Build a MERN Stack e\_commerce Web App - Part 4

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## In the fourth part, we will build the REST APIs required for the handling cart and the orders in this project. We will be using Stripe Checkout to handle payments.

So, this is the fourth part of the MERN Stack series we have recently started. In the first part, we all learnt how to set up the project and had explanations about various things we are going to use in the project and we developed all our models for the project in the second part with the help of Mongoose and MongoDB.

In the third part, we started to build the REST APIs which handles the authentication and items in our project. Now, in this fourth part, we will be wrapping up our backend part by building the REST APIs to handle the cart and orders aspect of the web application and also to handle payments using Stripe Checkout.

So, as we saw in the previous part, we created the folders named *routes*and *controllers*in our root directory.

We also created four files inside each of these two folders —representing the **auth, item, cart**and**order** respectively.

***Note****: We will handle the Routes and Controllers related to only Cart and Orders in this tutorial as we have already dealt with the Authentication and Items in the previous tutorial.*

Routes

Cart Routes

This file deals with all the routes related to the cart in our application. It has three routes — get cart items, add an item to cart and delete items from the cart.

Cart Routes

So, let us deal with each route one by one:-

1. **get\_cart\_items** — This route makes a *get*request which fetches all the items in the cart of a particular user. The id of the user requesting his cart is passed along as a param. So, we find the user’s cart and return all the cart items.
2. **add\_cart\_item** — This route makes a *post*request which is used to add an item to the cart. It also has a param *id* to identify which user is adding the item to the cart and thus we can find his cart and add the item to his cart or create a new cart for him.
3. **delete\_item** — This is a *delete*request and it removes an item from the cart. It takes in two params — userId and itemId. userId is used to get the cart of that particular user and itemId is used to search for that item and remove it from the cart.

Order Routes

This file would deal with all the routes we need for handling our orders. It has two routes — getting all our orders and placing an order (checkout).

So, let’s check these two routes one by one:-

1. **get\_orders** — It is a *get*request and fetches all the orders we have made till now in our application. It accepts a param *id*which is the userId which helps us to return the correct user’s orders.
2. **checkout** — It is a *post* request which also has a param *id*for finding the user. Its function is to create a new order. All the payments part is handled by this route. We will see those in its controller.

Controllers

Cart Controllers

This controller file would handle all the logic for the cart. It would handle adding items to cart, deleting items from the cart and getting the cart items to display along with the total cost.

This would consist of three functions, one each for the three routes we had, each handling the specific purpose.

So, we would discuss each of these three functions in detail. But before that, we need to require the Cart and the Item models into this file.

const Cart = require('../models/Cart');  
const Item = require('../models/Item');

Now, our first task would be to create the function which would fetch all the items in our cart for displaying in the application’s frontend.

We will get the user id of the user whose cart we want to access. Next, we would try to search for a cart with that username. If we find a cart with that username and the cart has non-zero items in it *i.e.* the cart is non-empty, then we would return the cart otherwise we would return null.

*We will handle the same in our frontend too by checking and informing users that cart is empty if we send a null value.*

module.exports.get\_cart\_items = async (req,res) => {  
 const userId = req.params.id;  
 try{  
 let cart = await Cart.findOne({userId});  
 if(cart && cart.items.length>0){  
 res.send(cart);  
 }  
 else{  
 res.send(null);  
 }  
 }  
 catch(err){  
 console.log(err);  
 res.status(500).send("Something went wrong");  
 }  
}

Next, we will handle adding items to our cart. This would be a little more involved since we need to handle more than one scenario.

In this function, we receive the userId by params and we also receive productId and the quantity via the request body. Here, we need userId to access the cart for that respective user and we need productId for finding the item to add to the cart.

So, we will first try to find a cart with the userId we got. Now, there are two scenarios — a user may have a cart or he might not have one.

Also, we find the item with the help of the productId we received. If the item is not found, we send a response stating the same.

If the user already has a cart, we then search for the item we need to add in the cart’s item *i.e.*if the item already exists in the cart. In this case, we take the item from the cart, increase its quantity by the quantity we received and then assign the updated item to the cart.

If the item is not present in the cart, we push it into the cart’s item’s array. We then update the cart’s bill in both cases and then save the cart. We then send the cart as a response.

Now, coming to the second case, if the user does not already have a cart, we create a new cart for the user with the userId, the item we need to add and the bill. We then return the new cart as a response.

module.exports.add\_cart\_item = async (req,res) => {  
 const userId = req.params.id;  
 const { productId, quantity } = req.body;  
  
 try{  
 let cart = await Cart.findOne({userId});  
 let item = await Item.findOne({\_id: productId});  
 if(!item){  
 res.status(404).send('Item not found!')  
 }  
 const price = item.price;  
 const name = item.title;  
   
 if(cart){  
 // if cart exists for the user  
 let itemIndex = cart.items.findIndex(p => p.productId == productId);  
  
 // Check if product exists or not  
 if(itemIndex > -1)  
 {  
 let productItem = cart.items[itemIndex];  
 productItem.quantity += quantity;  
 cart.items[itemIndex] = productItem;  
 }  
 else {  
 cart.items.push({ productId, name, quantity, price });  
 }  
 cart.bill += quantity\*price;  
 cart = await cart.save();  
 return res.status(201).send(cart);  
 }  
 else{  
 // no cart exists, create one  
 const newCart = await Cart.create({  
 userId,  
 items: [{ productId, name, quantity, price }],  
 bill: quantity\*price  
 });  
 return res.status(201).send(newCart);  
 }   
 }  
 catch (err) {  
 console.log(err);  
 res.status(500).send("Something went wrong");  
 }  
}

Now, finally, we move to the final function which is related to the cart — delete items from the cart.

In this, we receive two params — userId and productId. We try to search for the cart with the userId. We also search for the item using the productId we received.

If we have the item in the cart, we take that item from the cart and reduce the bill accordingly taking into account its price and quantity. We then use the *splice()*function to remove that item from the cart.

Next, we save the cart and return the cart as a response to the user.

module.exports.delete\_item = async (req,res) => {  
 const userId = req.params.userId;  
 const productId = req.params.itemId;  
 try{  
 let cart = await Cart.findOne({userId});  
 let itemIndex = cart.items.findIndex(p => p.productId == productId);  
 if(itemIndex > -1)  
 {  
 let productItem = cart.items[itemIndex];  
 cart.bill -= productItem.quantity\*productItem.price;  
 cart.items.splice(itemIndex,1);  
 }  
 cart = await cart.save();  
 return res.status(201).send(cart);  
 }  
 catch (err) {  
 console.log(err);  
 res.status(500).send("Something went wrong");  
 }  
}

So, finally, we have built all the required functions for the cart routes. Now, we are ready to handle all the requests relevant to the cart.

Cart Controller

Order Controller

This controller file would handle all the logic for the orders. It would handle viewing all our orders and also will allow us to place a new order from the items we have in our cart and we would handle payments via Stripe Checkout.

This would consist of two functions, one each for the two routes we had, each handling the specific purpose.

Before moving forward, we would need to install the Stripe library into our application.

So, to do it, we would use npm install stripe to install stripe. It would save it as a dependency in our *package.json* file.

Also, we need to add*StripeAPIKey* in our config file. So, our updated *default.json*file inside of the config folder would be:

{  
 "dbURI": "YOUR DB URI",  
 "jwtsecret": "your jwt secret",  
 "StripeAPIKey": "YOUR STRIPE SECRET API KEY"  
}

Now, we need to require a few into the orderControllers file. We would need to import the Order, Cart and the User models. We also need the config package to access Stripe Secret key. We also require Stripe into our function which would handle payments.

const Order = require('../models/order');  
const Cart = require('../models/Cart');  
const User = require('../models/User');  
const config = require('config');  
const stripe = require('stripe')(config.get('StripeAPIKey'));

So, we would start with the function required to fetch all the orders for a particular user. This is fairly straight forward and we just need to find the Orders using the userId provided with the params. We sort them in descending order by the date ordered and then return the orders as a response in JSON.

module.exports.get\_orders = async (req,res) => {  
 const userId = req.params.id;  
 Order.find({userId}).sort({date:-1}).then(orders => res.json(orders));  
}

Next, we have the checkout function. We receive the userId as a param with this request. We also receive a *source*as the request body from the frontend. This is to handle the payments via Stripe.

Now, we find the Cart and the User by using the userId provided. We get the email of the user.

We then check whether the cart exists or not If there is no cart, we send a response stating that the cart is empty.

Now, we create a *charge*using Stripe. We pass in the amount, the currency we want to receive payments in, the source object we received from the frontend and the receipt\_email.

If the charge was not successfully created, we throw an error stating payment failed.

If the charge was successful, we would create a new order userId, items using the cart’s items and the bill using the cart’s bill.

We then delete the cart using the cart’s id and then we send the order as a response to the frontend.

module.exports.checkout = async (req,res) => {  
 try{  
 const userId = req.params.id;  
 const {source} = req.body;  
 let cart = await Cart.findOne({userId});  
 let user = await User.findOne({\_id: userId});  
 const email = user.email;  
 if(cart){  
 const charge = await stripe.charges.create({  
 amount: cart.bill,  
 currency: 'inr',  
 source: source,  
 receipt\_email: email  
 })  
 if(!charge) throw Error('Payment failed');  
 if(charge){  
 const order = await Order.create({  
 userId,  
 items: cart.items,  
 bill: cart.bill  
 });  
 const data = await Cart.findByIdAndDelete({\_id:cart.id});  
 return res.status(201).send(order);  
 }  
 }  
 else{  
 res.status(500).send("You do not have items in cart");  
 }  
 }  
 catch(err){  
 console.log(err);  
 res.status(500).send("Something went wrong");  
 }  
}

Order Controller

Conclusion

So, this was all for the fourth part. We have finally summed up the backend part of this series and will now move forward to the client-side i.e. React and Redux code from the next tutorial.