Class 13 - Websockets

Agenda

1. Http and realtime connections
2. Websockets
    a. Features
    b. Pros
    c. Cons
3. Chat app example

## Client Server communication flow

1. When a client makes a request to a server over the web (e.g., entering a URL in a browser), it first establishes a TCP connection with the server.
2. Three way handshake happens. This involves the exchange of SYN (synchronize) and ACK (acknowledgment) packets.
3. Once the TCP connection is established, the client sends an HTTP request through this connection.
4. The server receives the HTTP request, processes it, and sends back an HTTP response.
5. The response is received by the client, and the information (like a webpage) is rendered for the user.
6. The TCP connection can be closed or kept open for further requests, depending on the HTTP headers (like Connection: keep-alive).

## Design choices of the communication system in place

1. By design http was designed to be stateless
2. This communication was designed to be a client initiated protocol.
    a. Client initiates request to which server responds back
3. We tried to remove certain inefficiencies like create a new tcp connection for every request
    a. In HTTP/1.1, persistent connections were introduced to overcome this inefficiency. With persistent connections, also known as HTTP keep-alive, a single TCP connection can be reused for multiple HTTP requests and responses between a client and server.
    b. HTTP/2, standardized in 2015, built upon the concept of persistent connections by introducing multiplexing. This allows multiple requests and responses to be simultaneously active on the same connection.
    c. Nevertheless they still remain client initiated model

## Limitations of http for real time communication

1. One-Way Communication
    a. The server cannot initiate communication with the client, which limits real-time interactions.
2. Overhead of HTTP Headers: Each HTTP request and response carries a significant amount of header data. This overhead is

inefficient, especially for applications that need to send frequent, small messages, like real-time chat messages or live price updates in trading applications.

3. No Persistent Connections
    a. it does not allow for continuous, full-duplex communication where data can flow in both directions simultaneously.
    b. Full-duplex means that data can flow in both directions simultaneously, which is crucial for real-time applications
4. Latency Issues: HTTP's request-response nature can lead to latency issues in real-time applications.
    a. I need market updates for live tracker. I need to send requests frequently to get the updates

Other techniques

1. Polling: In this approach, the client (typically a web browser) would regularly send HTTP requests to the server to check for new messages. This was a straightforward approach but inefficient, as it involved sending requests at regular intervals regardless of whether there were new messages, leading to unnecessary network traffic and server load.
2. Long Polling: An improvement over traditional polling, long polling involves the client sending a request to the server, which then keeps the request open until new data (like a chat message) becomes available. Once the data is sent to the client, the connection closes, and the client immediately opens a new

connection. This method reduces the amount of unnecessary HTTP requests but still has higher latency compared to

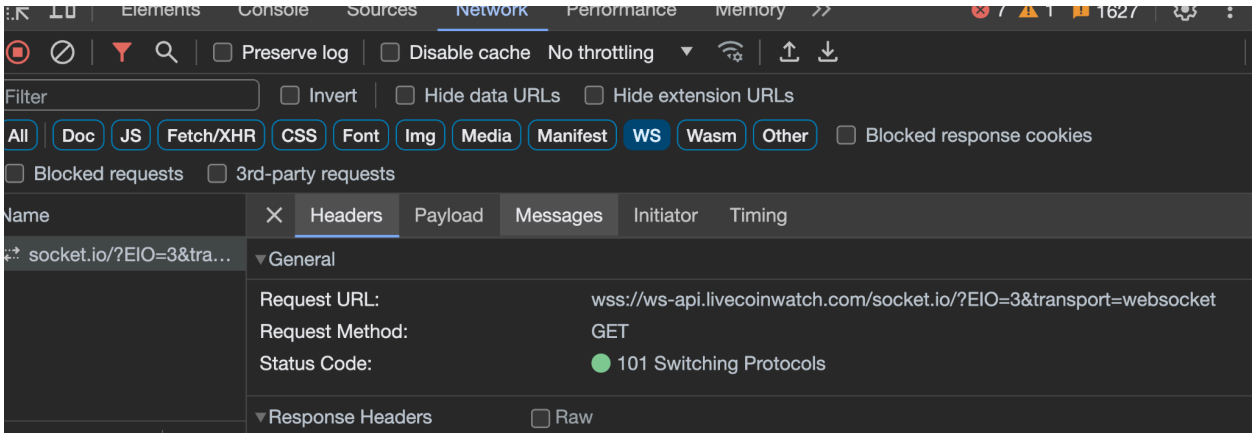
WEBSOCKETS

1. there's a dedicated channel for communication between the server and each client.
2. Persistent Connections: unlike HTTP, WebSocket creates a persistent connection between each client and the server. it remains open for two-way communication until explicitly closed by eithe client or a server
3. WebSocket connection as a "channel" or "pipeline" that directly links each client to the server. This channel is unique and private to the client-server pair.
4. Full-Duplex Communication: channels are full-duplex, meaning that both the server and the client can send messages to each other independently and simultaneously without waiting for a request-response sequence.
5. Real-time Interaction: Highlight that this setup allows for real-time interaction. As soon as the server has new data (like a message or update), it can immediately send it to the connected client(s), and vice versa.
6. WebSocket starts as an HTTP connection and then "upgrades" to a WebSocket connection through a handshake process. This upgrade is initiated with an HTTP request including a header

(Upgrade: websocket) indicating the desire to establish a WebSocket connection.

7. Another advantage that comes out is the header data in WebSocket communication is minimal because, unlike HTTP, it does not require the continuous transmission of cookies, user credentials, or other client-specific headers with each message, resulting in smaller packet sizes."

## Demo for websockets

1. go to https://www.livecoinwatch.com/
2. Go to network tab and select ws ( websockets ) and reload
3. 

4. Imp points to note
    a. When you select the 'WS' (WebSocket) filter in the Network tab of your browser's developer tools and see a request with a status code of 101, you're looking at the WebSocket handshake.

b. Status Code 101: This indicates 'Switching Protocols'. When you see this, it means the server understood the client's request to open a WebSocket connection and is agreeing to switch protocols from HTTP to WebSocket.
c. Response Headers: headers like Upgrade: websocket and Connection: Upgrade,
d. Messages Tab: After the handshake, you can switch to the 'Messages' tab within the WebSocket connection in the developer tools. This tab will show you the actual data frames being sent and received through the WebSocket. This is where you can see the real-time communication aspect of WebSockets.
    i. Go to the timestamp column to see the real time updates

CONS

1. Resource Utilization for Idle Connections: WebSocket connections, even when idle, continue to consume resources on the server, which can be inefficient compared to stateless HTTP connections that are closed after a transaction.
2. WebSocket does not have a built-in mechanism for back-pressure, which is the ability to handle situations where the server is sending data faster than the client can process it.
3. Each open WebSocket connection consumes resources on the client side, including memory and network ports. The client

device's capabilities (such as CPU, RAM, and network bandwidth) can thus limit the number of connections that can be practically managed.

4. the persistent nature of the connection can pose additional security challenges,potential for the server to be exposed to Denial of Service (DoS) attacks.
5. Smaller packet size can lead to fragmentation of large messages

## Websockets: Implementation

1. MDN docs - The WebSocket API is an advanced technology that makes it possible to open a two-way interactive communication session between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply..
2. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
3. Come to socket.io - https://socket.io/docs/v4/
4. Create a new folder and do npm init - -y
5. Npm i express
6. Create a file websockets.js under root folder and setup a basic route

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

const app = express();

app.get('/', (req, res) => {
```

```
 res.send("hello world");
});

app.listen(3000, () => console.log("listening at 3000"));
```

7. Now we will use http module from node to create the server

```
const express = require("express");
const http = require("http");

const app = express();
const server = http.createServer(app);

app.get('/', (req, res) => {
 res.send("hello world");
});

server.listen(3000, () => console.log("listening at
3000"));
```

   a. When using express, app.listen internally creates an HTTP server for us
   b. However, when we're working with WebSocket implementations such as using socket.io, we need direct access to the HTTP server object because socket.io needs to attach to it. This is why explicitly create the server using the http module..
   c. The HTTP module's createServer method here creates an HTTP server instance. This server listens for network

requests and can serve both HTTP and WebSocket requests.

d. The idea is
e. Express App (app): Manages the application logic, routing, middleware, etc., specific to handling web requests.
f. HTTP Server (server): Manages the lower-level HTTP communication between the client and server for bidirectional communication

8. Npm i socket.io
9. Nodemon websocket.js and open browser and go to localhost:3000 to see the response

## Adding socket io on backend

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

const app = express();
const server = http.createServer(app);
// this io is responsible for handling all the socket
connections
const io = new Server(server);

io.on('connection', (socket) => {
   console.log('a user connected');
 });
```

```
app.get('/', (req, res) => {
 res.send("hello world");
});

server.listen(3000, () => console.log("listening at 3000"));
```

Creating a client to initiate socket connection

1. Create a public folder to return the html where we will initiate a connection
2. Create index.html under public under websocketClient folder

```
<!DOCTYPE html>
<html lang="en">
 <head>
   <meta charset="UTF-8" />
   <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
   <title>Websockets</title>
 </head>
 <body>
   <h1>Websockets</h1>
    </body>
</html>
```

3. In the websockets.js, update this code

```
const app = express();
app.use(express.static("public"));
```

4. this is a built-in middleware function in Express. It specifies that files in the directory named "public" should be served as static resources. Static resources are files like HTML, CSS, JavaScript, images, etc., that don't need to be generated, modified, or processed on the server before being sent to the client.
5. if you have an index.html file in the public directory, it will be served as the default file when you access http://yourserver.com/.
6. In our index. html file, add the script for client side

```
  <script src="/socket.io/socket.io.js"></script>
  <script>
    const socket = io();
  </script>
```

7. In our Node.js code, when we create a Socket.IO instance and attach it to our HTTP server, Socket.IO sets up a special route (endpoint) to serve its client-side library. This endpoint is typically /socket.io/socket.io.js.
8. Reload localhost:3000 and see the console on the server
9. Now let us send some message from server to client

```
io.on("connection", (socket) => {
 console.log("a user connected");
 socket.emit("message", "message from server");
});
```

10. On our client code, add below

```
<script>
    const socket = io();
```

```
        socket.on("message", (data) => {
            console.log("receiving message", data);
        });
    </script>
```

Unique socket id for every connection

1. Every new connection from client will have a different socket id

```
io.on("connection", (socket) => {
 console.log("a user connected");
 socket.emit("message", "message from server",socket.id);
});
```

2. Open multiple tabs and all will have different socket ids
3. Let us add a date part to the msg so that we can see regular updates clearly

```
io.on("connection", (socket) => {
 console.log("a user connected");
 setInterval(() => {
   socket.emit(
     "message",
     "message from server" + "-" + socket.id + "at" + new
Date()
   );
 }, 2000);
```

4. Disconnect event
    a. Inside the io.on handler create socket.on listener for disconnect event

```
io.on("connection", (socket) => {
 console.log("a user connected");
 setInterval(() => {
   socket.emit(
     "message",
     "message from server" + "-" + socket.id + "at" +
new Date()
   );
 }, 2000);

 // disconnect event is fired when a user disconnects
from the server
   socket.on("disconnect", () => {
       console.log("user disconnected"+socket.id);
   });
});
```

Creating a simple broadcast messages to multiple clients

1. From the index.html, remove the script and move it to script.js under websocketClient folder

```
<script src="/socket.io/socket.io.js"></script>
   <script src="./script.js">


   </script>
```

2. In the script.js , move the copied code

```
const socket = io();
socket.on("message", (data) => {
```

```
 console.log("receiving message", data);
});
```

3. In index.html, add the input box and send btn

```
<h1>Websockets</h1>
   <input type="text" id="message" />
   <button id="send">Send</button>
   <script src="/socket.io/socket.io.js"></script>
   <script src="./script.js"></script>
```

4. Add this in script.js to send the message to the server

```
const btn   = document.getElementById("send");
const input = document.getElementById("message")
btn.addEventListener("click",()=>{
 const value = input.value;
   socket.emit("message",value);
   input.value = "";
})
```

5. On the server side , receive the message and publish it

```
// message event is fired when a user sends a message
 socket.on("message", (data) => {
   socket.broadcast.emit("broadcast", data);
 });
```

6. Broadcasts the received message to all other connected clients except the sender.
7. Let us add some style to show the messages differently

```
<h2>Messages</h2>
```

```
    <div class="messages">
      <ul></ul>
    </div>
    <script src="/socket.io/socket.io.js"></script>
    <script src="./script.js"></script>
```

8. Add style tag

```
<style>
ul{
        list-style: none;
      }

      .sender {
        background-color: lightcoral;
        margin-left: 4rem;
      padding:1rem;

      }
      .receiver {
        background-color: lightblue;
        margin-left: 2rem;
      padding:1rem;

      }

    </style>
  </head>
```

9. Update script.js to add some code in the sender's flow

```
btn.addEventListener("click",()=>{
 const value = input.value;
```

```
 const div = document.createElement("div");
 div.setAttribute("class","sender");
 const li = document.createElement("li");
 li.innerText = value;
 const para = document.createElement("p");
 para.innerText = "sender";
 div.appendChild(para);
 div.appendChild(li);
 ul.appendChild(div);
 input.value = "";
 socket.emit("message",value);
})
```

10. Add some code for the received broadcasted message

```
// brodcasted messages
socket.on("broadcast",(data)=>{
 console.log("broadcasted message",data);
 const div = document.createElement("div");
   div.setAttribute("class", "receiver");
   const li = document.createElement("li");
   li.innerText = data;
   const para = document.createElement("p");
   para.innerText = "receiver";
   div.appendChild(para);
   div.appendChild(li);
   ul.appendChild(div);
})
```

11. see for two tabs

Rooms in socket

1. Socket.IO provides a powerful feature called "rooms" that allows you to organize sockets into different groups. Rooms are a way to segregate clients based on certain criteria, such as users in a specific chat room or participants in a particular game. This makes it easy to broadcast messages to a subset of clients, rather than to every connected client.
2. Grouping Sockets: Rooms allow you to group sockets into different namespaces. Each socket can join multiple rooms.
3. Once sockets are grouped into rooms, you can emit events to all sockets in a room, which is useful for sending messages or updates to a specific group of users.
4. Sockets in one room are isolated from sockets in other rooms, which means broadcasting to one room does not affect sockets in other rooms.
5. Create a button in index.html to create group

```
   <button id="createGrp">Create Group</button>
```

6. In script.js, add an event listener to initiate room creation

```
const grpBtn = document.getElementById("createGrp");


grpBtn.addEventListener("click", () => {
 console.log("group created req")
 socket.emit("create_grp", Math.random(0, 1) * 1000);
}); // random room number
```

7. In websockets.js, listen to this even

```
let room
socket.on("create_grp", (roomId) => {
    console.log("group is created");
    // first participant
    room = roomId;
    socket.join(roomId);
  });


 // disconnect event is fired when a user disconnects from
the server
 socket.on("disconnect", () => {
   console.log("user disconnected" + socket.id);
 });
```

8. Create another button to join the group

```
   <button id="joinGrp">Join Group</button>
```

9. Script.js, add the handler

```
const joinGrp = document.getElementById("joinGrp");
joinGrp.addEventListener("click", () => {
 console.log("grp join req");
 socket.emit("join_room");
});
```

10. Now we will listen to this event on the server and make this socket join the room

```
 socket.on("join_room", () => {
```

```
    console.log(socket.id + " joined the room ", room);
    socket.join(room);
 });
```

11. Create a send to group button to send message privately

```
    <button id="stg">Send to group</button>
```

12. In script.js, add the handler

```
const stg = document.querySelector("#stg");
stg.addEventListener("click", function () {
 let value = input.value;
 if (value) {
    socket.emit("grp message", value);
 }
});
```

13. Listen to this event on the server and pass the messages to all sockets in that room

```
socket.on("grp message", function (data) {
    socket.to(room).emit("serv_grp_message", data);
 });
```

14. Finally we will have a listener that listens to the messages in the room

```
socket.on("serv_grp_message", function (data) {
 console.log("grp message", data);
});
```

## 15. Leaving the room

```
 <button id="stg">Send to group</button>
   <button id="leave">Leave group</button>
```

## In script.js

```
const leaveRoomBtn = document.getElementById("leave");

leaveRoomBtn.addEventListener("click", () => {
 socket.emit("leave_room");
});
```

## In websocket.js

```
socket.on("leave_room", () => {
       console.log(socket.id + " left the room", room);
       socket.leave(room);
     });
```