Build a MERN Stack e\_commerce Web App - Part 3

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

So, this is the third 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**.

Now, in the third part, we will build the backend part which would handle the authentication and items in our web application by building out the APIs with the help of Express Router and we will also be defining a custom middleware function to check whether a user is authenticated or not.

To keep things clean and simple, we would create a new folder named ***routes***in our root folder. This folder would contain all the routes we need for this project.

We will also create a folder named ***controllers***in which we would put all the function we would be calling once we hit an API endpoint. So, we would separate the function in a different folder and import them in routes folder to use them.

Inside the *routes*folder, we would create four files — **auth, item, cart**and**order.**These four files would contain the routes relevant to the authentication, items, cart and orders respectively.

Similarly, we would create four files inside the *controllers'* folder, one each for each file of *routes* folder. Those would be —**authControllers, itemControllers, cartControllers**and **orderControllers** respectively.

So, we would now start building our Routes folder which would be simple since we will put all the logic in the *controllers'* folder and not directly in the *routes* folder.

***Note****: We will handle the Routes and Controllers related to only Authentication and Items in this tutorial and will deal with the Cart and Order in the next tutorial.*

Routes

Auth Routes

This file would contain all the routes we need for authentication of a user in our web application.

***What comes to your mind when we talk about authentication?***

*Obviously registering a user, logging in and logging out. Also, we need to keep checking whether a user is currently logged in or not.*

Below is the code for the authentication routes we would be using in our application. First, check the code and then I would explain in detail about what is the purpose of each line present in our code.

Auth Routes

|  |
| --- |
| const { Router } = require('express'); |
|  | const authController = require('../controllers/authControllers'); |
|  | const router = Router(); |
|  | const auth = require('../middleware/auth'); |
|  |  |
|  | router.post('/register', authController.signup); |
|  | router.post('/login', authController.login); |
|  | router.get('/user', auth, authController.get\_user); |
|  |  |
|  | module.exports = router; |

So, first of all, we start by requiring ***Router*** from *express*in our file. We would be using the express router to build all our routes. We also bring in the *authControllers* from the controllers' folder and also our custom middleware function *auth*from the middleware folder. We would be building these files later on.

We then have three routes — **register, login** and **user.** Let’s see how each one functions.

1. **register**— This route handles a *post* request in which a user provides his name, email and password for registering in our system.
2. **login** — This route handles the user login part of the website. It allows users to log in and checks whether the credentials are correct.
3. **user**— This route is a*get* request and we try to retrieve whether a user is logged in or not using this route.

***Note****: Now, you may be wondering that we missed the logout route, but we do not need it. We will be using****Local Storage****for storing our****JWT****Token which handles it on the client side so we will handle logout of users on client side directly. We do not need to deal with server for this purpose.*

Item Routes

This file contains all the routes relevant to the items — getting the items, adding a new item, updating items and deleting items. So, let’s first check the code and then delve in more details.

Item Routes

|  |
| --- |
| const { Router } = require('express'); |
|  | const itemController = require('../controllers/itemControllers'); |
|  | const router = Router(); |
|  |  |
|  | router.get('/items', itemController.get\_items); |
|  | router.post('/items',itemController.post\_item); |
|  | router.put('/items/:id',itemController.update\_item); |
|  | router.delete('/items/:id',itemController.delete\_item); |
|  |  |
|  | module.exports = router; |

It has four routes in it. They each handle a specific functionality as we saw above.

1. **get\_items** — This route is a *get* request and the purpose of this route is to fetch all the items from the server.
2. **post\_item**— This route is a *post*request and its purpose is to add a new item to the database.
3. **update\_item** — This route is a *put*request. Its purpose is to update an existing item in the database.
4. **delete\_item** — This route is a *delete*request and its purpose is to delete an item from the database.

***Note****: delete\_item and update\_item both have a param field****‘id’****which is also passed along with the URL. It contains the id of the item which we want to delete or update. We then search for that item in the database using the id.*

Controllers

Auth Controller

This controller file would handle all the logic for the registration, login and fetching user to check whether the user is authenticated or not.

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 some things into this file.

const User = require('../models/User');  
const jwt = require('jsonwebtoken');  
const config = require('config');  
const bcrypt = require('bcrypt');

We require four things as seen above. Let’s discuss the responsibility of each of these:-

1. **User**— We require the User model we created in the previous tutorial. Since we are to deal with users, we will be needing it.
2. **jwt**— We require the *jsonwebtoken*in our file to create the JSON Web Tokens we need to store to verify whether a user has been authenticated.
3. **config** — This is required to bring in the config package to let us access the JSON we stored in the config folder. It lets us store the JWT secret code.
4. **bcrypt**— This is required to use the bcrypt library which is needed to hash the passwords before saving them in the database.

Now, we can start building our functions. So, we would be starting with the registration.

Here is the code for the registration function which we have named as *‘signup’.*

module.exports.signup = (req,res) => {  
 const { name, email, password } = req.body;  
  
 if(!name || !email || !password){  
 res.status(400).json({msg: 'Please enter all fields'});  
 }  
  
 User.findOne({email})  
 .then(user => {  
 if(user) return res.status(400).json({msg: 'User already exists'});  
  
 const newUser = new User({ name, email, password });  
  
 // Create salt and hash  
 bcrypt.genSalt(10, (err, salt) => {  
 bcrypt.hash(password, salt, (err, hash) => {  
 if(err) throw err;  
 newUser.password = hash;  
 newUser.save()  
 .then(user => {  
 jwt.sign(  
 { id: user.\_id },  
 config.get('jwtsecret'),  
 { expiresIn: 3600 },  
 (err, token) => {  
 if(err) throw err;  
 res.json({  
 token,  
 user: {  
 id: user.\_id,  
 name: user.name,  
 email: user.email  
 }  
 });  
 }  
 )  
 });  
 })  
 })  
 })  
}

As we can see, we have a request and response in the arrow function. First of all, we deconstruct the name, email and password fields from the request body which is being passed over to us with the API request.

Next, we would check whether any field is empty, if yes, then we send a response with a message telling users to fill out all the fields.

We then try to search for a user with the provided email. If we find the user with the email in our database, then we return a response to the user to tell him that the email id already exists in our system and the user should use a different email or log in with that email rather than registering.

Next, we create a new user instance with the name, email and password we received from the request body. We won’t save it to the database for now since we need to hash the password before saving.

Next, we generate a salt and then we hash the password using that salt. We then set the hashed value as our password and then we save the *newUser*instance in the database.

After saving the user in the database, we need to create a signed JWT token to be stored in the local storage. We create the token by providing the user id, a JWT secret and the expiry time. We then send the token as a response along with the user details without the password.

So, this is all about registering a new user into our system.

Next, we move on to log in part. It allows an already registered user to log into our application.

Here is the code for the login function which we have named as *‘login’.*

module.exports.login = async (req,res) => {  
 const { email, password } = req.body;  
 if(!email || !password){  
 res.status(400).json({msg: 'Please enter all fields'});  
 }  
 User.findOne({email})  
 .then(user => {  
 if(!user) return res.status(400).json({msg: 'User does not exist'});  
  
 // Validate password  
 bcrypt.compare(password, user.password)  
 .then(isMatch => {  
 if(!isMatch) return res.status(400).json({ msg: 'Invalid credentials'});  
  
 jwt.sign(  
 { id: user.\_id },  
 config.get('jwtsecret'),  
 { expiresIn: 3600 },  
 (err, token) => {  
 if(err) throw err;  
 res.json({  
 token,  
 user: {  
 id: user.\_id,  
 name: user.name,  
 email: user.email  
 }  
 });  
 }  
 )  
 })  
 })  
}

Similar to the signup, we start by deconstructing the request body to get the email and password value from it.

If any of the two is absent, we send a response to the user with a message stating that they need to enter both the email and password.

We then search for the user using the email id. If the user does not exist, we will send a response to the user stating that the user does not exist in the database and he/she should register first before logging in.

Next, we will compare the provided password with the user’s password present in the database. Now, you may be wondering how will we compare them since we have the hashed password in the database.

So, to compare, we will need to use bcrypt’s compare function. It takes the password provided and then hashes it and compares it to hashed password saved in the database. If they do not match, we return a message stating invalid credentials.

We then create a signed JWT token in the same way as we did in the signup function. We then return the token along with the details of the user without the password.

Next, we will be dealing with the *get\_user*function. It finds a user by its id and then returns the user without its password as the JSON response.

module.exports.get\_user = (req,res) => {  
 User.findById(req.user.id)  
 .select('-password')  
 .then(user => res.json(user));  
}

Here is the complete code for the Authentication Controllers file.

Auth controllers

|  |
| --- |
| const User = require('../models/User'); |
|  | const jwt = require('jsonwebtoken'); |
|  | const config = require('config'); |
|  | const bcrypt = require('bcrypt'); |
|  |  |
|  | module.exports.signup = (req,res) => { |
|  | const { name, email, password } = req.body; |
|  |  |
|  | if(!name || !email || !password){ |
|  | res.status(400).json({msg: 'Please enter all fields'}); |
|  | } |
|  |  |
|  | User.findOne({email}) |
|  | .then(user => { |
|  | if(user) return res.status(400).json({msg: 'User already exists'}); |
|  |  |
|  | const newUser = new User({ name, email, password }); |
|  |  |
|  | // Create salt and hash |
|  | bcrypt.genSalt(10, (err, salt) => { |
|  | bcrypt.hash(password, salt, (err, hash) => { |
|  | if(err) throw err; |
|  | newUser.password = hash; |
|  | newUser.save() |
|  | .then(user => { |
|  | jwt.sign( |
|  | { id: user.\_id }, |
|  | config.get('jwtsecret'), |
|  | { expiresIn: 3600 }, |
|  | (err, token) => { |
|  | if(err) throw err; |
|  | res.json({ |
|  | token, |
|  | user: { |
|  | id: user.\_id, |
|  | name: user.name, |
|  | email: user.email |
|  | } |
|  | }); |
|  | } |
|  | ) |
|  | }); |
|  | }) |
|  | }) |
|  | }) |
|  | } |
|  |  |
|  | module.exports.login = async (req,res) => { |
|  | const { email, password } = req.body; |
|  | if(!email || !password){ |
|  | res.status(400).json({msg: 'Please enter all fields'}); |
|  | } |
|  | User.findOne({email}) |
|  | .then(user => { |
|  | if(!user) return res.status(400).json({msg: 'User does not exist'}); |
|  |  |
|  | // Validate password |
|  | bcrypt.compare(password, user.password) |
|  | .then(isMatch => { |
|  | if(!isMatch) return res.status(400).json({ msg: 'Invalid credentials'}); |
|  |  |
|  | jwt.sign( |
|  | { id: user.\_id }, |
|  | config.get('jwtsecret'), |
|  | { expiresIn: 3600 }, |
|  | (err, token) => { |
|  | if(err) throw err; |
|  | res.json({ |
|  | token, |
|  | user: { |
|  | id: user.\_id, |
|  | name: user.name, |
|  | email: user.email |
|  | } |
|  | }); |
|  | } |
|  | ) |
|  | }) |
|  | }) |
|  | } |
|  |  |
|  | module.exports.get\_user = (req,res) => { |
|  | User.findById(req.user.id) |
|  | .select('-password') |
|  | .then(user => res.json(user)); |
|  | } |

Item Controller

This controller file would handle all the logic related to the items — add an item, get all items, delete an item or modify an item.

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

So, we would discuss each of these four functions in detail. We need to require only the Item model in this file.

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

Now, we would start with the function to get all items from the database. We will get all the items and sort them in decreasing order by date added. We then return these items in JSON format.

module.exports.get\_items = (req,res) => {  
 Item.find().sort({date:-1}).then(items => res.json(items));  
}

Next, we would be dealing with adding a new item to the cart. We will use the request’s body as the input for the new item since we are sending the request body from frontend in the same format as in our model. We could have deconstructed the request body and then provided the data while creating the new item as we did for the user but this was a cleaner way to do it.

We then save the item in the database and send the new item as the response in JSON format.

module.exports.post\_item = (req,res) => {  
 const newItem = new Item(req.body);  
 newItem.save().then(item => res.json(item));  
}

Next, we will deal with updating items. We receive updated information through the request body and the item id through the params. We will use the function *findByIdAndUpdate*to search for the item and update it with the new information. We then send the updated item as the response.

module.exports.update\_item = (req,res) => {  
 Item.findByIdAndUpdate({\_id: req.params.id},req.body).then(function(item){  
 Item.findOne({\_id: req.params.id}).then(function(item){  
 res.json(item);  
 });  
 });  
}

Lastly, we deal with the deletion of items from the database. We receive the item id through the params. Next, we find the item and delete it using findByIdAndDelete function. We then return a success response.

module.exports.delete\_item = (req,res) => {  
 Item.findByIdAndDelete({\_id: req.params.id}).then(function(item){  
 res.json({success: true});  
 });  
}

That would sum up our item controllers part. Here is the complete code for the item controller.

Item Controller

|  |
| --- |
| const Item = require('../models/Item'); |
|  |  |
|  | module.exports.get\_items = (req,res) => { |
|  | Item.find().sort({date:-1}).then(items => res.json(items)); |
|  | } |
|  |  |
|  | module.exports.post\_item = (req,res) => { |
|  | const newItem = new Item(req.body); |
|  | newItem.save().then(item => res.json(item)); |
|  | } |
|  |  |
|  | module.exports.update\_item = (req,res) => { |
|  | Item.findByIdAndUpdate({\_id: req.params.id},req.body).then(function(item){ |
|  | Item.findOne({\_id: req.params.id}).then(function(item){ |
|  | res.json(item); |
|  | }); |
|  | }); |
|  | } |
|  |  |
|  | module.exports.delete\_item = (req,res) => { |
|  | Item.findByIdAndDelete({\_id: req.params.id}).then(function(item){ |
|  | res.json({success: true}); |
|  | }); |
|  | } |

Next, we will deal with a custom middleware function to verify whether a user is logged in or not.

We will first require the config and jwt in our file. We then start making out auth middleware function.

We get the token from request’s header part named*‘x-auth-token’.*If there was no token, then we will verify the token and then send the decoded variable as the response.

We then use the *next()*function which allows us to move on to the next middleware function.

Auth middleware function

|  |
| --- |
| const config = require('config'); |
|  | const jwt = require('jsonwebtoken'); |
|  |  |
|  | function auth(req, res, next) { |
|  | const token = req.header('x-auth-token'); |
|  |  |
|  | // Check for token |
|  | if(!token){ |
|  | return res.status(401).json({ msg: 'No token, authorization denied'}); |
|  | } |
|  |  |
|  | try{ |
|  | // Verify token |
|  | const decoded = jwt.verify(token, config.get('jwtsecret')); |
|  | //Add user from payload |
|  | req.user = decoded; |
|  | next(); |
|  | } catch(e){ |
|  | res.status(400).json({ msg:'Token is not valid'}); |
|  | } |
|  | } |
|  |  |
|  | module.exports = auth; |

So, that was all about the middleware function. We have now covered everything we wanted to cover in the third part. In the fourth part, we will deal with the routes and controllers of Cart and the Order. We will handle the payments using Stripe Checkout in the next part of the series.