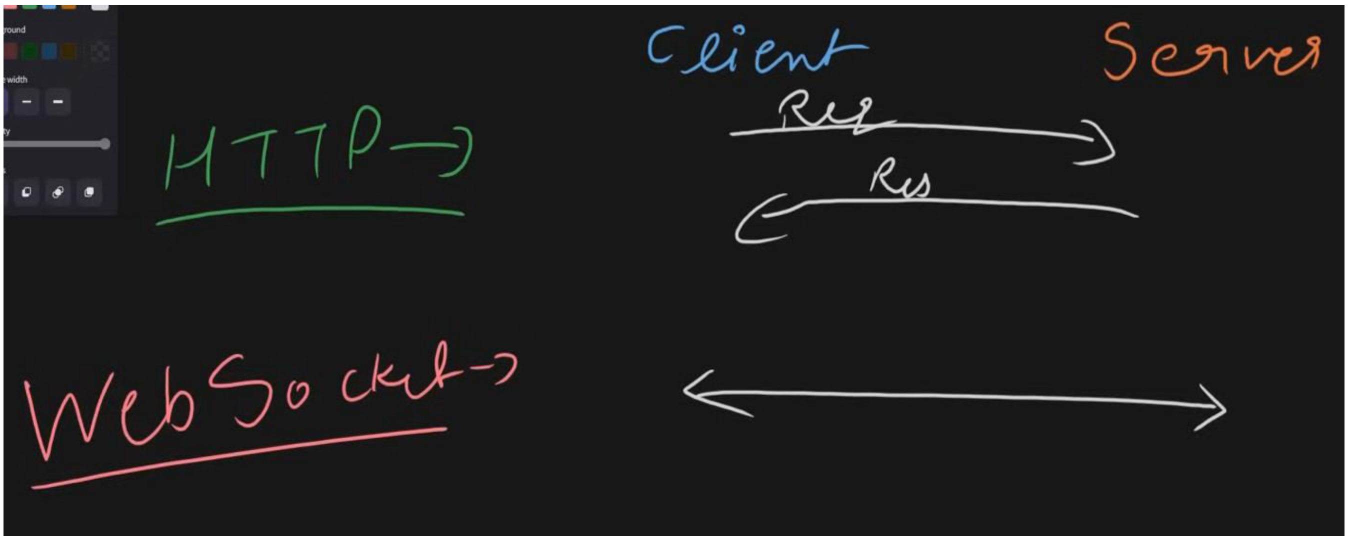
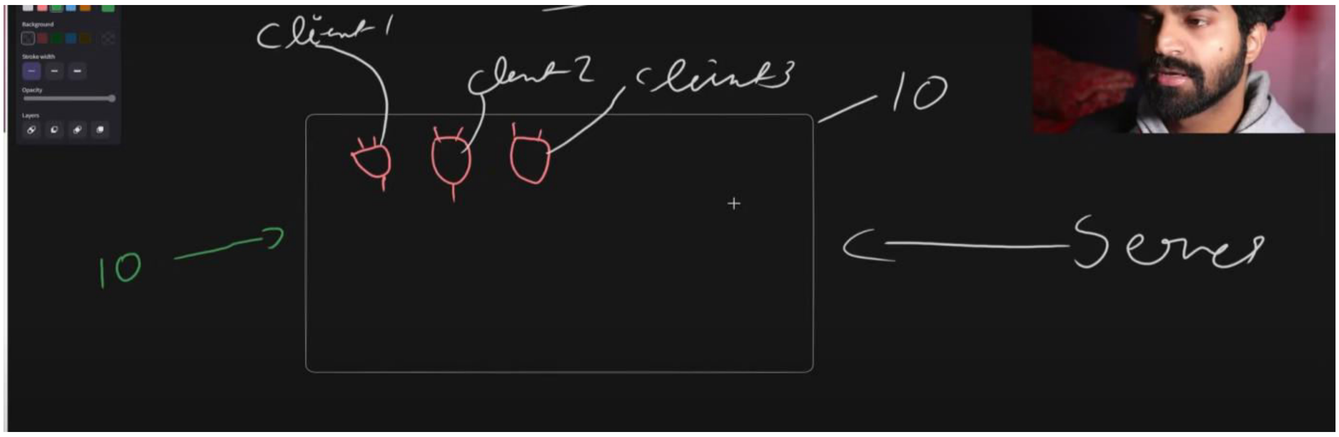
Socket.io and chat-app project

Socket.io is library of web socket. Web socket is a communication protocol like HTTP. In HTTP client makes request and then server provides the data.



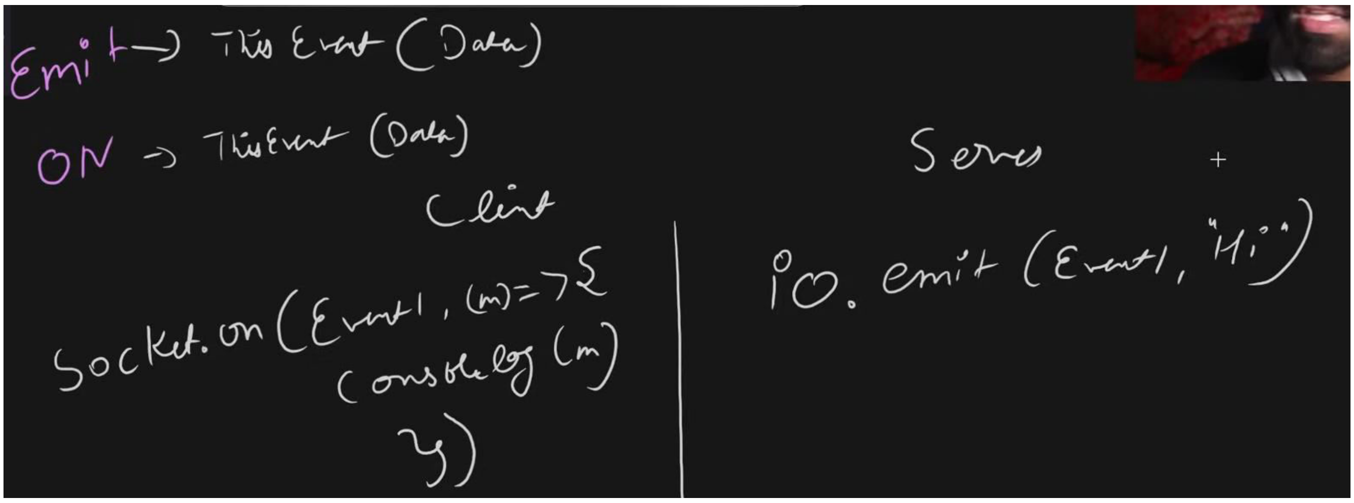
In websocket , server can send data irrespective of the request made by the user.

Here client and server both can send data to each other.



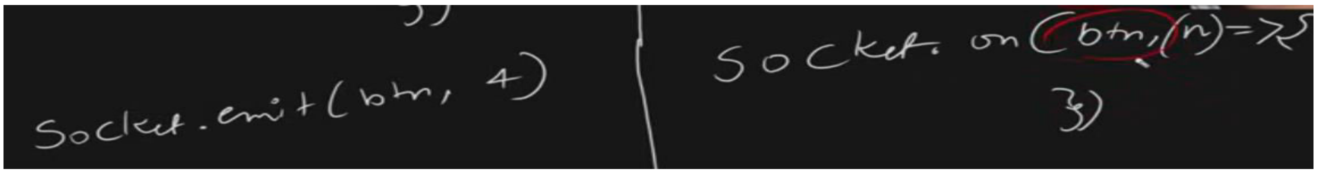
Here, in socket.io the IO is a server/circuit board where the client/socket attaches or joins like a socket connects with the circuit board. Now every socket has their unique ID.

There are two important keywords Emit and On , which means ,emit will emit the data and wherever the On is written it will receive the data emitted by Emit. Ex:-



Here Io has emitted data as a string “hi”, with the event name as event1, now socket has to receive the data, so the socket should have the same name of the event as in Io to receive the same data, Socket has functioned as an argument. The above picture is an example of how the server sends the data and the client receives the data.

Now we will see how to use emit on the client side.



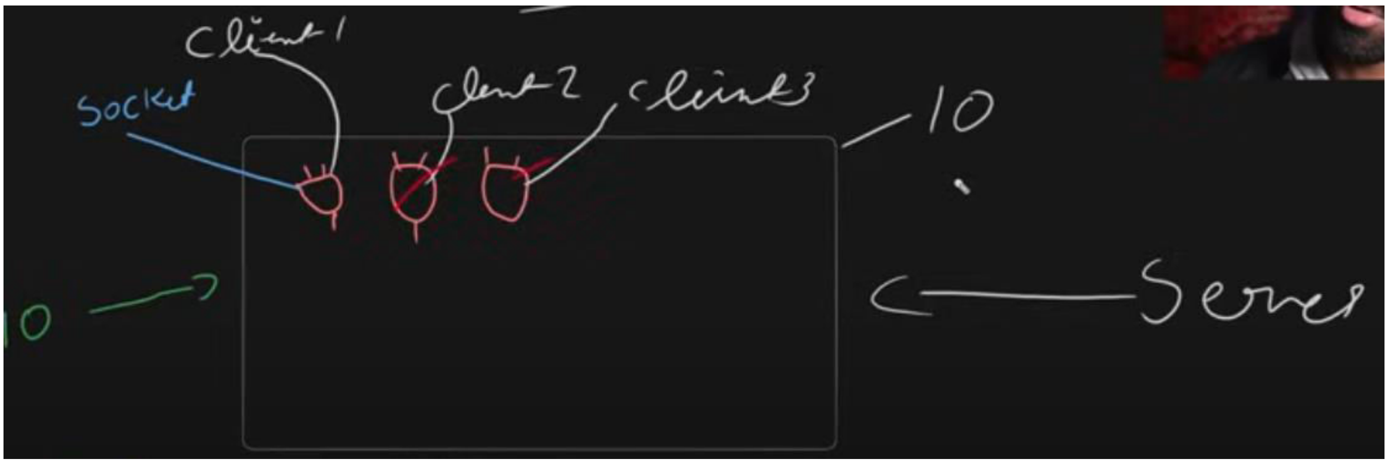
```
socket.emit(btn, 4)        socket.on(btn,(n)=>{
                           })
```

Here socket on the client side has used the emit function with the event name (btn).

On the server side, I have used used On function with the same name btn.

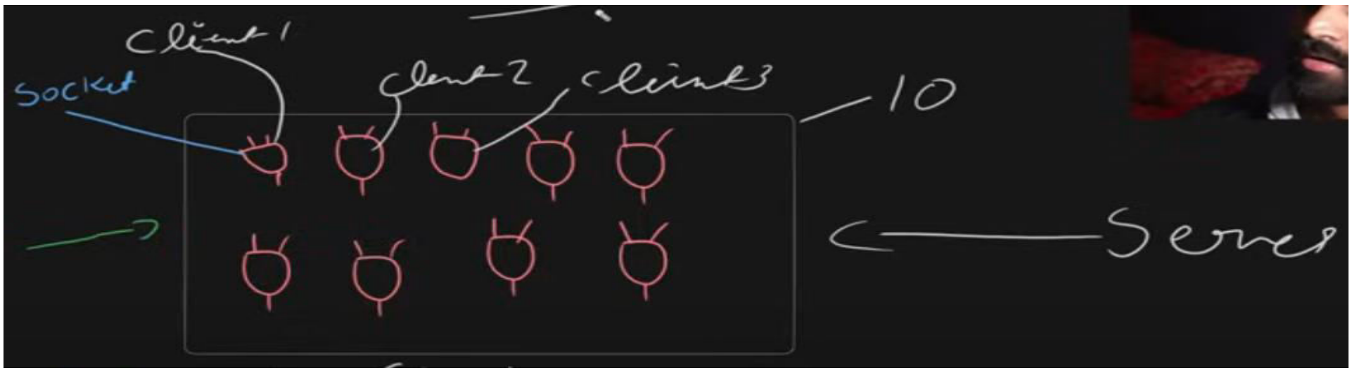
Here is the difference, when I do io.emit the data goes to every socket present in the io circuit. But when I do socket.emit the data will pass to a particular socket .

Now we will see socket.broadcast.emit();



Let's say the client 1 socket has done a function, socket.broadcast.emit(), so the data will pass to every socket except the client 1 socket.

Now, there are 2 more functions, (To and Join), let's say a particular socket wants to connect with another particular socket, then we do socket.to(id of a socket to which we want to connect).emit(), to connect with a particular room/socket.

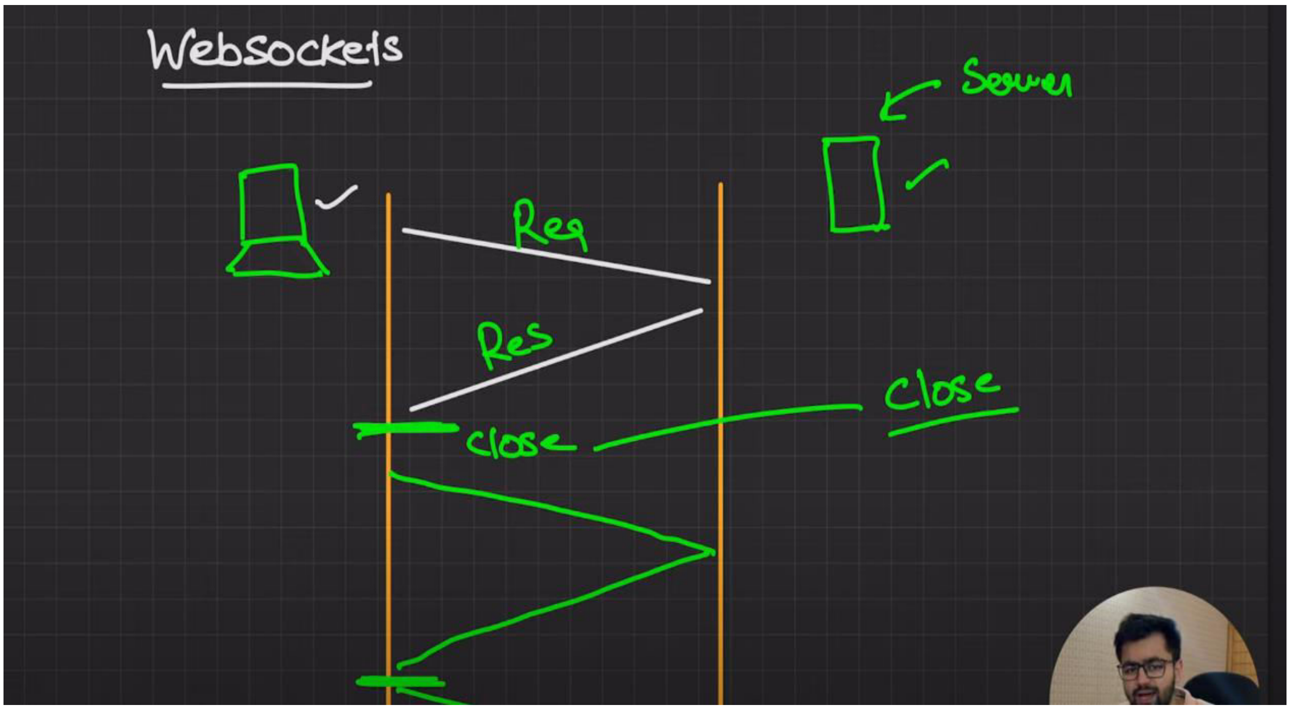


Now , if we want to prepare the scenario, where client 1 socket want to send message to the 4 sockets in the 2nd row only so we will use Join method,so now each socket is in individual room currently , but we can make room of 5 which consist of these sockets.

We will create a room for sockets like, socket.Join(name of the room), and by using the To function we can send data to the room where many participants are there.

<u>Socket by Piyush Garg</u>

Initially, the client makes the request to the server, then the server replies with the response, and after this req , res cycle is completed and the connection is closed.

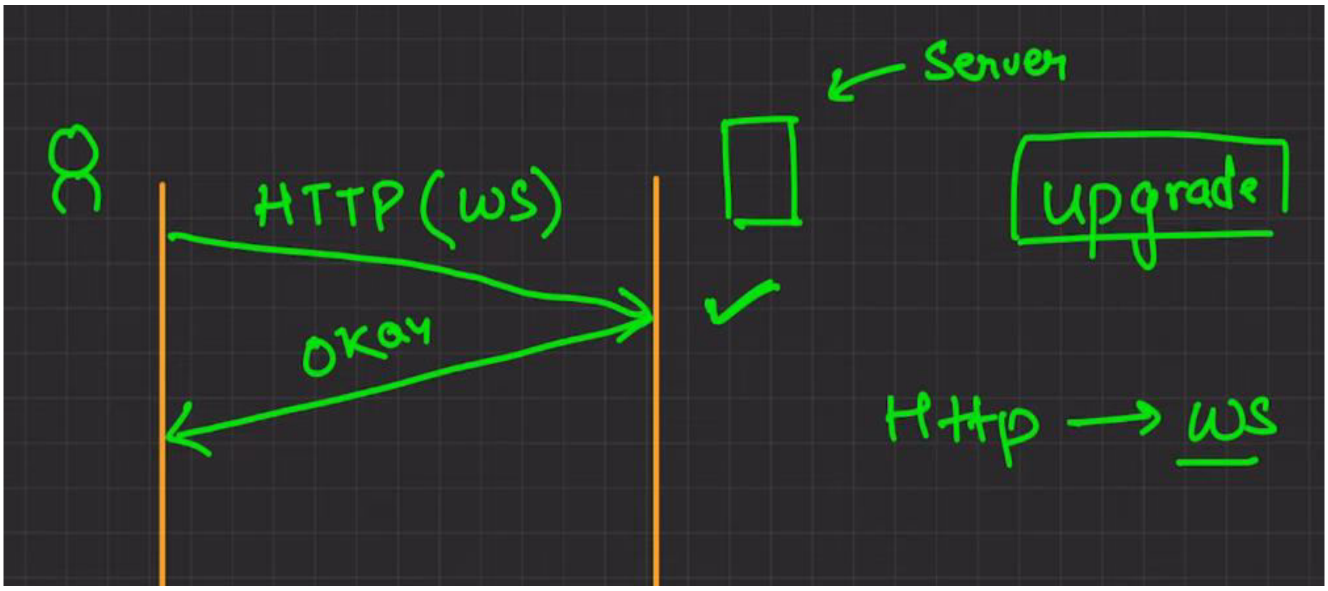


By this method, the client is communicating with the server and this method is simply a request-response cycle.

Let’s take an example where you are the server and two users A and B want to chat with each other, so direct communication is not possible for them, they need a server where they send or fetch data from the server, send data to other client or fetch data from the other client respectively. But the problem is, the server is not sending data to the client, client has to ask for it every time for the data, if the other user has sent it or not. This is called polling where the client is repetitively making the request to the server.

Polling causes to increase in load to the server.

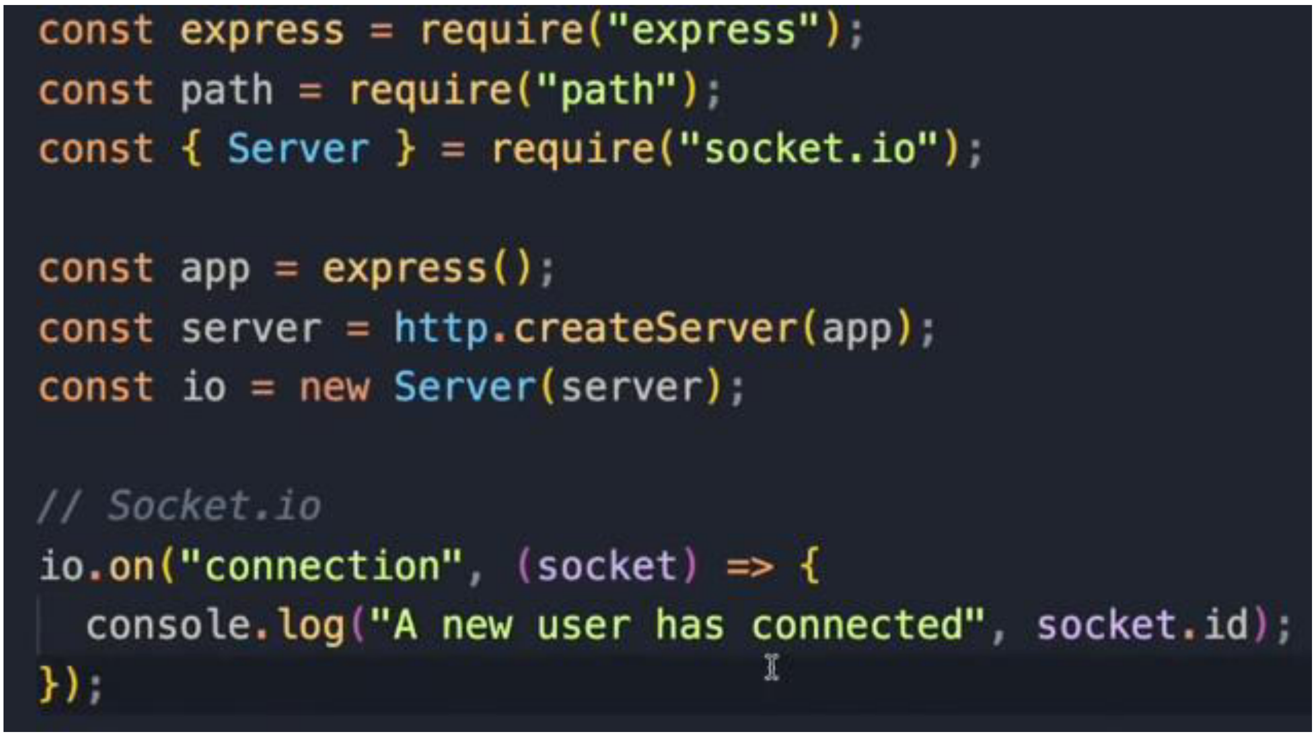
The solution of polling is WEBSOCKET.



Initially, the client sends the HTTP request to the server and asks the server to make a WebSocket Protocol connection, so there is one header in the request named upgrade, so the server accepts the request and upgrades the connection to a WebSocket Protocol connection.

Here, a websocket Protocol connection is a bi-directional connection where the server can also send data to the client, the client does not have to make requests to the server if any data for them is there or not, the server can itself send the data if it is there for any particular client.

To implement a WebSocket Protocol connection in node.js we use socket.io library.



```
const express = require("express");
const path = require("path");
const { Server } = require("socket.io");

const app = express();
const server = http.createServer(app);
const io = new Server(server);

// Socket.io
io.on("connection", (socket) => {
  console.log("A new user has connected", socket.id);
});
```

Here we have done:-

Const server=http.createServer(app), we have this so that we attach our server from http protocol to websocket protocol connection.

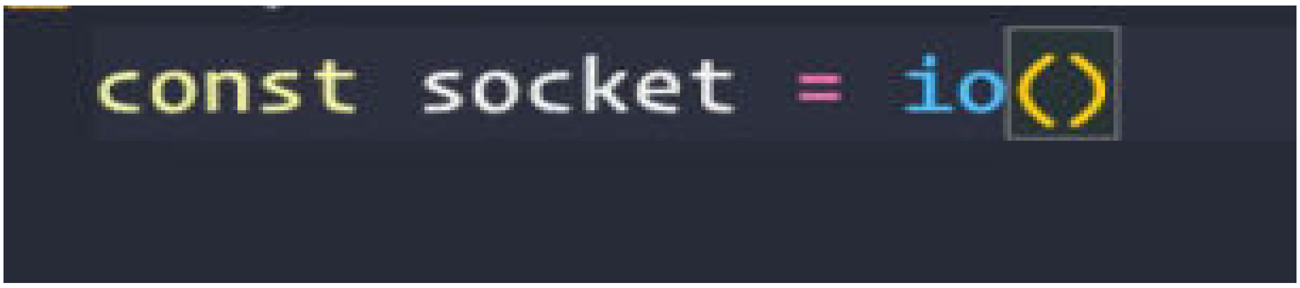
Now by doing, const io=new Server(server), we have created the io server, where connections are based on WebSocket protocol. The clients that are connected to the io server can get data from the io server without making the request. Through the io server, the connection between the user and the server is bi-directional.

By doing io.on(‘connection’,(socket)=>{});

The event connection is the predefined event.

We have created a connection of the socket parameter which comes from the client side.

And by doing this:-



```
const socket = io()
```

We make a socket on the client side that automatically connects to the io() server.

Every socket has a unique ID.

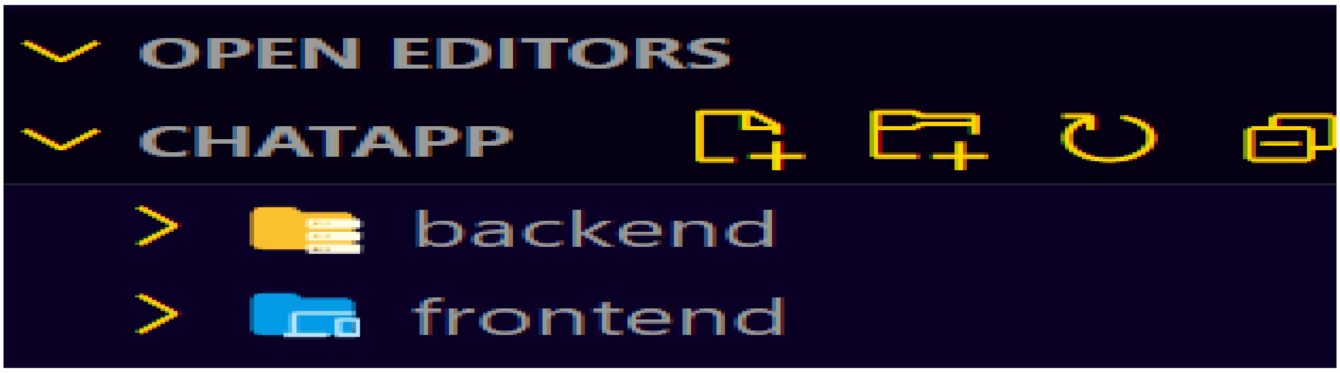
Whenever we do socket=io(), the socket will get created.

That is how we sockets and connect with the server.

CHAT APP BY USING WEB SOCKET:- 6 Pack-Programmer

CHAT-APPLICATION-PROJECT

1) Create a folder, and inside this folder create two new folder, for backend and frontend.

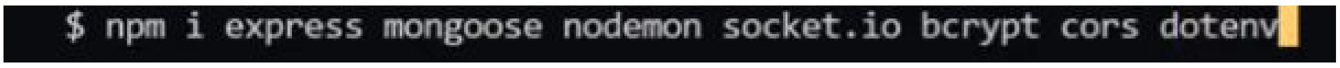


After doing this, open the terminal and for two folders, by doing cd backend and cd frontend for backend and frontend respectively.

2) In the backend terminal write (npm init) to download the package.json file and in

the frontend terminal write (npx create-react-app .).

3) Install some dependencies for the backend in the backend terminal.



```
$ npm i express mongoose nodemon socket.io bcrypt cors dotenv
```

In backend development, each of the technologies you mentioned serves a specific purpose and contributes to building robust and scalable web applications. Here's a breakdown of what each technology is used for:

1. **Express**:

   - **Purpose**: Express is a minimalist and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications.
   - **Use**: It simplifies the process of handling HTTP requests, routing, middleware integration, and more. Express is widely used to create APIs and server-side logic for web applications.

2. **Mongoose**:

   - **Purpose**: Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a straightforward schema-based solution to model your application data.
   - **Use**: Mongoose simplifies interactions with MongoDB by providing features like validation, type casting, query building, and more. It's commonly used to define schemas and models for MongoDB databases in Node.js applications.

3. **Nodemon**:

   - **Purpose**: Nodemon is a utility that monitors changes in your Node.js application and automatically restarts the server when file changes are detected.
   - **Use**: During development, Nodemon saves time by eliminating the need to manually restart the server after making changes to your code. It helps in maintaining a smooth development workflow.

4. **Socket.IO**:

    - **Purpose**: Socket.IO is a library that enables real-time, bidirectional communication between web clients and servers. It uses WebSocket protocol but provides fallbacks for environments where WebSocket is not supported.
    - **Use**: Socket.IO is used for implementing features like real-time chat, live notifications, multiplayer gaming, and more. It facilitates instant data exchange between clients and the server.

5. **bcrypt**:

    - **Purpose**: bcrypt is a library used for hashing passwords securely.
    - **Use**: Storing passwords in plaintext is unsafe. bcrypt allows you to hash passwords before storing them in a database. It uses a one-way hashing algorithm that is computationally intensive, making it resistant to brute-force attacks.

6. **dotenv**:

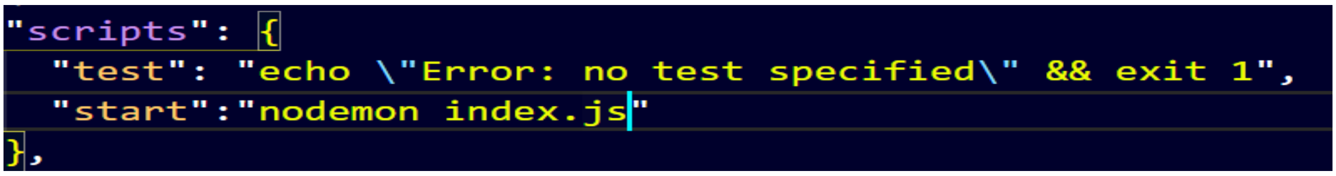
    - **Purpose**: dotenv is a module that loads environment variables from a `.env` file into `process.env`.
    - **Use**: It's used to manage sensitive configuration settings (like database credentials, API keys) in Node.js applications. By storing these variables in a `.env` file (which is not committed to version control), you can keep your application secure and portable across different environments.

7. **cors** (Cross-Origin Resource Sharing):

    - **Purpose**: cors is a Node.js middleware that enables Cross-Origin Resource Sharing in your application.
    - **Use**: When your frontend (hosted on a different domain) needs to access resources from your backend API, CORS headers must be set to allow cross-origin requests. The cors middleware handles this by adding the necessary headers to HTTP responses.

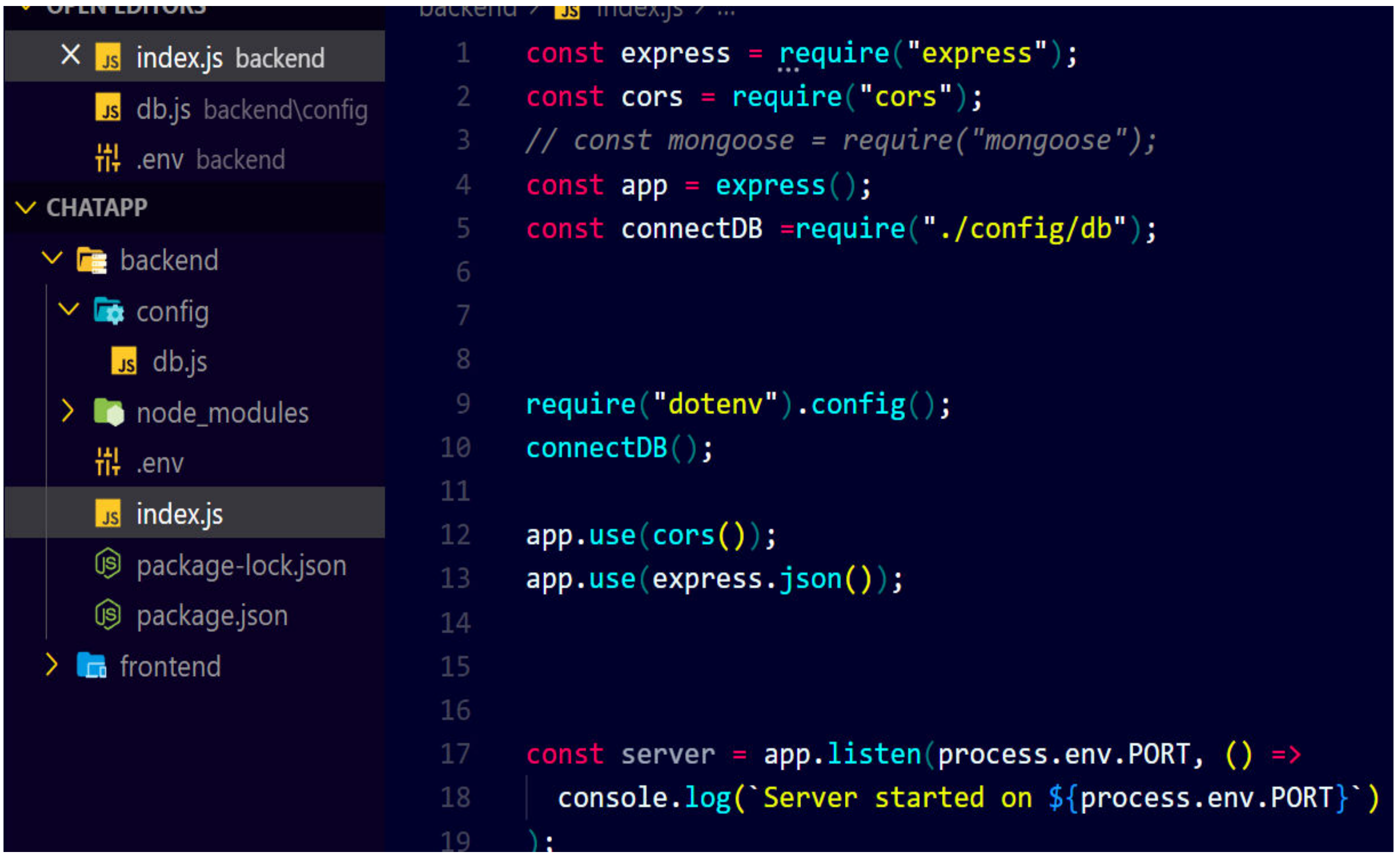
In summary, these backend technologies play critical roles in different aspects of web development, such as handling HTTP requests, managing databases, ensuring server-side security, enabling real-time communication, and simplifying development workflows. Understanding and effectively using these tools can significantly enhance the functionality, performance, and security of your backend applications.

4) Create a index.js file in backend and in package.json file of backend d this and create the .env file in backend folder:-



```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start":"nodemon index.js"
},
```

5) Now we are going to setup our server like this:-

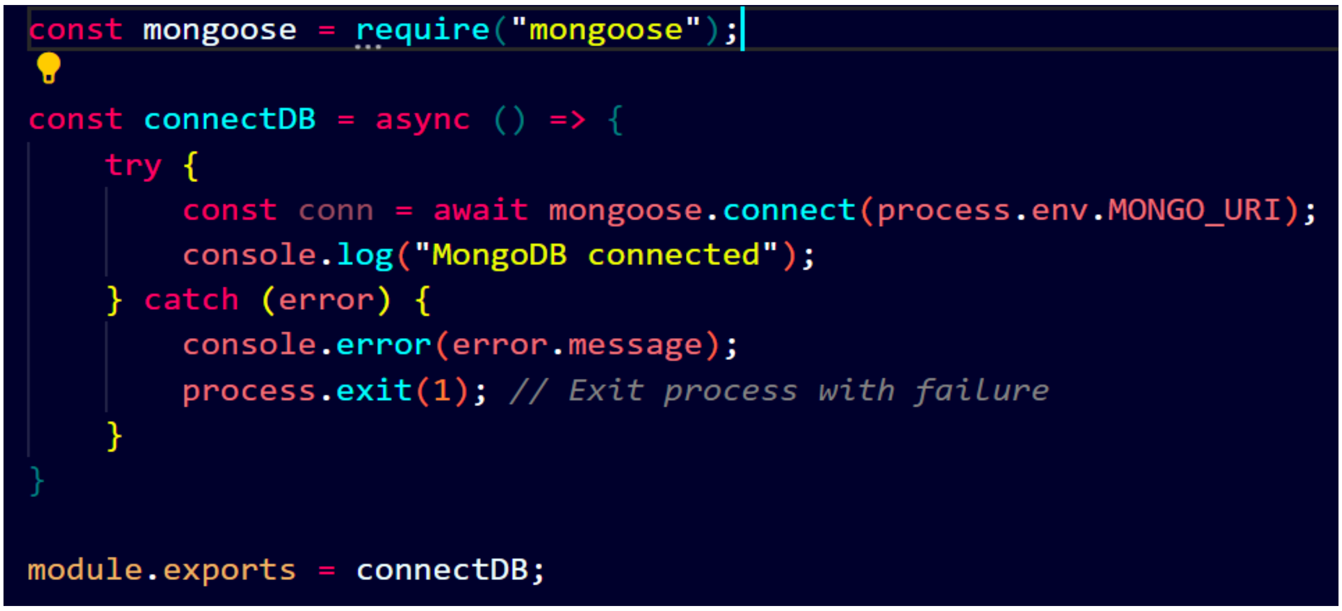


```
const express = require("express");
const cors = require("cors");
// const mongoose = require("mongoose");
const app = express();
const connectDB =require("./config/db");



require("dotenv").config();
connectDB();

app.use(cors());
app.use(express.json());



const server = app.listen(process.env.PORT, () =>
  console.log(`Server started on ${process.env.PORT}`)
);
```

the above conguration is the base setup.

6) For mongodb connection , create a folder named as config inside the

backend folder ,and inside the config folder create file name db.js and do this:-



```
const mongoose = require("mongoose");

const connectDB = async () => {
    try {
        const conn = await mongoose.connect(process.env.MONGO_URI);
        console.log("MongoDB connected");
    } catch (error) {
        console.error(error.message);
        process.exit(1); // Exit process with failure
    }
}

module.exports = connectDB;
```

7) Now in the frontend terminal, install some of the dependencies:- axios , styled-components, react-router-dom.

Axios

- **Purpose**: Axios is a popular JavaScript library used for making HTTP requests from the browser. It supports both the browser's XMLHttpRequests (XHR) and Node.js's HTTP API.
- **Use**: Axios simplifies the process of sending asynchronous HTTP requests and handling responses in a straightforward manner. It provides features such as interceptors, request/response transformations, and automatic JSON data parsing.

styled-components

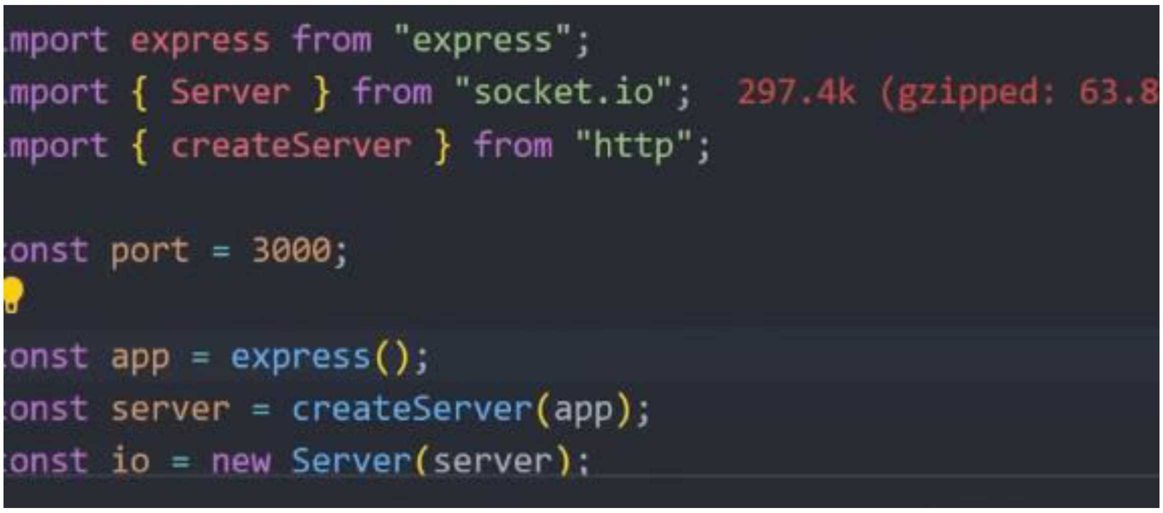
- **Purpose**: styled-components is a library for styling React components with CSS-in-JS. It allows you to write actual CSS code as JavaScript template literals, enabling scoped styles and dynamic styling based on component props.

- **Use**: styled-components promotes component-based styling, where styles are defined alongside components, making it easier to manage and maintain styles for individual components.

react-router-dom

- **Purpose**: react-router-dom is a library for implementing routing and navigation in React applications. It provides components like `BrowserRouter`, `Route`, `Link`, `Switch`, and `Redirect` for declaratively defining routing rules and rendering different components based on URL paths.
- **Use**: react-router-dom allows you to create single-page applications (SPAs) with multiple "pages" or views managed by the client-side router, without full-page reloads.

8) In frontend under the folder src , create a folder named as utils and under utils create the js the file APIRoutes.js where we store all the routes. We just store these backend routes for our covinience. And before this I have created register page and coded in register.js and app.js page.

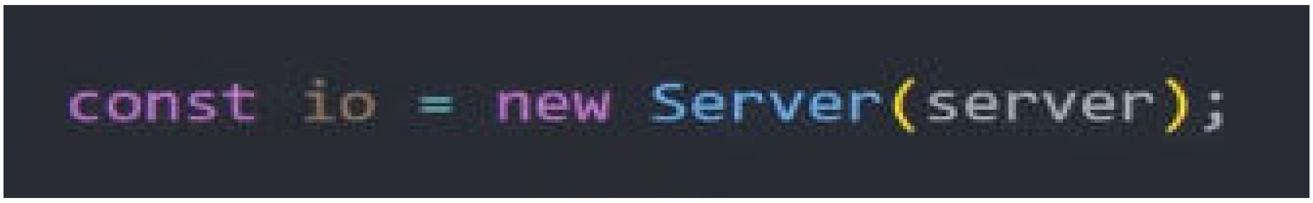
9) 

```
mport express from "express";
mport { Server } from "socket.io";  297.4k (gzipped: 63.8
mport { createServer } from "http";

onst port = 3000;

onst app = express();
onst server = createServer(app);
onst io = new Server(server);
```

Last me app.listen ki jgh server.listen krna h.

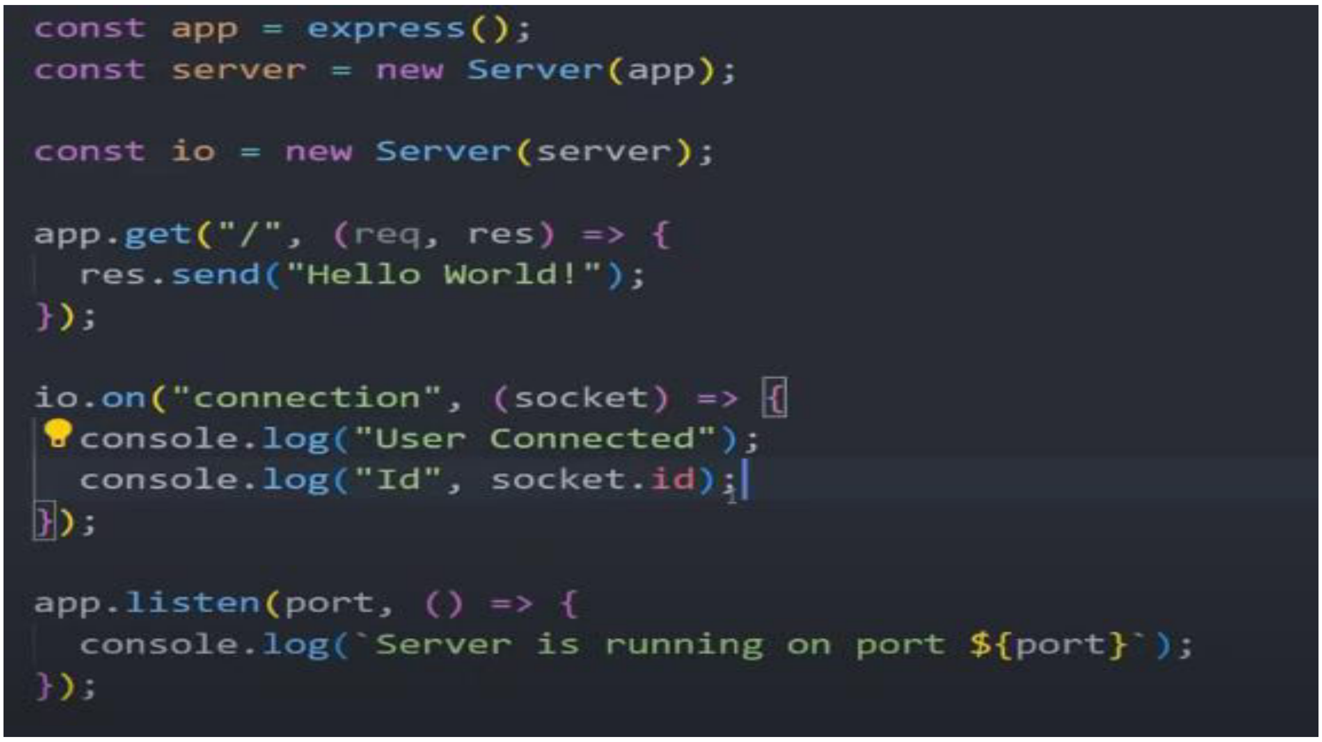
Here we have done server=createServer(app); where have passed our app, which changes the HTTP protocol connection to the WebSocket protocol connection.

Now we have to create the WebSocket server so we do this:-



```
const io = new Server(server);
```

Here we have created the server, but now will create the function that automatically connects the client from frontend with this server.



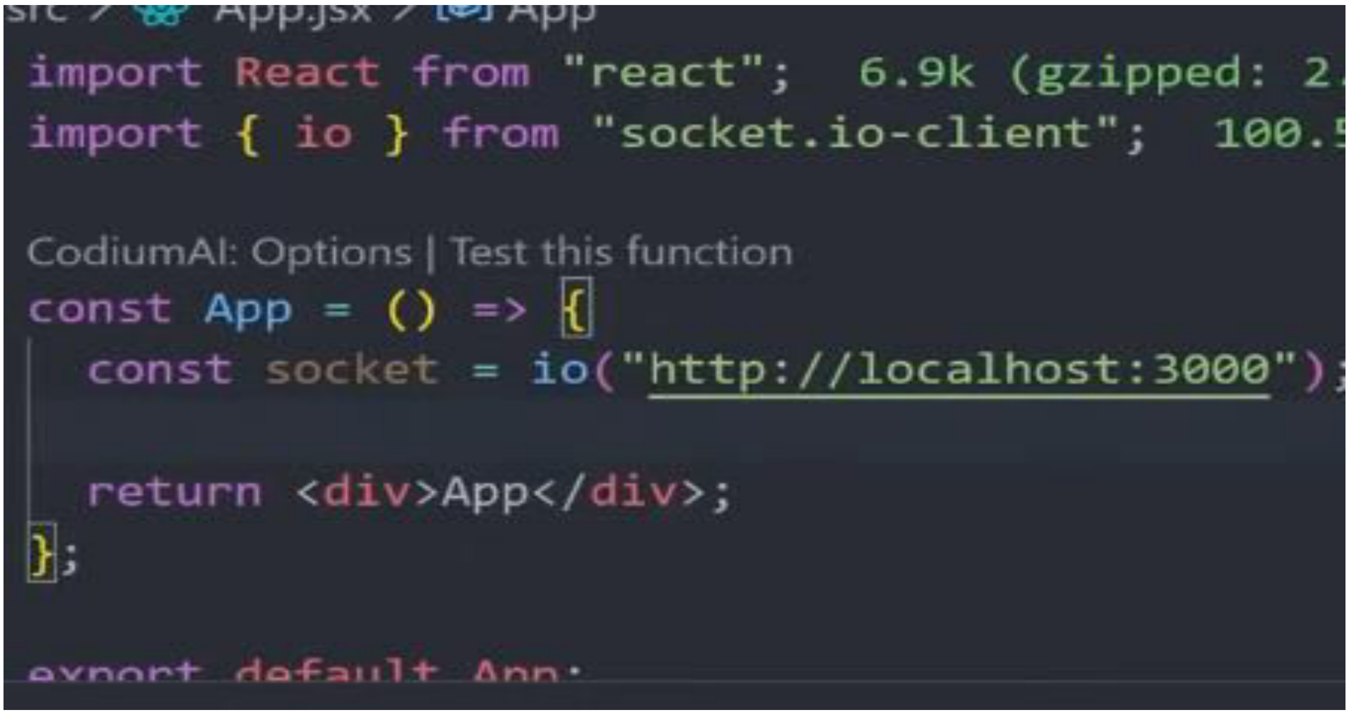
```
const app = express();
const server = new Server(app);

const io = new Server(server);

app.get("/", (req, res) => {
  res.send("Hello World!");
});

io.on("connection", (socket) => {
  console.log("User Connected");
  console.log("Id", socket.id);
});

app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

app

Here io.on(‘connection’,(socket)=>{});

The connection is the pre-defined event, and the parameter socket comes from the client side, whenever the socket is created then it will directly get connected with the io server, by using this code:-
io.on(‘connection’,(socket)=>{});

Now, in frontend on your file where you have to create the socket do this:-



```
src > App.jsx > App
import React from "react";  6.9k (gzipped: 2
import { io } from "socket.io-client";  100.5

CodiumAI: Options | Test this function
const App = () => {
  const socket = io("http://localhost:3000");

  return <div>App</div>;
};

export default App;
```

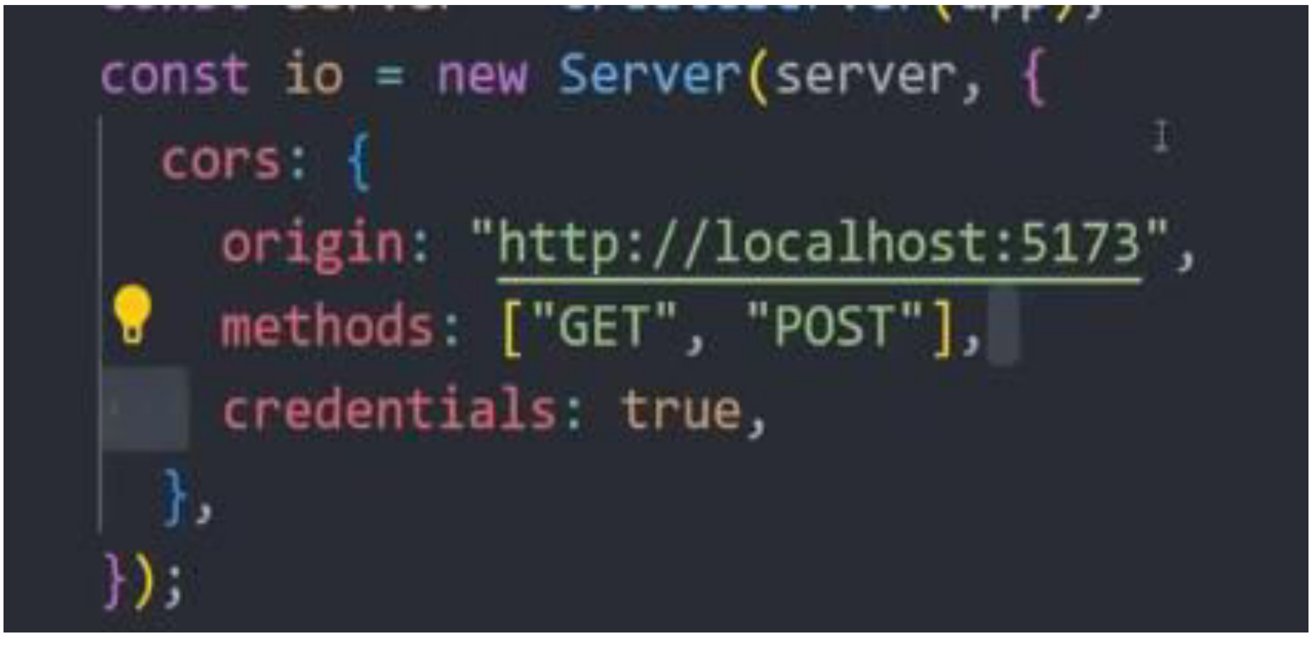
Here 1st we import io from socket.io-client, now we do:- const socket=io('url of backend where io server is created');

Due to this whenever this pages renders it will create the socket, because const socket=io();

Is called.

Agr kuch pass nhi krte, to by default frontend ka url dedeta.

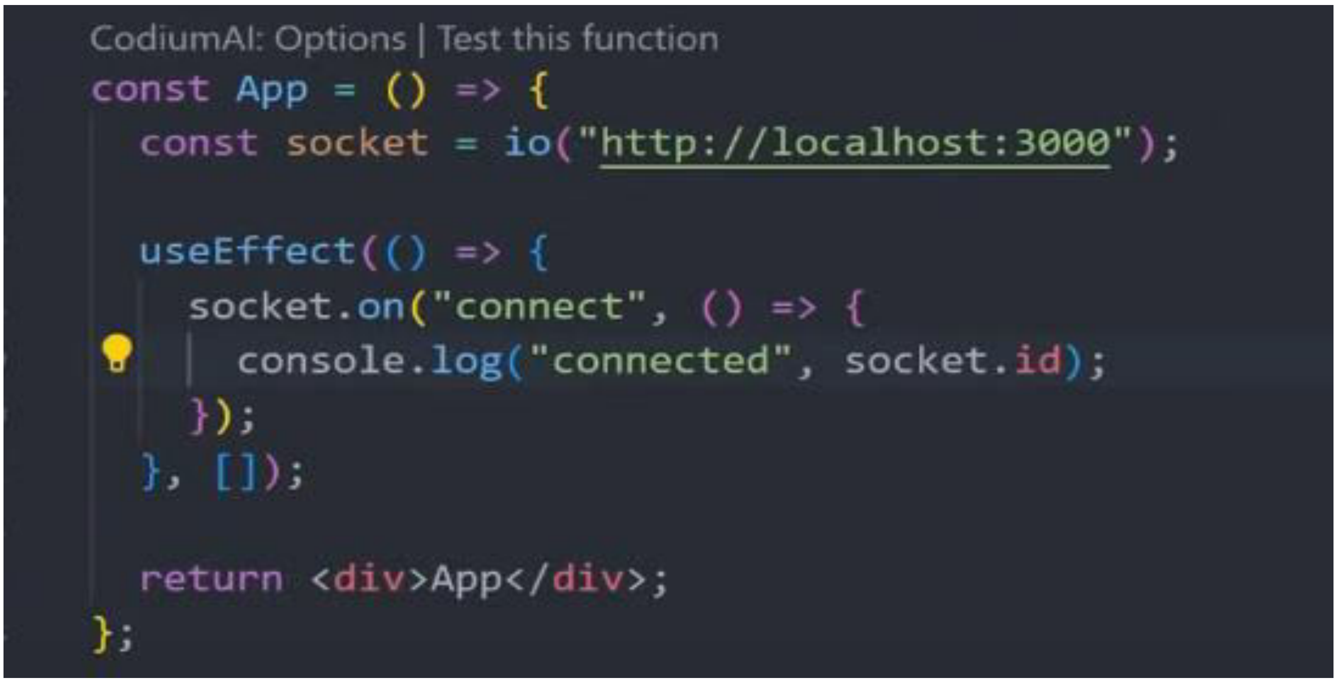
Since humne backend ka url dia h , aur chala frontend me rhe h, to dono ka url alag h, to cors ki wjh se block hojaega to isko resolve krne ke liye hum ye karenge:-



```
const io = new Server(server, {
  cors: {
    origin: "http://localhost:5173",
    methods: ["GET", "POST"],
    credentials: true,
  },
});
```

When have created the io server some extra information we have to pass. In origin we can also give “*” but here we have given frontend origin. In Methods we have given what methods sockets can use, credential true means we enables the exchange of cookies.

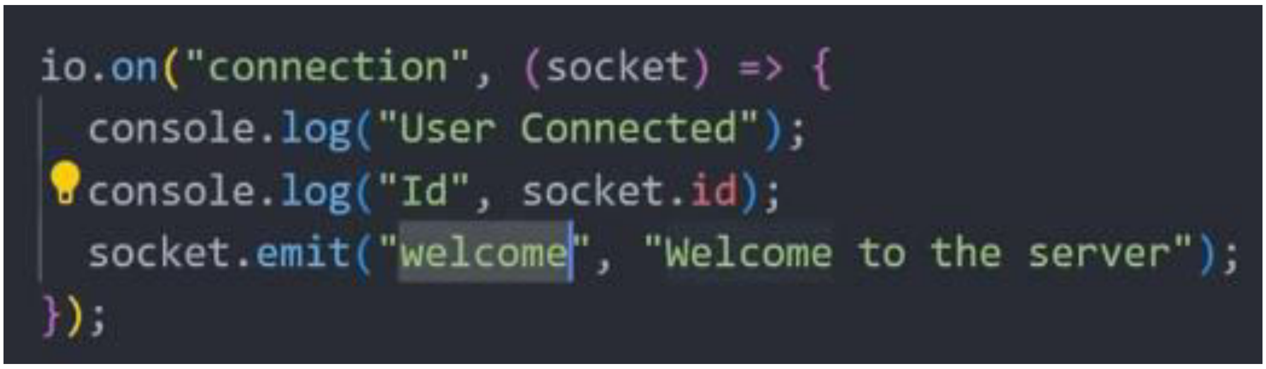
The cors we install by doing npm cors is used for another purpose like cors() is a middleware used APIs created at the backend.



```
CodiumAI: Options | Test this function
const App = () => {
  const socket = io("http://localhost:3000");

  useEffect(() => {
    socket.on("connect", () => {
      console.log("connected", socket.id);
    });
  }, []);

  return <div>App</div>;
};
```

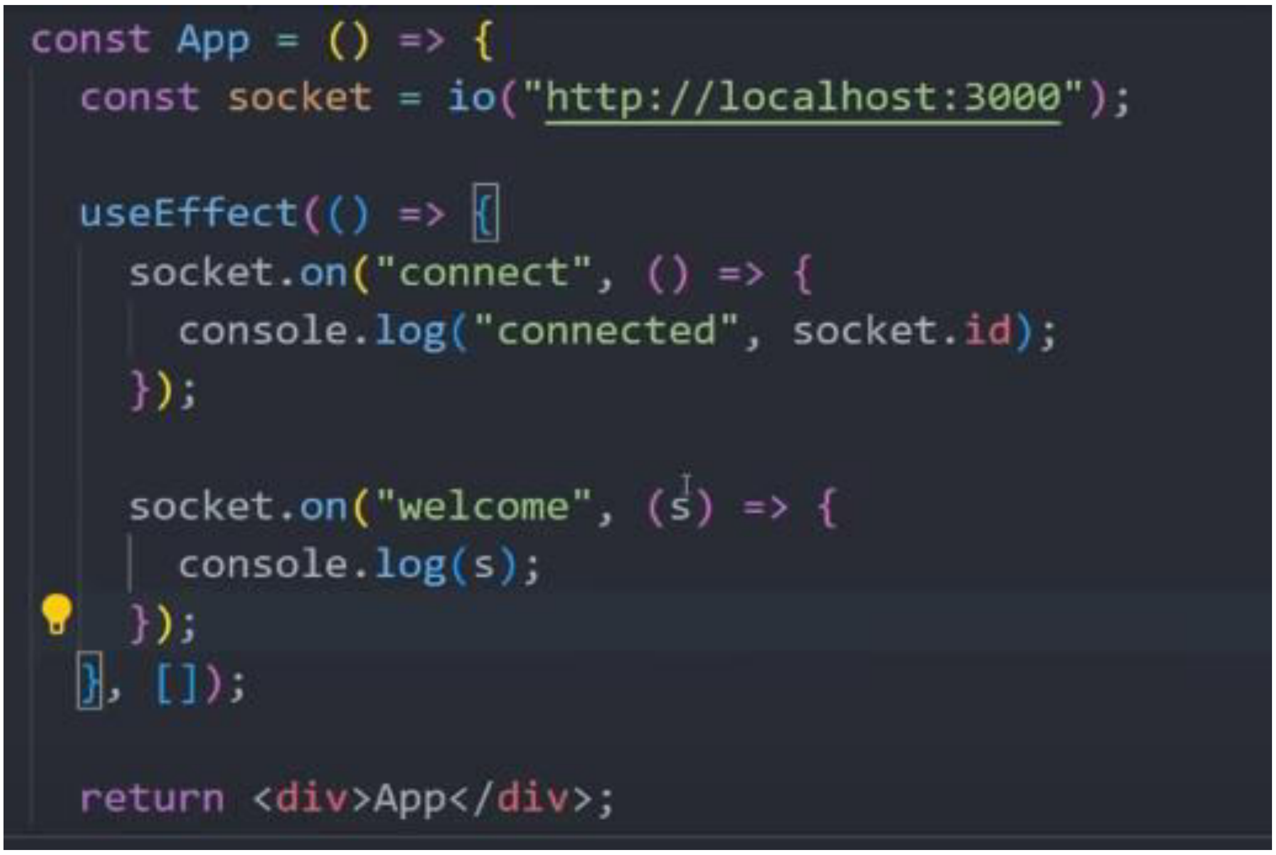
Inside useEffect we created an another event named connect (pre-defined) event, so whenever the socket is connected to the server this event will be triggered. And we have written this inside the useEffect because we do not want infinite re-rendering of this event so it will re-render whenever the page undergoes refresh.

Now from the server if want to give some messages to server, so we do this:-



```
io.on("connection", (socket) => {
  console.log("User Connected");
  console.log("Id", socket.id);
  socket.emit("welcome", "Welcome to the server");
});
```

Here inside the io server, we have created the socket.emit function which will emit the function, now we have to receive this message in frontend so we do this:-

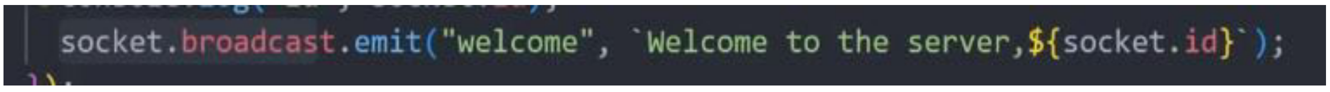


```
const App = () => {
  const socket = io("http://localhost:3000");

  useEffect(() => {
    socket.on("connect", () => {
      console.log("connected", socket.id);
    });

    socket.on("welcome", (s) => {
      console.log(s);
    });
  }, []);

  return <div>App</div>;
```

Here we have done socket.on("same event name as provided by the server/the event name from the server sent by socket.emit from which you want to fetch the data ",(function of what you want to do)=>{});

Note:- socket.emit sends message to every socket present in the circuit.

Now if we want to send the message by socket to another socket except itself so we have to use this:-

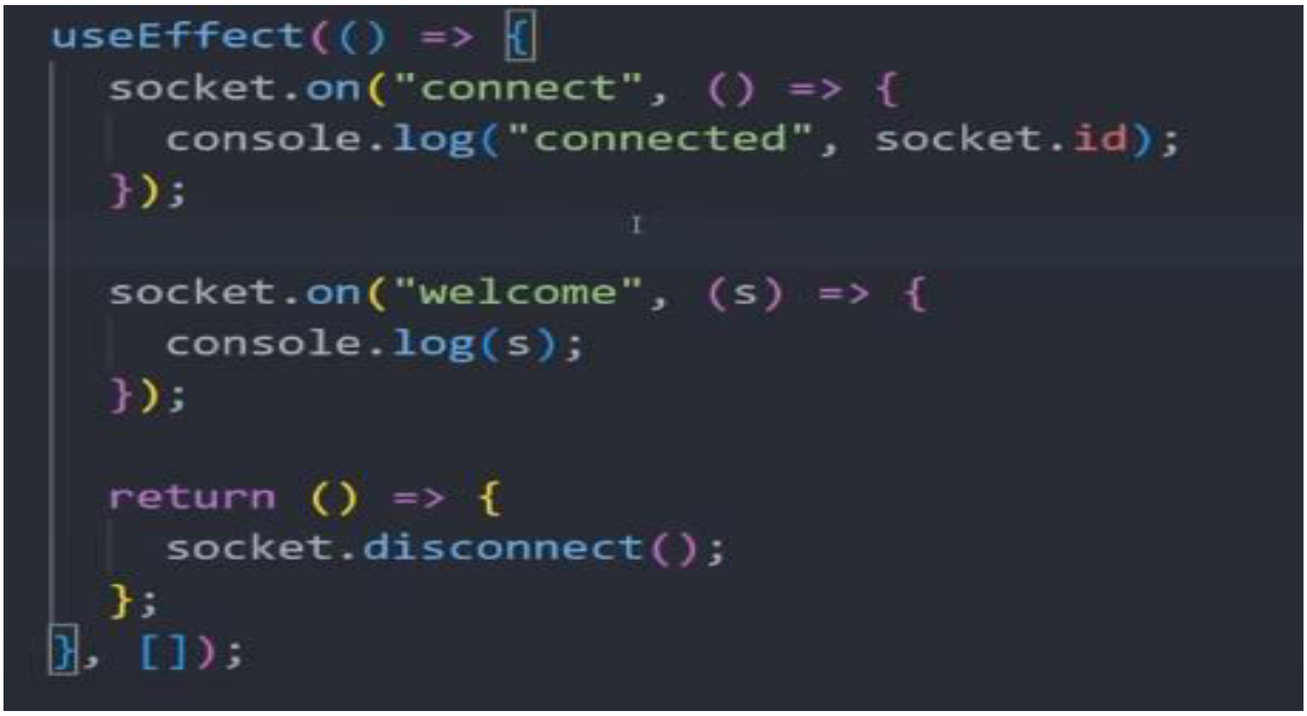


```
  socket.broadcast.emit("welcome", `Welcome to the server,${socket.id}`);
```

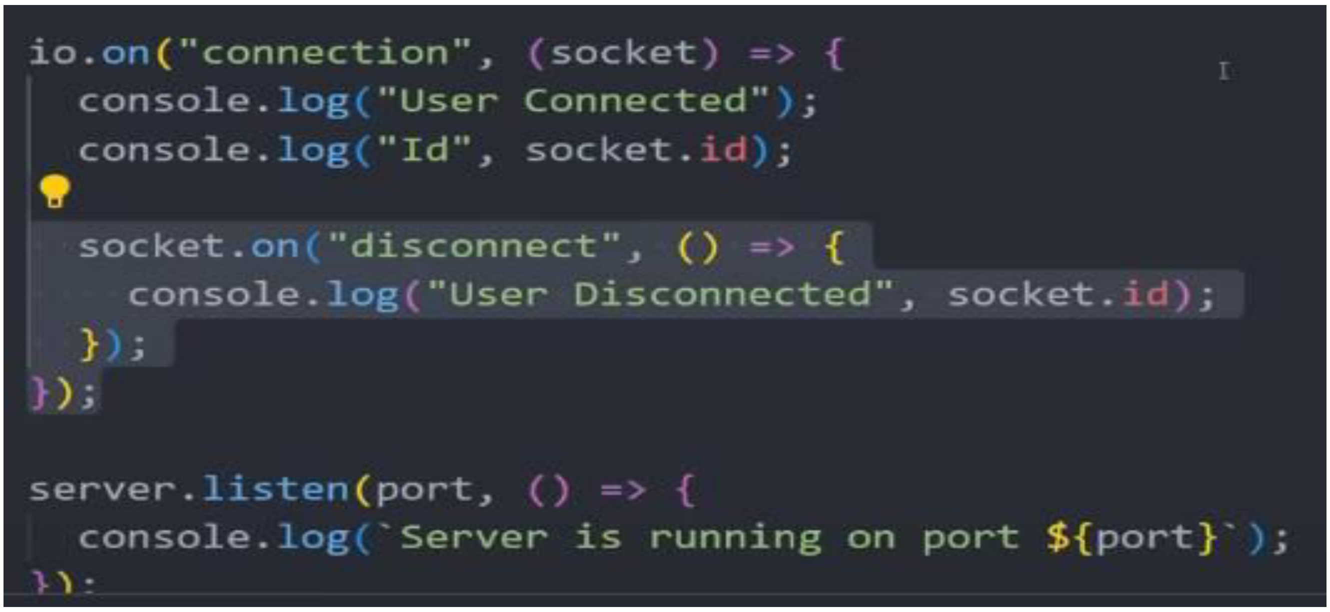
Maanlo 2 tab pe khole hue ho , aur dono user hue kyuki page dono jgh page render hua h , ab page render hua to const socket=io() bhi aaya hoga to kul mila ke 2 user h. ab jis page ki refresh karoge udhr se socket.broadcast.emit hoga aur dusre user pe message jaega.

Par usually hum server side se emit krte nhi abhi bhi example ke liye dikhaya h.

Ab server side se disconnect message bhi deskte h, to pehle to useEffect ke anday changes karna padega:-



```
useEffect(() => {
  socket.on("connect", () => {
    console.log("connected", socket.id);
  });

  socket.on("welcome", (s) => {
    console.log(s);
  });

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

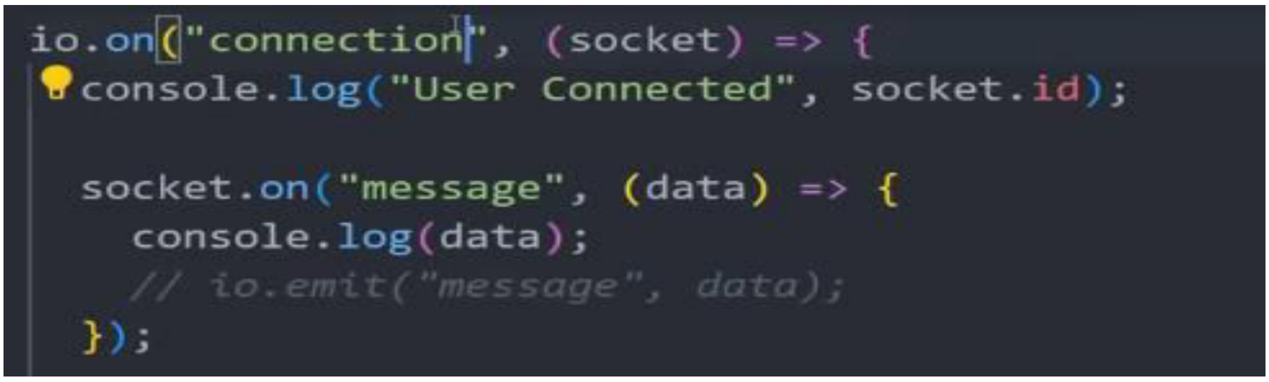


```
io.on("connection", (socket) => {
  console.log("User Connected");
  console.log("Id", socket.id);

  socket.on("disconnect", () => {
    console.log("User Disconnected", socket.id);
  });
});

server.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

Now whenever we refresh the page current user will get disconnected and new user will be created.

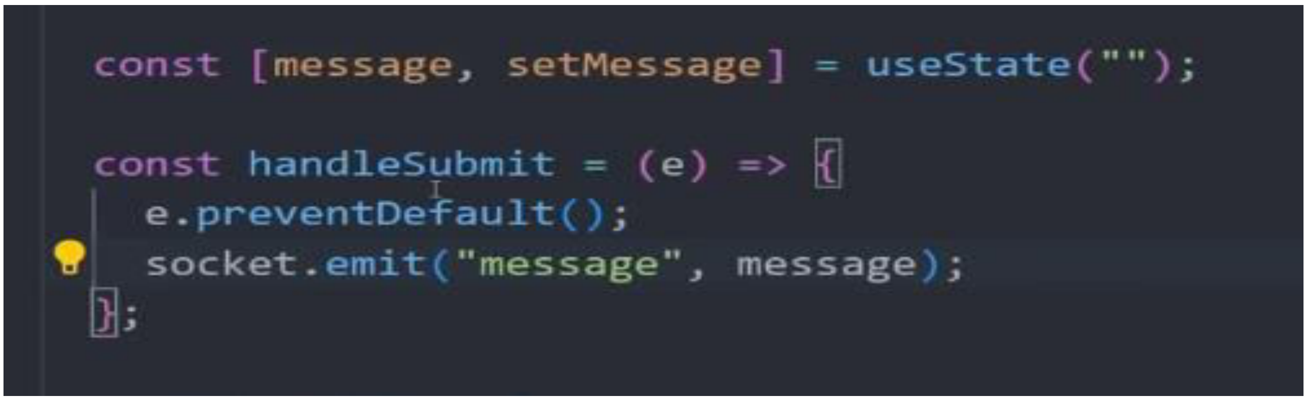
Ye disconnect bus seekhne ke liye tha.

Now we will see how we send and take messages.



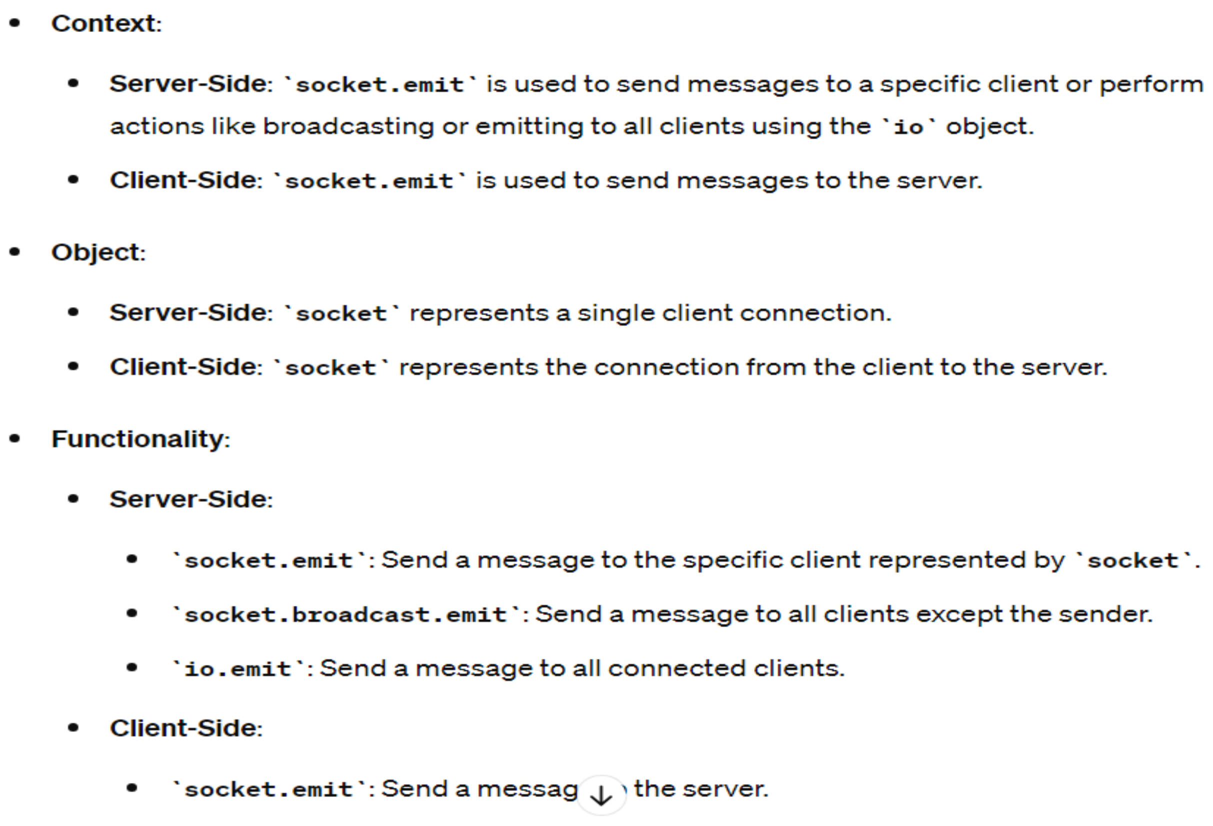
```
io.on("connection", (socket) => {
  console.log("User Connected", socket.id);

  socket.on("message", (data) => {
    console.log(data);
    // io.emit("message", data);
  });
```

Here we have created a listener event named as “message”.



```
const [message, setMessage] = useState("");

const handleSubmit = (e) => {
  e.preventDefault();
  socket.emit("message", message);
};
```

Neeche ek function banaya hua h handleSubmit jhn se message variable update ho rha h, aur ho jb ye function trigger hot ha h to socket message emit krde rha h.



- **Context**:
  - **Server-Side**: `socket.emit` is used to send messages to a specific client or perform actions like broadcasting or emitting to all clients using the `io` object.
  - **Client-Side**: `socket.emit` is used to send messages to the server.
- **Object**:
  - **Server-Side**: `socket` represents a single client connection.
  - **Client-Side**: `socket` represents the connection from the client to the server.
- **Functionality**:
  - **Server-Side**:
    - `socket.emit`: Send a message to the specific client represented by `socket`.
    - `socket.broadcast.emit`: Send a message to all clients except the sender.
    - `io.emit`: Send a message to all connected clients.
  - **Client-Side**:
    - `socket.emit`: Send a messag ↓ the server.

Here you can see, when we do socket.emit from server side, so it will send message to every socket connected to that server, because in client page we have used socket.on(eventname), all the sockets have listener event name, and this can also be implemented by io.emit() where instead of socket circuit/server sending the message to all the sockets connected to it. But if we do socket.emit() in client side and listen this event on server so only server will get the data. Matlb hame ye pata lag gya ki connections sockets ke beech me kaise hota h so ab unme kisko kya dena h nhi dena h khud dekh skte h.