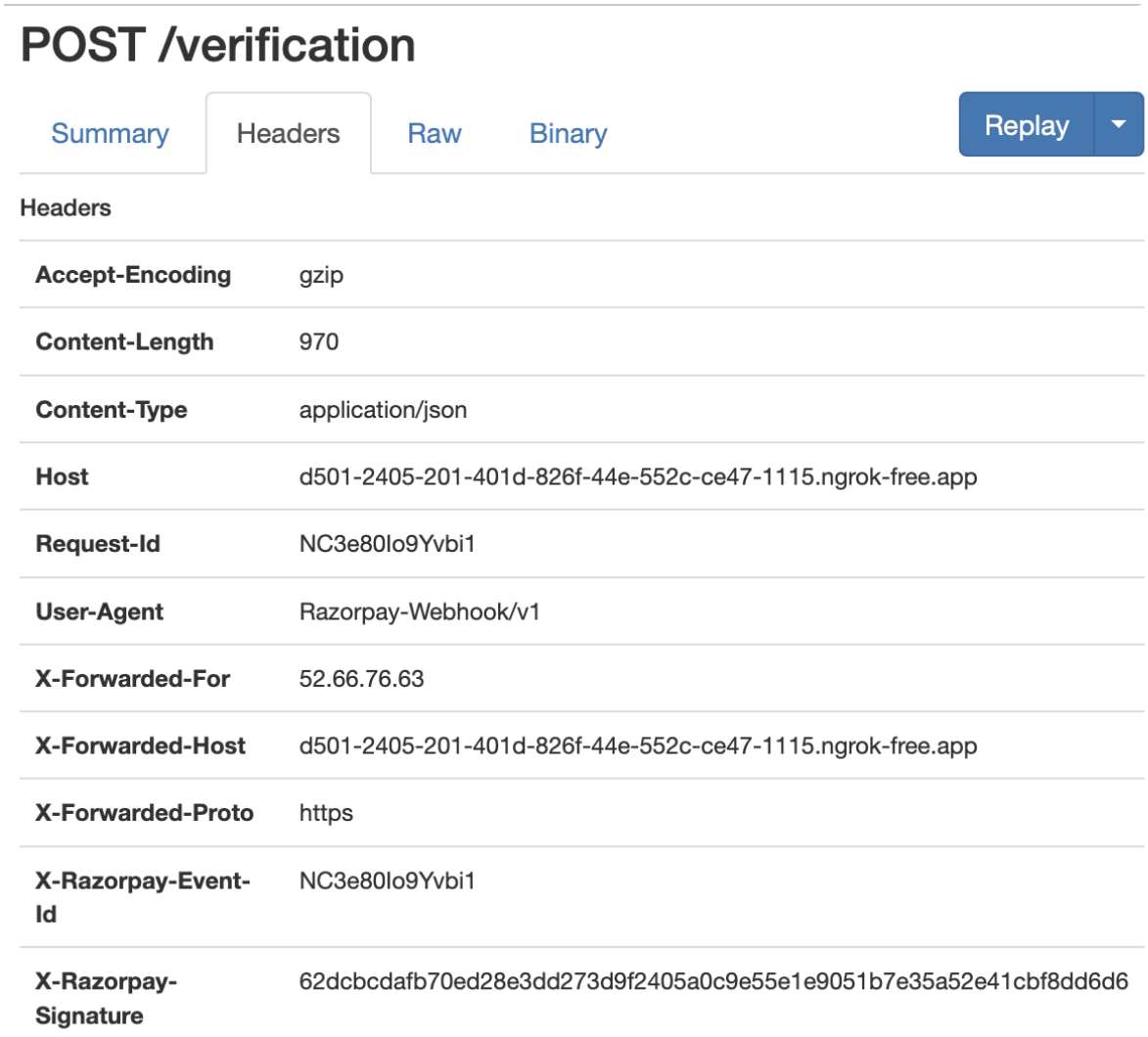
Class 10 - adding reviews, indexed, averages

Agenda

1. Payment verification
1. Data modeling in Mongoose
2. Creating data model for purchases
    a. Purchases are done when a user buys a product
    b. So we'll learn how to take care when there is a relationship between models
3. Creating data model for reviews
    a. Relationship between user and product
4. Calculating average reviews of product

Verifying Payments

1. A small but crucial bit is remaining where we need to confirm that the message received in the webhook is indeed from the razorpay
2. Open the route from ngrok - http://127.0.0.1:4040/inspect/http
3. Check the header of the post request



POST /verification

Summary | Headers | Raw | Binary | Replay

Headers

Accept-Encoding	gzip
Content-Length	970
Content-Type	application/json
Host	d501-2405-201-401d-826f-44e-552c-ce47-1115.ngrok-free.app
Request-Id	NC3e80lo9Yvbi1
User-Agent	Razorpay-Webhook/v1
X-Forwarded-For	52.66.76.63
X-Forwarded-Host	d501-2405-201-401d-826f-44e-552c-ce47-1115.ngrok-free.app
X-Forwarded-Proto	https
X-Razorpay-Event-Id	NC3e80lo9Yvbi1
X-Razorpay-Signature	62dcbcdafb70ed28e3dd273d9f2405a0c9e55e1e9051b7e35a52e41cbf8dd6d6

4.
5. The signature at the bottom is created using the payload and the secret that we had defined
6. We will create a signature at backend using the received response and the secret and then match both the signature
7. This ensures that the response was not tampered with

Updating verification route

```
app.post("/verification", (req, res) => {
```

```
 try {
   // console.log("request ",req.body);
   console.log("secret",process.env.WEBHOOK_SECRET)
   const shasum = crypto.createHmac("sha256",
process.env.WEBHOOK_SECRET);
   shasum.update(JSON.stringify(req.body)); // adding payload
to the hash
   const freshSignature = shasum.digest("hex"); // creating
hexadecimal digest
   console.log("coMPARING",freshSignature,
req.headers["x-razorpay-signature"]);
   if (freshSignature == req.headers["x-razorpay-signature"])
{
     console.log("request is legit");
     res.json({ status: "ok" });
   } else {
     return res.status(400).json({ message: "Invalid
signature" });
   }
   console.log("webhook called", req.body);
 } catch (err) {
   console.log(err);
 }
});
```

Need to import crypto - node module

```
const crypto = require("crypto");
```

Code walkthrough:

1. Create Hmac Object: crypto.createHmac("sha256", process.env.WEBHOOK_SECRET) creates an Hmac (Hash-based Message Authentication Code) object using the SHA-256 algorithm and your Razorpay webhook secret as the key.
2. Update Hmac with Payload: shasum.update(JSON.stringify(req.body)) adds the request payload (the body of the POST request from Razorpay) to the Hmac object. This payload contains the data that Razorpay sends, which we need to verify.
3. Generate Hex Digest: const freshSignature = shasum.digest("hex") generates a hexadecimal digest (hash) of the Hmac object. This digest is our computed signature based on the request's payload and your webhook secret.
4. Comparison of the signatures
5. It's a common naming convention in cryptography-related code to use terms like 'sum' or 'checksum' for variables that hold a hash value or digest.
6. The variable name shasum is likely derived from "SHA checksum" or "SHA summary."
7. Currently we are doing console statements after verification.But other additional logic can go here
    a. Updating database with payment status
    b. Updating inventory
    c. Sending email

Let us work on some of the model changes to capture bookings

Model for Booking

1. What attributes can be there
    a. bookingId - mongodb will provide
    b. priceAtBooking
    c. bookingDate
    d. Product
        i. Rather than have the whole document, we will have a reference
        ii. This reference will be used to **populate** the product while sharing the output
    e. User
        i. Reference for user

BookingModel code

Create a new file bookingModel under models

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

const bookingSchema = new mongoose.Schema({
 bookedAt: {
   type: Date,
   default: Date.now,
 },
 priceAtBooking: {
   type: Number,
```

```
    required: true,
  },
  status: {
    type: String,
    enum: ["pending", "confirmed", "cancelled"],
    default: "pending",
  },
  user:{
    type: mongoose.Schema.Types.ObjectId,
    ref: "User",
    required: true,
  },
  product:{
    type: mongoose.Schema.Types.ObjectId,
    ref: "Product",
    required: true,
  },
});

const bookingModel = mongoose.model("Booking", bookingSchema);

module.exports = bookingModel;
```

BookingRouter

1. Create a new router for booking under router folder

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

const bookingRouter = express.Router();
```

```
module.exports = bookingRouter;
```

2. Import in app.js

```
app.use("/api/auth",authRouter);
app.use("/api/booking",bookingRouter);
```

Adding a route to initiate on booking router

```
bookingRouter.post('/:productId', (req, res) => {
    try{
       var options = {
           amount: 11000, // 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,
           });
         });

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

1. Let us update our hard coded values from req and params

```
bookingRouter.post('/:productId',protectRoute, async (req,
res) => {
    try{
       const userId = req.userId; // protect route
//middleware will add the userid
       const productId = req.params.productId;
       const {priceAtBooking} = req.body;
       const bookingObj = {
           priceAtBooking,
           user:userId,
           product:productId
       }
// now we create the same booking that is created on
razorpay on our system as well
       const booking = await Booking.create(bookingObj);
       var options = {
           amount: priceAtBooking, // amount in the
smallest currency unit
           currency: "INR",
           receipt: booking._id.toString(),
         };
         instance.orders.create(options, function (err,
order) {
           console.log(order);
           res.status(200).json({
             message: "Order created",
             data: order,
           });
         });

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

```
    }
})
```

2. Add necessary imports in the booking router -

```
const {protectRoute} =
require("../controllers/authController");
const Booking = require("../models/bookingModel");
const Razorpay = require("razorpay");

var instance = new Razorpay({
    key_id: process.env.RAZORPAY_KEY_ID,
    key_secret: process.env.RAZORPAY_SECRET_KEY,
  });
```

3. In the postman , hit url with product id

   localhost:3000/api/booking/656e8a16513f8087b901c8d0

```
{
  "priceAtBooking":100
}
```

4. See the order on postman
5. See the booking object on mongoDB

Get All Bookings

```
bookingRouter.get('/',protectRoute, async (req, res) => {
 try{
   const allBookings = await Booking.find();
   res.status(200).json({
```

```
    message:"List of all bookings",
    data:allBookings
  })

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

1. Now we would want to populate the user and products in this response

```
bookingRouter.get('/',protectRoute, async (req, res) => {
 try{
   const allBookings = await
Booking.find().populate("user").populate("product");
   res.status(200).json({
     message:"List of all bookings",
     data:allBookings
   })

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

2. If we dont want the entire object but only few fields

```
const allBookings = await
Booking.find().populate({path:'user',select:"name email"})
   .populate({path:"product",select:"name price"});
```

3. populate is used to automatically replace the specified paths in the document with document(s) from other collection(s).
4. This is like fetching data using foreign keys in sql table

Adding a payment order id

1. In our schema we dont have a payment order id yet for internal tracking that is mapped to the razorpay payment
    a. When we create an order on razorpay, that time we create the same entry in our database with status as pending
    b. When the payment is done, we want to use the same id and update the status in our db
    c.
2. In Booking model

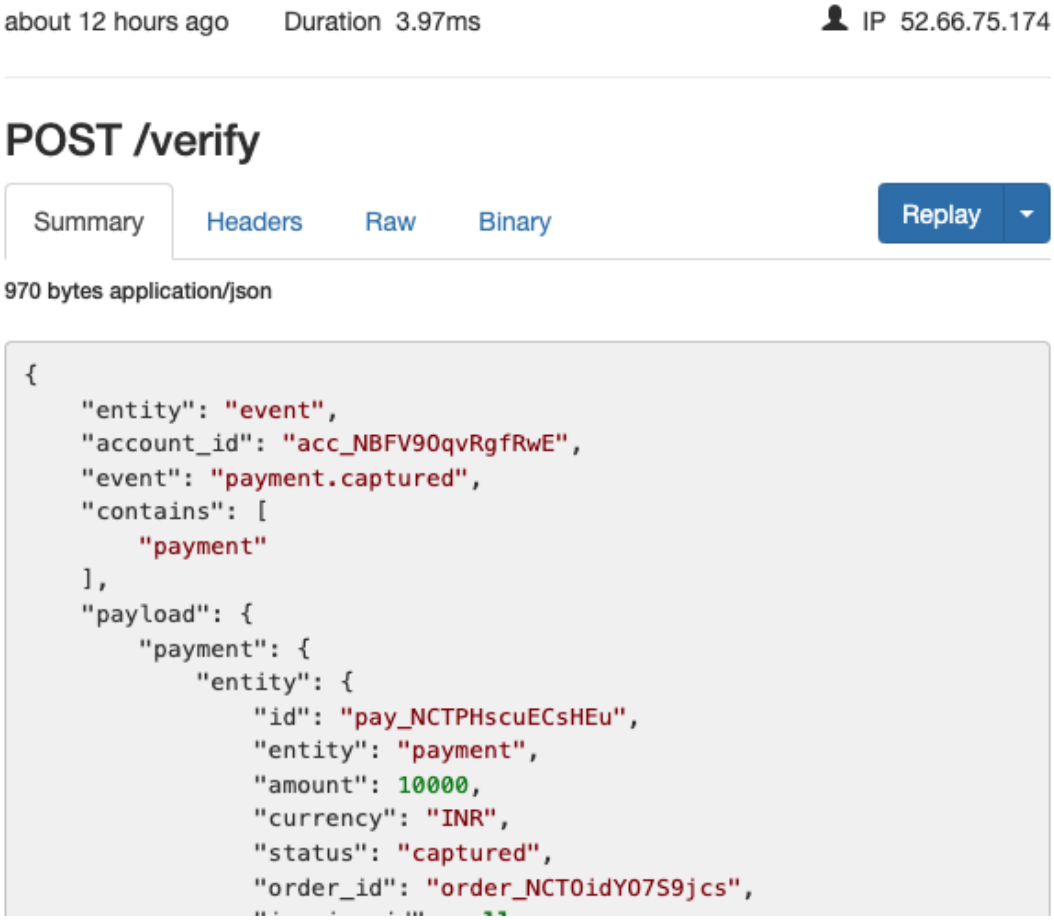
```
product:{
   type: mongoose.Schema.Types.ObjectId,
   ref: "Product",
   required: true,
 },
 paymentOrderId:String
```

3. Now we we will include the payment id received from razor pay and map it to the paymentOrderId in the model

```
bookingRouter.post("/:productId", protectRoute, async (req,
res) => {
 try {
   const userId = req.userId;
```

```
    const productId = req.params.productId;
    const { priceAtBooking } = req.body;
    const bookingObj = {
      priceAtBooking,
      user: userId,
      product: productId,
    };
    const booking = await Booking.create(bookingObj);
    var options = {
      amount: priceAtBooking, // amount in the smallest
currency unit
      currency: "INR",
      receipt: booking._id.toString(),
    };
    const order = await instance.orders.create(options);
     console.log("created order",order);
    booking.paymentOrderId = order.id;
    await booking.save();
    res.status(200).json({
      message: "Booking created",
      data: booking,
      order,
    });
  } catch (err) {
    console.log(err);
  }
});
```

Use of payment order id in the verification step

1. See the order id in the ngrok request details
2. 

---

3. We will find this order id in the booking collection and update the status

```
bookingRouter.post("/verify", async (req, res) => {
 try {
   // console.log("request ",req.body);
   console.log("secret", process.env.WEBHOOK_SECRET);
   const shasum = crypto.createHmac("sha256",
process.env.WEBHOOK_SECRET);
   shasum.update(JSON.stringify(req.body)); // adding
payload to the hash
```

```
   const freshSignature = shasum.digest("hex"); // creating
hexadecimal digest
   console.log(
     "coMPARING",
     freshSignature,
     req.headers["x-razorpay-signature"]
   );
   if (freshSignature ==
req.headers["x-razorpay-signature"]) {
     console.log("request is legit");
     const booking = await Booking.findOne({
       paymentOrderId:
req.body.payload.payment.entity.order_id,
     });
     booking.status = "confirmed";
     delete booking.paymentOrderId; // this step could be
// a design decision
      await booking.save();

     res.json({ status: "ok" });
   } else {
     return res.status(400).json({ message: "Invalid
signature" });
   }
   console.log("webhook called", req.body);
 } catch (err) {
   console.log(err);
 }
});
```

Reviews

1. For a review to be made, you should be a buyer
2. Think about the review schema
    a. Rating
    b. Review text
    c. User
    d. Product
3. Create reviewModel.js under Models folder

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

const reviewSchema = new mongoose.Schema({
    review:{
        type:String,
        required:[true,"Review cannot be empty"]
    },
    rating:{
        type:Number,
        min:1,
        max:5
    },
    createdAt:{
        type:Date,
        default:Date.now()
    },
    user:{
        type:mongoose.Schema.Types.ObjectId,
        ref:"User",
        required:[true,"Review must belong to a user"]
```

```
    },
    product:{
        type:mongoose.Schema.Types.ObjectId,
        ref:"Product",
        required:[true,"Review must belong to a product"]
    }
})
const Review = mongoose.model("Review",reviewSchema);

module.exports = Review;
```

4. One change that we need to make for User model is that a user can have multiple bookings under his name and would want to see the bookings on his previous orders page
5. In the userModel make the changes below

```
 role:{
     type: String,
     default: "user"
   },
  bookings:{
      type: [mongoose.Schema.Types.ObjectId],
      ref: "Booking"
    }
```

6. Now when we initiated the booking, we saved the details on booking collection. We also will need to save it in the user collection

```
bookingRouter.post("/:productId", protectRoute, async (req, res)
=> {
```

```
try {
  const userId = req.userId;
  const productId = req.params.productId;
  const { priceAtBooking } = req.body;
  const bookingObj = {
    priceAtBooking,
    user: userId,
    product: productId,
  };
  const booking = await Booking.create(bookingObj);
  /** update user with the booking details */
  const user = await User.findById(userId);
  user.bookings.push(booking._id);
  await user.save();
  /** create order */
  var options = {
    amount: priceAtBooking, // amount in the smallest currency
unit
    currency: "INR",
    receipt: booking._id.toString(),
  };
  const order = await instance.orders.create(options);

  /** updating booking with razor pay payment id */
  booking.paymentOrderId = order.id;
  await booking.save();
  res.status(200).json({
    message: "Booking created",
    data: booking,
    order,
  });
} catch (err) {
```

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

Review Router

1. Adding a review router in api.js

```
app.use("/api/reviews",reviewRouter);
```

2. Create a reviewRouter.js file

```
const express = require("express");
const reviewRouter = express.Router();
const Review = require("../models/reviewModel");
const {protectRoute} =
require("../controllers/authController");

reviewRouter.post("/:productId",protectRoute,async
function(req,res){})
reviewRouter.get("/:productId",async function(req,res){})

module.exports = reviewRouter;
```

3. And we now need to add a review property in the ProductModel as well
4. In the product model, add below

```
brand:{
        type:String,
        required:[true,'Please provide brand']
    },
```

```
    reviews:{
        type:[mongoose.Schema.Types.ObjectId],
        ref:"Review"
    },
averageRating:{
         type:Number,
         default:0,
         min:0,
         max:5,
     }
```

5. Update create review route

```
reviewRouter.post("/:productId", protectRoute, async function
(req, res) {
 /**
   * 1. get the product id from the params
   * 2. get the review from the body
   * 3. get the user id from the req.userId
   * 7. update average rating of the product
   * 4. create a review object
   * 5. save the review object
   * 6. push the review id in the product reviews array
  */
}
```

Adding the logic

```
reviewRouter.post("/:productId", protectRoute, async function
(req, res) {
 /**
```

```
  * 1. get the product id from the params
  * 2. get the review from the body
  * 3. get the user id from the req.userId
  * 7. update average rating of the product
  * 4. create a review object
  * 5. save the review object
  * 6. push the review id in the product reviews array
  */
 try {
   const userId = req.userId;
   const productId = req.params.productId;
   const { review, rating } = req.body;

   const reviewObj = await Review.create({
     review,
     rating,
     user: userId,
     product: productId,
   });

   const productObj = await Product.findById(productId);
   const averageRating = productObj.averageRating;
   if (averageRating) {
     let sum = averageRating * productObj.reviews.length;
     let finalAverageRating = (sum + rating) /
(productObj.reviews.length + 1);
     productObj.averageRating = finalAverageRating;
   } else {
     productObj.averageRating = rating;
   }
   productObj.reviews.push(reviewObj._id);
   console.log("product",productObj);
```

```
    await productObj.save();
    res.status(200).json({
      message: "Review created",
      data: reviewObj,
    });
  } catch (err) {
    res.status(500).json({
      message: "Server error",
      error: err.message,
    });
  }
});
```

The central takeaway from today's class is that we need to be cognizant of the interdependence between the models