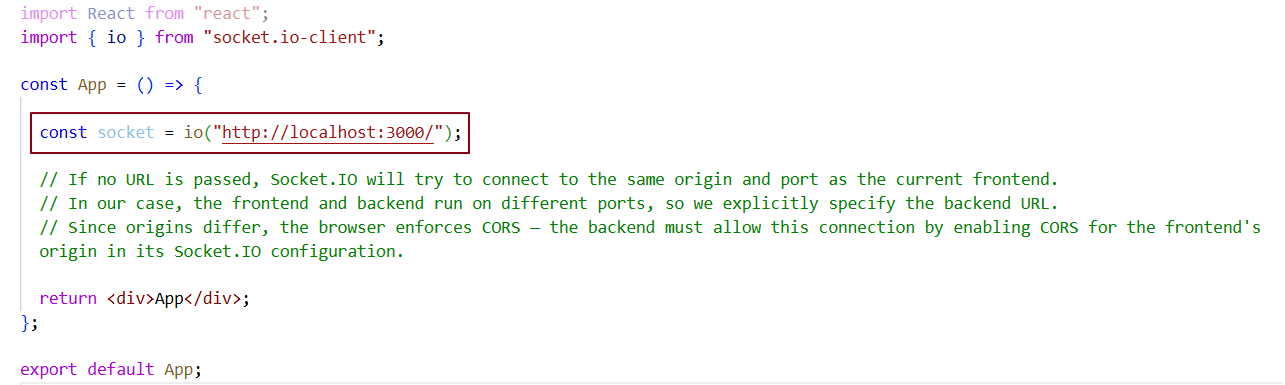
### io.on("connection", handler) -

* io is the server instance / circuit. All connected clients are attached here. When io emits an event, all connected clients that are listening for that event will receive it (broadcast).
* When a client connects to the WebSocket, Socket.IO **internally** triggers connection event.

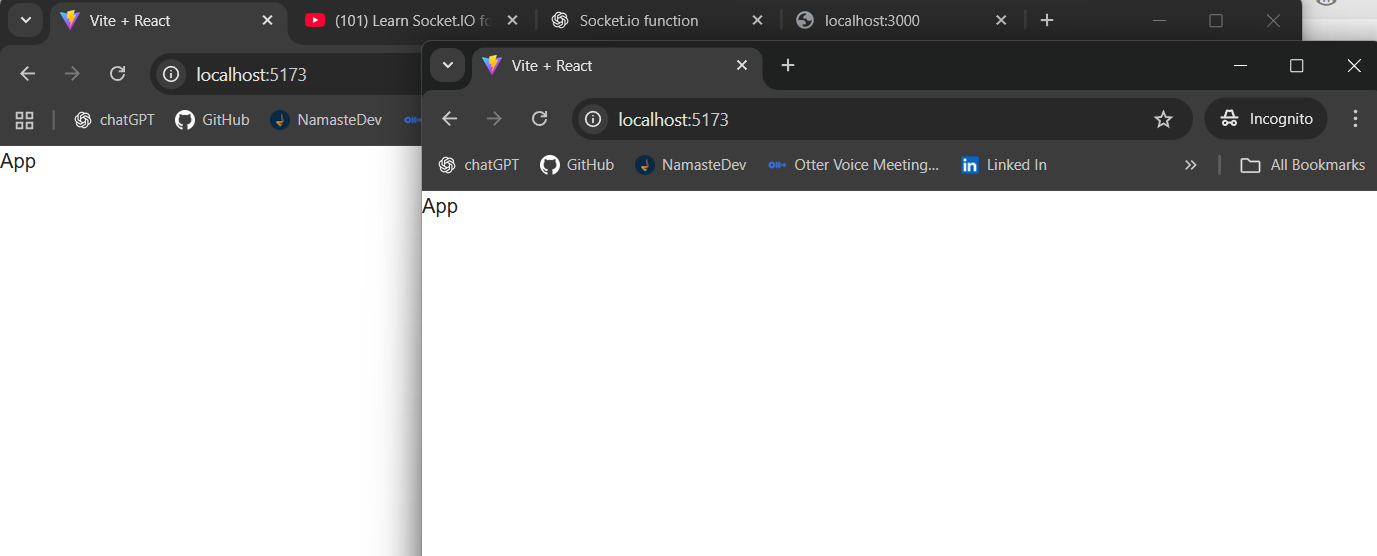
io.emit("connection", socket).

* To handle this built-in event, we must explicitly write io.on("connection", callback) in our code - that’s when our callback runs.

***Frontend –***



***Two clients connected to the Circuit / server –***

**

***Logging their socket ID in server -***



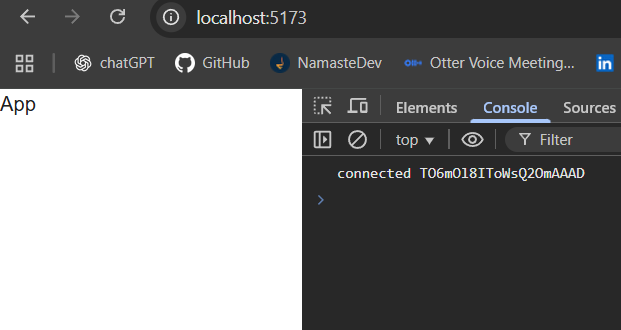
***Logging socket id in client –***



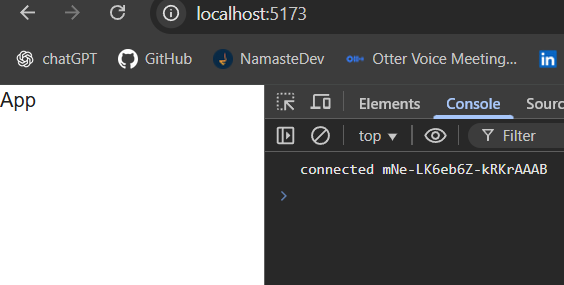
***socket.on("connect", ...) – Execution Flow***

1️. Server is running → Must already be up and listening for Socket.IO connections.  
2️. Server connection listener → io.on("connection", handler) waits for clients.  
3️. No listener? → Client’s connect attempt fails.  
4️. Client initiates connection → Happens when io("http://localhost:3000") runs.  
5️. Handshake sequence → TCP connection → HTTP long-polling → optional WebSocket upgrade.  
6️. Server confirms → Assigns socket.id and marks client as “connected”.  
7️.Client’s connect fires → Callback runs, you can now emit/listen to events.

***Client 1 -***



***Client 2 –***



## *Socket.IO Basics*

IO - The Socket.IO server instance / circuit. All connected clients are attached here. When io emits an event, all connected clients that are listening for that event will receive it (broadcast).

Socket - Represents one specific client connection to the server, identified by a unique socket.id. This allows the server to target events or messages to that exact client.

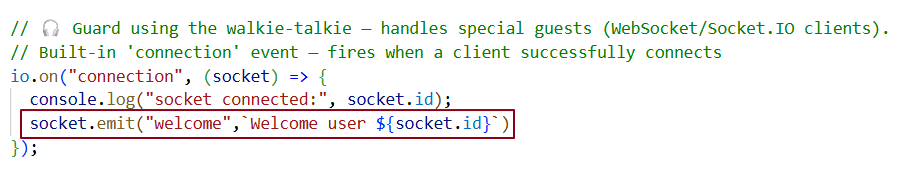
Emit - Sends (emits) an event. Can be called on either the server (io or socket) or client to trigger an event.

On - Listens for an event. Can be called on either the server or client to react when that event is emitted.

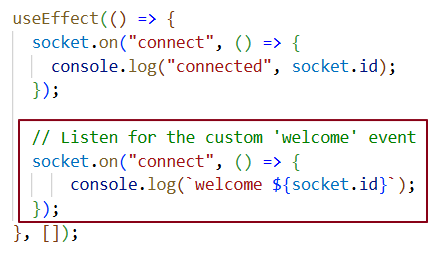
Emitting an Event When a Client Connects –

When a new client connects to a Socket.IO server, we can **emit a custom event from the server** and **listen for it on the client.**

***Backend –***



***Frontend-***



***Execution Flow -***

io.on("connection", handler) (Server)

* Triggered automatically when a new client connects.
* Gives you the socket object for that specific client.

socket.emit(eventName, data) (Server)

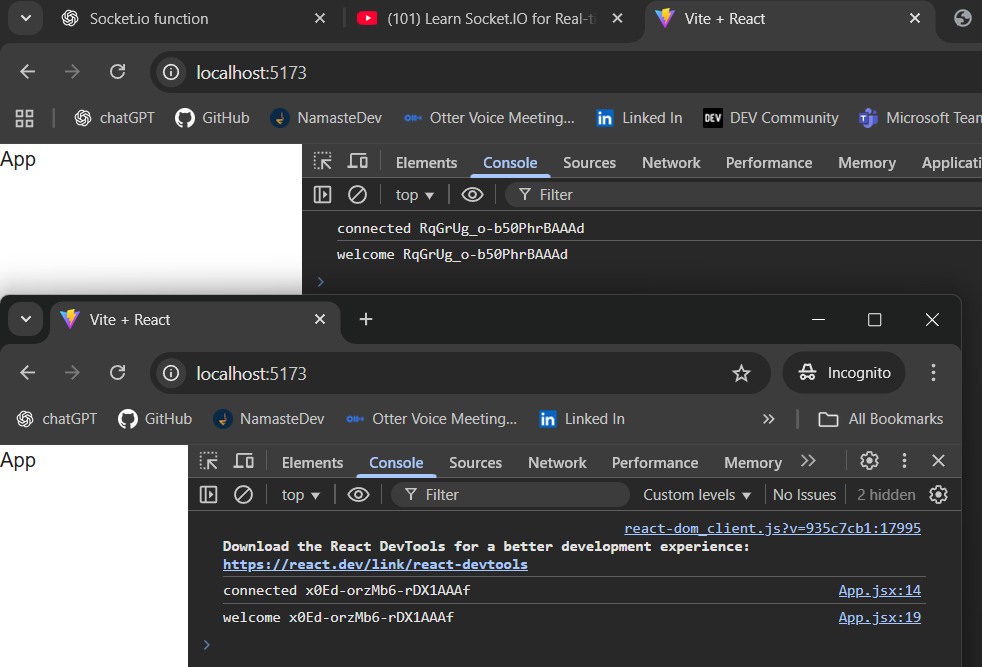
* Sends a **custom event** to that specific connected client.

socket.on(eventName, callback) (Client)

* Listens for events from the server.
* Runs the callback when that event is received.

***In summary -***

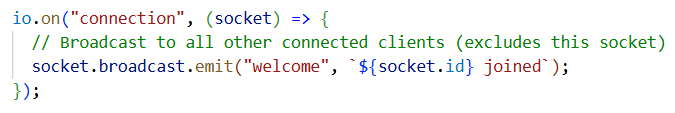
1. Client connects → Server’s connection event triggers.
2. Server emits "welcome" event with message.
3. Client listens for "welcome" → executes callback → logs the message.



On the server -socket.emit("welcome", ...) sends the event only to this connected client (this socket). Other clients do not receive it.

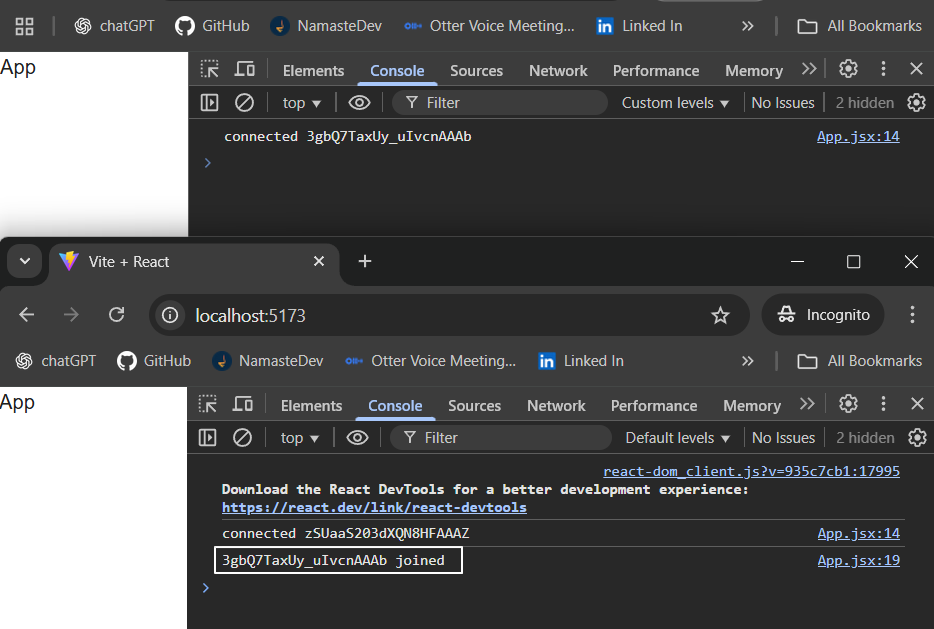
Server broadcast on client connect (others only) -

***Backend -***



***Frontend –***





## *Who receives the event?*

* **Newly connected client (the sender): ❌** does **not** receive welcome (because it’s a broadcast).
* **All other currently connected clients: ✅ do** receive welcome.

## *How to test (what you saw)*

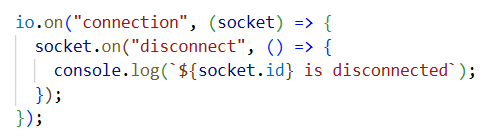
* Open **two tabs** (or one tab + Incognito).
* When **Tab2** connects:
  + Tab2 logs: connected <id2> which is its own connect).
  + **Tab1** logs: <id2> joined (received the broadcast).
  + Tab2 **does not** see <id2> joined (it’s excluded).

Disconnecting a Socket -

***Frontend –***



***Backend –***

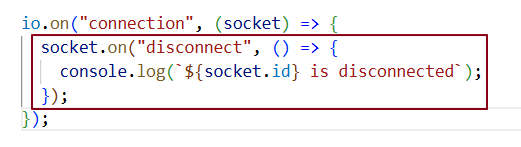


# ***Disconnect Flow -***

## *A) Client triggers disconnect (React Cleanup)*

* Component unmounts / page refresh → useEffect **Cleanup** runs.
* Cleanup calls socket.disconnect().
* The **client** closes the connection and **disables auto-reconnect** for that socket.
* The **server** detects the close and **locally fires** the built-in "disconnect" event (after "disconnecting").

Server handler:

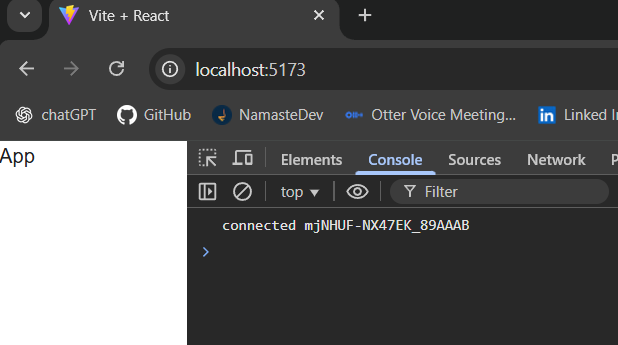


You never emit "disconnect" yourself—Socket.IO fires it automatically.

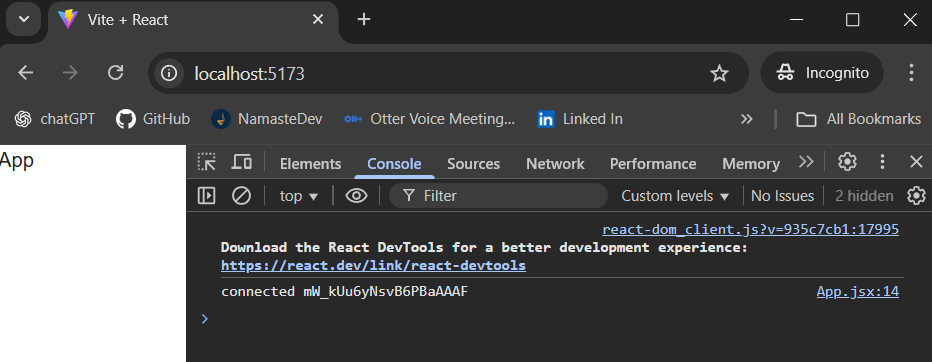
## *B) Page/tab closed or hard refresh (no explicit disconnect)*

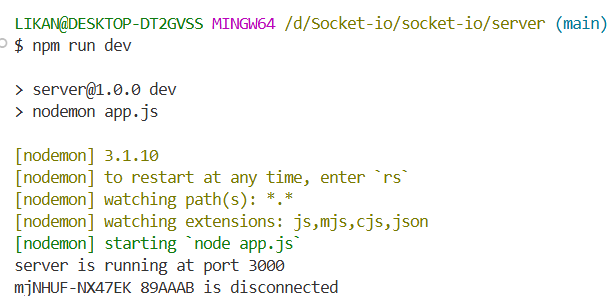
1. Browser kills the page → transport closes abruptly.
2. Server fires "disconnect" with reason **"transport close"**.
3. Client Cleanup may not run on abrupt closes.

***Testing –***



On a page refresh, a new socket connection is created — meaning the old socket is automatically closed (disconnected) by the browser.





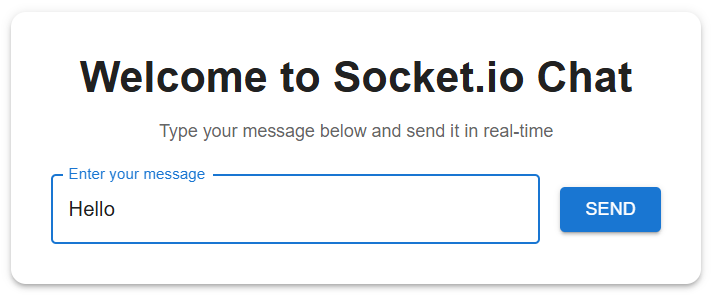
Message Events –

Let’s emit message events from the frontend and handle them in the backend.

***Frontend -***

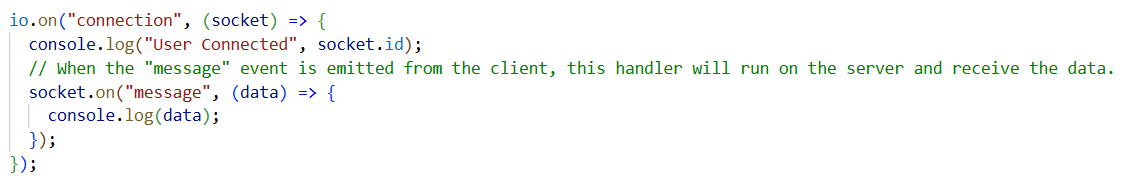
* When the user submits the form (clicks Send or presses Enter), the handleSubmit function runs.
* Inside handleSubmit, the code **fires a Socket.IO event** named "message" using socket.emit("message", message).
* This sends the message text from the browser to the server over the active WebSocket connection.

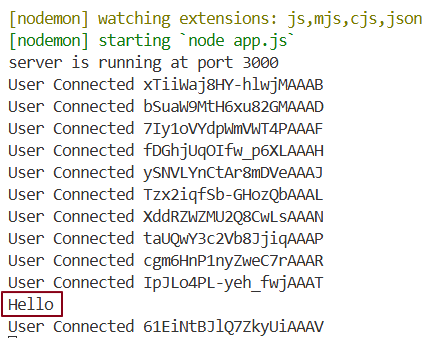




***Backend (Node + Socket.IO server) -***

* The server is listening for incoming socket connections with io.on("connection", ...).
* For each connected client, it sets up an event handler: socket.on("message", (data) => { ... }).
* When the "message" event arrives from the frontend, this handler runs and **logs the received data** to the server console.





## *****Problem Statement*****

Every time the user types a message and clicks **Send**, the backend logs the message (which is correct), **but** it also logs a new "User Connected <socketId>".  
This means a **new socket connection** is being created **on every send** instead of reusing the existing one.

## *****Cause*****

You have: const socket = io("http://localhost:3000/"); inside your component function body.

React re-runs this function **on every render.**  
When you call setMessage("") after sending a message, you trigger a **re-render**, so:

1. The component body runs again.
2. io(...) is executed again.
3. A brand-new socket instance is created.
4. The server sees this as a **new client connection** and fires the connection event again.
5. A new socket.id is generated and logged.

## *****Detailed Flow: How the re-render happens*****

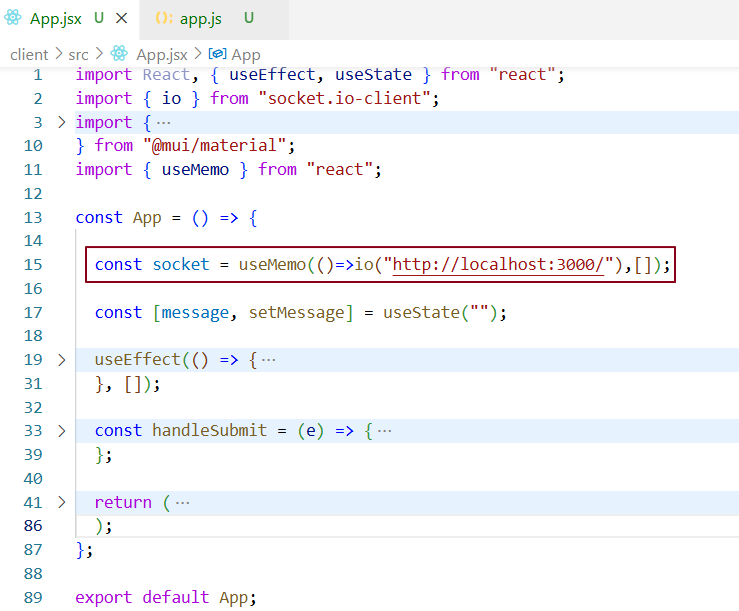
1. **Initial render.**
   * React calls your component function.
   * io("http://localhost:3000/") runs → creates a socket → server logs "User Connected 123".
   * UI shows with the empty input.
2. **User types a message.**
   * onChange calls setMessage("Hello").
   * State changes → React re-renders the component.
   * On re-render, io(...) runs again → new socket connection → server logs "User Connected 456".  
     (Even before clicking send, this can happen if your state updates)
3. **User clicks Send.**
   * handleSubmit runs → emits "message" event to backend → backend logs "Hello".
   * Then setMessage("") runs to clear the input.
   * State changes again → triggers **another** re-render.
   * Component function runs from the top again → io(...) runs again → new socket connection → server logs "User Connected 789".

This is why you’re getting multiple "User Connected" logs — **every render is making a new socket.**

## *****Fix: Keep the socket stable across renders*****

You need to ensure io(...) only runs **once** for the lifetime of the component.  
That way, re-renders caused by setMessage won’t create a new connection.

**Using useMemo -**



### ***Why this works?***

* useMemo with [] dependencies run only on the first mount → returns the same socket instance on every render.
* setMessage can cause as many re-renders as it wants; the socket reference never changes.
* Backend "connection" fires **only once** (unless you refresh or manually disconnect).

## *****Detailed Fixed Flow*****

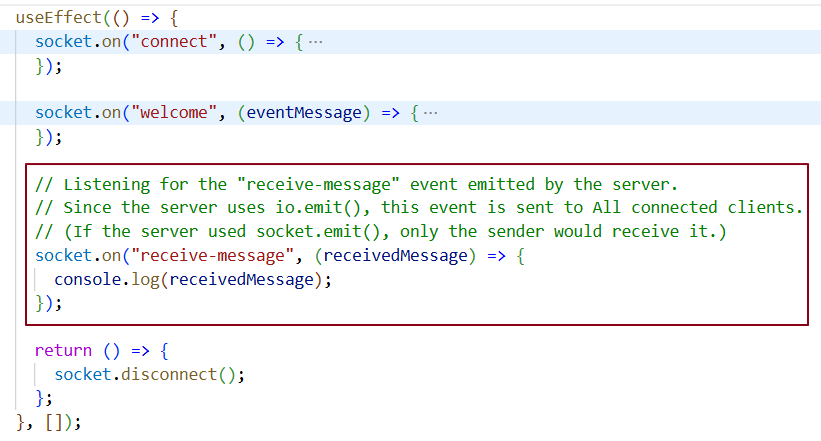
1. **Initial render**
   * useMemo runs once → creates socket → server logs "User Connected 123".
   * useEffect sets up event listeners.
2. **User types a message**
   * setMessage updates state → triggers re-render.
   * useMemo does **not** run again → same socket instance is reused.
   * No new "connection" on backend.
3. **User clicks Send**
   * socket.emit("message", ...) sends message.
   * Backend logs "Hello".
   * setMessage("") clears input → re-render happens.
   * Socket is **not** recreated → backend sees no new connection.

If I just log the messages in the backend, it’s of no use. I need to emit a "receive-message" event so that all clients connected to the server will be notified when it’s properly handled in the frontend code.

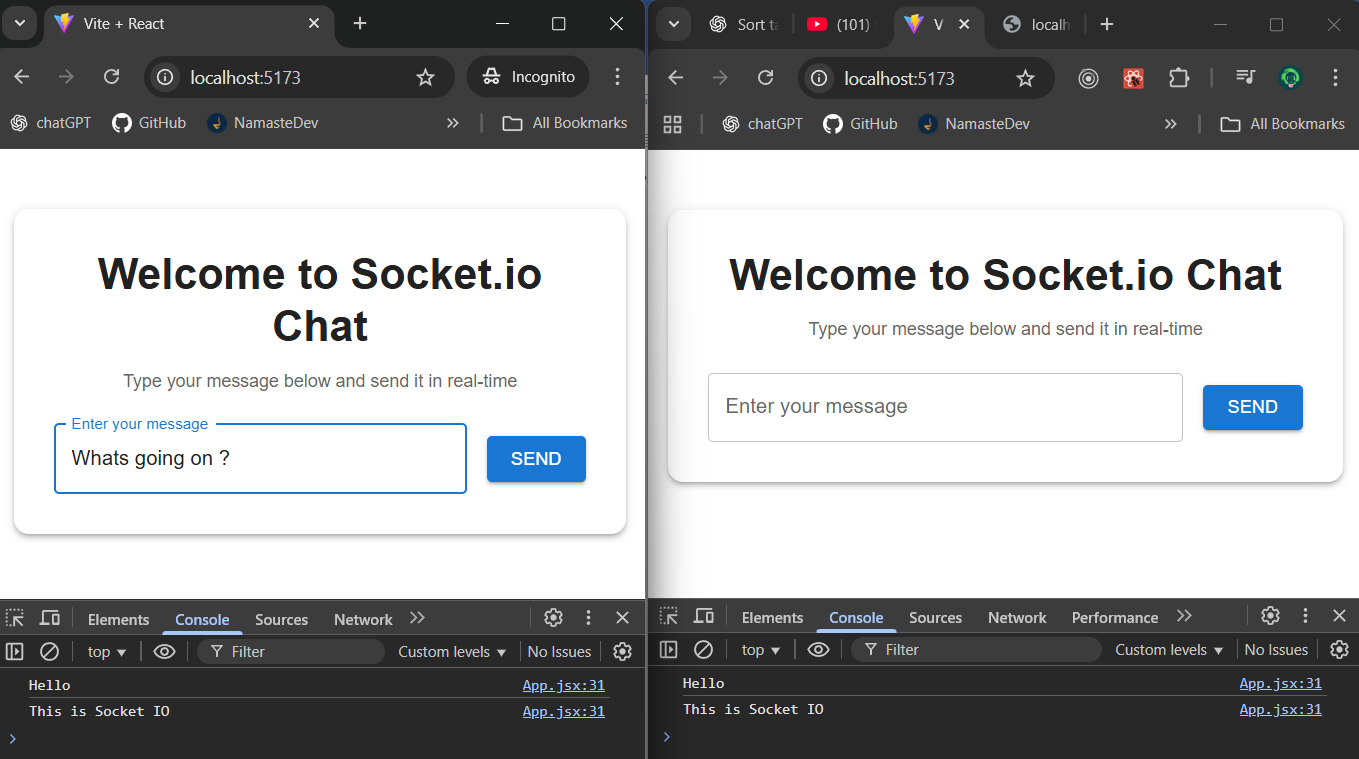
***Backend –***



***Frontend –***



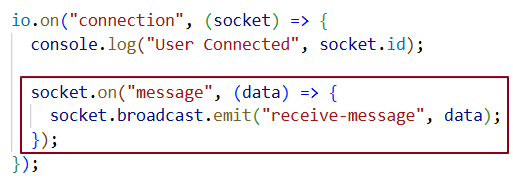
***Testing –***



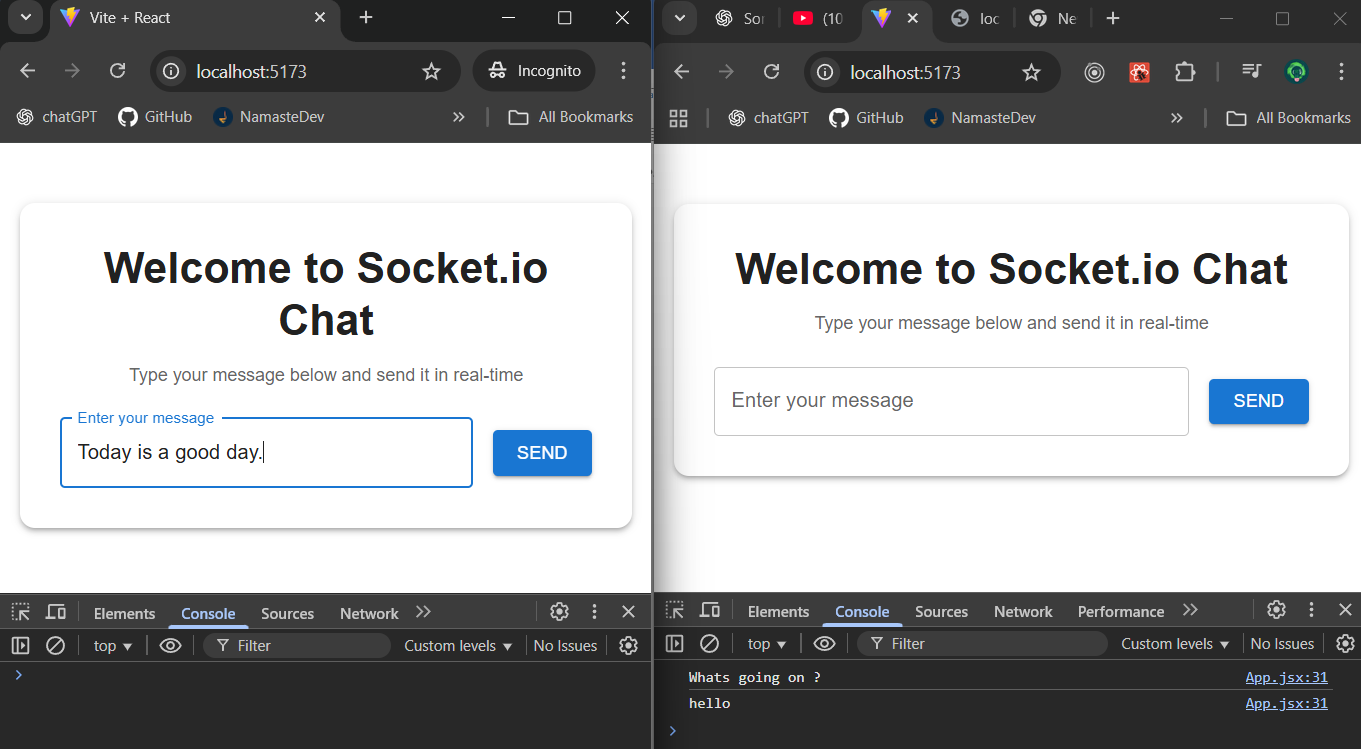
Now, if I send a message from any client, it will be logged on all clients connected to the server.

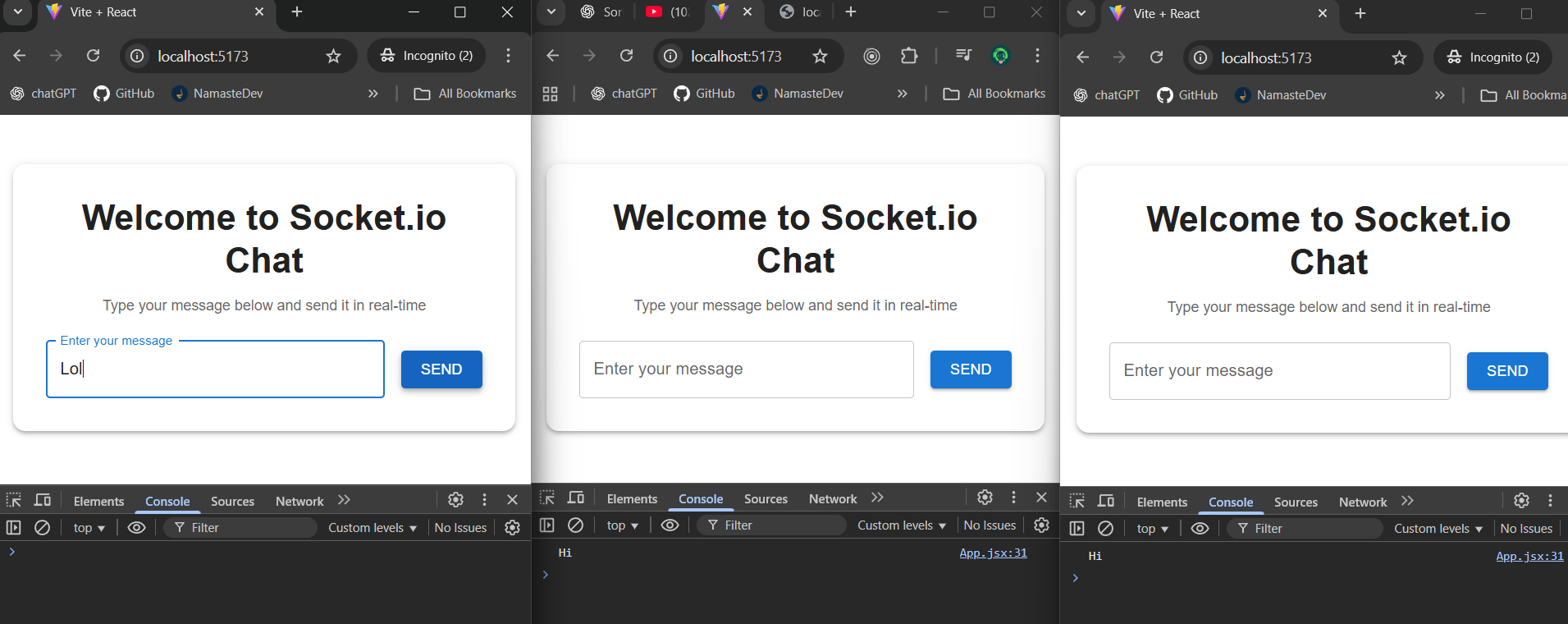
What if the messages sent from a client should be displayed to other clients but not to the sender? In the backend, we can use socket.broadcast to achieve this. This can be configured in the backend.

***Backend –***



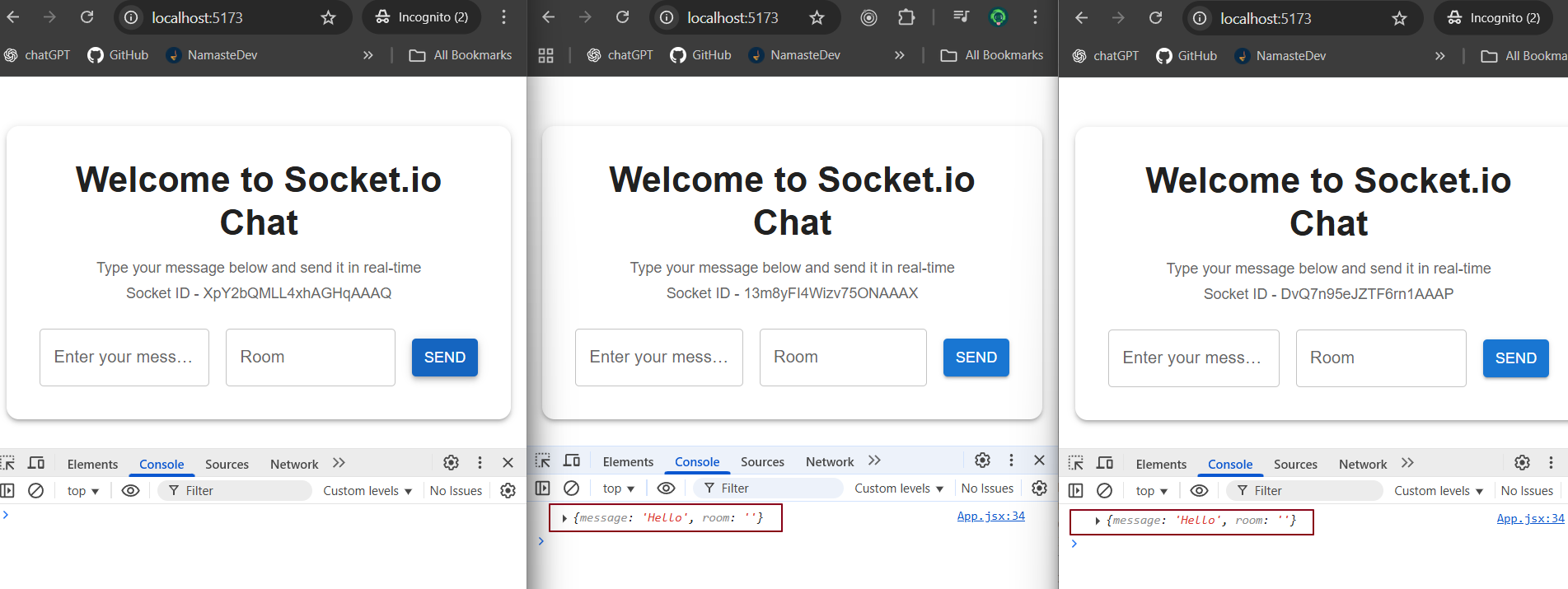
***Testing –***





ROOM -

In Socket.IO, a room is basically a named channel inside the server where you can group sockets together so you can send messages only to those sockets in that group.



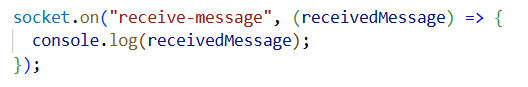
Right now, we’re broadcasting from the current socket(clinet1) to all other clients (client 2 & client 3). We don’t want that. We want **one-to-one** messaging between two clients connected to the same server (two-way communication). For that, we can either **emit directly to a socket ID** or use a **room** that both clients join.

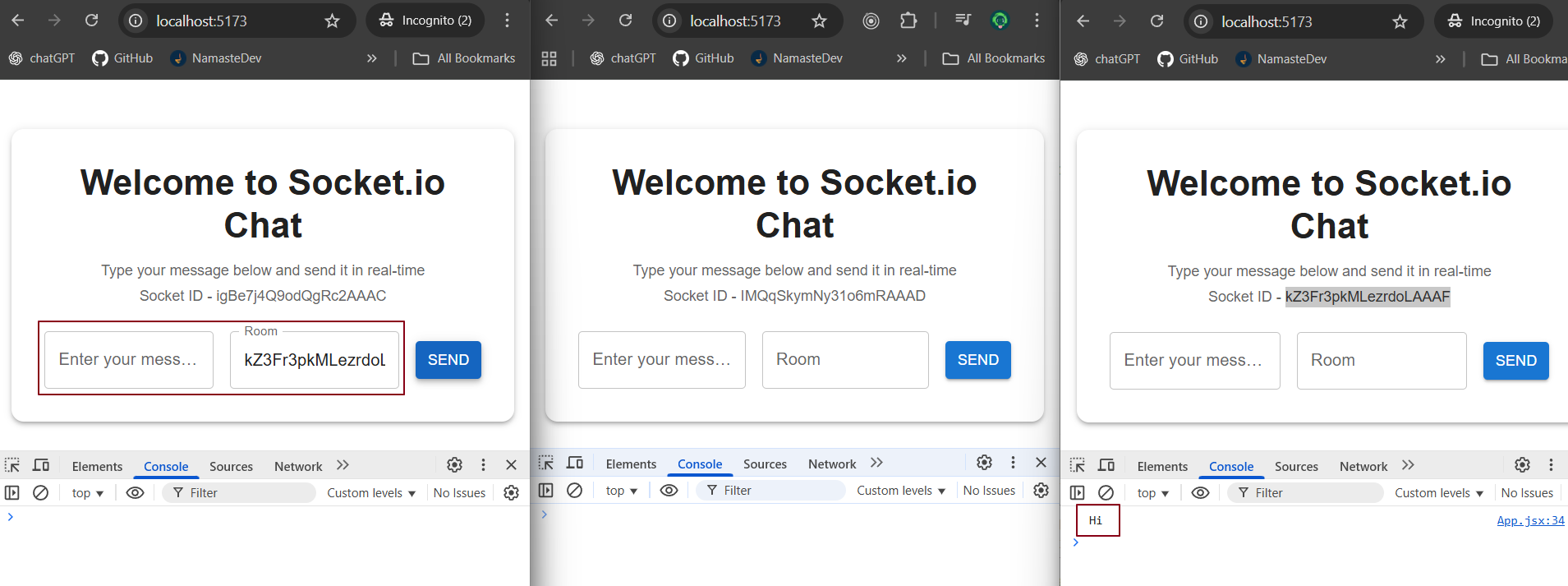
Emitting event directly to a socket ID -

***Backend –***



***Frontend –***





***Execution Flow -***

**1. Client 1 input -** Client 1 enters a message and the roomId/socketId of the client it wants to send the message to.  
(In our case, Client 1 is sending a message to Client 3.)

**2. Triggering the event -** When Client 1 clicks the **Send** button, the "message" event is emitted from the frontend to the server.



**3. Server receives the event**- The server’s "message" event handler receives two things:

* The **message texts.**
* The **roomId** (or socketId) of the target client

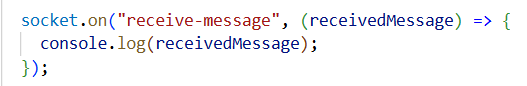
**4. Forwarding to target client** - Inside the server’s "message" handler, we emit a "receive-message" event **only** to the target room/socket:



socket.to(room) → Sends an event to all clients in the given room **except the sender.**

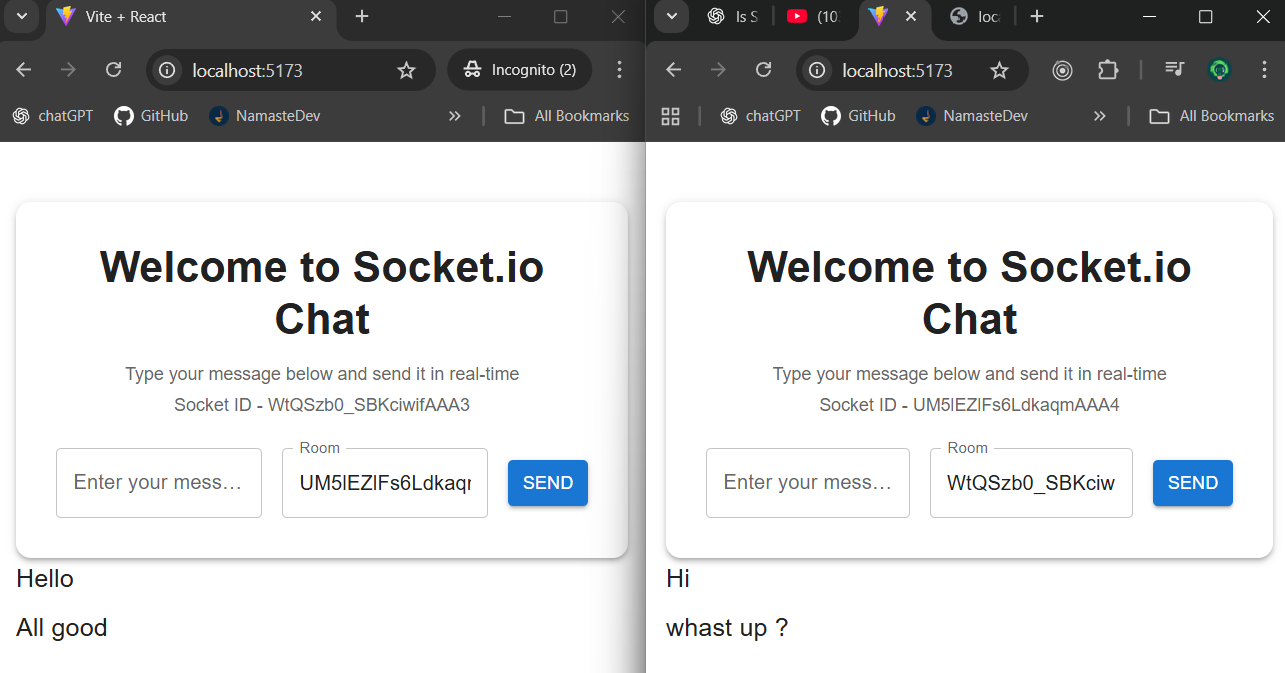
In our current setup, the **room ID we’re using is actually just the target client’s (client 3) socket.id.**

**5. Client 3 receives the message** - On the frontend, Client 3 has a listener for "receive-message" inside a useEffect hook. When the event arrives, Client 3 logs the message (or could display it in the UI).

  
  
**6. Communication established** - Now Client 1 can send messages directly to Client 3 without broadcasting to everyone else.

**Let’s Display communication between two users in UI -**



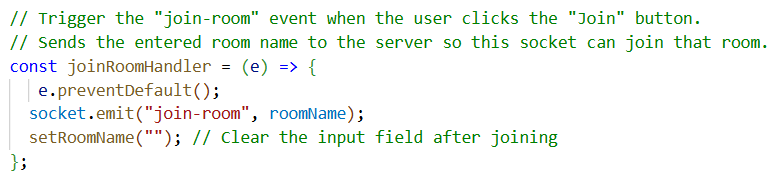


Emitting Events to a Room (Where Multiple Clients Reside) / Join a Room –

Only these clients will notify about the triggered events.

**Join a Room -** Adds the socket to a named group so only members of that group can send and receive messages with each other.

***Front end -***



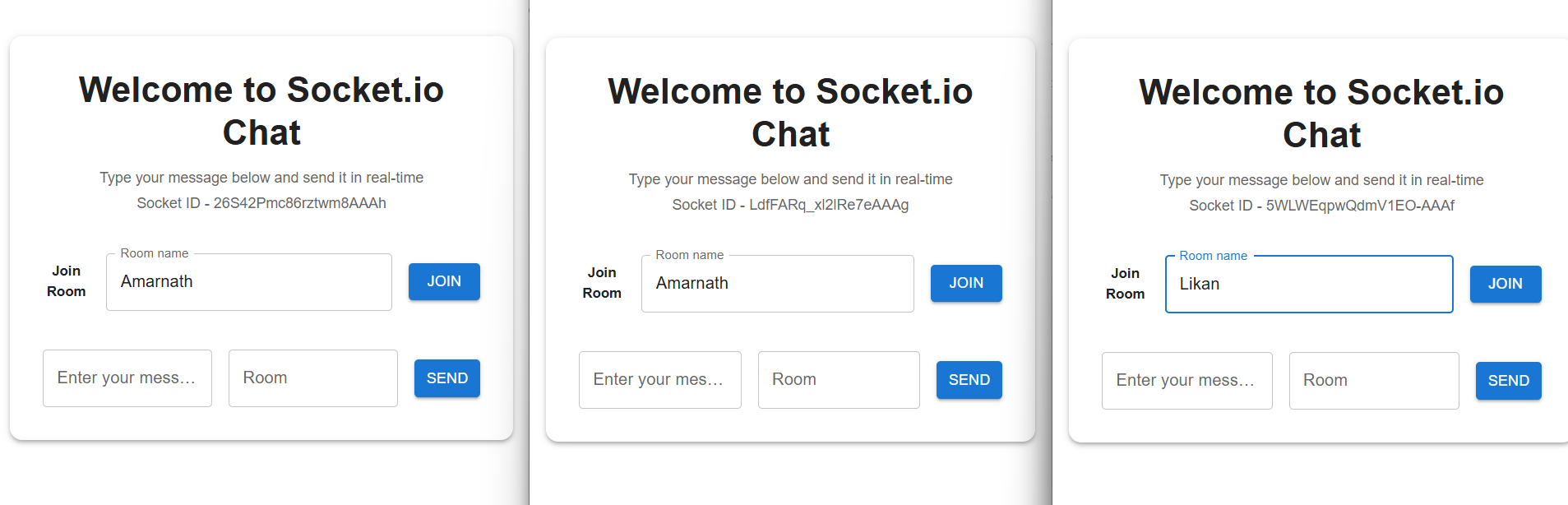
***Backend -***

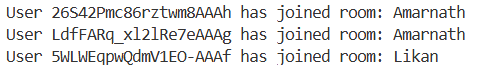


* **socket. join(room)** → Adds this socket (user’s connection) to a *room* — just like adding someone to a WhatsApp group.
* **Room** → A named collection of sockets. Anyone in that room can receive messages sent to that room.
* **io.to(room). emit(...)** → Sends a message to *everyone in that group* (including the sender, if you want).
* **socket.to(room). emit(...)** → Sends a message to *everyone in that group except the sender*.

***Testing-***

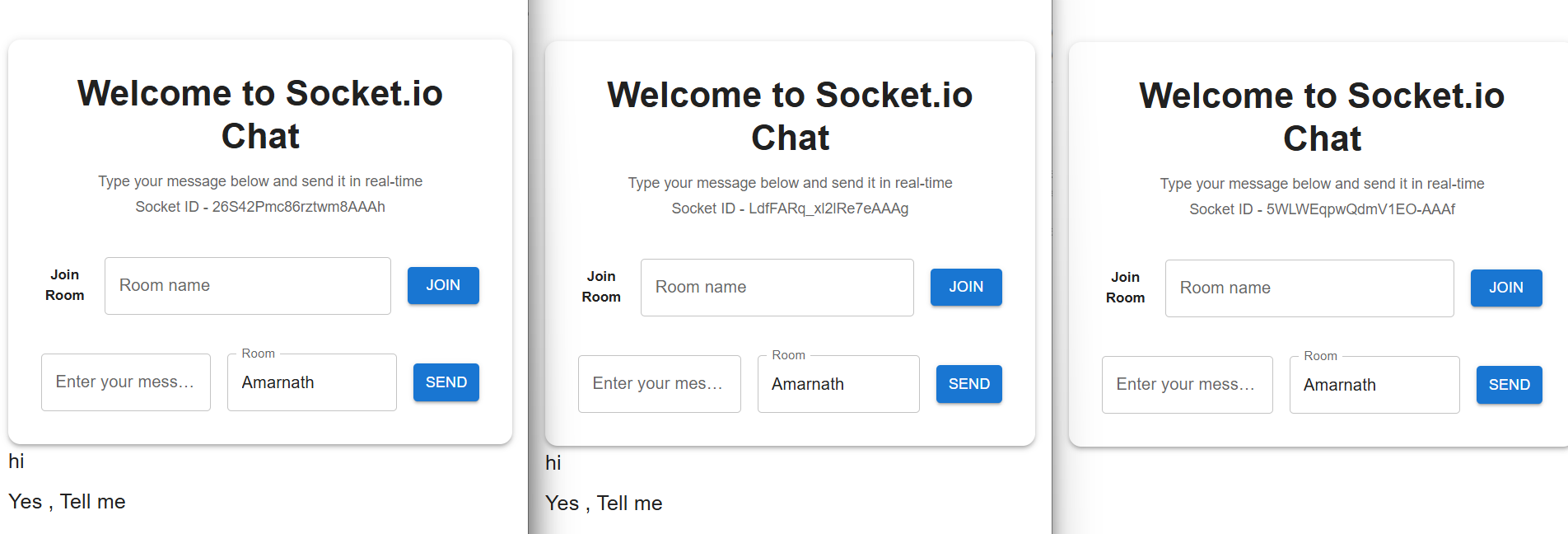
Step 1 - Client 1 and Client 2 are joining a room Name called Amarnath and Client 3 is joining Likan roomname .



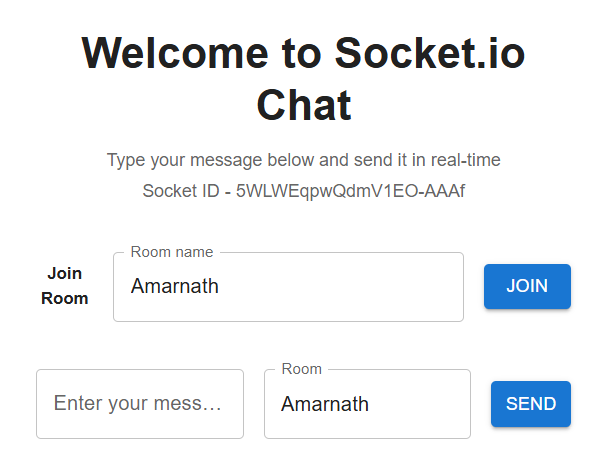


Step 2 - Client 1 is connected with Client 2 and Client 3, but Client 1 and Client 2 receive the message while Client 3 does not, because Client 3 is in a different room (Likan) rather than the room where Client 1 and Client 2 are (Amarnath).

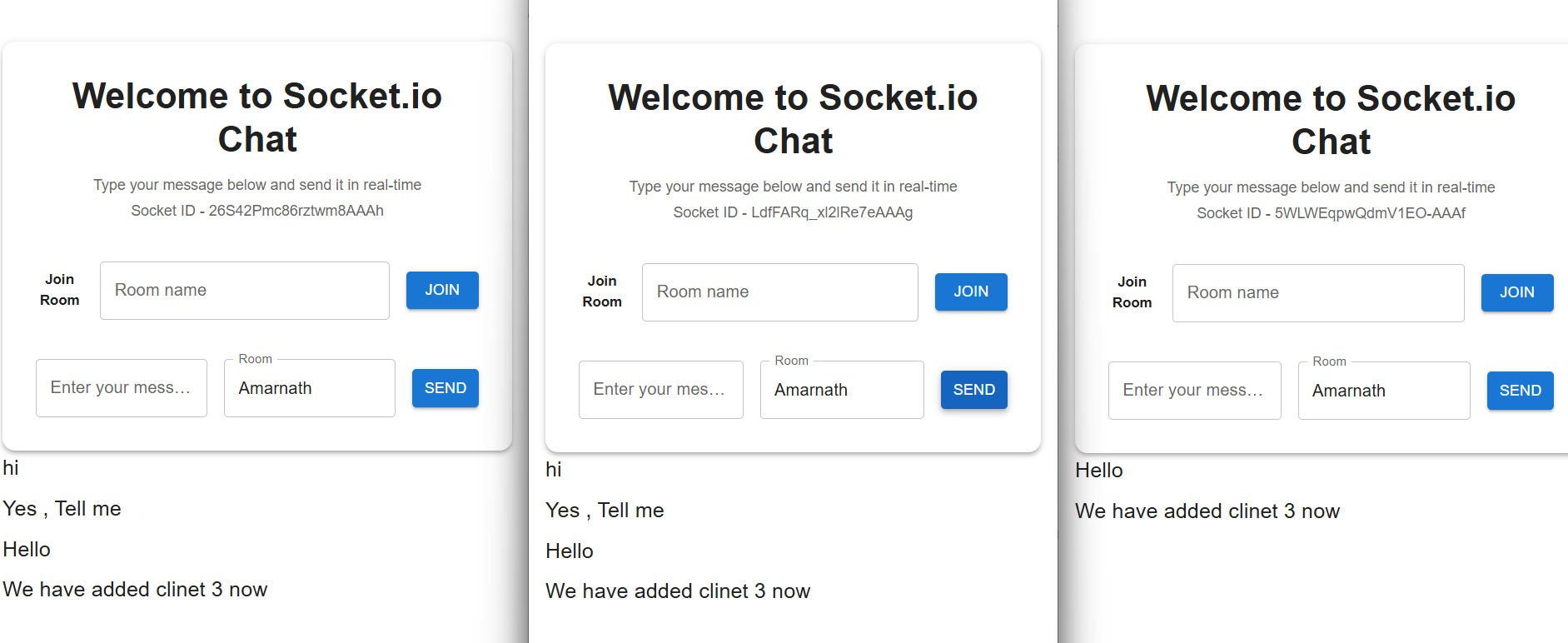
This means both Client 1 and Client 2 can send and receive messages with each other, but Client 3 is excluded since they are in a separate room.



Step 3 - Let’s add Client 3 to the Amarnath room and see whether Client 3 can communicate with Client 1 and Client 2.







Client 3 can communicate with client 1 and 2 now

Usage of middleware in socket.io -



***Reference -*** <https://www.youtube.com/watch?v=_h7Pc1woq-I&t=2883s>

To cover - 1.08.08 – 1.18.43 (10 mins)