Class 9 - Product Sales Data model and Razor Pay Payments

Agenda:

1. Checkout flow and mode of payment
2. Payment gateway
    a. What problem statement is solved
    b. How it is solved
3. Integrating Razorpay as payment gateway
4. Webhooks
5. Payment confirmation using webhooks

Flow on the application

1. Select a product -> Buy Now
2. Cart Review , add / delete , promo codes, etc.
3. Proceed to Checkout page - order id is generated , order created
4. Make payment Action
5. Payment gateway
6. Back to the orders page with the status

Payment related challenges - why we cant have our own payment gateways

1. Multiple modes of payment
    a. UPI
    b. Credit / Debit
    c. Netbanking
2. Security
    a. Compliance with Payment Card Industry Data Security Standard (PCI DSS)
        i. Requirement: Compliance with PCI DSS is mandatory for any system that handles credit card transactions. It includes a wide range of security measures, such as maintaining a secure network, protecting cardholder data, implementing strong access control measures, regularly monitoring and testing networks, etc.
        ii. PCI DSS is a global standard and is applicable to any organization, regardless of its location, that processes, stores, or transmits cardholder data
        iii. Complexity: Achieving and maintaining PCI DSS compliance is a complex, time-consuming, and costly process.
    b. Data Encryption
        i. All sensitive data, including credit card numbers and personal information, must be encrypted both in transit and at rest.

    c. Fraud Detection and Prevention
        i. This involves building and constantly updating algorithms that can detect unusual patterns and potential security breaches
    d. Legal and Regulatory Considerations
        i. Payment gateways must adhere to various financial regulations that vary from country to country

Payment Gateways - responsibilities

1. Include multiple mode of payments - payment aggregators
2. Inform the server the final status of payment
    a. Call a predefined route (webhook) to inform about the status
3. Security

CHECKOUT FLOW

1. Choose a Payment Gateway
    a. Selection: Begin by selecting a suitable payment gateway like Razorpay, Stripe. Consider factors like fees, payment methods supported, and regional availability.
    b. Registration: Register with the payment gateway to get access to its APIs. This typically involves creating an account on their platform.
2. Obtain API Keys

    a. API Access: Once registered, obtain API keys from the payment gateway. These keys are essential for authenticating requests to the payment gateway from your application.

3. Backend Integration
    a. Setup on Server: Integrate the payment gateway into your Node.js backend. This includes setting up the necessary configurations using the API keys.
    b. Order Creation: Implement a mechanism to create a payment order on your server when a user initiates a transaction. This is usually done in response to a user's action, like clicking a 'Procced to checkout' button.
4. Frontend Integration
    a. Payment Interface: On the frontend, set up a user interface for payment. This might include a form which will initiate the payment // payment gateway's widgets / scripts
    b. Connecting Frontend with Backend: Ensure the frontend communicates with your backend to get the necessary payment order details.
5. Handling the Payment
    a. Initiate Payment Process: Once the user proceeds to make a payment, the payment gateway's interface or widget takes over to facilitate the actual transaction.
    b. User Interaction: The user completes the payment process, which may involve entering card details, using net banking, or other payment methods.

6. Verifying Payment Success
    a. Backend Verification: After a transaction, your backend should verify the payment to ensure its authenticity. This is a crucial step for security.
    b. Update Records: Once verified, update the transaction's status in your database to reflect whether it was successful, failed, or is pending.
7. Communicating Transaction Outcome
    a. User Feedback: Inform the user of the transaction outcome through a confirmation message, email, or redirection to a success or failure page.
8. Security and Compliance
    a. Security Measures: Ensure that sensitive information is handled securely, and maintain compliance with legal standards for online transactions.

Conclusion

Integrating a payment gateway is a multi-step process involving choosing a gateway, setting up the backend and frontend to handle transactions, verifying transactions, and ensuring security and compliance. Understanding this high-level flow is crucial before diving into the technical specifics of implementation.

Webhooks

1. webhook is essentially a route (or URL endpoint) in your application that the payment gateway calls to notify you of

various events, such as payment completions, failures, or chargebacks. This route is set up to listen for incoming HTTP requests (usually POST requests) from the payment gateway.

2. In your payment gateway's dashboard (e.g., Stripe, PayPal, Razorpay), you usually have an option to configure webhooks.

Flow revisited

1. From the product page, we initiate a request to get the product details
2. Checkout page, we get the details and move to place the order
3. Make the payment through the integrated payment gateway
4. Redirected to orders page (loading state)
5. Webhook to notify about the payment status

Payment Flow - low level

1. Razor pay gives two keys - public and private
    a. Asymmetric Encryption: This use of public and private keys is known as asymmetric encryption because two different keys are used for encryption and decryption, as opposed to symmetric encryption, where the same key is used for both.

b. Secure Data Transmission: This method ensures that sensitive data can be safely transmitted over the internet. Even if the encrypted data is intercepted, it remains secure because the private key, which is needed to decrypt it, is kept secret.
c. Public Key in Client: When integrating Razorpay's API, you'll use the public key in your client-side code. This key is used to initialize the payment interface and encrypt transaction requests.
d. Private Key in Server: The private key is used in your server-side code to authenticate API requests to Razorpay's servers. It's used to decrypt information or to sign requests, ensuring that the communication is authenticated and originates from a trusted source.

CODE

1. Create a front end . go to root folder
2. Run npm create vite@latest
3. Enter folder name like frontend
4. Use arrow keys to select react and javascript
5. Run the npm run dev command

Backend

1. Open razorpay nodejs integration

2. https://razorpay.com/docs/payments/server-integration/nodejs/payment-gateway/build-integration/
3.
4. Go to build step
5. npm i razorpay
6. We will need public and private keys from razorpay
7. Create a new file called payments.js
    a. Razorpay Instance: A Razorpay instance is created using your API keys. This instance will be used to interact with Razorpay's API.

```
const express = require("express");
const Razorpay = require("razorpay");
require('dotenv').config();

const app = express();
var instance = new Razorpay({
   key_id: process.env.RAZORPAY_KEY_ID,
   key_secret: process.env.RAZORPAY_SECRET_KEY,
 });
app.listen(3000, () => {
   console.log("Server is running on port 3000");
});
```

8. Next step is to create an order which has the amount, currency and initiates the transaction process

```
var options = {
   amount: 50000,  // amount in the smallest currency unit
   currency: "INR",
   receipt: "order_rcptid_11"
 };
 instance.orders.create(options, function(err, order) {
   console.log(order);
 });

app.listen(3000, () => {
   console.log("Server is running on port 3000");
});
```

9. Install the shortid package and require it

10.
```
const shortid = require("shortid");
```

11. Generate a unique id that is used for internal tracking of the order

```
var options = {
   amount: 50000,  // amount in the smallest currency unit
   currency: "INR",
   receipt: shortid.generate(),
 };
```

12. Create a checkout route

```
app.post('/checkout', (req, res) => {
// the details below in a real world app will be fetched
//via an internal order id
    // when the user finalized their cart, an order id will
//be generated and passed to this route
```

```
    var options = {
      amount: 50000,  // amount in the smallest currency
unit
      currency: "INR",
      receipt: shortid.generate(),
    };
    instance.orders.create(options, function(err, order) {
      console.log(order);
    });
 })
```

Checkout Endpoint: The /checkout POST endpoint is set up to create an order with Razorpay.

When this endpoint is hit, it constructs an order object with details like amount, currency, and a unique receipt ID.
It then calls instance.orders.create() to create an order with Razorpay.
Razorpay responds with order details, including the order_id.
our backend sends this order information back in the response to the frontend

13. Run this in postman and see the order created in razorpay

FrontEnd

1. Update app.jsx

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

function App() {
 const displayRazorpay = () => {

 }

 return (
   <>
     <div>
       <a href="https://react.dev" target="_blank">
         <img src={reactLogo} className="logo react"
alt="React logo" />
       </a>
     </div>
     <h1>React payments demo</h1>
     <div className="card">
       <a onClick={displayRazorpay} target='_blank' >Make
payment</a>
     </div>
   </>
 )
}

export default App
```

2. React specific integration is available at razorpay -
3. https://razorpay.com/docs/payments/server-integration/nodejs/faqs/#3-can-i-integrate-razorpay-checkout-with-reactjs
4. Update App.jsx

```
import { useState } from "react";
import reactLogo from "./assets/react.svg";
import viteLogo from "/vite.svg";
import "./App.css";

function loadScript(src) {
 return new Promise((resolve) => {
   const script = document.createElement("script");
   script.src = src;
   script.onload = () => {
     resolve(true);
   };
   script.onerror = () => {
     resolve(false);
   };
   document.body.appendChild(script);
 });
}

function App() {
 async function displayRazorpay() {
   const res = await loadScript(
     "https://checkout.razorpay.com/v1/checkout.js"
   );

   if (!res) {
      alert("Razropay failed to load!!");
      return;
    }

    const responseObj = await
fetch("http://localhost:3000/checkout", { // postman url
      method: "POST",
    });
    const paymentResponse = await responseObj.json();

    console.log(paymentResponse);
     const {id, currency, amount} = paymentResponse.data;

   // const options = {
    //   key: 'rzp_test_y9KN7luxGUi2Vc', // Enter the Key ID generated from the
Dashboard
    //   amount: data.data.amount, // Amount is in currency subunits. Default
currency is INR. Hence, 50000 refers to 50000 paise
    //   currency: data.data.currency,
    //   name: "Acme Corp",
    //   description: "Test Transaction",
    //   image: "https://example.com/your_logo",
    //   order_id: data.data.id, //This is a sample Order ID. Pass the `id`
obtained in the response of Step 1
    //   callback_url: "http://localhost:3000/verify",
    //   notes: {
    //     address: "Razorpay Corporate Office",
    //   },
    //   theme: {
    //     color: "#3399cc",
    //   },
    // };
    // console.log("options", options)
    // const paymentObject = new window.Razorpay(options);
    // paymentObject.open();


  }
```

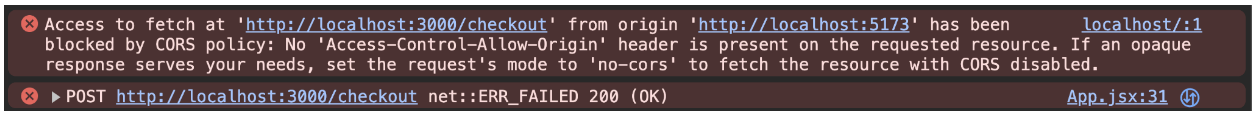
```
 return (
   <>
     <div>
       <a href="https://react.dev" target="_blank">
         <img src={reactLogo} className="logo react"
alt="React logo" />
       </a>
     </div>
     <h1>React payments demo</h1>
     <div className="card">
       <button onClick={displayRazorpay}>Pay now</button>{"
"}
     </div>
   </>
 );
}

export default App;
```

5. Load Razorpay Script: When the user is ready to make a payment, the Razorpay checkout script is loaded into the page using the loadScript function.
6. Fetching Order Details:
    a. A POST request is made to the backend's /checkout endpoint.
    b. The backend responds with the created order details

Return from checkout route

```
app.post('/checkout', (req, res) => {
   // the details below in a real world app will be fetched
//via an internal order id
       // when the user finalized their cart, an order id will
//be generated and passed to this route
        var options = {
          amount: 50100,  // amount in the smallest currency
unit
          currency: "INR",
          receipt: shortid.generate(),
        };
        instance.orders.create(options, function(err, order) {
          console.log(order);
          res.status(200).json({
           message:"Order created",
           data:order
       })
        });
    })
```

7.
8. Should get a CORS error when clicking on Pay now button

9. 

```
Access to fetch at 'http://localhost:3000/checkout' from origin 'http://localhost:5173' has been        localhost/:1
blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque
response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
▶ POST http://localhost:3000/checkout net::ERR_FAILED 200 (OK)                                          App.jsx:31
```

10. CORS
    a. CORS, or Cross-Origin Resource Sharing, is a security feature in web browsers that controls how a web page can request resources (like scripts, images, or fonts) from a different domain than the one that served the web page.

b. By default, browsers follow a "same-origin policy," which is like saying you can only order dishes from the restaurant you're currently in. This policy prevents potentially malicious scripts on one site from obtaining access to sensitive data on another.

c. web page at http://example.com tries to request a JavaScript file from http://another-domain.com/script.js.

d. With CORS, the server at http://another-domain.com can say, "It's okay for http://example.com to request my JavaScript file," and the browser will allow the script to be included in the example.com page.

11. Add CORS to backend

    a. Npm i cors

```
const cors = require("cors");
require('dotenv').config();

const app = express();
app.use(cors());
```

12. Make the payment on frontend again and see the console.
13. The data object should be the same that was sent from checkout route
14. Now an order is created
15. Replace the commented code with the following options object

```
const options = {
    key: 'rzp_test_y9KN7luxGUi2Vc', // Enter the Key ID
generated from the Dashboard
    amount: amount, // Amount is in currency subunits.
Default currency is INR. Hence, 50000 refers to 50000 paise
    currency: currency,
    name: "Acme Corp",
    description: "Test Transaction",
    image: "",
    order_id: id, //This is a sample Order ID. Pass the
`id` obtained in the response of Step 1
    // callback_url: "http://localhost:3000/verify",
    // notes: {
    //   address: "Razorpay Corporate Office",
    // },
    handler:function(response){
      alert(response.razorpay_payment_id);
      alert(response.razorpay_order_id);
      alert(response.razorpay_signature)
    },
    theme: {
      color: "#3399cc",
    },
  };
  console.log("options", options)
  const paymentObject = new window.Razorpay(options);
  paymentObject.open();
 }
```

16. Razorpay Payment Object:
    a. we create a Razorpay payment object using these details.

        i. The key used here ('rzp_test_y9KN7luxGUi2Vc') should match the key_id used in the backend.
        ii. The amount, currency, and order_id from the backend's response are used to configure this object.
    b. Opening Razorpay Checkout:
        i. The paymentObject.open() method triggers the Razorpay checkout interface.
        ii. The user completes the payment process through this interface.
    c. Payment Completion:
    d. Upon successful completion of the payment, Razorpay's checkout interface calls the handler function.
        i. This function receives details like razorpay_payment_id, razorpay_order_id, and razorpay_signature, which can be used to verify the payment on the backend.

Updating Backend with the status

1. Till now we created an order, passed the details from the front end to razorpay and made the payment
2. But the backend is not aware of the status yet
3. Current Challenge: After a payment is processed, the backend server needs to be informed about the payment status. This is

crucial for updating the order status in your database and for any post-payment processing.

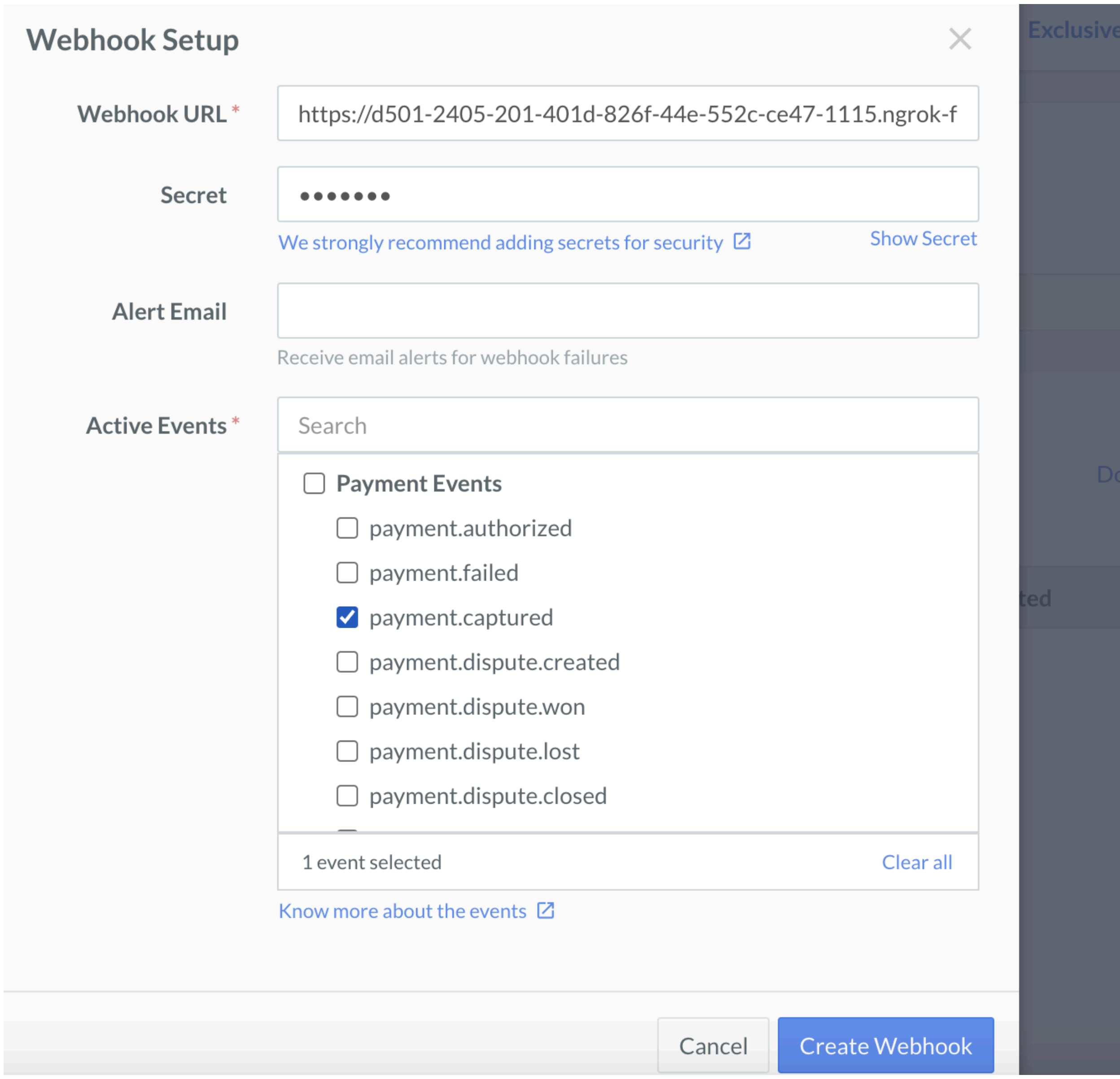
4. Solution - Webhooks: Razorpay, like many payment gateways, uses webhooks to notify your server about various events, including payment success, failure, or other related updates.
    a. Webhook Explanation: webhooks as automated messages sent from Razorpay to a specified URL on your server in response to payment events.

Making Local Backend Publicly Accessible

1. Problem with Local Development: In a development environment, our backend server is usually running locally and is not accessible from the internet, which means Razorpay cannot send webhook notifications to it.
2. HTTP Tunneling: Introducing HTTP Tunneling as a solution.
3. Tunneling Explanation: HTTP tunneling allows us to expose our local server to the internet via a public URL. Tools like Ngrok are commonly used for this purpose.
4. So notification sent to the proxy url will forward it to our local dev server

Installing ngrok

1. Do the signup and get the tokem
2. Start the ngrok service - ngrok http 3000

3. Copy the http://127.0.0.1:4040 url from the terminal
4. On opening this url , we will see the generated url for our local server
5. Open razorpay account and settings -> webhooks
6. Create a webhook
    a. Paste the generated ngrok url from the browser here
    b. 

Webhook Setup

Webhook URL * https://d501-2405-201-401d-826f-44e-552c-ce47-1115.ngrok-f

Secret •••••••

We strongly recommend adding secrets for security

Alert Email

Receive email alerts for webhook failures

Active Events *

- [ ] Payment Events
  - [ ] payment.authorized
  - [ ] payment.failed
  - [x] payment.captured
  - [ ] payment.dispute.created
  - [ ] payment.dispute.won
  - [ ] payment.dispute.lost
  - [ ] payment.dispute.closed

1 event selected

    c. Notice the /verification added after the ngrok url

d. Create a secret (12345678) and add it in the env file as well

Create the verification route

```
app.post('/verification', (req, res) => {
      try{
        console.log("webhook called",req.body);
        res.json({status:"ok"})

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

1. Check the razor pay
2. Check the ngrok url for requests

Chweck stripe node js integration as well