Topic	Video Chat App - PeerJS	
Class Description	**Student 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!**	
Class	**C-217**	
Class time	**45 mins**	
Goal	● Learning about PeerJS ● Implementation of PeerJS and PeerJS server	
Resources Required	● Teacher Resources: ○ Laptop with internet connectivity ○ Earphones with mic ○ Notebook and pen ○ Visual Studio Code ● Student Resources: ○ Laptop with internet connectivity ○ Earphones with mic ○ Notebook and pen ○ Visual Studio Code	
Class structure	**Warm-Up** **Teacher - led Activity 1** **Student - led Activity 1** **Wrap-Up**	**10 mins** **15 mins** **15 mins** **5 mins**

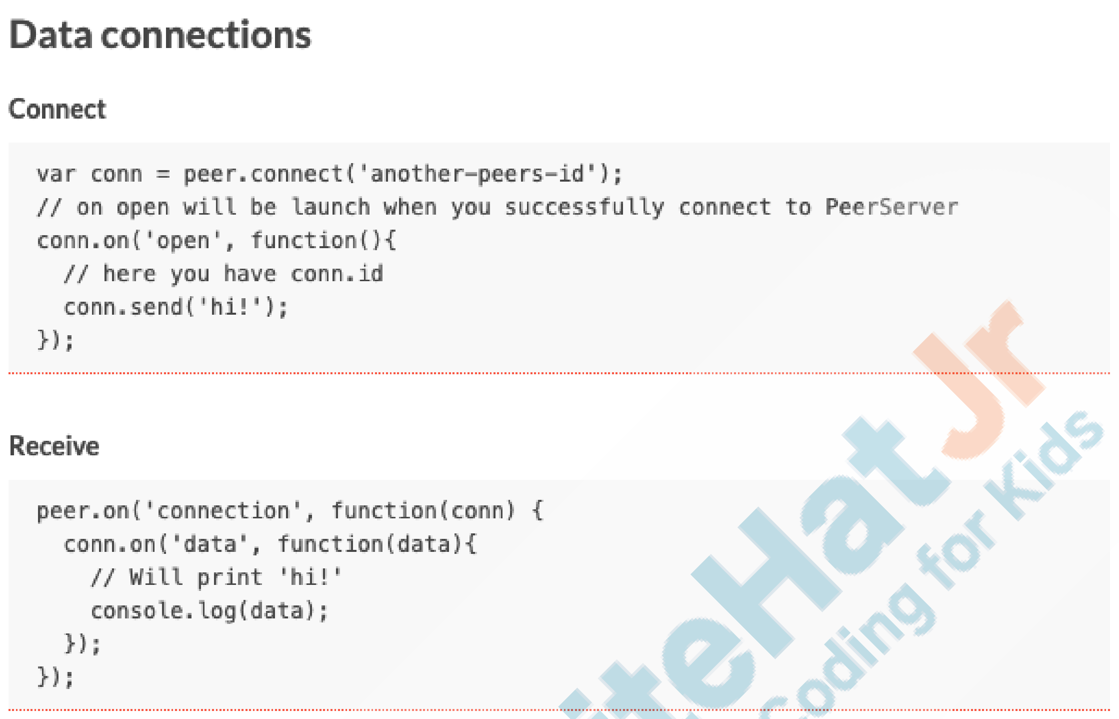
WARM UP SESSION - 10mins

Teacher Action	Student Action

Hey <student's name>. How are you? It's great to see you! Are you excited to learn something new today?	**ESR**: Hi, thanks, yes, I am excited about it!
Q&A Session	
Question	**Answer**
What are sockets? A. Sending only B. Receiving only C. Used for sending and receiving D. None of the above	C
.js represents what? A. Javascript B. Jscript C. Jquery D. None of the above	A
TEACHER-LED ACTIVITY - 15mins	
Teacher Initiates Screen Share	
<u>ACTIVITY</u> ● **Creating PeerServer locally** ● **Connecting with PeerServer**	
Teacher Action	**Student Action**

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. Do you have any doubts in the last class? *Teacher clears the doubts, if any* Great! Now to tackle this problem, we would be looking into a very nice JS library known as PeerJS. Are you excited? Let's get started then!	**ESR:** Varied **ESR:** Yes
In the beginning of this project, we have installed a library called PeerJS into our project. Do you know what a peer means? A peer usually refers to someone who is similar to you. They could be the same age, or doing something that you're also doing. Now, here in our Video Chat Application, if you are doing a video call, then your peers would be other people with whom you're doing the video call. Their clients would be peers to your client. Now 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.	**ESR:** Varied

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. In our use case, there would be certain events in which we would want to know what's going on in our Peer Clients. For example, as soon as they open up the room, our Video should stream to their client, and their video should stream to ours! It will also help us maintain what peers are connecting on which rooms, that we separated with unique URLs, and thus, we can separate the video streams and chats happening in different rooms too! Let's refer to ***PeerJS*** website once! *Teacher refers to Teacher Activity 2 for PeerJS documentation and website.* *Note - The previous class code is available in Teacher and Student Activity 1*	*Student refers to Student Activity 2 for PeerJS documentation and website*



Data connections

Connect

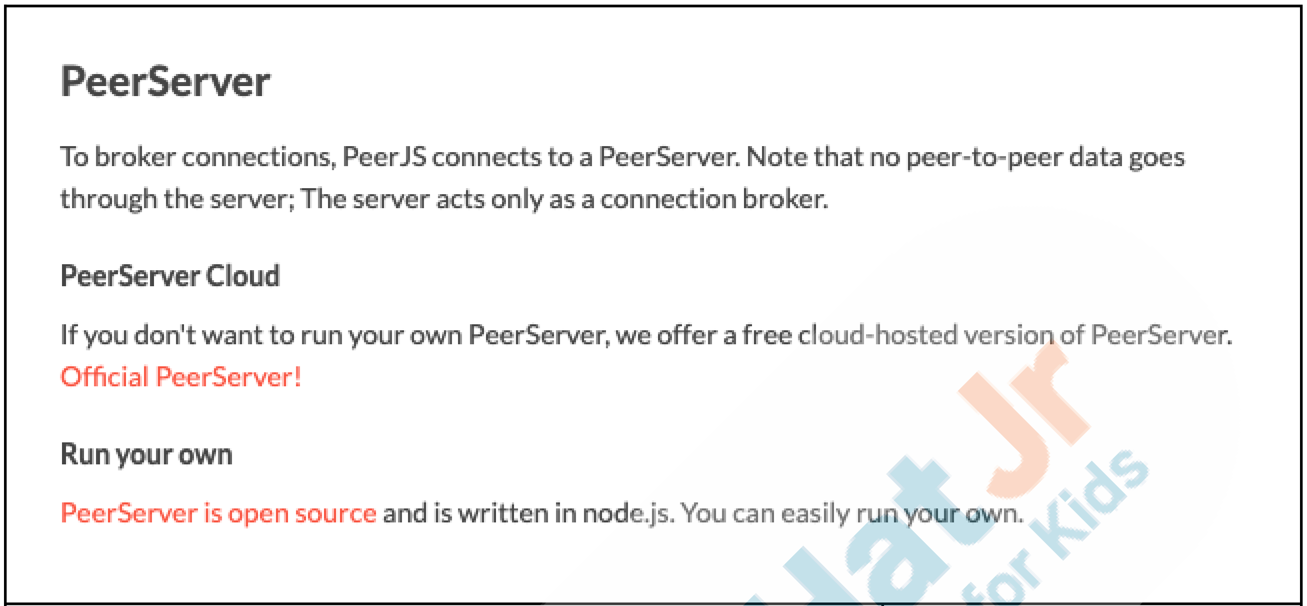
```
var conn = peer.connect('another-peers-id');
// on open will be launch when you successfully connect to PeerServer
conn.on('open', function(){
  // here you have conn.id
  conn.send('hi!');
});
```

Receive

```
peer.on('connection', function(conn) {
  conn.on('data', function(data){
    // Will print 'hi!'
    console.log(data);
  });
});
```

On scrolling down a bit, we can see that there is some code provided for Connecting with a fellow Peer, as well as to receive some data!

On scrolling down further, we can see a section called ***PeerServer -***



PeerServer

To broker connections, PeerJS connects to a PeerServer. Note that no peer-to-peer data goes through the server; The server acts only as a connection broker.

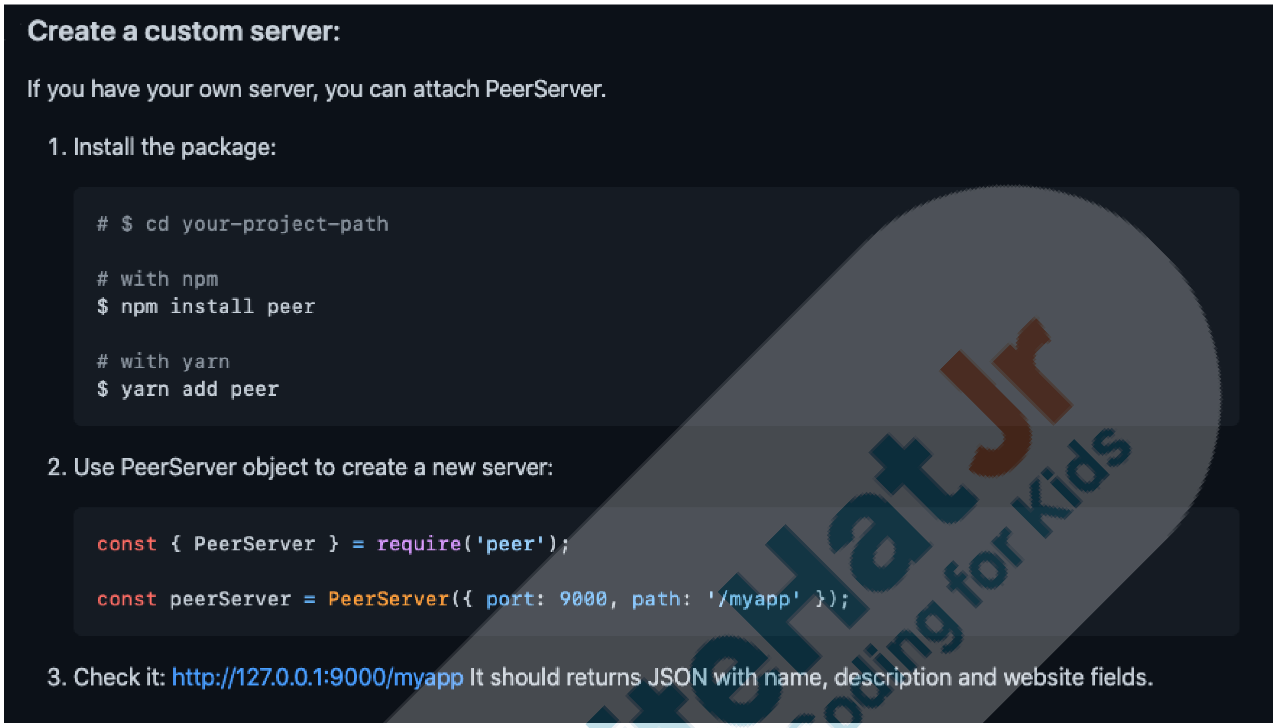
PeerServer Cloud

If you don't want to run your own PeerServer, we offer a free cloud-hosted version of PeerServer. Official PeerServer!

Run your own

PeerServer is open source and is written in node.js. You can easily run your own.

Now, as they have mentioned it here, PeerJS connects to a PeerServer. It is only to establish a connection between the 2 peers, so that their clients can be connected with each other and use PeerJS, but the data is not transferred through it. 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! Now for creating and running our own PeerServer, it says that we need ***node.js***, which is exactly what we are using! Let's click on it's link! *Teacher opens the link and the Github repo opens*	*Student clicks on the link and the Github repo opens*



Create a custom server:

If you have your own server, you can attach PeerServer.

1. Install the package:

```
# $ cd your-project-path

# with npm
$ npm install peer

# with yarn
$ yarn add peer
```

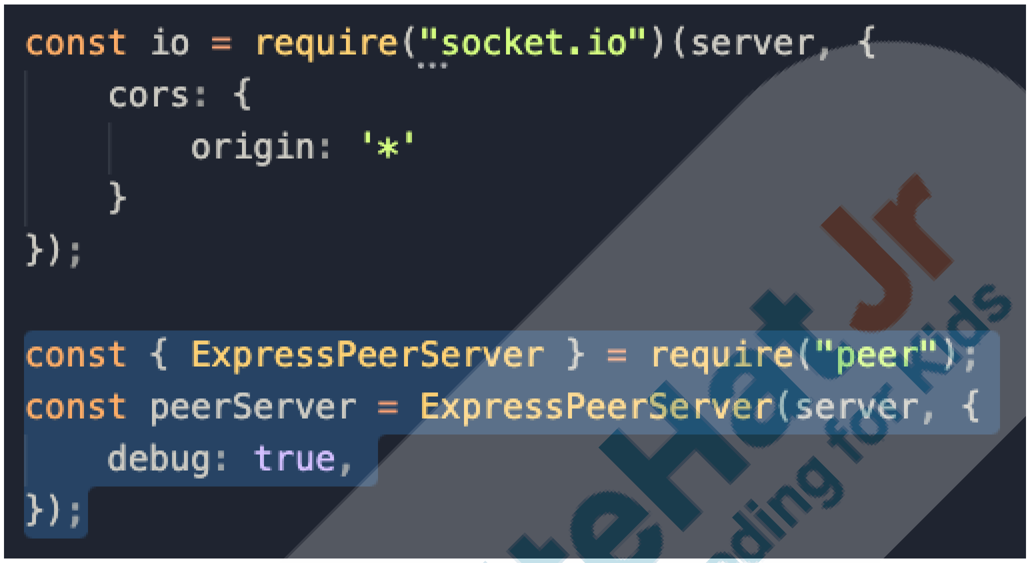
2. Use PeerServer object to create a new server:

```
const { PeerServer } = require('peer');

const peerServer = PeerServer({ port: 9000, path: '/myapp' });
```

3. Check it: http://127.0.0.1:9000/myapp It should returns JSON with name, description and website fields.

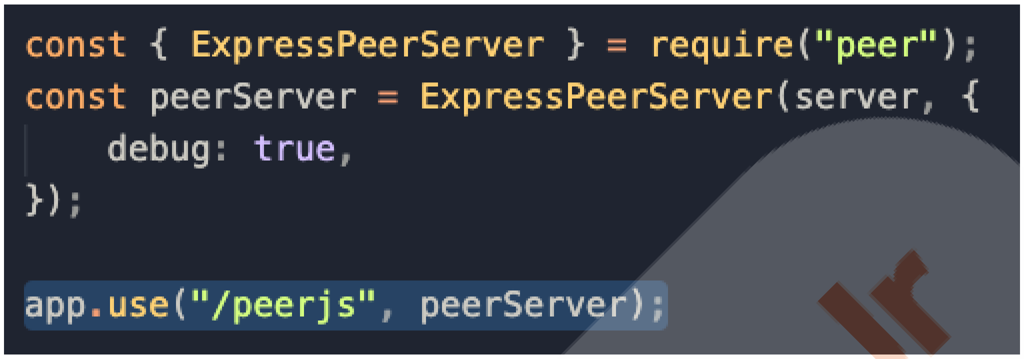
Okay, now here, if we scroll down a little and read the instructions provided in the github repository, we would find a section where there are instructions to create a custom server for PeerJS! It contains 3 steps - 1. To install the package from ***npm*** or ***yarn*** 2. To ***require*** it in our NodeJS app and create a ***PeerServer*** 3. To check if it works! We can skip the third step. For now, it is important to open our ***server.js*** file and create a ***PeerServer***. Let's do that! *Teacher opens the previous class' code in Visual Studio Code and make the changes to **server.js***	

Note - The previous class code is available in Teacher and Student Activity 1	


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

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

Here, as we can see, right after requiring socket.io and using it in our ***server.js***, we are requiring ***peer*** that we installed earlier and using it to create a ***PeerServer***. One thing that you will notice, is different than the instructions is that here, we are using the server that we created with ***ExpressJS*** similar to how we use it above with ***socket.io***, and also, we are setting the ***debug mode to true*** for our PeerServer, in case we want to debug errors. In the instructions on Github, we were defining a Port and a Route for our Peer Server, which was ***/myapp***, so what according to you will it take here? Since no port or route has been given to it, it will try to use the HTTPS port by default, which is what? Now we have created the server, but we need our App to use it as well.	**ESR:** Varied! **ESR:** 443 **ESR:** Varied

Let's do that -



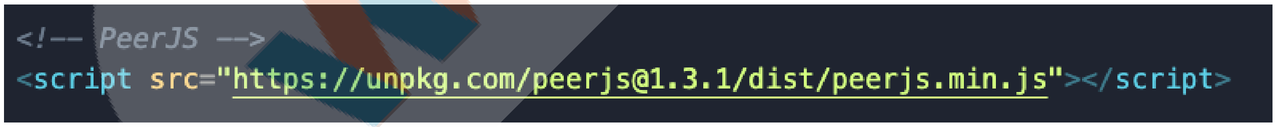
```
const { ExpressPeerServer } = require("peer");
const peerServer = ExpressPeerServer(server, {
    debug: true,
});

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

Here, we have defined the route it should use in our already created ***app***, called ***"/peerjs"***.

Now, we have the ***PeerJS*** set up on our Server Side! We also want to connect it to our client side too!

Let's do that! First, we will need to add it's script in our ***index.ejs*** in the ***head*** tag. Copy and paste the following there -

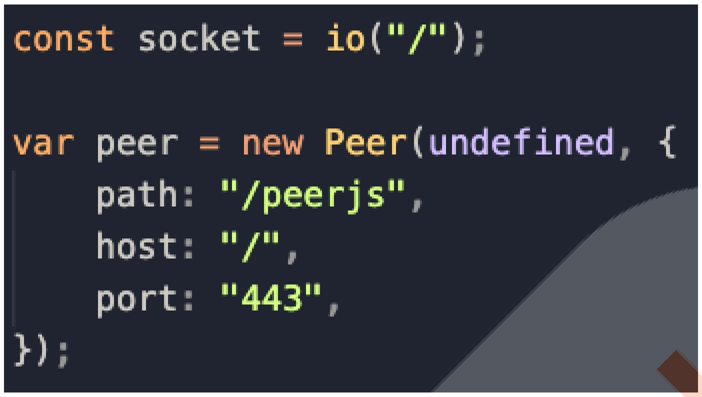
```
<script
src="https://unpkg.com/peerjs@1.3.1/dist/pee
rjs.min.js"></script>
```



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

Next, we need to connect the client side to the PeerServer that we have just created!

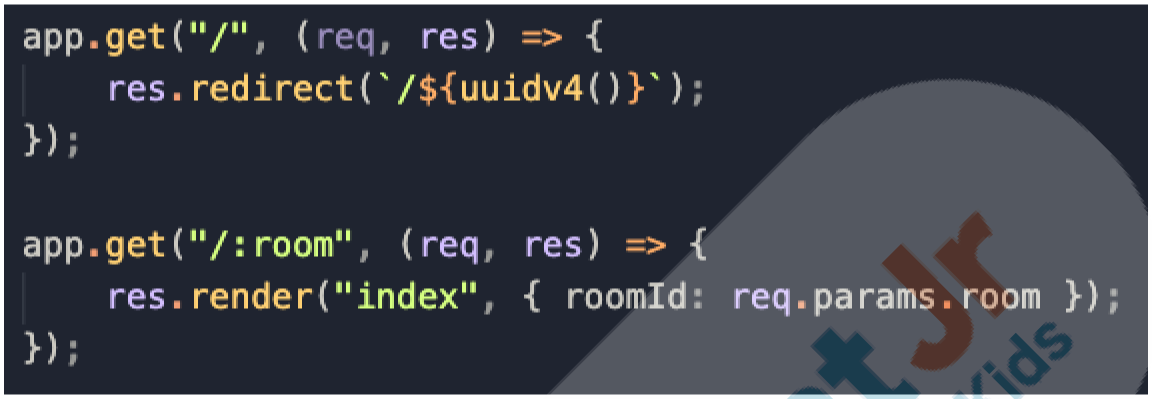
We will need to do that in the ***script.js*** file in the ***public*** folder.



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

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

Here, right after we created a socket with the ***io()*** function, we are creating a ***Peer*** with the ***Peer()*** function. The ***new*** keyword suggests that this is a new peer. Now, inside the ***Peer*** function, we have passed the first argument as ***undefined*** because we do not have a server running on the client side! We are passing some other arguments to our ***Peer()*** function to use while connecting, where we are giving it the ***path, host and port*** to use while making the connection with the ***PeerServer***. For ***path,*** we are telling it to use ***"/peerjs"*** which we just set in the backend. For the ***host***, we are telling it to use ***"/"***, so that it runs on our server. The ***port*** is ***443***, which it takes by default by creating the PeerServer. Now, what was the problem that we were facing by the end of the last class? That's right! Now, the first that we want to do it to ensure that the roomId that we are creating is getting used properly!	**ESR:** Different rooms were not having different chats!

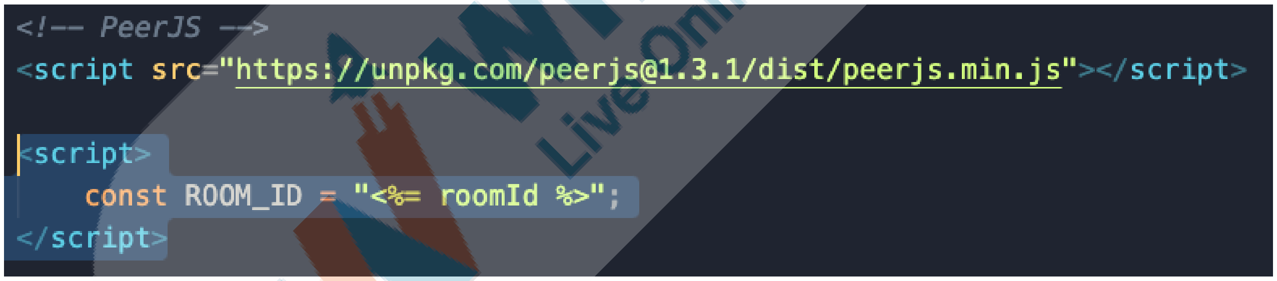
For that, let’s just take a look at our ***server.js*** and see the *“/”* route -



```
app.get("/", (req, res) => {
    res.redirect(`/${uuidv4()}`);
});

app.get("/:room", (req, res) => {
    res.render("index", { roomId: req.params.room });
});
```

Here, in these functions, we would notice that we already have passed a ***roomId*** variable when we are rendering the ***index.ejs*** file, but we are not yet using it on our client side!

Let’s start by saving it into a variable in our client side. For that, we will add some code in our ***index.ejs*** in the ***head*** tag -

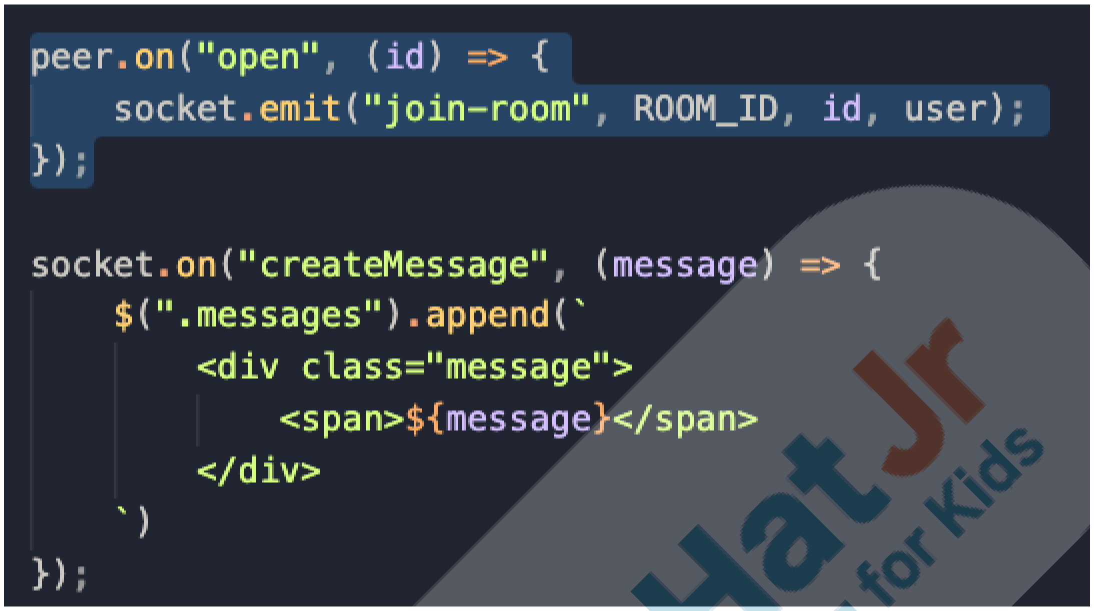


```
<!-- PeerJS -->
<script src="https://unpkg.com/peerjs@1.3.1/dist/peerjs.min.js"></script>

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

Here, we have added a small script. Inside this script, we are creating a variable called ***ROOM_ID***, and we are assigning the variable ***roomId*** that we were sending from the backend.

Now we know that we were using the ***ejs*** as our ***view engine***, and this is why the view engines are used. Here, with the syntax ***<%= variable %>*** we can use the variables from the backend in the frontend.

In our case, the variable is the ***roomId***. Now, we have our room's ID saved into a variable called ***ROOM_ID***. Now can you tell me what ***PeerJS*** does exactly? It helps connect peers, which are clients in our case, and also it can detect different events that the peer (or fellow clients) trigger! Now, PeerJS works based on the URL! This means that since our rooms have unique URLs, it would look for other peers (or clients) connecting to the same room and only get triggered for their events. Now, in a video chat application, we want to show the user's video and share it with others as soon as the page is opened, so what should be our first trigger here? That's right! 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 -	**ESR:** Varied! **ESR:** Opening the page!

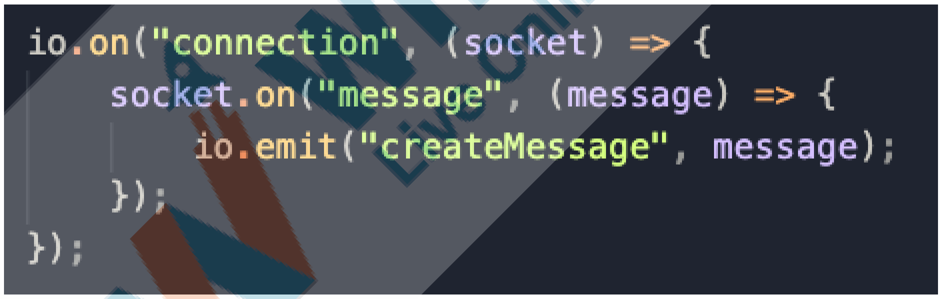


```
peer.on("open", (id) => {
    socket.emit("join-room", ROOM_ID, id, user);
});

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

In our ***script.js*** file, we are creating a trigger with the ***peer.on()*** function, just like how we have used ***socket.on()*** to handle socket events. Next, we are defining the type of trigger it is, which is ***open*** in our case. This means that we want to shoot the function inside it, as soon as a peer (or a client) opens the page. Now, ***PeerJS*** manages all the peers on a page by giving them a unique ID! For this, the arrow function that we pass into it will contain an argument called ***id***, which is the unique ID generated by the PeerJS of the peer that is connected! Inside this arrow function, we will ***emit()*** a socket message for joining the room. We will call this socket event ***join-room*** and we will pass 3 arguments to it -	**ESR:** Varied! **ESR:** Yes!

1. ***ROOM_ID*** - To know which room are we talking about 2. ***id*** - Which can act as the unique ID of our user and can be useful later 3. ***user*** - Which is the name of the user we took earlier from the prompt() function of JavaScript. Great! Now, we know 3 things - 1. Which room is joined 2. Who joined it 3. Name of the user who joined it Amazing, isn't it! With the help of this, let's ensure that different rooms maintain their separate chats, a problem that we faced in the last class!	
Teacher Stops Screen Share	
STUDENT-LED ACTIVITY - 15 mins	
• **Ask the student to press the ESC key to come back to the panel.** • **Guide the student to start Screen Share.** • **The teacher gets into Full Screen.**	
ACTIVITY • **Joining socket to room** • **Creating room based chat functionality**	

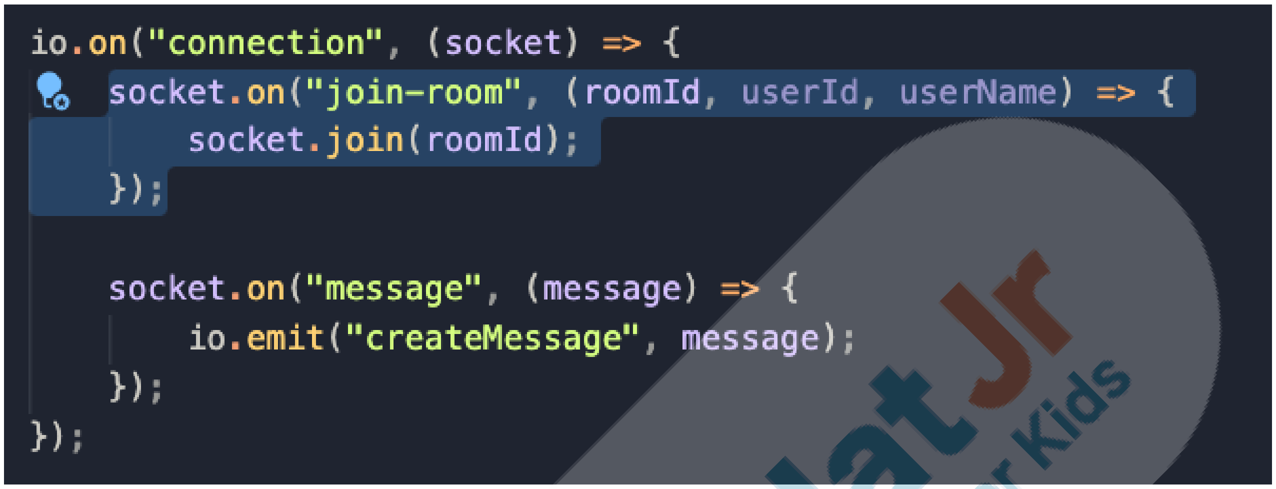
Teacher Action	Student Action
Guide the student to get the boilerplate code from [Student Activity 3]()	*Student clones the code from [Student Activity 3]()*
Okay, now let's start by installing the modules in your project with the following command - ***yarn install*** *Teacher helps the student in running the command*	*Student runs the command*
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. *Teacher helps the student in opening the* **server.js** *file and seeing the functions*	*Student sees the function*


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

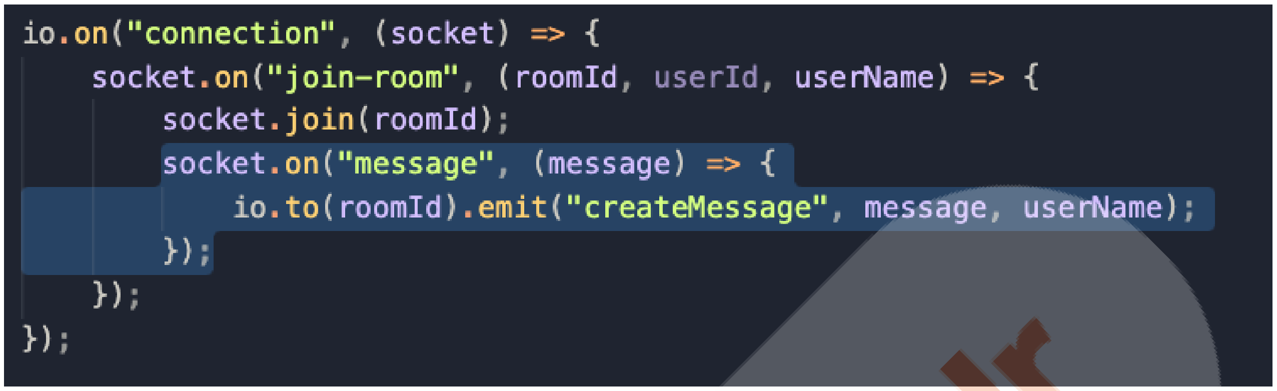
Here, we want to first handle the ***join-room*** socket event that we just created on the client side ***script.js*** As soon as we handle that socket event, we want to let the socket maintain the events based on the ***roomId,*** so we can separate out the events of different rooms! Let's write the code for that!	

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

Teacher guides the student in writing the code	Student writes the code

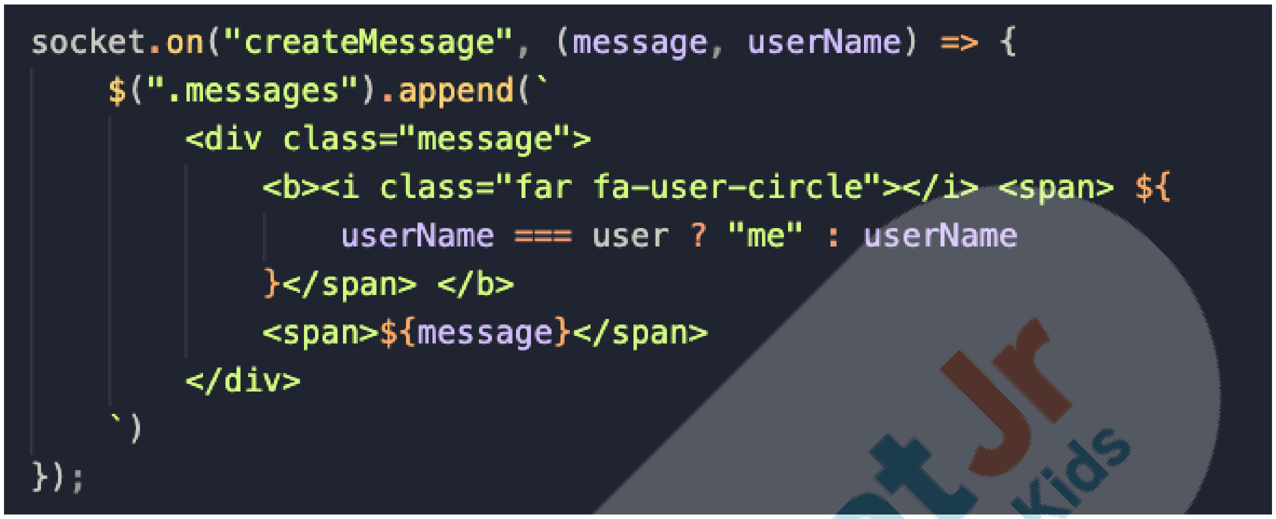

```
io.on("connection", (socket) => {
    socket.on("join-room", (roomId, userId, userName) => {
        socket.join(roomId);
    });

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

Here, we are handling the socket event ***join-room***, where we are passing the ***roomId, userId and userName***, just what we are receiving from the client side! Inside this socket event, we just do ***socket.join()*** and provide the ***roomId*** into it so that our socket knows which room we are referring to! So far, it looks all good! Now, 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! *Teacher guides the student in writing the code*	*Student writes the code*



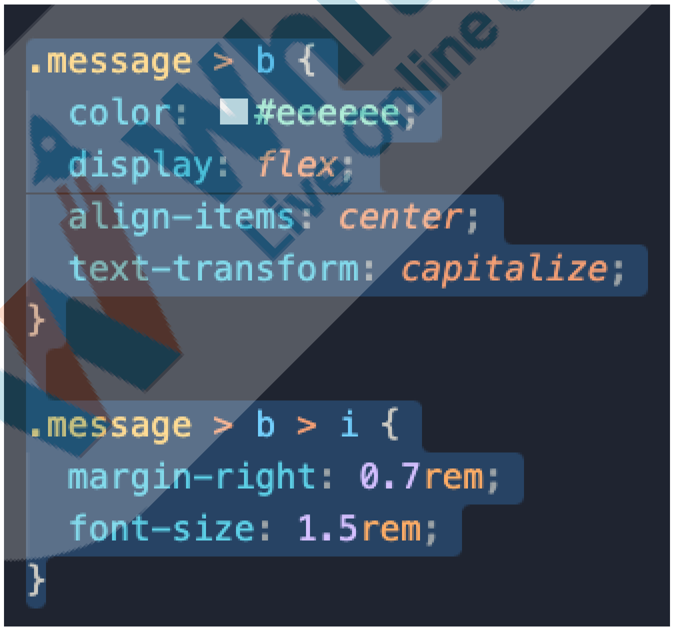
```
io.on("connection", (socket) => {
    socket.on("join-room", (roomId, userId, userName) => {
        socket.join(roomId);
        socket.on("message", (message) => {
            io.to(roomId).emit("createMessage", message, userName);
        });
    });
});
```

Here, if you notice, we have made some changes to the code as well. This time, instead of doing ***io.emit()***, we are doing ***io.to().emit()*** where the ***to()*** function takes the room ID. This will tell the socket which room it needs to emit this message to! Also, in the ***createMessage*** event, we are also sending the ***userName*** to the client, so that it knows which client is sending this message. We have this name from the ***join-room*** event inside which we are handling this event. Now, similar changes would be required in the ***createMessage*** event handler on the client side, to display the name of the user as well! Let's do that! *Teacher guides the student in writing the code*	*Student writes the code*



```
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>
    `)
});
```

Now, for the corresponding HTML that we have added, let's add styling too, so it looks good! *Teacher guides the student in writing the code*	*Student writes the code*



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

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

Awesome! So far, we've done pretty well! Now the only bit of our chat application's main feature remains is that, we	

want to stream the user's video to it's peer. In the last class, we tested our application through NGROK, but this time around, it's not possible because we just introduced PeerJS into our app. PeerJS keeps making constant requests to the server, which the NGROK can't handle. NGROK only allows limited requests that can be made to the server. Due to this, we will not be able to test the app with this method! To tackle this, in the next class, we will be deploying our video chat application online on a remote server!	
Teacher Guides Student to Stop Screen Share	
WRAP UP SESSION - 5 Mins	
Quiz time - Click on in-class quiz	
Question	**Answer**
What is WebRTC? A. Make a list of connections B. Peer to Peer connections C. Break Connections D. None of the above	**B**
What do you mean by peer.js? E. Real time Communication F. Real time Capabilities G. Real life communication H. None of the above	**A**

What are the steps to install peer.JS? A. Install peer B. Npm peer C. npm install peer D. None of the above	C
End the quiz panel	
FEEDBACK ● **Appreciate the students for their efforts in the class.** ● **Ask the student to make notes for the reflection journal along with the code they wrote in today's class.**	
Teacher Action	**Student Action**
You get Hats off for your excellent work! In the next class	*Make sure you have given at least 2 Hats Off during the class for:* Creatively Solved Activities +10 Great Question +10 Strong Concentration +10
Project Discussion	
Teacher Clicks ✖ End Class	

ADDITIONAL ACTIVITIES	
Additional Activities *Encourage the student to write reflection notes in their reflection journal using markdown.* Use these as guiding questions: ● What happened today? ○ Describe what happened. ○ The code I wrote. ● How did I feel after the class? ● What have I learned about programming and developing games? ● What aspects of the class helped me? What did I find difficult?	*The student uses the markdown editor to write her/his reflections in the reflection journal.*

ACTIVITY LINKS		
Activity Name	**Description**	**Link**
Teacher Activity 1	Previous Class Code	https://github.com/pro-whitehatjr/PRO-C216-ReferenceCode
Teacher Activity 2	PeerJS	https://peerjs.com/
Teacher Activity 3	Reference Code	https://github.com/pro-whitehatjr/PRO-C217-ReferenceCode
Student Activity 1	Previous Class Code	https://github.com/pro-whit

		ehatjr/PRO-C216-ReferenceCode
Student Activity 2	PeerJS	https://peerjs.com/
Student Activity 3	Boilerplate Code	https://github.com/pro-whitehatjr/PRO-C217-StudentBoilerplate