öne24

Title :

Node.js Socket communication

Introduction :

This project is aimed to demonstrate the client and the server real-time communication using WebSocket which is a protocol providing full-duplex communication channels over a single TCP connection.

Using this, one can easily connect with a server and exchange packets without any need of refreshing the page. Event-based communication ensures that both the client and the server are listening to events by carrying data, unlike a HTTP protocol where the data packet has to be sent in order to invoke the communication.

Architecture :

External dependencies/packages involved in the project:

1. **Express**

   Express is a Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework −

- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Allows to dynamically render HTML Pages based on passing arguments to templates.

2. **Socket.io**

   Socket.IO is a JavaScript library that enables real-time bi-directional(client to server & server to client) event-based communication for real-time web applications. It works on every platform, browser or device, focusing equally on reliability and speed. Socket.IO is built on top of the WebSockets API (Client side) and Node.js. It is one of the most depended upon library on **npm** (Node Package Manager).

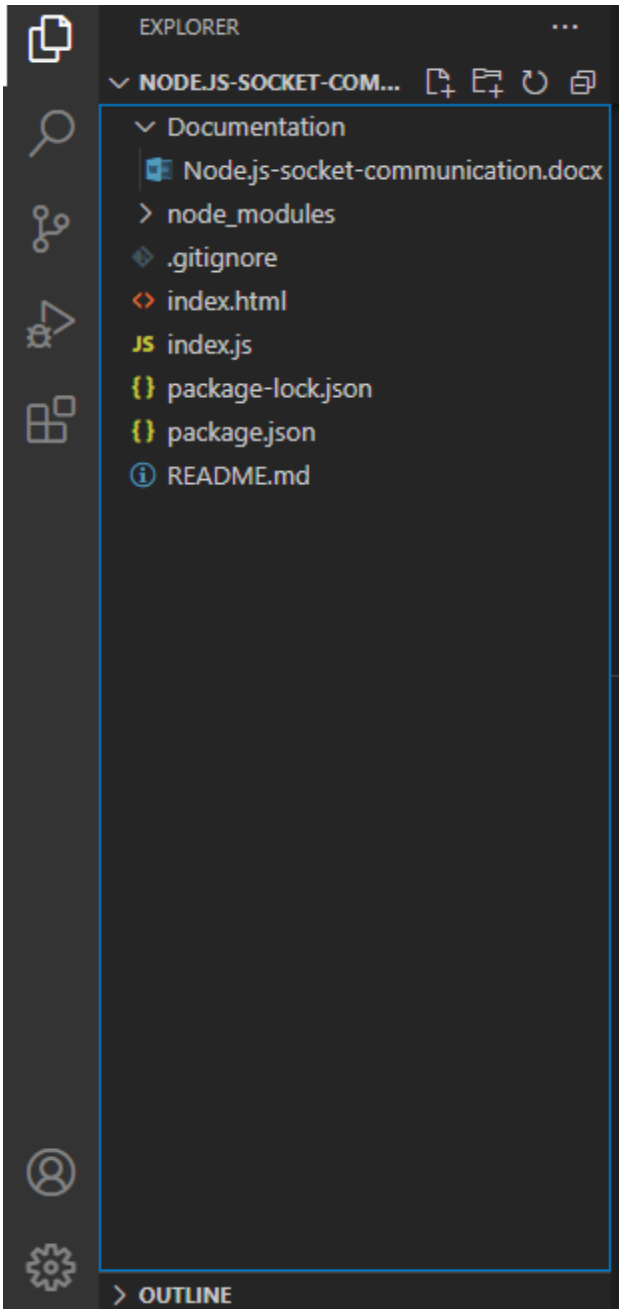
Real-time Applications

A real-time application (RTA) is an application that functions within a period that the user senses as immediate or current.

Some examples of real-time applications are:

- **Instant messengers**: Chat apps like Whatsapp, Facebook Messenger, etc. You need not refresh your app/website to receive new messages.
- **Push Notifications**: When someone tags you in a picture on Facebook, you receive a notification instantly.
- **Collaboration Applications**: Apps like google docs, which allow multiple people to update same documents simultaneously and apply changes to all people's instances.
- **Online Gaming**: Games like Counter Strike, Call of Duty, etc., are also some examples of real-time applications.

Directory Architecture :



Picture: Directory architecture for the project

Code Implementation :

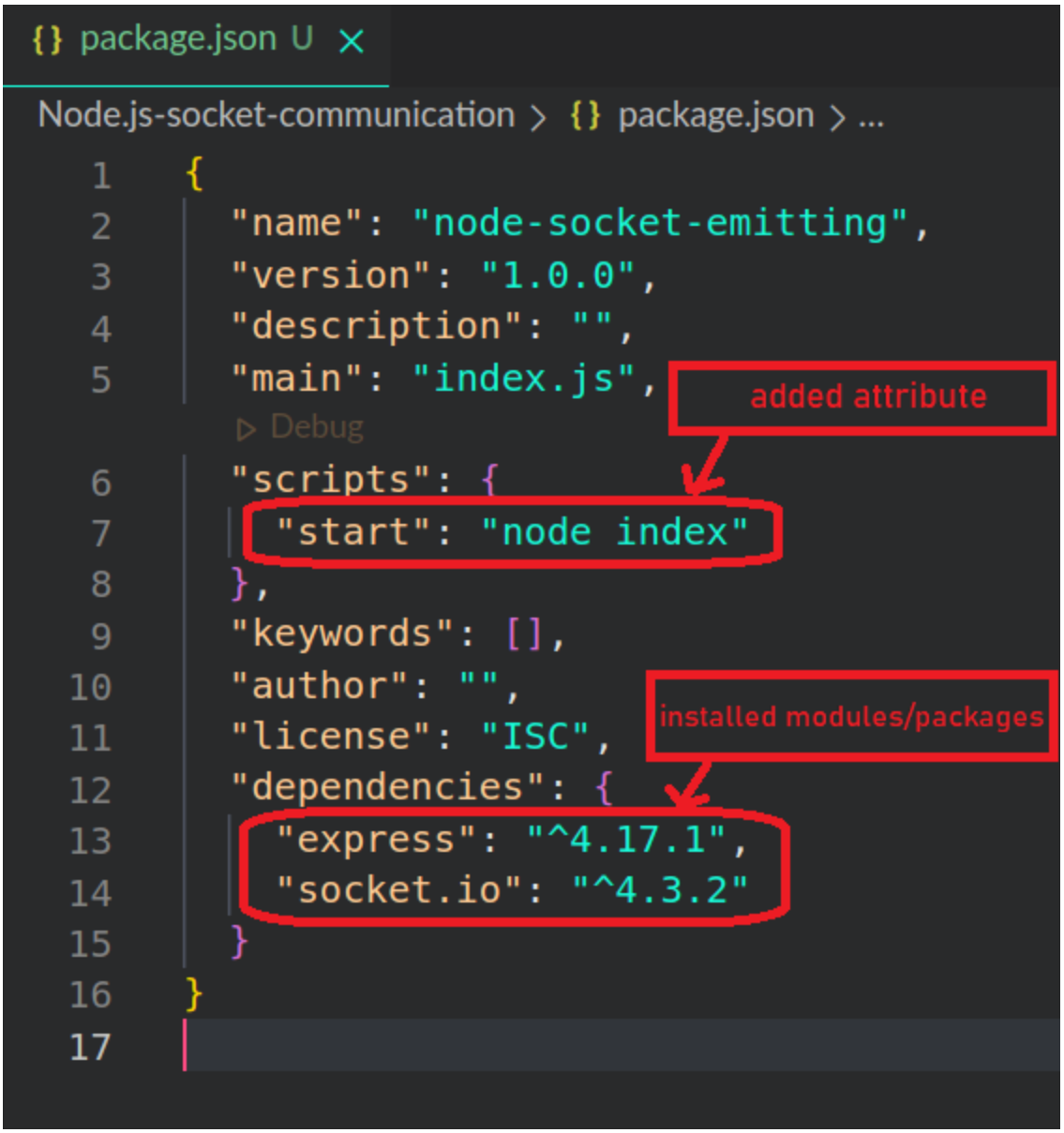
To start the project, in the project directory, execute the following command:
(Make sure that the latest version of Node.js and npm are installed on the system)

npm init -y

This will create a package.json file (using -y flag does not prompt for any details)

Thereafter, to install the external dependencies required for this project, execute the command:

npm install --save express, socket.io



Also, I have edited the ***start*** script, so that node will serve ***index.js*** on start.

Let's implementing the code structure and learn how coordination being formed in between the client and the server.

Now, coming onto the implementation part, first we will create the index.js server file in the same directory where these package.json is kept and start with importing the modules or packages required whichever be installed using the npm packages *i.e.,* express and socket.io.

```
const express = require('express');
const http = require('http');
const socketio = require('socket.io');
```

What is http ?

Http is a built-in module, which allows Node.js application to transfer the data over the **Hyper Text Transfer Protocol(HTTP).**

To include the HTTP module, use the require() method:

const http = require('http');

After the declaration of all the packages to a constant variable, just provide the PORT number so that the browser identifies the transaction over a network by specifying the host(URL).
Let suppose,

const PORT = 3000;

once the constant variable PORT is declared, we call an express() function so that we use the feature of express.
As express() function creates a new express application for you and it internally calls a createApplication() function from the lib/express.js file which is the default export.

For example,

const app = express();

where the constant variable `app` returned from this function is one that we use in our application code.

```
const PORT = 3000;
const app = express();
const server = http.createServer(app);
const io = socketio(server);
```

The above code snippet has a function createServer() that is present in the http package which is used to create a server by listen to the server ports and gives a response back to the client.

For Example,

```
http.createServer(function (req, res) {
  res.write('Hello World!'); //write a response to the client
  res.end(); //end the response
}).listen(8080);
```

We pass a constant variable `app` to the createServer() function in the above code because we further get the response to the `app` variable by the use of the **Basic Routing** feature of express.

Next,

const io = socketio(server);
or *i.e.,* on that minimal line code you write it as,
const io = require('socket.io')(http);

The require ('socket.io')(http) creates a new socket.io instance attached to the http server.

The socket.io package having a function called on() function which is an event handler that handles connection, disconnection, etc. events in it, using the socket object.

```
io.on('connection', (socket) => {
    console.log('New websocket communication');
    let id = Math.floor(Math.random() * (10 - 0 + 1)) + 0;// random i bw 0 and 10

    io.emit('joined', id );
    socket.on('disconnect', () => {
        io.emit('left', id)
        console.log('A web socket connection has disconnected');
    })
})
```

On every connection / disconnection of a client, a message is displayed in the console, then we have generated a random ID for a user (between 0 and 10, inclusive)., and an event ***"joined"*** is emitted with that ***ID*** as the payload.

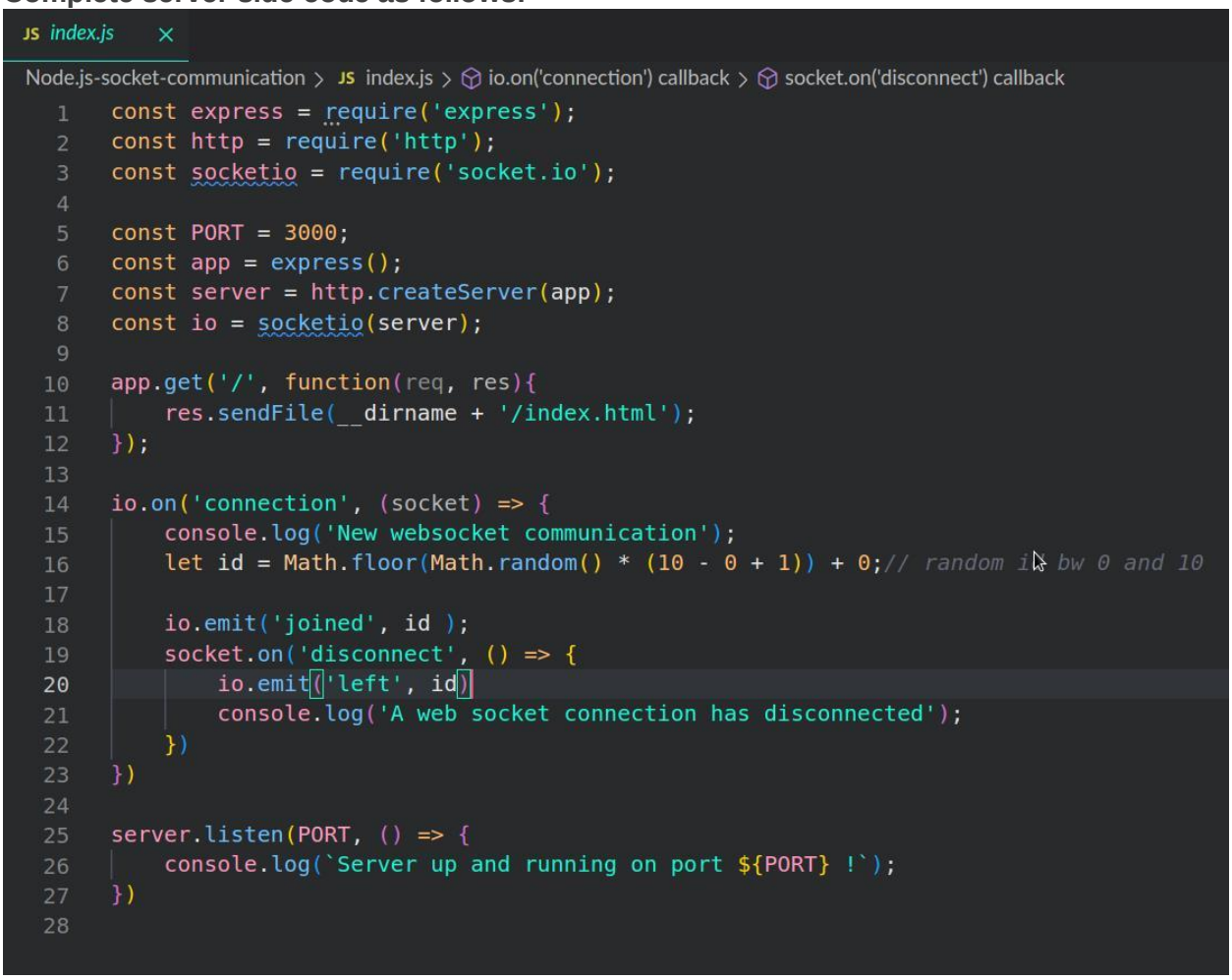
Message was a built-in event provided by the API, but is of not much use in a real application, as we need to be able to differentiate between events.

emit() function ensures that the all the sockets listen to that custom event, including the sender socket itself
socket.on('disconnect') will listen whenever any socket disconnects from the client side, so whenever a socket disconnects, an io event "***left***" will be emitted informing all the sockets that a particular user having that random assigned ***ID*** has been disconnected.

```
server.listen(PORT, () => {
    console.log(`Server up and running on port ${PORT} !`);
})
```

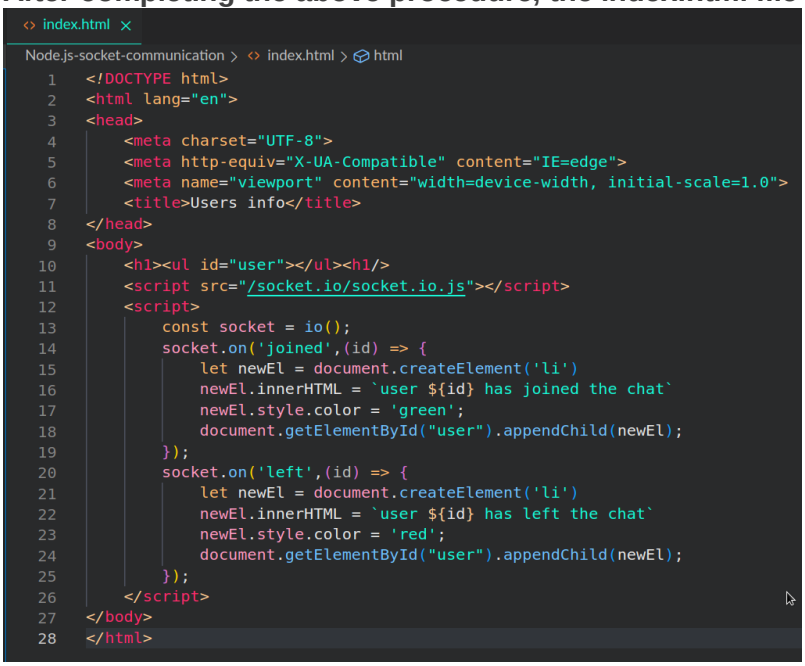
The above listen() function bind and listen the connections on the specified host and port. This method is identical to both http.Server.listen() method and express.listen() method.

Complete server side code as follows:



We have set up our server to log messages on connections and disconnections. We now have to include the client script and initialize the socket object there, so that clients can establish connections when required. The script is served by our io server at **'/socket.io/socket.io.js'.**

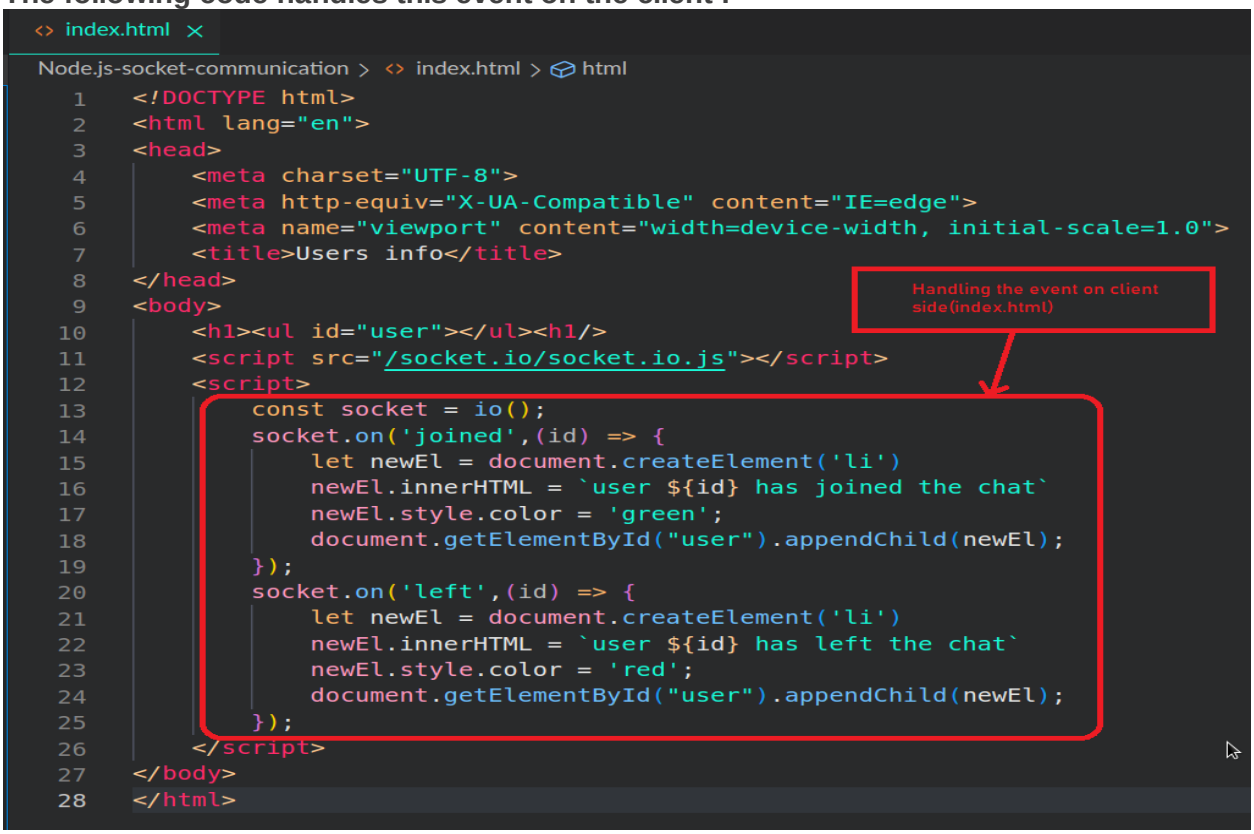
After completing the above procedure, the index.html file will look as follows :



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Users info</title>
</head>
<body>
    <h1><ul id="user"></ul><h1/>
    <script src="/socket.io/socket.io.js"></script>
    <script>
        const socket = io();
        socket.on('joined',(id) => {
            let newEl = document.createElement('li')
            newEl.innerHTML = `user ${id} has joined the chat`
            newEl.style.color = 'green';
            document.getElementById("user").appendChild(newEl);
        });
        socket.on('left',(id) => {
            let newEl = document.createElement('li')
            newEl.innerHTML = `user ${id} has left the chat`
            newEl.style.color = 'red';
            document.getElementById("user").appendChild(newEl);
        });
    </script>
</body>
</html>
```

To handle this custom event on client(index.html) we need a listener that listens for the events `joined` and `left`.

The following code handles this event on the client :




```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Users info</title>
</head>
<body>
    <h1><ul id="user"></ul><h1/>
    <script src="/socket.io/socket.io.js"></script>
    <script>
        const socket = io();
        socket.on('joined',(id) => {
            let newEl = document.createElement('li')
            newEl.innerHTML = `user ${id} has joined the chat`
            newEl.style.color = 'green';
            document.getElementById("user").appendChild(newEl);
        });
        socket.on('left',(id) => {
            let newEl = document.createElement('li')
            newEl.innerHTML = `user ${id} has left the chat`
            newEl.style.color = 'red';
            document.getElementById("user").appendChild(newEl);
        });
    </script>
</body>
</html>
```

For example,
We can also emit events from the client. To emit an event from your client, use the emit function on the socket object.

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io();
      socket.emit('clientEvent', 'Sent an event from the client!');
   </script>
   <body>Hello world</body>
</html>
```

To handle these events, use the **on() function** on the socket object on your server.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
   res.sendFile('E:/test/index.html');
});
io.on('connection', function(socket){
   socket.on('clientEvent', function(data){
      console.log(data);
   });
});

http.listen(3000, function(){
   console.log('listening on localhost:3000');
});
```

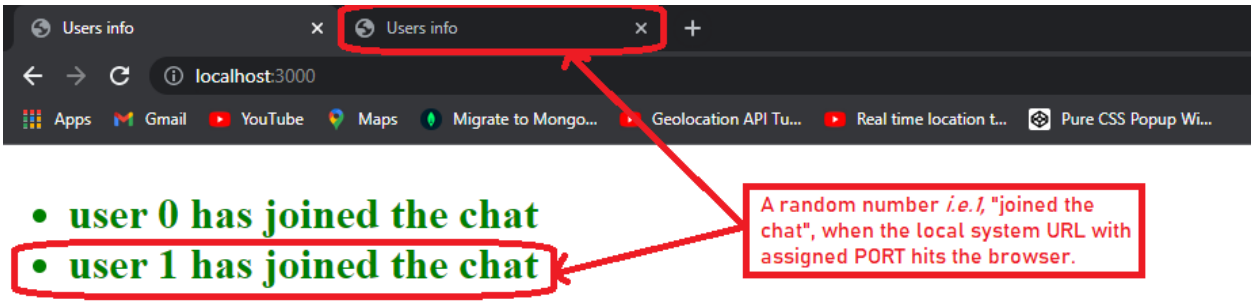
So, now if we go to localhost:3000, we will get a custom event called **clientEvent** fired. This event will be handled on the server by logging :

Output:

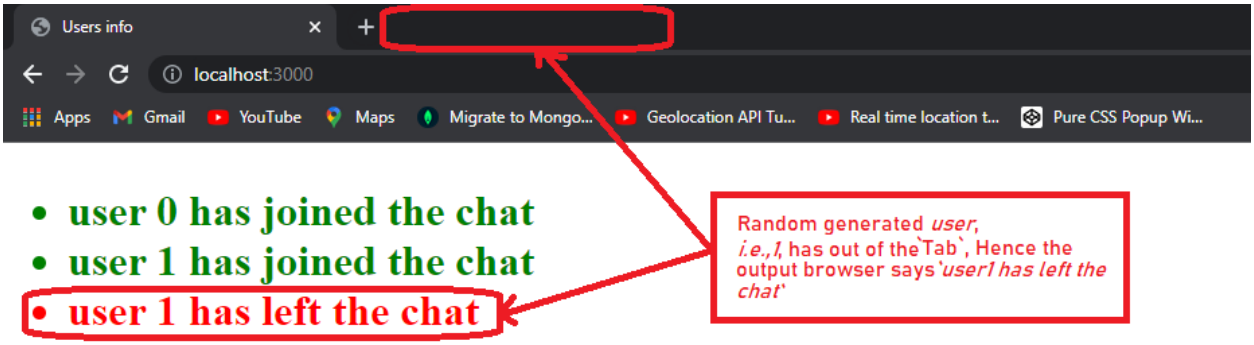
```
Sent an event from the client!
```

As we have imported the client side dependency required for socket.io and defined information about users to be shown as a list followed by bullets with red and green color.

Client will listen to the ***"joined"*** event emitted from the server, and will use the ***ID*** payload received to display the user joining information in a ***green*** text.
Same goes with the ***"left"*** event which was emitted from the server side, but displaying the user disconnecting information in a ***red*** text.



Picture-1: A random number *i.e.1*,"joined the chat" which shows in green color, when the local URL *(http://localhost:3000)* hits to the browser



Picture-1: A random generated number *i.e.1*,"left the chat" which shows in red color, when the `Tab` is discarded from the browser.

Subsequent testing of the application will display results like above ***pictures.***

References:

- https://www.w3schools.com/nodejs/nodejs_http.asp
- https://www.geeksforgeeks.org/express-js-app-listen-function/
- https://www.tutorialspoint.com/socket.io/index.htm
- Node.js official site
- Express.js official site
- Socket.io official site and docs

öne24

Thank you.

Paper written by:
Anukul Kumar
National Institute of Technology(N.I.T), Agartala.
Company: One24

Paper reviewed by:
Rahul Ranjan
(Signature)

One24.