VIDEO CHAT APP - PeerJS

What is our GOAL for this CLASS?

In this class, we will learn about PeerJS, it's use cases and also, how it can be implemented to how clients can remain updated about their peers on the browser.

What did we ACHIEVE in the class TODAY?

- Learning about PeerJS
- Implementation of PeerJS and PeerJS server

Which CONCEPTS/ CODING BLOCKS did we cover today?

- PeerJS**:**

PeerJS is a very famous tool used commonly with WebRTC! WebRTC stands for ***Web Real Time Chat.*** It enables people to share data in realtime.

In Video Chat Applications, We want to stream the video to other users in Real Time, therefore **WebRTC** is used very commonly and heavily in Video Chat Applications!

Google Meet, a very famous tool for video meetings by Google, is based on ***WebRTC*** and ***PeerJS***.

What PeerJS helps with is, it provides Peer to Peer connections.

They offer 2 types of servers -

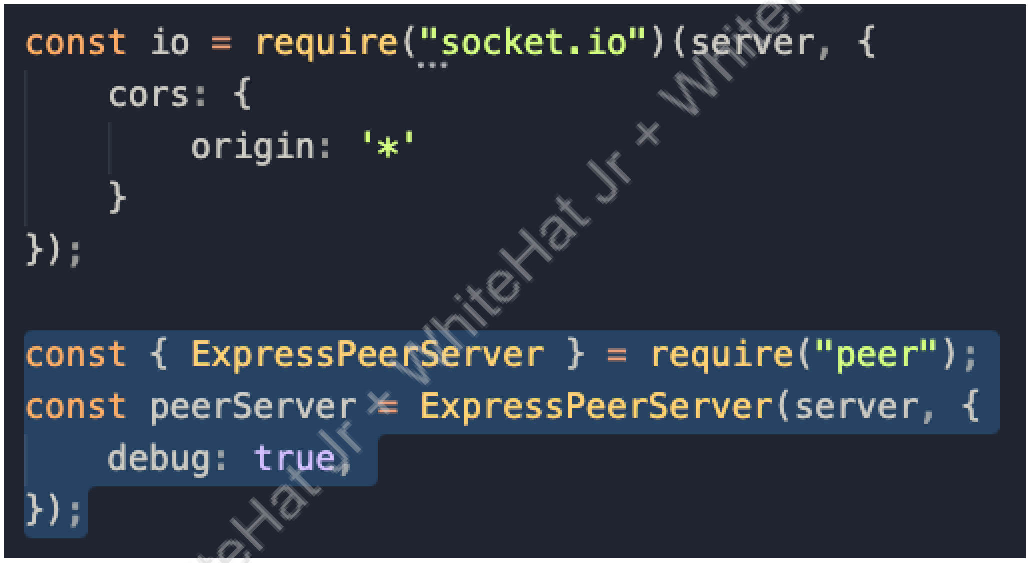
1. PeerServer Cloud - A cloud based PeerServer that can be used with PeerJS!
2. Local PeerServer - A server that you could create and run locally!

How did we DO the activities?

In the last class, we successfully implemented socket.io into our project, which enabled us to make the chat functionality work, but it wasn't as expected as different people from different chat rooms could chat with each other, instead of their chats being specific to the room. Today we will solve this issue.

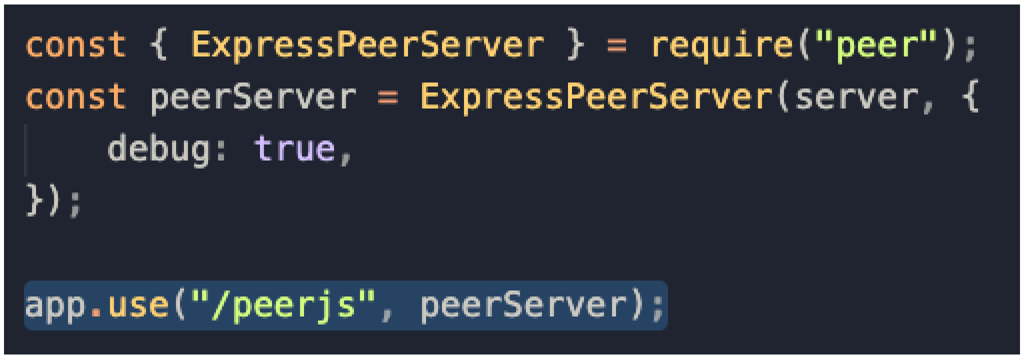
Activity:

1. Install the package **peer**.
2. Open ***server.js*** file and create a ***PeerServer***.



```
const io = require("socket.io")(server, {
    cors: {
        origin: '*'
    }
});

const { ExpressPeerServer } = require("peer");
const peerServer = ExpressPeerServer(server, {
    debug: true,
});
```

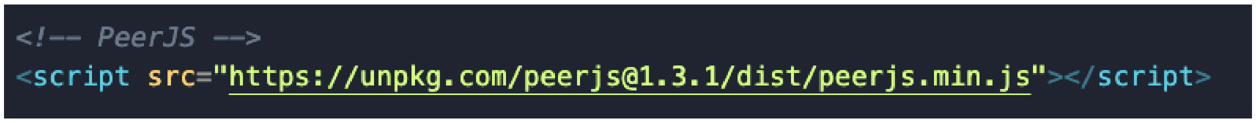
3. Now, let's make the App use the server.



```
const { ExpressPeerServer } = require("peer");
const peerServer = ExpressPeerServer(server, {
    debug: true,
});

app.use("/peerjs", peerServer);
```

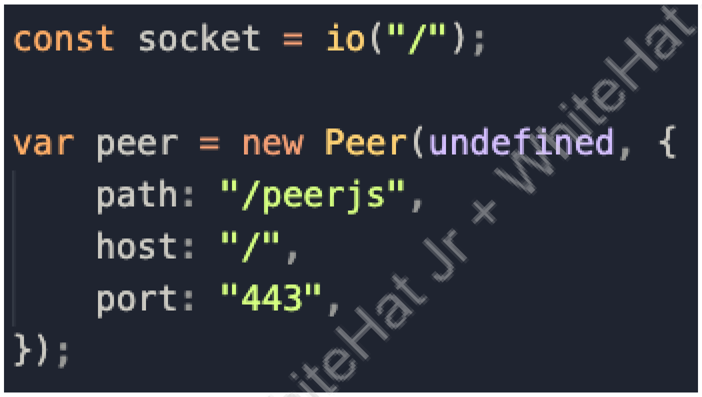
4. Let’s connect PeerJs to the client. For that:

    - First, we will add it’s script in our index.ejs in the head tag.



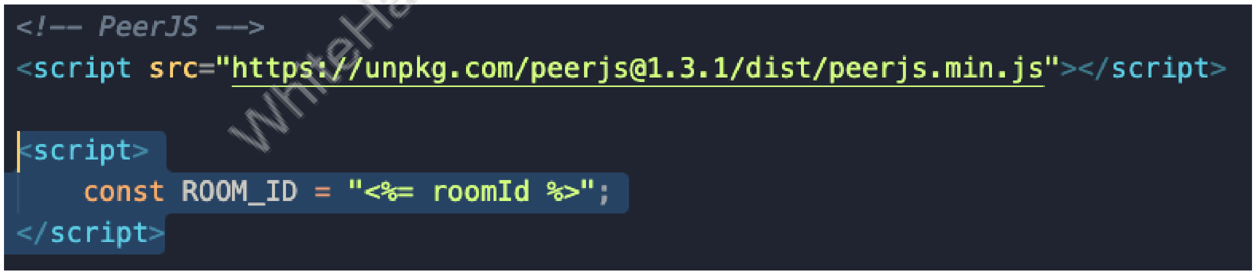
```
<!-- PeerJS -->
<script src="https://unpkg.com/peerjs@1.3.1/dist/peerjs.min.js"></script>
```

    - Second, we need to connect the client side to the PeerServer that we have just created.For that we write the instruction in script.js in the public folder.



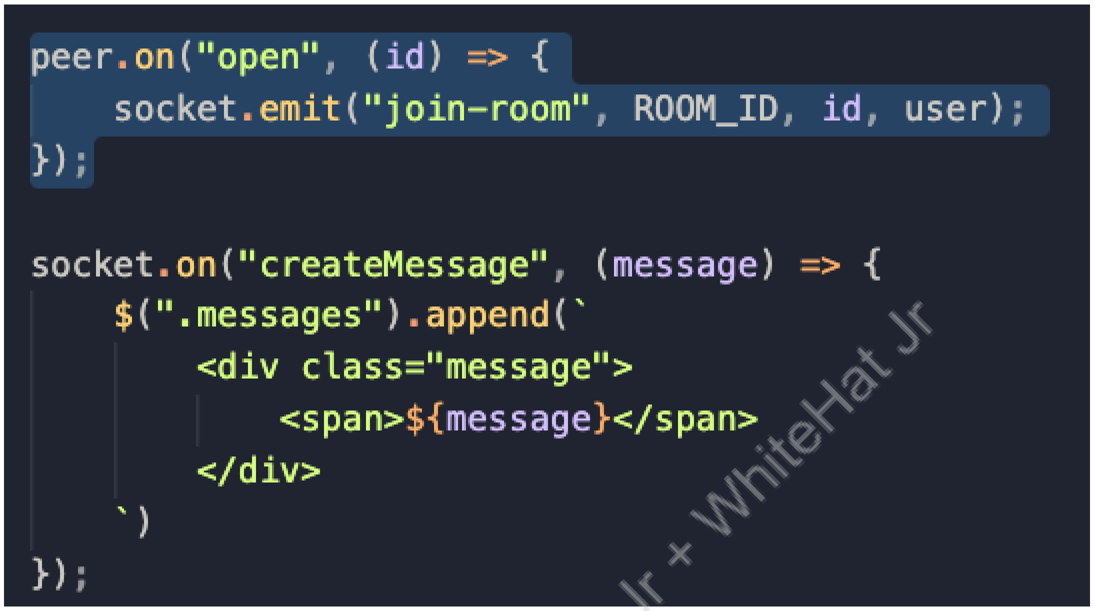
```
const socket = io("/");

var peer = new Peer(undefined, {
    path: "/peerjs",
    host: "/",
    port: "443",
});
```

5. Let’s save the “***room id***” in a variable in our client side. For that, we will add some code in our ***index.ejs*** in the ***head*** tag -



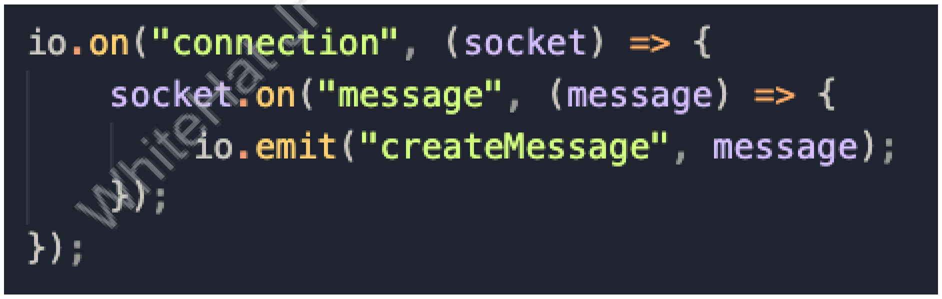
```
<!-- PeerJS -->
<script src="https://unpkg.com/peerjs@1.3.1/dist/peerjs.min.js"></script>

<script>
    const ROOM_ID = "<%= roomId %>";
</script>
```

6. We want to check if another peer has joined the room or not, and we can check it by seeing if any peer has ***opened*** the page.Let’s code this trigger -



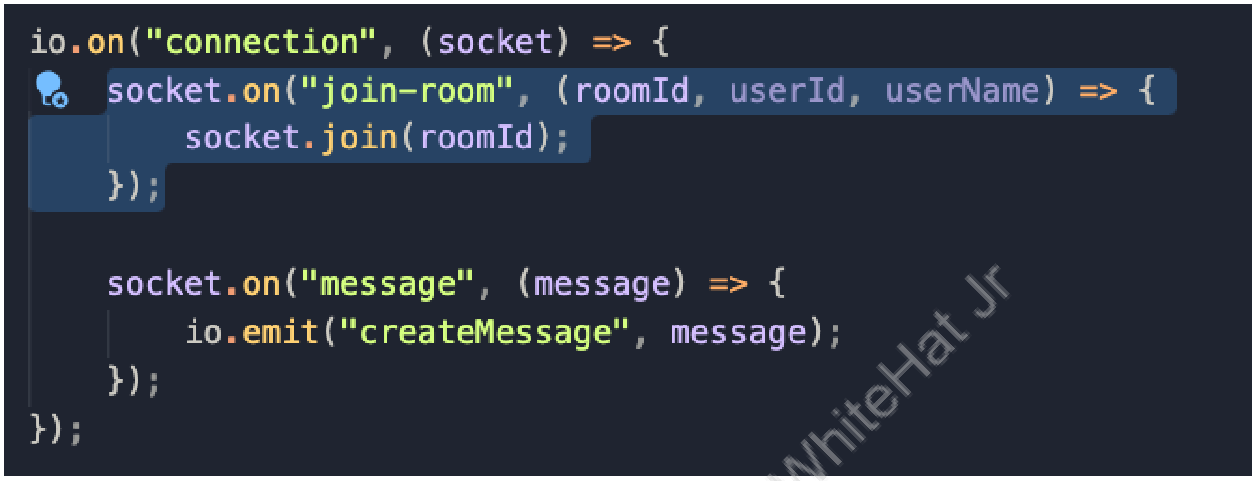
```
peer.on("open", (id) => {
    socket.emit("join-room", ROOM_ID, id, user);
});

socket.on("createMessage", (message) => {
    $(".messages").append(`
        <div class="message">
            <span>${message}</span>
        </div>
    `)
});
```

7. Now in our ***server.js*** file, we have created a ***"connection"*** event on the socket, in which we have created a socket event on ***"messages"*** which emits a ***createMessage*** event for the client.



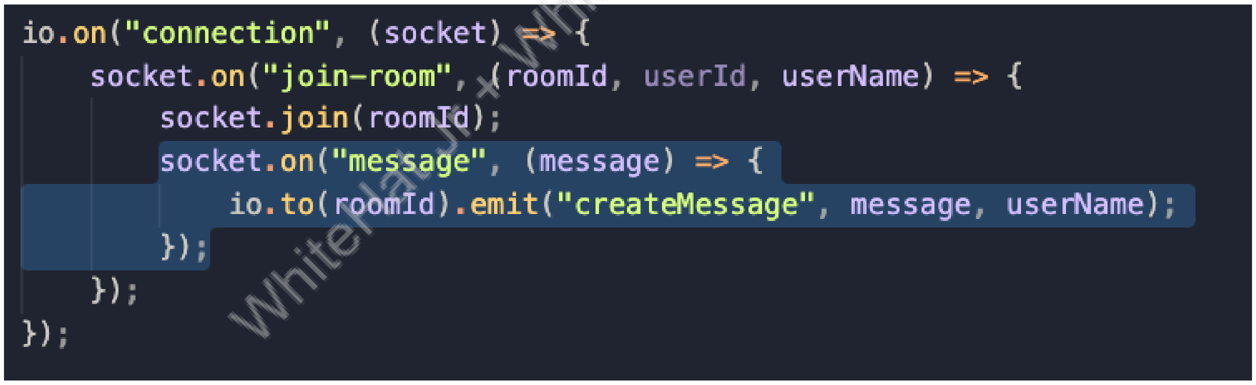
```
io.on("connection", (socket) => {
    socket.on("message", (message) => {
        io.emit("createMessage", message);
    });
});
```

8. As soon as we handle the socket event, we want to let the socket maintain the events based on the ***roomId,*** so we can separate out the events of different rooms. Let's write the code for that!



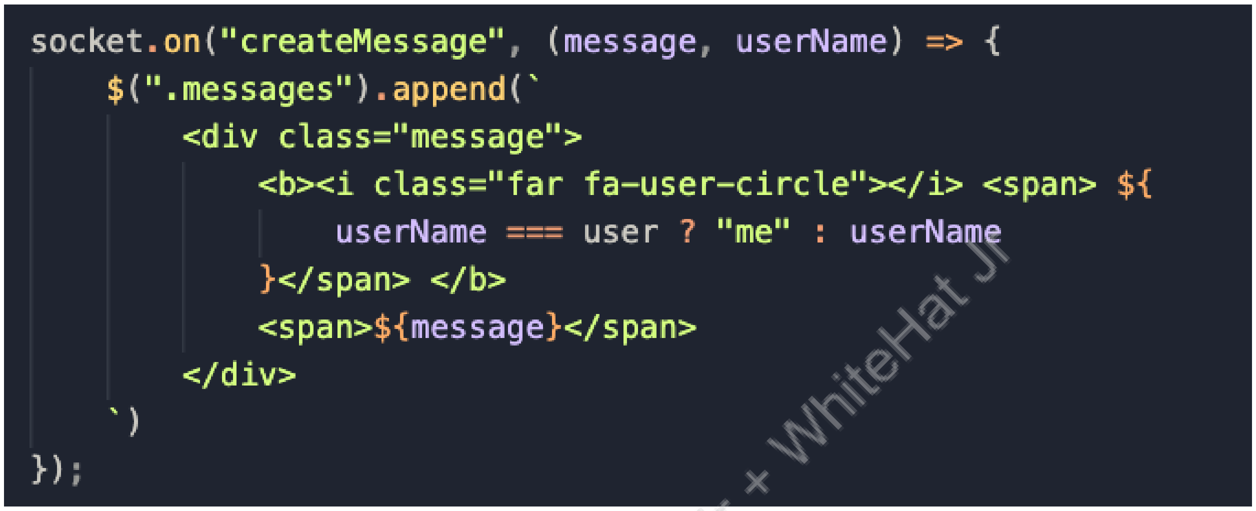
```
io.on("connection", (socket) => {
    socket.on("join-room", (roomId, userId, userName) => {
        socket.join(roomId);
    });

    socket.on("message", (message) => {
        io.emit("createMessage", message);
    });
});
```

9. Let's just move our ***message*** socket event inside the ***join-room*** socket event, so it knows that this event is to be triggered only after a peer has joined the room.



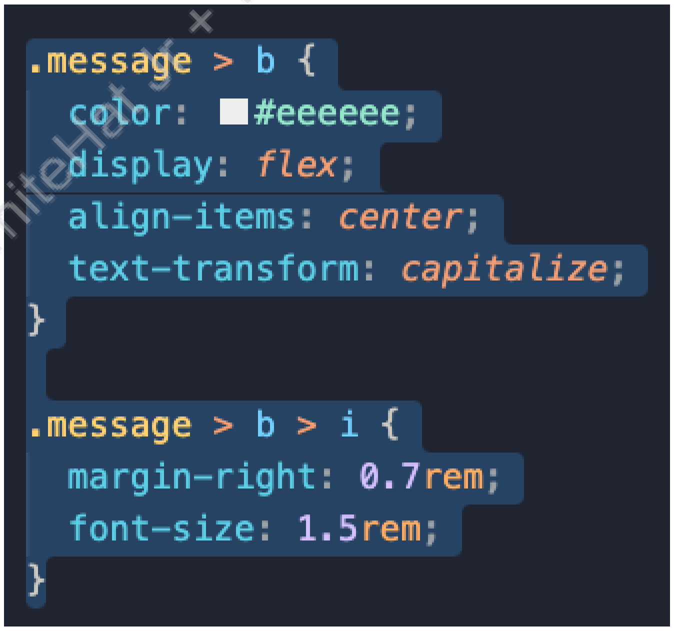
```
io.on("connection", (socket) => {
    socket.on("join-room", (roomId, userId, userName) => {
        socket.join(roomId);
        socket.on("message", (message) => {
            io.to(roomId).emit("createMessage", message, userName);
        });
    });
});
```

10. In the ***createMessage*** event at the client side, we will also send the ***userName*** to the client, so that it knows which client is sending this message.



```
socket.on("createMessage", (message, userName) => {
    $(".messages").append(`
        <div class="message">
            <b><i class="far fa-user-circle"></i> <span> ${
                userName === user ? "me" : userName
            }</span> </b>
            <span>${message}</span>
        </div>
    `)
});
```

11. Let's add styling for corresponding HTML.



```
.message > b {
  color: #eeeeee;
  display: flex;
  align-items: center;
  text-transform: capitalize;
}

.message > b > i {
  margin-right: 0.7rem;
  font-size: 1.5rem;
}
```

12. Since, PeerJS makes constant requests to the server, which the NGROK can't handle. Using NGROK to test the app is not possible.We will handle this in the next class.

What's NEXT?

In the next class, we will be deploying our video chat application online on a remote server.

Expand Your Knowledge:

Explore more about PeerJS click here.