NAMASTE NODE.JS SEASON 3

Full Notes-episode[1-9]

Hand Written Notes

-By Shanmuga Priya

www.linkedin.com/in/shanmuga-priya-e-tech2

Season: 3

Episode: 1

Launching AWS Instance and deploying frontend

1) what is AWS?

Amazon web Service (AWS) is a cloud computing platform that offer a variety of services over the internet.

2) what is cloud computing platform?

→ A cloud computing platform is a network of servers that provide variety of services such as servers, storage, databases, networking & so on to users over the internet.

→ Users can rent access to these services on demand, paying only for what they use.

Steps involved in Setting up AWS for deployment

Step1: Sign up in AWS

Step2: Search for EC2 in the search bar provided in Console Home Page.

what is EC2?

EC2 is called Elastic Compute Cloud is a web service that allow users to create and run virtual machines, called instances in the cloud.

Step 3: click on Launch instance and add a name to that instance and select the OS which we need to run on that instance from the given list. (Ubuntu Preferable) and select the instance type.

Step 4: Create a new Key pair (.pem) (.ppk)

→ It is like a secret key which is used to connect to the instance securely.

→ Enter a Key Pair name of your choice and select the RSA algorithm and Pem file format. and click on create Key pair.

→ It will generate a file with the secret key & download automatically.

Step 5: click on Launch instance it will create a new instance for us with all the configuration that we setup in the previous step.

→ the instance will be in pending state later it will turn into running state.

Step 6: click on the instance ID it will give all the info about the virtual machine that we have rented like Ip address. Click on connect it will give us a various way of connecting to that instance.

Step 7: Select SSH client from it. It is used to connect to the instance using terminal. In terminal follow the steps provided to login to the instance. Now, we have successfully logged in to our instance. To logout from the instance type exit.

Step 8: Install a node in the new instance that we created. make sure to use the same version which is used in the project to avoid misfunctio

How to connect back to instance once logged out?

TO connect back to the instance we need to run the same SSH -i command which we have done in step 7. It basically contains a SSH -i + secret file + instance name.

(connecting method)

How to run our project in the new instance that we created?

Step 1: get a https link and place it in the terminal with the command

git clone

"git clone your projects https link from code section."

It will clone our project in the new instance that we created.

Steps involved in ~~setting up~~ deploying a project.

Once the instance is setup we will deploy our project. Let's see how to deploy our frontend part.

→ Step 1: Building a project.

before deploying a project we need to bundle it up to a single 'dist' folder which contains the entire project code & packages needed for our project. This can be done by running a npm run build in project's terminal.

→ Step 2: Build the project in our new instance.

before building a project we run npm install to download all the packages that our project needs. then we run npm run build to build our project.

→ Step 3. Download nginx

what is nginx?

→ Nginx is a opensource software that can be used as web server, load balancer, reverse proxy & so on.

→ It is widely used for serving static web content, handling high traffic and distributing requests to backend servers.

→ we use Nginx to host or manage our app on AWS instance.

sudo apt update → to update ubuntu version.
sudo apt install nginx → to install nginx
sudo systemctl start nginx → to start nginx
sudo systemctl enable nginx → to enable nginx

→ step 4: copy the code from dist folder to nginx http server.

to copy a dist folder to http server we run a command

sudo scp -r dist/* /var/www/html/

↓ ↓ ↓ ↓

copy recursively everything from dist → http server.

→ This will run our app on port "80" we need to connect this port to our instance's Public Ip address to see our app live.

Step 5: Connecting nginx & our instance.

→ to enable port 80 on our instance click on security tab and go to security group and add a inbound rules to include port 80.

Step 6: application Live.

Now we go to Public Ip address our app will be live.

———— x ————

Episode - 2

Backend APP Deployment and connecting Frontend & Backend

Steps involved in deploying Backend APP:

we have already created an instance and cloned our backend project in previous episode now we will deploy it.

Step 1: Move to Backend folder & install necessary dependencies

```
npm install
```

Step 2: Copy Ipaddress to MongoDB

Inorder to run a project in instance we use npm start. Before that we need to allow access of our DB to the instance Ip Address. Save the local Ip address of instance in the DB atlas. Now the DB connection is successfull

Step 3: Enable the port

to enable the port go to security then securitygroups there we set a new inbound rules to include the port NO 7777.

Issue with current implementation:

whenever the terminal is closed the ~~server~~ app will also be shutdown but we cannot keep the terminal open all the time we need to run this application in the background all the time for that we use a package called "PM2".

what is PM2?

→ PM2 is a process manager that will help us manage and keep our application online 24/7. It is an open source Process manager for Node.js application

```
npm install pm2 -g.
```

→

Step4: Start our app in Pm2.

Now we start our app in the Pm2 process manager to run our application 24/7 using

Pm2 start npm -- start

This command will run npm start in the background by creating a new process that keeps on running 24/7 in the background.

→ This make our app running even after the terminal is closed or logged out from the instance.

Pm2 logs → it gives logs of all running process.

Pm2 flush name of the process/ app → to clear the logs.

Pm2 list → list of processess started by Pm2.

Pm2 stop processname → to stop the process.

Pm2 delete Processname → to delete the process.

we can also give a custom name to the process before starting

Pm2 start npm --name "devtinder" -- start.

Connecting frontend & backend

Step1: Configure nginx.

we need to configure nginx to map /api to port num 7777 inorder to ease access for users this way user need not deal with ports in URL's.

without nginx:

frontend: http://ex.com

Backend: http://ex.com:7777/

with nginx:

http://ex.com/api (both frontend & backend)

Behind the scence when the browser sends a req to http://ex.com/api nginx receives this & forward to http://localhost:7777/api.

frontend → nginx → backend

req ⟶ ⟶

Step a: Open the nginx configuration file

sudo nano /etc/nginx/sites-available/default

Step b: edit the nginx file.

1) Change the server name from _ to domainname (or) localIPaddress.

server_name Domainname/Ipaddress

2) add the proxy-pass lines below it.

```
location /api/ {
    Proxy_pass http://localhost:7777/;
    Proxy_http_version 1.1;
    Proxy_set_header Upgrade $http_upgrade;
    Proxy_set_header Connection 'upgrade';
    Proxy_set_header Host $host;
    Proxy_cache_bypass $http_upgrade;
}
```

It says whenever you see a ~~location~~ URL /api just redirect it to localhost:7

Step c: Save the file.

Ctrl + C → to exit it will ask to save it type Y. and enter.

Step 2: Restart nginx.

after the nginx file updated we need to restart the nginx

sudo systemctl restart nginx

Step 3: Replace the backend URL in the frontend file with the "/api".

const BASEURL = "http://localhost:7777/" → const BASEURL="/api".

and push it to github.

Step 4: Make a pull req in the instance terminal to get updated with latest commit.

git pull.

Step 5: Build the frontend once again.

As we updated the code in frontend inorder to reflect the changes made we need to bundle it again and copy the dist folder to nginx. like we did when deploying frontend.

```
npm run build

sudo scp -r dist/* /var/www/html.
```

Now, both frontend & backend are connected successfully.

×

Episode-3

Adding a custom Domain Name

-> The website which we will be using to purchase Domain name is from GODaddy..

Step1: Search for the Domain name.

Create an account in godaddy and search for a domain name that you want to setup for your project. It will give you a list of names similar to the one you provided. choose the one among them & finish the Payment Process.

Step2: Map our Instance Public Ip address to DNS Management in godaddy.

In Dashboard, go to my products / all products select the Domain name you have purchased click on DNS and create a DNS Record & Point our Ip.

(or)

In cloudflare.

Step 1: Signup on cloudflare.

Create an account cloudflare. we will be using cloudflare to manage the DNS Records instead of godaddy. becoz it provide free SSL certificate

What is cloudflare?

cloudflare is a service that provide a combination of:

*) Content Delivery Network (CDN): speeds up a website by caching content at servers.

*) Security: protects site from attacks

*) DNS Management: Centralized control of DNS records.

*) Free SSL: provides free SSL certificate to secure website.

what is SSL certificate?

→ An SSL (Secure Sockets Layer) certificate ensures that data transferred between a user and a website is encrypted and secure.

→ It is critical for securing websites, especially those handling sensitive information like password (or) credit card details.

→ SSL also enables HTTPS, which is more secure.

Step: 2 Add a domain name.

Add the domain name that purchased from crodaddy in cloudflare after creating an account and click on continue and select a free plan and click continue it will redirect to a page to review DNS records. click on continue to activation.

Step 3: Adding a nameserver from cloudflare to crodaddy.

→ After the 2nd step it will gives us 2 nameservers we need to add those nameservers to godaddy.

→ In godaddy DNS management → you will see a tab Nameservers, click on change Nameservers and select an option that "I'll use my own nameservers" and add the cloudflare nameservers.

→ Once its done cloudflare will take some time to update Nameservers and get the DNS record from the go daddy.

What is Nameserver?

Nameserver is a place where your DNS records is hosted. When someone tries to access a domain, the browser contacts the name server to retrive the DNS record and find the correct server to direct the req.

Why to add cloudflare's nameserver to crodaddy?

Since we are using cloudflare to manage DNS records instead of godaddy we are replacing the godaddy's nameservers with the cloudflare nameserver.

What is DNS Records?

DNS records are the configuration in the DNS system that tell how to handle your domain. There are various types of DNS records:

A record: Maps a domain name to an Ip address

CNAME record: Maps one domain to another (eg: www.ex.com → ex.com)

Step 4: Add Our Instance Ip address to DNS Records.

In sidebar click on DNS → Records → edit the A record with the Instance Ip address. There will be a 2 A record by default which points to random Ip address. Delete anyone of them and edit the other one with the Instance Ip address.

Step 5: Enabling SSL certificate:

→ In sidebar click on SSL/TLS → click on configure and select custom SSL/TLS in that to flexible and save it.

→ Click on edge certificate in sidebar under SSL/TLS → enable "Automatic HTTPS rewrites". It changes http to https.

→ Now we have successfully created a custom domain name and protected it with SSL certificate.

What is DNS?

DNS (Domain Name System) is like Phonebook of the internet. It translates human-readable domain names to its Ip address that computer use to communicate.

What is DNS Registrar?

It is a place where we purchase a domain name from. eg: godaddy

———— × ————

Episode-4

Keeping our Credentials safe using dotenv files

→ we need to keep our project sensitive data like Passwords, API key, DB connection string from others accessing it to do that we use a package called dotenv. "npm install dotenv."

Step:1 Create a .env file.

On root level create a .env file and we will place all the secrets in this file. we should never push this file to github. to do this we include it in .gitignore file so that it will automatically be not included.

Step2: Creating a secrets and accessing it inside code:

Inside a .env file we just create a var without var, let, const Keywords and it is preferred to be capital letters and asign it with value. eg: .env file.

```
DB_STR = "your DB connection str"
```

this is called environmental variable.

→ we can have all the secrets in a single file.

How to use this inside code?

we can access the environmental variable stored inside a .env file using Process.env.name of the environmental variable.

eg:

```
Const ConnectDB = async() => {
    await mongoose.connect(process.env.DB_STR)
}
```

How to write a comments in .env file?

we can write a comment in .env file using #

Step3: Configure / connect dotenv & app.

in app.js on top level of the code we need to config this dotenv Package we installed.

eg: app.js

```
require("dotenv").config()
```

Step 4: Deploy it in instance and add our .env file to instance

Step a: git pull

go to the backend project in the instance and do a git pull to get the lastest code that we Pushed to github.

Step b: adding a .env file

→ Inorder to use .env we need to install dotenv Package for that run a command npm install. It will install all the dependencies

→ to create a new file run

```
sudo nano .env.
```

and copy all the code from .env file to this new file.

Step c: Restart The Process.

→ we need to restart the Process to reflect the changes made by using Pm2 restart Id of the process.

→ we can get the id of The currently running process from

```
Pm2 list.
```

Episode - 5

Sending Emails using Amazon SES (Simple Email Service)

Step1: Set a new IAM user.

IAM (Identify and Access Management)

↳ it gives Permission to different services inside AWS.

→ To create a new user search "IAM" in search bar present in AWS Console.

→ click on "user" then "create user" btn. add a username as you wish and click on next and set permission click on "attach policies directly" it will display a list of policies search "amazonSES" in it and

• select "AmazonSESFullAccess" and click next it will take you to the summary page click on "create new user".

→ Now user is successfully created.

Step 2: Setup SES account.

Once the IAM user is created search for "SES" in searchbar to setup SES.

Step1: Account creation:

click on "View get setup page" in the account dashboard and it will take you to "get set up page" click on "create identity"

→ we can create identity either by using domain name (or) email address Since we created a domain name already click on "domain" option. and enter a domain name. and verify the domain name using "Easy DKIM" type. select the key length of your choice. and click on "Create Identity".

→ our Identity is created but its verification is in pending state to complete its verification, configure DKIM in DNS record of cloudflare.

Step 2: configuring DKIM in cloudflare

→ copy the CNAME records provided by DKIM identity to cloudflare's DNS record.

→ In cloudflare, go to DNS Records click on add record choose the type as "CNAME" and copy the CNAME Records from DKIM identity and turn off the proxy.

→ Once the records are needed SES takes some time to verify the records. once the identity is verified we can see the identity status as "verified."

Step 3: Request Production action

→ Move back to setup page and click on "Request Production access" click on "Transactional" as mail type and provide our website url. and submit it.

→ It will take some time to grand permission.

Step 4: Using it in the code for sending Email

→ For that we need to get the Credentials of the IAM user that we have created in Step 1.

→ go to the user and click on "Security Credentials" click on "create Access Key" and select the usecase as others and give a tag name for it as your wish (or) just skip it and click on "Create access key"

→ Now, the Secret key is generated copy it and place it in the .env file of our Project and also a access Key.

Step5: Copy the code from AWS SDK V3.

→ Google the AWS SDK V3 documentation and move onto Amazon SES search for sendEmail.

Step a: Create Sesclient:

→ In utils folder of our project, create a new file called "sesclient.js" and copy the sesclient code from github repo.

→ Inorder to use this code we need to install package

npm i @aws-sdk/client-ses

and edit the code to include access key.

eg: //Sesclient.js

```
Const {SESClient} =require("@aws-sdk/client-ses")

Const REGION = "us-east-1" → it should be equal to the region where we sending from.

Const Sesclient = new SESclient({
    region: REGION,
    credentials: {
        accessKeyId: process.env.AWS_ACCESS_KEY,
        SecretAccessKey: Process.env.AWS_SECRET_KEY,
    },
})

module.exports = {sesclient}
```

Step b: write a code for sending emails.

→ copy the code of sendmail from github repo of aws SDK V3.

and edit the reciplent & sender email address.

→ the email we are using here should be verified as user in IAM. if not add a new user in IAM with this email.

Step6: Using the sendemail fn in project:

whenever a connection req is ~~send~~ made it should send a email to the other user.

eg:

```
//api for sending connection req
const sendEmail = require("../utils/sendEmail")

requestRouter.post("/request/send/:status/:touserId", userAuth,
async (req, res)=>{
    try{
        ----
        ---- Logic for sending a connection req
        ----
        const emailRes = await sendEmail.run()
        console.log(emailRes)

        ..... sending res to user ---
    }catch(){
        ---
    }
})
```

whenever Person1 makes connectn req to Person 2, email is send to Person 2 stating that Person1 sent you a connection req.

Episode-6

Scheduling cron Jobs

what is Scheduling cron Jobs?

Eg we need to run a particular task at certain intervals daily we schedule those task with cron to run those tasks automatically after certain periods.

```
npm i node-cron
```

Step 1: create a New file to run cron job

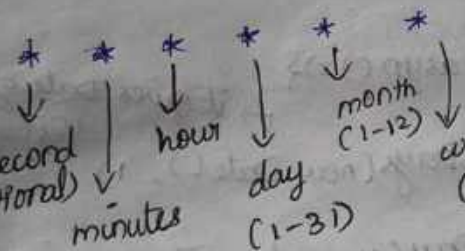
In utils folder create a new file called "cronjobs.js" (it can be anything) and inside it require a node-cron module that we installed.

eg: //cronjob.js

```
const cron = require ("node-cron")

cron.schedule ("* * * * * *", () => {
    console.log ("Hello world" + new Date())
})
```

these star represents time



-> if we put 6 stars means the task will run for every sec
-> if 5 stars, the task will run for every min & so on.

Step 2: Place it in app.js

inorder to run this cron jobs as soon as application starts we need to place it in the index file (i.e) app.js file. we just need to require this file in app.js tats it.

Crontab.guru -> is a website where we can experiment with this star strings. eg: for 8'o clock → 0 8 * * * this will schedule a task to run at 8 A.m everyday.

min hr day month week

code for sending Email to users who recieved connection request Yesterday:

Our task is to send a email everyday at 8:00 AM for the users who have received the connection request previous day.

eg code:

```
const cron = require ("node-cron")
const { subDays, startOfDay, endOfDay } = require ("date-fns")
                    ↳ we use "date-fns" package to calculate yesterday.
const sendEmail = require ("./sendEmail")
const connectReq = require ("../models/connectionRequest")

//Scheduling cron Job
cron.Schedule ("0 8 * * *", asyn ()=>{
    try {
        const yesterday = subDays(new Date(), 1)   → it gives Date of yesterday
        const start = startOfDay(yesterday)
        const End = endOfDay (yesterday)           → it gives time stamp
        //querying Collection
        const pendingReq = await connectReq.find({
            status : "interested",
            createdAt: { $gte : start , $lt: End }
        }).popdate ("fromUserId toUserId")
```

//extracting the email Id of the req receivers.

```
const listOfEmails = [...new Set(pendingReq.map((req) =>
                                    req.toUserId.emailId ))]
```

we are using Set to find a unique email Id beoz one emailId can recieve so much req.

```
for(const emails of listOfEmails){
    try {
        const res = await sendEmail.run(
            "New friend Requests are pending for" + email,
            "Please log in to DevTinder to accept the request"
        )

        console.log(res)

    } catch(err){
        console.log(err)
    }

} catch(err){
    console.log(err)
}
})
```

→ using the mail fn we created earlier on episode 5.

Note:

→ This is perfect for small application as we are looping the email and sending it blocks the code execution.

→ for larger app we should do either queueing (or) batching using some packages like "bee Queue" (or) "BullMQ" (or) simply can use Amazon SES Bulk email sending option.

Episode - 1

Payment Gateway Integration (Razorpay)

workflow:

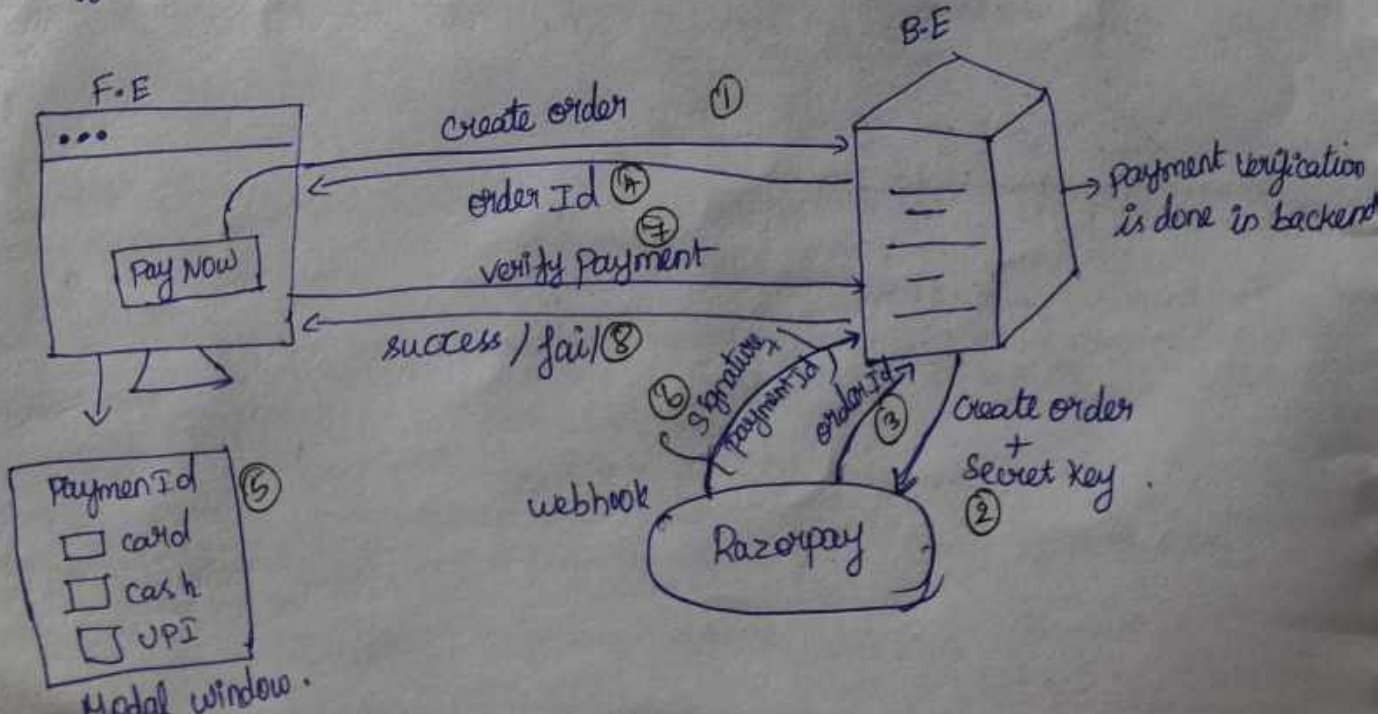
→ there is no connection between frontend and Razorpay whenever a button (Pay Now) is clicked it sends an api call to Backend then Backend will send it to Razorpay along with the secret key.

→ the Razorpay will generate an order and sends an orderId to Backend then Backend sends this order id as response to frontend.

→ Now, the pop up modal will appear along with the orderId and different mode of payments. Once the payment is done Razorpay will automatically inform the backend that the payment is finished along with the signature & payment Id using webhook.

→ After few minutes frontend will make an another API call to Backend to verify the Payment whether it is success (or) fail.

→ there is 2 API call made one to create (or) place an order, 2nd is to verify the Payment.



Steps Involved in Integrating Razorpay in a Project.

Step 1: Creation of Razorpay account:

→ Signup in a Razorpay using Id card such as aadhar card, website URL & so on. It will take 3-5 days for approval of the account.

Step 2: Including links in our Project.

→ In order for the approval of Razorpay, we need to create few components such as "Privacy Policy", "Terms of Use", "Refund and cancellation Policy", "Contact Information", "About us" ~~is the~~ and include them as links in the footer section.

→ This step is mandatory as Razorpay explicitly mentions that business need to display these for approval. without, these links, Razorpay might reject your application (or) delay approval.

Step 3: Create a new page for Premium Membership:

→ In frontend we create a new component "Premium" which consits of diff Premium plans like Silver, gold with a button and link the Page to the Navbar for ease of access.

Step 4: Create an API for placing an order.

Before creating an endpoint we need to create an instance for the Ra in utils folder create a new file "razorpay.js".

Step 1: Download Razorpay.

```
npm i razorpay
```

Step 2: creating instance.

```
const Razorpay = require("razorpay")
```

let instance = new Razorpay ({

key_id = "your key from razorpay dashboard",

key_secret = "your secret key from razorpay dashboard"

Bg?

w to get a key_id & key_secret from dashboard?

→ go to dashboard → choose Test mode → go to Accounts & settings → click on API Keys → find key_id & key_secret.

Remember to place this in .env file.

5: Connecting the instance with the Backend API.

Create a new Route in "Routes" folder as "payment.js" and add this te in "app.js" like we do always.

→ import the razorpay instance in this file and connect it with Backend.

eg: const express = require ("express")

const {userAuth} = require ("../middleware/auth")

const razorpayInstance = require ("../utils/razorpay")

//creating Router

Const paymentRouter = express.Router ()

↳ Post method as we are creating new order

PaymentRouter.post ("/payment/create", userAuth, async (req, res) => {

try {

//connecting razorpayInstance.

const order = await razorpayInstance.orders.create ({

amount: 50000, → in Paise

currency: "INR",

receipt: "receipt #1",

notes: {
firstName: "value",
membershipType: "silver",
},
});

→ notes is the metadata that we need to send along with the order.

// once order is created save it in DB.
-> create a new Payment model and import it.
-> create a new Payment in the Payment model

```
const payment = new Payment({
  userId: req.user._id,   → we get user from userAuth.
  orderId: order.id,
  status: order.status,
  amount: order.amount,
  currency: order.currency
  ...
});
const savedPayment = await payment.save();
```

// Return back order detail to frontend.

```
res.json({ ...savedPayment.toJSON() });
} catch(err) {
  return res.status(500).json({ msg: err.message });
}
});
```

Step 6: Connecting this API in frontend.

-> attach a handler fn on btn and call this API whenever the button is clicked
-> this handler fn should send the membershipType selected. Based on that backend data is dynamically generated.

Step 7: Displaying the popup modal window for completing the Payment.

→ Once the API call is made successfull from frontend we need to display a modal. for that first we need to attach a razorpay modal script file to our index.html.

→ The script tag you will get from razorpay documentation in Add Checkout options.

→ copy that script tag & Paste it in our "index.html".

→ extract the necessary data that we need to pass to the dialogue box from API response & pass it as "options".

eg: const { amount, keyId, currency, notes, orderId } = order.data ← api response

```
Const options = {
        key: keyId,
        amount,
        currency,
        name: "Dev Tinder", → Name of the app to display in modal
        order_id: OrderId,
        prefill: {
            name: notes.firstname + " " + notes.lastname,
            email: notes.emailId,
        },
        theme: {
            color: "#F37254", → Color of the modal.
        }
    }
```

(8) (9) //code for display popup

Const rzp = new window.Razorpay(options) ← passing the options.

↳ It comes from the script tag we attached

rzp.open()

↳ it opens the modal

Step8: Creating a new webhook in Razorpay.

→ webhook is used to communicate the backend whether the Payment is successfull (or) not.

→ in Razorpay Dashboard → go to Account & settings → then webhooks → click on Add New webhook. which ask for a "webhook URL" which is nothing but a what API should razorpay call if the payment is success (or) fail. and select the "payment Events" as "payment.failed" and "payment.captured". and add "Secret" this can be anything of our choice. and click on "create webhook".

Step9: Creating an API for webhook

→ we need to create an API endpoint that we added in "webhook URL" while creating a webhook. for that we need to validate whether the webhook is valid (or) not.

eg:

```
Const { validateWebhookSignature } = require("razorpay/dist/utils/razorpay-utils");

//Route
PaymentRouter.post("/payment/webhook", async (req, res) => {
    try {
        // check whether webhook is valid
        Const webhookSignature = req.headers["X-Razorpay-Signature"]
                                 ↳ this will be automatically attached by RazorPay.

        Const isWebhookValid = validateWebhookSignature(
            JSON.stringify(req.body), webhookSignature,
            process.env.Razorpay_webhook_Secret
                                 ↳ it is the same Secret that we created while adding new web hook.
        )
```

```
if (!isValidWebhook) {
    return res.status(400).json({ msg: "webhook signature invalid" })
}

// If there is valid webhook update the payment status in DB.
const PaymentDetails = req.body.payload.payment.entity
const Payment = await Payment.findOne({ orderId: PaymentDetails.order_id })

payment.status = paymentDetails.status
await payment.save()

//update the user as premium
Before that update the user model to include membershipType, isPremium
validity e so on.
const user = await User.findOne({ _id: Payment.userId })
user.isPremium = true
user.membershipType = payment.notes.membershipType
await user.save()

//return success response to Razorpay.
return res.status(200).json({ msg: "webhook received succesfully" })
} catch(err){
    return res.status(500).json({ msg: err.message })
}
```

Step 10: verify payment and update the UI.

→ create a router to check whether the user is premium (or) not.

eg:

```
PaymentRouter.get("/premium/verify", userAuth, async(req, res) => {
    const user = req.user.toJSON()
    if(user.isPremium){
        return res.json({ isPremium: true })
    }
    return res.json({ isPremium: false })
})
```

Step 11: update the UI

→ make an API call to this verify endpoint and update the UI based on isPremium status.

→ eg:

```
const [isPremium, setIsPremium] = useState(false)

useEffect(() => {
    verifyPremium()
}, [])

// handler fn
const verifyPremium = async() => {
    const res = await axios.get(BASE_URL + "/premium/verify", {
        withCredentials: true
    });
    if(res.data.isPremium){
        setIsPremium(true)
    }
}

// attach this handler fn to modal popup.
const options = {
    ...
    handler: verifyPremium
}
```

this fn will be called once the popup modal is closed

→ we return diff UI based on the isPremium state that we defined.

Step :12 Push the code to the instance

As a final step Push all the code to the instance and update the .env file.

Episode-8

web socket and Socket.io

what is Socket.io?

Socket.io is a library that enables low-latency (fast, smooth), bidirectional (client ⇄ server) and event-based communication between a client and a server.

what is webSocket?

→ web socket is a communication protocol that enables full-duplex (communication occurs simultaneously, in both direction), bidirectional communication between a client and a server over a single, persistent connection which allows real-time data exchange.

→ web socket connections are established over the standard TCP/IP Protocal, typically on port 80 (HTTP) (or) 443 (HTTPS).

Steps involved in implementing chat feature:

Step 1: Building UI:

→ Add a button to each connections with the content "chat" on clicking it it should take to the chat Page along with the selected person's user id.

→ Build a new chat component to display msgs and also a input box and button to send new msgs.

Step 2: Implementing server side connection

Step 1: Install socket.io

```
npm i socket.io
```

Step 2: Configuration of socket.io in app.js (or) new file & include it in app.js

→ earlier we created a server using express but now for configuration of socket.io we need to create it using "http" module (built-in module)

eg:
```
Const http = require ("http")
Const socket = require ("socket.io")
//Create server using http.
Const server = http.createServer (app)   → we pass the express app
//Socket configuration
  Const io = socket (server, {
                cors: {
                  origin: frontend url,
                }
  })

// listening for the events
io.on("connection", (socket) => {
   // handle events
   socket.on ("joinchat", () => { })
})

// Replace the express server with this http server.
connectDB().then(()=>{
    server.listen (8000, ()=> { console.log("server connected")
})
```

Step 3: Configuration of socket.io in frontend.

→ As its bidirectional we need to configure socket in both frontend & Backend for that install a package

```
npm i socket.io-client
```

→ Configure this socket.io in a seperate file in utils folder.

```
//socket.js
import io from "socket.io-client"
import {BaseURL} from "./constants"

export const createSocketConnection = () => {
    return io(BaseURL)          → this configuration only works in development not in production.
                  └> Backendurl
}
```

Step4: update the code in chat component

→ In chat component as soon as the page loads we need to establish a connection for that we use useEffect.

eg: // chat.jsx.

```
useEffect(() => {                    ↱ fn created in socket.js in above step
    const socket = createSocketConnection()
// socket connection is made & join chat event is emitted
    socket.emit("joinchat", {userId, targetUserId})   → data that we need to send to backend.
                ↳ this event should match the backend events

// disconnect socket whenever necessary (X)(X)
    return () => {
        socket.disconnect()
    }
}, [])
```

Step5: Handling the events in the backend

Step 1) Creating room for joining chat

we need to create a seperate room in a server for each chats where both can communicate.

→ Each room should have a unique room Id and when to 2 person is chatting lets say x & y, the chat roomId of x→y and y→x should be same.

eg: // app.js

```
io.on("connection", (socket) => {
  // handling join the room event
  socket.on("joinchat", ({ firstName, userId, targetUserId }) => {
                          ↳ these are the data passed from the frontend
    const roomId = [userId, targetUserId].sort().join("-")
                                  ↓
                  we are sorting it in order to avoid creating
    2 seperate room for same users (ie x→y & y→x)

    socket.join(roomId)
      ↳ is a method used to join the room.
  })

  // handling the event for sending Message.
  socket.on("sendMessage", ({ firstName, userId, targetUserId, msg }) =>
                                  ↓
                              data from frontend.
  {
    const roomId = [userId, targetUserId].sort().join("-")

    io.to(roomId).emit("messageReceived", { firstName, text })
      ↳ sending a message to a room
                                  ↳ once msg received we are emitting an event we need to listen this up in frontend.
  })
})
```

Step 6: Updating the frontend to send message to backend

→ Creating a state var to store a content from the input box and attach a handler fn to send button to send a msg to backend.

eg: const [msg, setMsg] = useState("")

//handler fn to send a msg

const sendMessage = () => {

const socket = CreateSocketConnection()

socket.emit("sendMessage", {

firstName, userId, targetUserId, msg

})

}.

//attach it to btn & input box

<input type="text" value={msg} onChange={(e) => setmsg(e.target.value)} />

<button onClick={sendMessage}>Send</button>.

Step 7: Listening up the "MessageReceived" event emitted from Backend in frontend.

-> create a new state var to store the msgs to display it in UI.

-> whenever the event is emitted from a BE we listen it in F.E and push that message to the state var.

eg: const [messages, setMessages] = useState([])

//Inside useEffect

socket.on("messageReceived", ({ firstName, text }) => {

setMessages((messages) => {[...messages, { firstname, text }]})

}

-> Loop through this state var & display it in UI.

Step 8: Make the RoomId encrypted in Backend

→ Instead of using the userId & targetuserId as if we can hash it for better security.

eg: const crypto = require("crypto")

```
// fn to hash Id's
Const getSecretRoomId = (userId, targetUserId) => {
    return crypto.createHash("sha256")
        .update([userId, targetuserId].sort().join("_"))
        .digest("hex")
}

// use it to generate roomId.
socket.on("joinchat", ({ firstName, userId, targetuserId }) => {
    const roomId = getSecretRoomId(userId, targetuserId)

    socket.join(roomId)
})
```

Homework:

Refer the authentication in socket.io documentation & implement it in the project.

Episode-9

Building Real time Live chat feature

->It is the continuation of the episode-8, where we have just stored the messages in the state var. The problem with that is whenever we refresh the Page the chats also get erased. In this episode we will learn how to store that chat in DB and retrieve it.

Step1: Create a Message Model in B-E

```
const mongoose = require("mongoose")

const messageSchema = new mongoose.Schema(
    { senderId:{
        type: mongoose.schema.Types.ObjectId,
        ref:"User",
        required: true,
      },
      text:{ type:String, required: true },
    },
    {timestamps: true}
)

const ChatSchema = new mongoose.schema({
  Participants: [ { type: mongoose.Schema.Types.ObjectId,
                    ref, "User", required: true } ],
  messages: [messageSchema] -> messageschema defined above
})

const Chat = mongoose.model("Chat", chatSchema)

module.exports = Chat.
```

sender & receiverId

Step 2: Save the messages in DB

→ write a logic to save the msgs in DB inside a "send Message" event in socket.js file.

eg: socket.on ("sendMessage", async ({ firstName, userId, targetUserId, text }) => {

```
try {
  const roomId = getSecretRoomId(userId, targetUserId)

  // find whether there is a chat present in DB of these participants
  // if present - push the new chat to it
  // else - create a new ~~chat~~ empty chat

  let chat = await Chat.findOne({ participants: { $all: [userId, targetUserId] },
  })

  if(!chat){
    chat = new Chat({ participants: [userId, targetUserId],
                      messages: [ ]
    });
  }

  // if already exists
  // Push this new msg to it
  chat.messages.push({ senderId: userId, text, });

  await chat.save() → Save it in DB

  io.to(roomId).emit("messageReceived", { firstName, text });
      ↳ emitting an event once message Received to indicate frontend.

} catch(err){
  console.log(err)
}
```

Create a new chat

Step3: Create an endpoint to fetch the messages from DB

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

const Chat = require("../models/chat")

const chatRouter = express.Router()

chatRouter.get("/chat/:targetUserId", userAuth, async (req, res) => {
    try {
        const userId = req.user._id  → getting userId from userAuth.
        const { targetUserId } = req.params

        let chat = await Chat.findOne({ participants: { $all: [userId, targetUserId] },
        }).populate({ path: "messages.senderId",
                      select: "firstName lastName"
        })

        if (!chat) {
            chat = new Chat({ participants: [userId, targetUserId],
                              messages: [] })
            await chat.save()
        }
        res.json(chat)
    } catch(err) {
        console.log(err)
    }
})

module.exports = chatRouter
```

if no chat send an empty msg.

→ place this router in app.js file

```
app.use("/", chatRouter)
```

Step 4: Retrieve from DB & display it in UI.

→ create a fn to fetch messages in chat component & update it with state var.

```
const fetchchatMessages = async() =>{
//make an API call
const chat = await axios.get(BackendURL+"/chat/"+targetUserId,{
                withCredentials: true, })

const chatmessages = chat?.data?.messages.map((msg)=>{
        const {senderId, text} = msg
        return {
            firstName: senderId?.firstName,
            text
        }
    })
    setMessages(chatMessages) → update the state var with the api response.
}

useEffect(()=>{ fetchChatMessages() }, []) → need to called as soon as pageload
```

Homework:

1) write a functionality to check whether that a person is friend (or) not before sending a msg to avoid manual typing of targetUserId.

2) Display green dot when online & last seen status

3) Limit messages when fetching from DB.

4) Build a tic tac toe (or) chess game (or) type racer.

Step 5: Deploy it to instance:

→ first, update the code in F.E Socket.js file before deploying it to the instance.

```
export const createSocketConnection = () => {
    if (location.hostname === "localhost") {
        return io(BaseURL)
    } else {
        return io("/", {path: "/api/socket.io"})
    }
}
```

deploy it in instance.