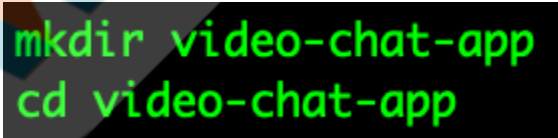


Topic	Video Chat App - NodeJS	
Class Description	Student will learn about a new technology called NodeJS, through which they can build servers and APIs	
Class	C-215	
Class time	45 mins	
Goal	<ul style="list-style-type: none"> <li>• Learning about NodeJS</li> <li>• Learning about ExpressJS</li> <li>• Creating a server to host the HTML files</li> </ul>	
Resources Required	<ul style="list-style-type: none"> <li>• Teacher Resources:               <ul style="list-style-type: none"> <li>○ Laptop with internet connectivity</li> <li>○ Earphones with mic</li> <li>○ Notebook and pen</li> <li>○ Visual Studio Code</li> </ul> </li> <li>• Student Resources:               <ul style="list-style-type: none"> <li>○ Laptop with internet connectivity</li> <li>○ Earphones with mic</li> <li>○ Notebook and pen</li> <li>○ Visual Studio Code</li> </ul> </li> </ul>	
Class structure	<b>Warm-Up</b> <b>Teacher - led Activity 1</b> <b>Student - led Activity 1</b> <b>Wrap-Up</b>	<b>10 mins</b> <b>15 mins</b> <b>15 mins</b> <b>5 mins</b>
<b>WARM UP SESSION - 10mins</b>		
<b>Teacher Action</b>		<b>Student Action</b>
<i>Hey &lt;student's name&gt;. How are you? It's great to see you!            Are you excited to learn something new today?</i>		<b>ESR:</b> Hi, thanks, yes, I am excited about it!

Q&A Session	
Question	Answer
TEACHER-LED ACTIVITY - 15mins	
Teacher Initiates Screen Share	
<u>ACTIVITY</u> <ul style="list-style-type: none"> <li>Understanding the HTML and Bootstrap Code</li> <li>Adding relevant HTML and CSS for responsiveness</li> </ul>	
Teacher Action	Student Action
<p>In the last class, we built a frontend for our Video Chat App and made it responsive by using Bootstrap, jQuery and Media Queries in CSS.</p> <p>In today's class, we will start building a backend server for our Video chat app!</p> <p>We already understand that data can be communicated from one client to the other through sockets that use a server. In our case, we can see that the HTML frontend that we have can be considered a client, and we need a server for it to work.</p> <p>Up until now, we have learnt a technology called Flask, which can be used to create servers and APIs, and it's based on Python, but it's not the only technology that can do so.</p>	

<p>All our experience until now has told us that JavaScript is an equally powerful technology that can be used to build games, mobile apps, websites, and whatnot!</p> <p>Just like how we used React Native to build mobile applications, JavaScript has another very powerful and famous technology called NodeJS, which can be used to create backend servers and APIs.</p> <p>Have you ever heard of it before?</p>	<p><b>ESR:</b> Varied!</p>
<p>Up until now, we have faced errors and issues where we have checked our NPM and Node versions in our terminal.</p> <p>Can you tell me what NPM stands for?</p>	<p><b>ESR:</b> Node Package Manager</p>
<p>Yes, and NPM is named so, because it was originally built for NodeJS, to manage all the packages that our backend server that we create with it is using.</p> <p>Again with NodeJS too, you can create an empty project template with the help of a command, just like how it's done in React Native.</p> <p>Let's start by creating an empty NodeJS project.</p> <p><i>Teacher opens command prompt / terminal</i></p>	
	
<p>Create a new directory called <b>video-chat-app</b> and navigate into this directory.</p> <p>Now, to create an empty project template for NodeJS, we can run the following command -</p>	

## npm init

On running the following command in the project directory, there are a couple of things that you will be asked to confirm for the project -

```
Press ^C at any time to quit.  
package name: (video-chat-app)  
version: (1.0.0)  
description:  
entry point: (index.js)  
test command:  
git repository:  
keywords:  
author:  
license: (ISC)
```

Simply press enter, for all of these options. We do not want to change or configure any of this while setting up the project.

About to write to /Users/apoorvelous/video-chat-code/video-chat-app/package.json:

```
{
  "name": "video-chat-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Is this OK? (yes) yes

It will then tell you how your **package.json** file would look like. Enter **yes** in the prompt and press enter.

It should now create the project's **package.json** file for you.

Next, we will use the following command to install a few libraries into this project -

```
yarn add express ejs socket.io uuid peer
```

This command will install 5 most important libraries that we need to build our project -

1. ExpressJS
2. EJS
3. Socket.io
4. UUID
5. Peer

Let's try to understand these libraries one by one -

**ExpressJS -**

It is the most famous, minimal and very flexible framework of NodeJS which provides you with all you need to create any modern and robust web application. In many ways, it is similar to Flask but works with JavaScript.

**EJS -**

It is an **E**MBEDDED **J**AVASCRIPT **T**EMPLATING system used by NodeJS. HTML is usually hardcoded, and cannot be programmed but EJS enables it to have some logic added into it. You may find the need to use loops or if-else conditions in HTML while building a page, or the use of a variable to display the user's name! EJS can enable you with all those logic buildings in HTML

**Socket.io -**

It is a JS library for socket programming, similar to the socket library we were using in Python.

**UUID -**

It is a JS library to create Unique IDs that can be used for multiple things. In our case, we can give a unique ID to different rooms on our video chat app, so that multiple video chats can happen at once.

**PeerJS -**

It is again, one of the most famous JS libraries that helps communicating one client with the other, without the use of a server. There are certain events like if another peer client is on the same page as you, etc. that we may need and this will help with just that, saving us to write a lot of code ourselves.

Don't worry about PeerJS just yet, we will get into it's details and see how it works with a lot of depth in the later

<p>sessions.</p> <p>For now, let's open our project in VS Code editor and create a new file called <b>server.js</b></p> <p><i>Teacher opens the project in VS Code and creates a new file called <b>server.js</b></i></p>	
	
<p>With this, our project setup is complete, and we can start building our application!</p>	
<p>Now the first thing that we want to do is to import our <b>express.js</b>, which is our framework that we will use to create our server in NodeJS.</p> <p>We will then create an <b>http</b> server using <b>ExpressJS</b>.</p> <p>Let's do that in our <b>server.js</b> -</p>	
	
<p>Here, we are first importing <b>express</b> with <b>require("express");</b> and saving it into a constant called <b>express</b></p>	

Unlike in React Native, the convention in which we import libraries in NodeJS is with the **require()** function, and save them into constants.

Next, we are creating our “server app” with the **express()** function that we just required in our project, and saving it into another constant called **app**.

Finally, we are requiring (or importing) another library called **http**, which is used to create HTTP servers, and creating a server with the **Server()** function with our **app** constant that we created with **express**, we are saving this http server that we just created in a constant called **server**.

Now, to understand how it works, let’s start by creating a simple GET API, displaying **Hello World** to see how we can create APIs.

To create the API, we will write the following code -

```
const express = require("express");
const app = express();
const server = require("http").Server(app);

app.get("/", (req, res) => {
  res.status(200).send("Hello World");
});

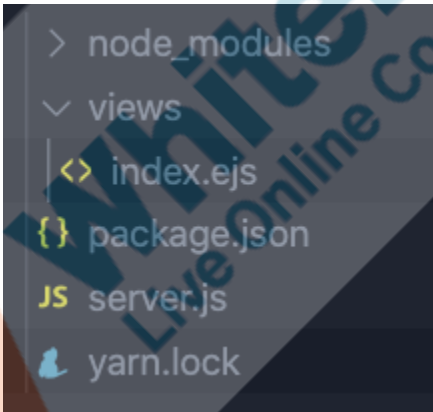
server.listen(3030);
```

Here, right after creating the server, we are defining a **get** request on the app with the **app.get()** notation, inside which, we are first writing a route on which it will work - “/” - and then creating an arrow function that contains 2 arguments - **req** and **res**.

These 2 arguments are mandatory for all the APIs of any type that you create in NodeJS.



<p><b>req</b> stands for request, and contains all the data that is sent in the request (mostly used in POST requests where you are saving some data into the API), and a <b>res</b> keyword, stands for Response, which is the response that this API will send back.</p> <p>Inside this arrow function, we have written -</p> <p><b>res.status(200).send("Hello World");</b></p> <p>This means that the response of this API would have a status 200, which is the default <b>Okay</b> response for all the <b>GET</b> apis, and we are sending <b>"Hello World"</b> in it.</p> <p>After this, finally we are saying that this server will listen on port 3030, with <b>server.listen(3030)</b></p> <p>On running the <b>npm start</b> command on the command prompt / terminal, and checking the browser at <b>localhost:3030</b>, we will see the following output -</p>	
<div data-bbox="633 1102 938 1171" data-label="Text"> <pre>npm start</pre> </div> <p>Gives the following output in the browser at <b>localhost:3030</b></p> <div data-bbox="665 1291 852 1354" data-label="Text"> <hr/> <p>Hello World</p> </div>	
<p>Awesome! We just created our very first API with NodeJS using ExpressJS!</p> <p>Now, currently we are just displaying some plain text on the browser, which is <b>"Hello World"</b>, but we want to display HTML instead, and we want to display it through <b>EJS</b> so we can add some logic to our HTML file as well!</p> <p>For this, we will have to set our <b>view engine</b> of the app to <b>ejs</b>, so that NodeJS knows that we are using EJS for rendering our HTML!</p>	

Let's do that -	
<pre>const express = require("express"); const app = express(); const server = require("http").Server(app); app.set("view engine", "ejs");</pre>	
<p>By adding the following line, we are setting our app's "<b>view engine</b>" to "<b>ejs</b>" with the <b>set()</b> function on the app.</p> <p>Now just like how we have <b>template</b> folder that <b>flask</b> used to find HTML files, <b>ejs</b> uses a folder called <b>views</b>, so let's create that folder, and inside that folder, we can create a file called "<b>index.ejs</b>"</p>	
	
<p>That's how our folder structure should look so far!</p> <p>Now, we already have the front-end (or the client side) of our app designed in the last class, so it's time that you add it here and render it instead of "<b>Hello World</b>".</p>	

Teacher Stops Screen Share	
STUDENT-LED ACTIVITY - 15 mins	
<ul style="list-style-type: none"> <li>Ask the student to press the ESC key to come back to the panel.</li> <li>Guide the student to start Screen Share.</li> <li>The teacher gets into Full Screen.</li> </ul>	
<p align="center"><b><u>ACTIVITY</u></b></p> <ul style="list-style-type: none"> <li>Integrate HTML code into NodeJS server</li> <li>Make all pages have a unique ID</li> </ul>	
Teacher Action	Student Action
<i>Guide the student to get the boilerplate code from <a href="#">Student Activity 2</a></i>	<i>Student clones the code from <a href="#">Student Activity 2</a></i>
<p>Okay, now let's start by installing the modules in your project with the following command -</p> <p><b>yarn install</b></p> <p><i>Teacher helps the student in running the command</i></p>	<p><i>Student runs the command</i></p>
<p>Okay, now the first thing that we would want to do is to copy our <b>index.html</b> code from last class into <b>index.ejs</b> file</p> <p><i>Teacher helps the student in copying and pasting the code</i></p> <p><i>Note - You can refer to Teacher / Student Activity 1 for previous class' code</i></p>	<p><i>Student copies and pastes the code</i></p>

```

<> index.ejs  x
213t > views > <> index.ejs > html
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="UTF-8" />
6    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7    <title>Video Chat App</title>
8    <link rel="stylesheet" href="style.css" />
9    <script src="https://kit.fontawesome.com/c939d0e917.js"></script>
10
11    <!-- Bootstrap -->
12    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap
13    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
14    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"></scr
15    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></script>
16  </head>
17
18  <body>
19    <div class="row" style="overflow: hidden;">
20      <div class="col-sm-12 col-md-12 col-lg-12 text-center p-3" style="background-color: #1d
21        <div class="header_back">
22          <i class="fas fa-angle-left"></i>
23        </div>
24        <h3 class='text-white'>Video Chat</h3>
25      </div>
26    </div>
27    <div class="row main">
28      <div class="col-sm-12 col-md-12 col-lg-9 left-window">
29        <div class="row">
30          <div class="col-sm-12 col-md-12 col-lg-12" style="height: 81vh; background-color:

```

Now, we also had a **style.css** and a **script.js** that we would want to use.

Just like how we had a folder called **static** in **flask** to host such files for us, we can create a folder called **public** for the same here, and have our **style.css** and **script.js** in this folder

*Teacher guides the student in creating a folder called **public** and move the **style.css** and **script.js** in this folder*

*Student creates the folder and moves the files*

```
> node_modules
  ✓ public
    JS script.js
    # style.css
  ✓ views
    <> index.ejs
  .gitignore
  {} package.json
  JS server.js
  yarn.lock
```

Great! Now we need to define in our NodeJS **server.js** file, that our public folder is hosting static data!

For that, let's add one line of code in our **server.js** file -

*Teacher guides the student in writing the code*

*Student writes the code*

```
const express = require("express");
const app = express();
const server = require("http").Server(app);
app.set("view engine", "ejs");
app.use(express.static("public"));
```

With the following line of code, we are letting our app know that it needs to **use** the **"public"** folder to host the **static** data with the **express.static()** function.

Now, you must have used Video Apps yourself. How do video apps differentiate between different rooms?

**ESR:**  
With unique URLs!

That's right! Now, we too, want to use unique URLs for creating different rooms, for which, we will now require **uuid** that we installed earlier into our **server.js**, with the following line of code.

*Teacher guides the student in writing the code*

*Student writes the code*

```
const express = require("express");
const app = express();
const server = require("http").Server(app);
app.set("view engine", "ejs");
app.use(express.static("public"));

const { v4: uuidv4 } = require("uuid");
```

Here, in the last line, we have required **"uuid"** and we are using its version 4.

Next, we want to make sure that when someone comes to the URL **"/"**, we are redirecting them to a unique URL, so our code in the **"/"** API would also change.

*Teacher guides the student in writing the code*

*Student writes the code*

```
const { v4: uuidv4 } = require("uuid");

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

This time, instead of sending **"Hello World"** back, we are redirecting the user with the **redirect()** function to a url that is unique! Remember how we can use backticks (``) in JS for string formatting? We are doing exactly that!

Our URL would now look like “**{random unique ID}**”

Now, we also want to display our “**index.ejs**” file on this URL, so that the user sees the client side frontend code.

Let’s create a new API for that!

*Teacher helps the student in writing the code*

*Student writes the code*

```
app.get("/", (req, res) => {
  res.redirect(`/${uidv4()}`);
});

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

This time, we created a new **GET API** with our app on URL “**/:room**”, where we know that the **room** here is the unique ID of the room.

Inside this API, we are using the **res.render()** function this time, to render our **index.ejs** file. We do not have to mention the extension of the file, as we have already defined earlier that we want to use EJS as our view engine!

While rendering it, do note that we are also sending the **room**, which is our unique room ID in a variable called **roomId** to the client. Since we are using EJS, this room ID variable would come in handy later, and we can fetch it from the **req** variable, which contains all the data coming in the request. Since our room’s unique ID comes from the URL as a parameter, we can see it can be fetched from the **params**.

Now let’s again start the server with **npm start** command in

command prompt / terminal to see the output on the browser at **localhost:3030**

*Teacher guides the student in running the command and checking the output on the browser at **localhost:3030***

*Student runs the command and checks the output*






And the URL -

 **localhost:3030/cebc1167-9716-4553-9912-fa546297532f**

We can see that on opening **localhost:3030** on the browser, the server automatically added a unique ID to the URL, which means it redirects as we expected, and opens the HTML page (or the client side) for us.

This means that our server now works perfectly! In the next class, we will first implement the text chat functionality of our video chat app, something that we have done in the past, but this time, with JavaScript!



Teacher Guides Student to Stop Screen Share	
WRAP UP SESSION - 5 Mins	
Quiz time - Click on in-class quiz	
Question	Answer
End the quiz panel	
<b>FEEDBACK</b> <ul style="list-style-type: none"> <li>• Appreciate the students for their efforts in the class.</li> <li>• Ask the student to make notes for the reflection journal along with the code they wrote in today's class.</li> </ul>	
Teacher Action	Student Action
<p>You get Hats off for your excellent work!</p> <p>In the next class</p>	<p><i>Make sure you have given at least 2 Hats Off during the class for:</i></p> <div>           Creatively Solved Activities  +10         </div> <div>           Great Question  +10         </div> <div>           Strong Concentration  +10         </div>
<b>Project Discussion</b>	

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	<a href="https://github.com/pro-whitehatjr/PRO-C214-ReferenceCode">https://github.com/pro-whitehatjr/PRO-C214-ReferenceCode</a>
Teacher Activity 2	Reference Code	<a href="https://github.com/pro-whitehatjr/PRO-C215-ReferenceCode">https://github.com/pro-whitehatjr/PRO-C215-ReferenceCode</a>
Student Activity 1	Previous Class Code	<a href="https://github.com/pro-whitehatjr/PRO-C214-ReferenceCode">https://github.com/pro-whitehatjr/PRO-C214-ReferenceCode</a>

© 2021 - WhiteHat Education Technology Private Limited.

Note: This document is the original copyright of WhiteHat Education Technology Private Limited.

Please don't share, download or copy this file without permission.

		<a href="#">ehatjr/PRO-C214-ReferenceCode</a>
Student Activity 2	Boilerplate Code	<a href="https://github.com/pro-whitehatjr/PRO-C215-StudentBoilerplate">https://github.com/pro-whitehatjr/PRO-C215-StudentBoilerplate</a>

