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

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

In the fifth part, we will set up the client-side for our application with React and will use Redux for the state management in our application.

So, this is the fifth part of the MERN Stack series. In the first four parts, we discussed the backend part of our application in complete detail — from setting up the routes to accepting payments via stripe, we did all the backend work in those four parts.

So, from the fifth part onwards, we will start focusing on the frontend part. In this part, we will start setting up the client side of our project with React and will also make use of the Redux library to manage all our state in the React application.

So, first of all, we would need to make a new folder inside our root folder (where we have all our backend files). We will name this folder as *‘client’* and we will have all files related to the client-side inside this folder.

We will make use of the *create-react-app*to set up a React project for us and so we don’t need to deal with various complex things like babel and webpack. Using this command will make the process much easier and we will be able to focus on the things that really matter.

So, after creating the folder named *client,*we would move into that folder and run the following command to create a new react app inside the folder.

npx create-react-app .

Or, if you have not yet created the *client* folder, you can just type in this command to set up a new React project in a folder named *client* and then you can move inside the *client* folder.

npx create-react-app client

This will set up a new React project in our application. We can run both the server and the client at the same time as we installed *concurrently*in part 1 of the series and defined the node script for this purpose. We just need to run npm run dev to run them both at the same time.

Now, open the *package.json*file inside the client folder. We will see that it contains various dependencies installed. We would also install some more dependencies we would be using in our project.

Here is the *package.json* file of the client-side. See there are many dependencies mentioned here. We will need all of these in our project.

So, now after you have seen the dependencies, let’s go through them one by one to see what all we installed and what function does it serves us.

1. **Axios** — Axios will be used to interact with the REST APIs and fetch the data from the server.
2. **Bootstrap**— Bootstrap will be the frontend CSS library we will be using to design the frontend.
3. **redux —**This is the state management library we will be using to manage our state.
4. **react-redux —**This is the React version of Redux which will be used to manage the state in our application. (since Redux can be used with various frameworks and libraries. This one is for React.)
5. **react-router-dom —**This will manage the routes of our application. It will help us define routes and allow them to go from one route to another.
6. **redux-thunk**— It is the middleware we will be using in our application which will help us in state management of the application.
7. **reactstrap** — It is the React version for Bootstrap styling which allows us to use Bootstrap classes as React components.
8. **react-stripe-checkout —**It is used for using Stripe for accepting payments in our application.

Also, we will set up a proxy which will allow us to transfer the request to *localhost:4000*. Since we are running our application on port number 3000 and we will not be wanting to write complete URL for interacting with the APIs so we will define a proxy to transfer requests from 3000 to 4000.

So, now we will look into the files which are present in the client folder. So, we will look into the *src*folder. We have an *index.js*file in which we do not need to make any changes. Here is the *index.js*file:-

import React from 'react';  
import ReactDOM from 'react-dom';  
import App from './App';  
  
ReactDOM.render(  
 <React.StrictMode>  
 <App />  
 </React.StrictMode>,  
 document.getElementById('root')  
);

Next, we move to look at the *App.js* file:-

import { Component } from 'react';  
import 'bootstrap/dist/css/bootstrap.min.css';  
  
class App extends Component {  
 render(){  
 return (   
 <div className="App">  
 <h1>Hello everyone!</h1>  
 </div>   
 );  
 }  
}  
  
export default App;

We have imported *bootstrap*into this file. Also, we will clear out all the pre-written code.

We would make some changes in this file later on to incorporate the Redux state management and the routing.

Next, we would start setting up our Redux state management. We will create a new file named *store.js* in the *src*folder.

So, this is our *store.js* file. It will serve as a store for our state. To understand what is going on, it is highly recommended to have some knowledge about Redux.

import { createStore, applyMiddleware, compose } from 'redux';  
import thunk from 'redux-thunk';  
import rootReducer from './reducers';  
  
const initialState = {};  
  
const middleWare = [thunk];  
  
const store = createStore(rootReducer, initialState, compose(  
 applyMiddleware(...middleWare),  
 window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()  
));  
  
export default store;

If you look closely, we have imported something we have not actually created till now, the *rootReducer*file from reducers folder. We will make that in a while.

If you do not understand what is going on here, I will recommend you first learn how Redux works to understand it in a better way. I too believe Redux is not easy.

But the good news here is, we do not need to ever visit this file again for any purpose. It is the standard way to set up a Redux store. We cannot do much about it as this is how Redux works.

Now, we will start building the *actions*folder inside the *src*folder. This is standard Redux way to have *store, actions*and *reducers.*

So, first of all, we start with the *types.js*file inside the *actions*folder. Here we will define all the action types. This is the Redux way of doing things.

export const GET\_ITEMS = 'GET\_ITEMS';  
export const ADD\_ITEM = 'ADD\_ITEM';  
export const DELETE\_ITEM = 'DELETE\_ITEM';  
export const UPDATE\_ITEM = 'UPDATE\_ITEM';  
export const ITEMS\_LOADING = 'ITEMS\_LOADING';export const CART\_LOADING = 'CART\_LOADING';  
export const GET\_CART = 'GET\_CART';  
export const ADD\_TO\_CART = 'ADD\_TO\_CART';  
export const DELETE\_FROM\_CART = 'DELETE\_FROM\_CART';export const ORDERS\_LOADING = 'ORDERS\_LOADING';  
export const GET\_ORDERS = 'GET\_ORDERS';  
export const CHECKOUT = 'CHECKOUT';export const USER\_LOADING = 'USER\_LOADING';  
export const USER\_LOADED = 'USER\_LOADED';  
export const AUTH\_ERROR = 'AUTH\_ERROR';  
export const LOGIN\_SUCCESS = 'LOGIN\_SUCCESS';  
export const LOGIN\_FAIL = 'LOGIN\_FAIL';  
export const LOGOUT\_SUCCESS = 'LOGOUT\_SUCCESS';  
export const REGISTER\_SUCCESS = 'REGISTER\_SUCCESS';  
export const REGISTER\_FAIL = 'REGISTER\_FAIL';export const GET\_ERRORS = 'GET\_ERRORS';  
export const CLEAR\_ERRORS = 'CLEAR\_ERRORS';

First of all, we have all the types related to the items, next, we have for cart, orders and then for the user and then finally for errors.

So, we would have five more files in this *actions*folder which would be — *itemActions, authActions, cartActions, orderActions*and *errorActions*.

So, we will deal with all of them one by one here. So, we would be starting with the errorActions as this is very easy and does not require interacting with the server.

errorActions

We have two functions in this actions file. We have one for returning any error we have in our application and the next is for clearing out these errors when we do not need to display them.

The first function will take a message, status and id in the function and will return them as payload with the type of GET\_ERRORS.

The next function would be to just clear the errors by sending the type as CLEAR\_ERRORS.

These will be handled in the errors reducers file we will build in the next part which will then handle the state as specified by these functions.

authActions.js

This is the most involved part of the actions file since it would handle all the authentication part. We have four functions in this part — *loadUser, register, login* and*logout*.

And we also have a helper function *tokenconfig*which would get token from local storage and set up the config to send a request using loadUser for fetching the currently logged in user details.

So now we can go through all these four functions in details now:-

1. **loadUser**— It first sets the type as USER\_LOADING to indicate that the user is currently loading. We then make a request using Axios to the API endpoint*‘/api/user’*along with the token obtained from the *tokenconfig* which will then get the result and set the payload as the data fetched from the API endpoint. The type would be set as USER\_LOADED since we have successfully loaded the user. In case of any error, we will call the returnErrors function and set the type to AUTH\_ERROR.
2. **register**— It takes in the name, email and password from the frontend and then makes these a JSON object. We then hit the API endpoint for register and pass in the data. We then receive a response and set the data received as payload and the type is set to REGISTER\_SUCCESS. We handle errors in the same way as we did in the previous function and set the error type as REFGISTER\_FAIL.
3. **login**— It works similarly as register function works. The difference is that the login function gets email and password only and then it hits the API endpoint meant for login. We get a response and set the payload as the data received from the response and set the type as LOGIN\_SUCCESS. We handle errors in the same way and set the error type as LOGIN\_FAIL.
4. **logout**— We just set the type as LOGOUT\_SUCCESS and that is all we need to do for logout.

We will handle all these responses type and their payload in the reducers file we describe in the next part.

|  |
| --- |
| import axios from 'axios'; |
|  | import { returnErrors } from './errorActions'; |
|  | import { USER\_LOADING, USER\_LOADED, AUTH\_ERROR, LOGIN\_SUCCESS, LOGIN\_FAIL, LOGOUT\_SUCCESS, REGISTER\_SUCCESS, REGISTER\_FAIL} from './types'; |
|  |  |
|  | export const loadUser = () => (dispatch, getState) => { |
|  | // User loading |
|  | dispatch({ type: USER\_LOADING }); |
|  |  |
|  | axios.get('/api/user', tokenConfig(getState)) |
|  | .then(res => dispatch({ |
|  | type: USER\_LOADED, |
|  | payload: res.data |
|  | })) |
|  | .catch(err => { |
|  | dispatch(returnErrors(err.response.data, err.response.status)); |
|  | dispatch({ |
|  | type: AUTH\_ERROR |
|  | }); |
|  | }); |
|  | } |
|  |  |
|  | export const register = ({name, email, password}) => dispatch => { |
|  | // headers |
|  | const config = { |
|  | headers: { |
|  | 'Content-Type': 'application/json' |
|  | } |
|  | } |
|  |  |
|  | //request body |
|  | const body = JSON.stringify({name, email, password}); |
|  |  |
|  | axios.post('/api/register',body,config) |
|  | .then(res => dispatch({ |
|  | type: REGISTER\_SUCCESS, |
|  | payload: res.data |
|  | })) |
|  | .catch(err => { |
|  | dispatch(returnErrors(err.response.data, err.response.status, 'REGISTER\_FAIL')); |
|  | dispatch({ |
|  | type: REGISTER\_FAIL |
|  | }); |
|  | }); |
|  | } |
|  |  |
|  | export const login = ({email, password}) => dispatch => { |
|  | // headers |
|  | const config = { |
|  | headers: { |
|  | 'Content-Type': 'application/json' |
|  | } |
|  | } |
|  |  |
|  | //request body |
|  | const body = JSON.stringify({email, password}); |
|  |  |
|  | axios.post('/api/login',body,config) |
|  | .then(res => dispatch({ |
|  | type: LOGIN\_SUCCESS, |
|  | payload: res.data |
|  | })) |
|  | .catch(err => { |
|  | dispatch(returnErrors(err.response.data, err.response.status, 'LOGIN\_FAIL')); |
|  | dispatch({ |
|  | type: LOGIN\_FAIL |
|  | }); |
|  | }); |
|  | } |
|  | // logout user |
|  | export const logout = () => { |
|  | return { |
|  | type: LOGOUT\_SUCCESS |
|  | } |
|  | } |
|  |  |
|  |  |
|  | // Setup config/headers and token |
|  | export const tokenConfig = getState => { |
|  | //Get token from local storage |
|  | const token = getState().auth.token; |
|  |  |
|  | // Headers |
|  | const config = { |
|  | headers:{ |
|  | "Content-type": "application/json", |
|  | } |
|  | } |
|  |  |
|  | if(token){ |
|  | config.headers['x-auth-token'] = token; |
|  | } |
|  |  |
|  | return config; |
|  | } |

itemActions

Here, we would handle all the actions related to the items i.e. products which we would be displaying on the website.

It has five functions which would manage fetching items, adding a new item, deleting an item, updating an item and for setting items state as loading.

Though we will not be using deleting and updating of items in our application components in this series but having everything ready is a good choice in case we need to add these later on.

*Note: You can add a separate portal for managing all items like adding them, deleting them and updating them. We only cover adding items and getting all items in the components though we have APIs, actions and reducers ready for all these tasks.*

1. **getItems** — This function is used for fetching all the items from the backend using API endpoint designed for getting them. We first set items as loading and then we reach the API endpoint to get all items. We then set the type as GET\_ITEMS and set payload as the data received as a response.
2. **addItem**— This function is used for adding a new item to the database. For this, we take in the item object through frontend forms and then send this data to the API endpoint responsible for adding the item. We then set the type as ADD\_ITEM and set payload as the data received from the response.
3. **deleteItem —**This function is used to delete an existing item from the database. It takes in the id of the item we want to delete and sends the id using a delete request to the API endpoint meant for this purpose. We then set the type as DELETE\_ITEM and the payload as the id of the item which was deleted.
4. **updateItem —**This function is used to update an existing item present in our inventory. It makes a put request to the API endpoint with the help of id and also sends in the new item object. We then set the type as UPDATE\_ITEM and set payload as the id and the data we receive as the response from the server.
5. **setItemsLoading** — This function just sets the type as ITEMS\_LOADING.

|  |
| --- |
| import axios from 'axios'; |
|  | import { GET\_ITEMS, ADD\_ITEM, DELETE\_ITEM, UPDATE\_ITEM, ITEMS\_LOADING } from './types'; |
|  | import { returnErrors } from './errorActions'; |
|  |  |
|  | export const getItems = () => dispatch => { |
|  | dispatch(setItemsLoading()); |
|  | axios.get('/api/items') |
|  | .then(res => dispatch({ |
|  | type: GET\_ITEMS, |
|  | payload: res.data |
|  | })) |
|  | .catch(err => dispatch(returnErrors(err.response.data, err.response.status))); |
|  | } |
|  |  |
|  | export const addItem = (item) => (dispatch) => { |
|  | axios.post('/api/items', item) |
|  | .then(res => dispatch({ |
|  | type: ADD\_ITEM, |
|  | payload: res.data |
|  | })) |
|  | .catch(err => dispatch(returnErrors(err.response.data, err.response.status))) |
|  | } |
|  |  |
|  | export const deleteItem = (id) => (dispatch) => { |
|  | axios.delete(`/api/items/${id}`) |
|  | .then(res => dispatch({ |
|  | type: DELETE\_ITEM, |
|  | payload: id |
|  | })) |
|  | .catch(err => dispatch(returnErrors(err.response.data, err.response.status))) |
|  | } |
|  |  |
|  | export const updateItem = (id, item) => (dispatch) => { |
|  | axios.put(`/api/items/${id}`, item) |
|  | .then(res => dispatch({ |
|  | type: UPDATE\_ITEM, |
|  | payload: Promise.all([id, res.data]) |
|  | })) |
|  | .catch(err => dispatch(returnErrors(err.response.data, err.response.status))) |
|  | } |
|  |  |
|  | export const setItemsLoading = () => { |
|  | return{ |
|  | type: ITEMS\_LOADING |
|  | } |
|  | } |

cartActions

This one handles all the actions related to the cart of any user. It has four functions which are getting the cart, adding items to cart, deleting items from the cart and set the cart status to loading.

1. **getCart** — This function is used to fetch the cart for any user. First of all, we set cart as loading. This function passes on the id as a param with the API endpoint and receives a response consisting of the cart of the user. We set the type as GET\_CART.
2. **addToCart**— This function is used to add an item to cart. It takes in the id of the user which it uses as param and also passes on the productId and quantity as the request body. We then receive a response which we assign to the payload and set the type as ADD\_TO\_CART.
3. **deleteFromCart** — This function is used to delete an item from the cart. It takes in the userId and the itemId and passes these as params to the API endpoint. We then set the type to DELETE\_FROM\_CART and set payload as the response’s data.
4. **setCartLoading** — This just sets the type as CART\_LOADING.

|  |
| --- |
| import axios from 'axios'; |
|  | import { returnErrors } from './errorActions'; |
|  | import { GET\_CART, ADD\_TO\_CART, DELETE\_FROM\_CART, CART\_LOADING } from './types'; |
|  |  |
|  | export const getCart = (id) => dispatch => { |
|  | dispatch(setCartLoading()); |
|  | axios.get(`/api/cart/${id}`) |
|  | .then(res => dispatch({ |
|  | type: GET\_CART, |
|  | payload: res.data |
|  | })) |
|  | .catch(err => dispatch(returnErrors(err.response.data, err.response.status))); |
|  | } |
|  |  |
|  | export const addToCart = (id, productId, quantity) => dispatch => { |
|  | axios.post(`/api/cart/${id}`, {productId, quantity}) |
|  | .then(res => dispatch({ |
|  | type: ADD\_TO\_CART, |
|  | payload: res.data |
|  | })) |
|  | .catch(err => dispatch(returnErrors(err.response.data, err.response.status))); |
|  | } |
|  |  |
|  | export const deleteFromCart = (userId, itemId) => dispatch => { |
|  | axios.delete(`/api/cart/${userId}/${itemId}`) |
|  | .then(res => dispatch({ |
|  | type: DELETE\_FROM\_CART, |
|  | payload: res.data |
|  | })) |
|  | .catch(err => dispatch(returnErrors(err.response.data, err.response.status))); |
|  | } |
|  |  |
|  | export const setCartLoading = () => { |
|  | return{ |
|  | type: CART\_LOADING |
|  | } |
|  | } |

**orderActions**

This action file handles all the actions related to the orders in our application. It has three functions which are used to get all orders of a user, place a new order (checkout) and set the orders as loading.

1. **getOrders** — This function first sets the orders as loading. Next, it uses the id of the user received to use it as a param in making a GET request. We then set the type as GET\_ORDERS and set payload as data received as the response.
2. **checkout** — This function is used to place an order. It receives two parameters from the components which are the id of the user and source. The source is generated from stripe checkout functions which we will deal with in detail when we actually create them in the later part. We then use the id as a param and source as request body and make a POST request. We then set the type as CHECKOUT and set the payload as response’s data.
3. **setOrdersLoading** — This function sets the order type as ORDERS\_LOADING.

|  |
| --- |
| import axios from 'axios'; |
|  | import { returnErrors } from './errorActions'; |
|  | import { GET\_ORDERS, CHECKOUT, ORDERS\_LOADING } from './types'; |
|  |  |
|  | export const getOrders = (id) => dispatch => { |
|  | dispatch(setOrdersLoading()); |
|  | axios.get(`/api/order/${id}`) |
|  | .then(res => dispatch({ |
|  | type: GET\_ORDERS, |
|  | payload: res.data |
|  | })) |
|  | .catch(err => dispatch(returnErrors(err.response.data, err.response.status))); |
|  | } |
|  |  |
|  | export const checkout = (id, source) => dispatch => { |
|  | axios.post(`/api/order/${id}`, {source}) |
|  | .then(res => dispatch({ |
|  | type: CHECKOUT, |
|  | payload: res.data |
|  | })) |
|  | .catch(err => dispatch(returnErrors(err.response.data, err.response.status))); |
|  | } |
|  |  |
|  | export const setOrdersLoading = () => { |
|  | return{ |
|  | type: ORDERS\_LOADING |
|  | } |
|  | } |

## Conclusion

So, this is all we have to deal with in this part. In the next part, we would deal with the reducers and start dealing with some components and then in the last part we would wrap up the series after dealing with the leftover components part.