Lecture 17 - Socket Programming

What is Socket Programming?

Socket programming is a technique that enables communication between computers over a network using sockets, which are endpoints for sending and receiving data.

Needs of Socket Programming

- **Real-time Communication**: Socket programming is crucial for enabling real-time communication between devices and applications.
- **Client-Server Model**: It allows multiple clients to connect to a server, enabling efficient sharing of resources and data.
- **Cross-Platform**: It works across different platforms and programming languages, making it versatile for diverse applications.

Socket Programming Working:

- **Socket Creation**: Sockets are created using programming libraries or APIs. They are identified by an IP address and port number.
- **Server Setup:** In a client-server model, a server socket listens for incoming client connections on a specific port.
- **Client Connection**: Clients create socket connections to the server's IP address and port.
- **Data Exchange**: Once connected, data can be sent and received between clients and the server.
- **Protocols**: Sockets often use protocols like TCP (reliable, connection-oriented) or UDP (fast, connectionless) for data transmission.
- **Bi-Directional**: Sockets support bidirectional communication, allowing both sending and receiving data simultaneously.

Applications of Socket Programming:

- **Chat Applications**: Sockets power real-time chat apps like WhatsApp, Slack, and IRC.
- **Online Gaming**: Multiplayer online games use sockets for real-time player interaction.
- **Web Applications**: WebSockets enable real-time updates in web applications, such as stock tickers or social media notifications.
- **Video Streaming**: Sockets are used for live video and audio streaming services.
- **IoT Devices**: Internet of Things (IoT) devices use sockets to send data to servers or other devices.
- **Remote Control**: Remote desktops and remote control applications utilise sockets for data transmission.
- **Network Services**: Network services like FTP, SMTP, and HTTP rely on socket programming for data transfer.

Setup server using socket.io

Setting up a server using Socket.IO is a crucial step in building real-time applications that require instant communication between clients and the server. Here's a brief overview of how to set up a server using Socket.IO:

Step 1: Setting Up the Backend Server

```
import express from 'express';
import http from 'http';
import { Server } from 'socket.io';
```

```
import cors from 'cors';

// Create an Express app
const app = express();
app.use(cors());

// Create an HTTP server
const server = http.createServer(app);

// Initialize Socket.io
const io = new Server(server, {
    cors: {
        origin: '*',
        methods: ["GET", "POST"]
    }
});

// Handle client connections
io.on("connection", (socket) => {
    console.log("Connection made.");

    // Handle events and interactions here

    // Handle disconnects
    socket.on("disconnect", () => {
        console.log("Connection disconnected.");
    });
});

// Start the server on port 3000
server.listen(3000, () => {
    console.log("Listening on port 3000");
```

```
});
```

Explanation:

- We import modules such as express, http, socket.io, and cors.
- We create an Express app and enable Cross-Origin Resource Sharing (CORS) to allow client-side connections from any origin.
- An HTTP server is created using Express.
- We initialise Socket.io on the server, specifying CORS options to allow connections from any origin.
- The io.on("connection", ...) block handles incoming socket connections. When a client connects, it logs a message.
- You can implement event handling and interactions within this connection block.
- We also handle client disconnects by listening to the "disconnect" event.

Step 2: Setting Up the Client-Side HTML

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Socket.io Example</title>
</head>

<body>
    <!-- Include the Socket.io client library script -->
    <script
src="http://localhost:3000/socket.io/socket.io.js"></script>
```

```
    <script>
        // Initialize a connection to the server
        const socket = io.connect('http://localhost:3000');

        // Implement client-side logic here

        // Handle events and interactions
        // For example:
        socket.on("message", (data) => {
            console.log("Received message from server:", data);
        });

        // Emit an event to the server
        // For example:

document.querySelector('#sendMessageBtn').addEventListener('click'
, () => {
            const message =
document.querySelector('#messageInput').value;
            socket.emit("sendMessage", message);
        });
    </script>
</body>

</html>
```

Explanation:

- In the client-side HTML, we include the Socket.io client library script, which connects to the server.
- We initialise a connection to the server by creating a socket object and connecting it to the server URL (http://localhost:3000 in this example).
- Inside the <script> tag, you can implement client-side logic.
- We listen for the "message" event from the server using socket.on(). When a message is received, we log it to the console.
- We also demonstrate how to emit an event to the server when clicking a button with the ID sendMessageBtn. The emitted event is named "sendMessage," and it sends a message from the client to the server.

Step 3: Implementing Client-Side Logic

- The client-side logic can be extended to handle various events and interactions based on your application's requirements. You can listen for server-sent events and emit events to the server to create real-time functionality.

Step 4: Running the Application

- Start the server by running node server.js in your terminal.
- Access the client page by opening the index.html file in a web browser.
- Interact with the client-side application by sending messages or triggering events. You'll see logs in the server console when clients connect or disconnect and when messages are received.

Socket.io's basic events

Socket.IO provides several basic events that you can use to implement real-time communication between clients and the server.

Listening for Events with on

In Socket.io, you can listen for events sent from the server or other clients using the on method.

Server-side (server.js):

```
io.on("connection", (socket) => {
    console.log("Connection made.");

    // Listen for a custom event named "chatMessage"
    socket.on("chatMessage", (message) => {
        console.log(`Received message from client: ${message}`);

        // Broadcast the message to all connected clients,
including the sender
        io.emit("chatMessage", message);
    });

    // ...
});
```

Client-side (index.html):

```
// Listen for the "chatMessage" event from the server
socket.on("chatMessage", (message) => {
    console.log(`Received message from server: ${message}`);
});
```

- On the server, we use socket.on("chatMessage", ...) to listen for the "chatMessage" event from clients.
- On the client, we use socket.on("chatMessage", ...) to listen for the same event from the server.

Sending Events with emit

Using the emit method, you can send custom events from the server to clients or from clients to the server.

Server-side (server.js):

```
// Sending a custom event named "serverMessage" to a specific client
socket.emit("serverMessage", "Hello from the server!");

// Sending a custom event to all connected clients, including the sender
io.emit("serverMessage", "Hello everyone!");
```

Client-side (index.html):

```
// Sending a custom event named "clientMessage" to the server
socket.emit("clientMessage", "Hello from the client!");
```

- On the server, we use socket.emit(...) or io.emit(...) to send custom events to clients.
- On the client, we use socket.emit(...) to send custom events to the server.

Broadcasting Events

Socket.io allows you to broadcast events to all connected clients or to all clients except the sender.

Server-side (server.js):

```
// Broadcast a custom event to all connected clients except the sender
socket.broadcast.emit("notification", "A new user has joined!");

// Broadcast a custom event to all clients in a specific room except the sender
socket.to("room1").emit("roomMessage", "A message for room 1!");
socket.broadcast.emit(...) sends an event to all clients except the sender.
socket.to("room").emit(...) sends an event to all clients in a specific room except the sender.
```

Client-side (index.html):

```
// Listen for a "notification" event from the server
socket.on("notification", (message) => {
    console.log(`Notification: ${message}`);
});
```

Clients can listen for broadcasted events from the server using socket.on(...).
In this example, clients will receive the "notification" event if a new user joins.

Storing with Mongoose

Install Mongoose and set up your MongoDB connection using ES6 modules.
Create a separate module for database connection.

```
import mongoose from 'mongoose';

const connectToDatabase = async () => {
    try {
        await
mongoose.connect('mongodb://localhost:27017/mydatabase', {
            useNewUrlParser: true,
            useUnifiedTopology: true,
        });
        console.log('Connected to MongoDB');
    } catch (error) {
        console.error('MongoDB connection error:', error);
    }
};

export default connectToDatabase;
```

Define a Schema and Model:

Define a Mongoose schema and model for the data you want to save.
This example uses a chat message schema.

```
import mongoose from 'mongoose';

const messageSchema = new mongoose.Schema({
    text: String,
    sender: String,
    timestamp: Date,
});

const Message = mongoose.model('Message', messageSchema);

export default Message;
```

Saving State to MongoDB

Create a function in a separate module to handle saving data to MongoDB.
Use the Mongoose model to save data.

```
import Message from './message.model.js'; // Import your Mongoose
model

export const saveMessageToMongoDB = async (text, sender) => {
    const newMessage = new Message({
        text,
        sender,
        timestamp: new Date(),
    });
```

```
    try {
        const savedMessage = await newMessage.save();
        console.log('Message saved:', savedMessage);
        return savedMessage;
    } catch (error) {
        console.error('Error saving message:', error);
        throw error;
    }
};
```

Retrieving State from MongoDB

Create a function that queries MongoDB to retrieve the desired data.
This example fetches all chat messages.

```
import Message from './message.model.js'; // Import your Mongoose
model

export const getAllMessagesFromMongoDB = async () => {
    try {
        const messages = await Message.find();
        console.log('All messages:', messages);
        return messages;
    } catch (error) {
        console.error('Error fetching messages:', error);
        throw error;
    }
};
```

Usage in Your Application

In your application code, import and use the functions to save and retrieve the state.

```
import connectToDatabase from './db.js';
import { saveMessageToMongoDB, getAllMessagesFromMongoDB } from
'./stateService.js';

// Connect to the database
connectToDatabase();

// Save a new message
saveMessageToMongoDB('Hello, world!', 'User1');

// Retrieve all messages
const messages = await getAllMessagesFromMongoDB();
```

Summarising it

- We have discussed what socket.io is, why it is needed, applications and working with it.
- We have made the connection to the backend and frontend with socket.io.
- We have seen how to run the application.
- We have discussed basic events, how they are emitted and how to listen to them.
- We have seen the integration of the mongodb database with our application.

Some Additional Resources:

- **More on Rooms**
- **Socket.io**