Lab: E-Commerce Data Rendering using React Toolkit

Estimated time needed: 60 minutes

Introduction

In this lab, you will learn how to use Redux Toolkit to manage state throughout your entire application in such a way that it can be accessed by any component without following the hierarchy between the components. You will build a simple E-Commerce application using React and Redux where you will display a list of products with an "Add to Cart" button for each product, display the items added to the cart, and allow users to remove items from the cart. All this information is going to be globally available throughout the application using Redux Toolkit.

Learning objectives

After completing this lab, you will be able to:

- Integrate React components with Redux for state management
- Implement basic E-Commerce features such as adding items to the cart, removing items from the cart, and clearing the cart
- Practice composing multiple React components to build a cohesive user interface

Prerequisites

- Basic knowledge of React composition of components
- Intermediate knowledge of JavaScript
- Knowledge of React functional component, state management using useState hook, and Redux Toolkit

Step 1: Setting up the environment

1. From the menu on top of the lab, click the **Terminal** tab at the top-right of the window shown at number 1 in the given screenshot, and then click **New Terminal** as shown at number 2.

2. Now, write the following command in the terminal to clone the boiler template for this React application and hit Enter.

```
1. 1
1. git clone https://github.com/ibm-developer-skills-network/ecommerce_rtk.git
```

Copied!

3. The above command will create a folder, "ecommerce_rtk" under the "Project" folder. You can see the structure in the screenshot.

4. Next, you need to make sure that the path of your terminal should have cloned folder to perform certain operations for this react application. Use the below command to navigate to the **ecommerce_rtk** folder in the terminal.

```
1. 1
1. cd ecommerce_rtk
```

Copied!

5. To ensure the code you have cloned is working correctly, you need to perform the following steps:
   - Write the given command in the terminal and hit Enter. This command will install all the necessary packages to execute the application.

```
1. 1
1. npm install
```

Copied!

   - Then execute the following command to run the application and this will provide you with port number 4173.

```
1. 1
1. npm run preview
```

Copied!

6. To view your React application, click the Skills Network icon on the left (refer to number 1). This action will open the **SKILLS NETWORK TOOLBOX**. Next, click **Launch Application** (refer to number 2). Enter the port number **4173** in **Application Port** (refer to number 3) and click .

7. The output will display as shown in the given screenshot.

8. You can preserve your latest work on this lab by adding, committing, and pushing it to your GitHub repository. This ensures that even if you're not working on the task continuously, your progress will be saved, allowing you to resume from where you left off.

Note: Step 8 is optional. We will recommend you to complete task in one sitting

Step 2: Implementing ProductList component

1. Next, navigate to the **ProductList.jsx** file located in the **Components** folder of the **src** directory in your cloned **ecommerce_rtk** folder.

2. The basic structure of this component will be as shown in the screenshot.

3. Now you need to show the product list in the front end. For this you need to apply the map method within the `<ul>` tag with class name "product-list-items".

```
1
2
3
4
5
6
7
8
```

```
{products.map(product => (
  <li key={product.id} className="product-list-item">
  <span>{product.name} - ${product.price}</span>
  <button>
    Add to Cart
  </button>
  </li>
))}
```

Copied!

4. Check the output and it will be displayed as shown in the given screenshot.

Step 3: Implement Redux logic

1. You need to apply logic for Redux toolkit to ensure that when you click the Add Product button to add a product to the cart, the information of product quantity entered by you should be available globally to any component.
2. Now navigate to the **CartSlice.jsx** file located in the **Components** folder of the **src** directory in your cloned **ecommerce_rtk** folder.
3. You will see the structure as given below:

```
1
2
3
```

```
const CartSlice = ({

});
```

Copied!

4. First, initialize an empty array named **cartItems** outside **CartSlice**.

```
1
2
3
```

```
const initialState = {
  cartItems: [],
};
```

Copied!

5. Now initialize **CartSlice** with one **createSlice** Redux Toolkit function. You need to install `@reduxjs/toolkit` and `react-redux` as a third-party module. For this lab, you do not need to install it separately as it is already provided in the `package.json` file and **createSlice** is a utility function provided by Redux Toolkit, a library built on top of Redux. It simplifies the process of creating Redux slices, which are portions of the Redux state, along with associated action creators and reducers.

```
1
2
3
```

```
const CartSlice = createSlice({

});
```

Copied!

6. Ensure that **createSlice** should be imported at the top of this component.

```
1
```

```
import { createSlice } from '@reduxjs/toolkit';
```

Copied!

Step 4: Actions and reducers creation

1. Slice Creation:
   - You call **createSlice** with an object containing configuration options for your slice.
   - The configuration options include:
     - name: A string value representing the name of your slice. It's used internally by Redux Toolkit for action type prefixing and other purposes.
     - initialState: An object representing the initial state of your slice.
     - reducers: An object containing reducer functions. Each key-value pair represents a single reducer, where the key is the name of the action and the value is the reducer function.

```
1
2
3
4
```

```
5. 5
6. 6
1. const CartSlice = createSlice({
2.   name: 'cart',
3.   initialState,
4.   reducers: {
5.   }
6. });
```
Copied!

Step 5: Reducers creation and export actions

1. Inside the **reducer** object, you need to create five functions out of which two are used to handle addition and removal of products in the shopping cart, one to clear all the items at once, and other two are to increase and decrease the quantity.
   - `addItemToCart`:
     - This reducer function handles the action of adding an item to the cart.
     - It takes two parameters: **state** (current state of the slice) and **action** (the dispatched action containing the payload).
     - It first checks if the item already exists in the cart by searching for its ID within **state.cartItems**.
     - If the item exists (**existingItem** is true), it increases the quantity of the existing item in the cart by 1.
     - If the item doesn't exist in the cart, it adds the item to the **cartItems** array with a quantity of 1.

     ```
     1. 1
     2. 2
     3. 3
     4. 4
     5. 5
     6. 6
     7. 7
     8. 8
     1. addItemToCart(state, action) {
     2.       const existingItem = state.cartItems.find(item => item.id === action.payload.id);
     3.       if (existingItem) {
     4.         existingItem.quantity += 1;
     5.       } else {
     6.         state.cartItems.push({ ...action.payload, quantity: 1 });
     7.       }
     8.     },
     ```
     Copied!

   - `removeItemFromCart`:
     - This reducer function handles the action of removing an item from the cart.
     - It takes two parameters: **state** and **action**.
     - It updates the **cartItems** array by filtering out the item with the ID provided in the **action payload**.

     ```
     1. 1
     2. 2
     3. 3
     1. removeItemFromCart(state, action) {
     2.       state.cartItems = state.cartItems.filter(item => item.id !== action.payload);
     3.     },
     ```
     Copied!

   - `clearCart`:
     - This reducer function handles the action of clearing the entire cart.
     - It takes only the state parameter.
     - It sets the **cartItems** array to an empty array, effectively clearing all items from the cart.

     ```
     1. 1
     2. 2
     3. 3
     1. clearCart(state) {
     2.       state.cartItems = [];
     3.     },
     ```
     Copied!

   - `increaseItemQuantity`:
     - This reducer function handles the action of increasing the quantity of a specific item in the cart.
     - It takes two parameters: **state** and **action**.
       - `state`: This represents the current state of the Redux store. It typically includes the data relevant to the application.
       - `action`: This is an object that describes the action that occurred. Redux actions are plain JavaScript objects that must have a type property indicating the type of action being performed. Additionally, they may contain additional data necessary to carry out the action. In this case, action.payload likely contains the identifier (id) of the item whose quantity needs to be increased.
     - The function logic:
       - It finds the item in the shopping cart whose identifier (id) matches the identifier passed in the action payload.
       - If the item is found (itemToIncrease is not undefined), it increments the quantity property of that item by 1.

     ```
     1. 1
     2. 2
     3. 3
     4. 4
     5. 5
     6. 6
     1. increaseItemQuantity(state, action) {
     2.       const itemToIncrease = state.cartItems.find(item => item.id === action.payload);
     3.       if (itemToIncrease) {
     4.         itemToIncrease.quantity += 1;
     5.       }
     6.     },
     ```
     Copied!

- `decreaseItemQuantity`:
  - This reducer function handles the action of decreasing the quantity of a specific item in the cart.
  - It takes two parameters: **state** and **action**.
    - state: This represents the current state of the Redux store, typically containing all the data relevant to the application.
    - action: Similar to the previous function, it's an object describing the action being performed. It's expected to have a type property indicating the action type and may include additional data needed to carry out the action. Here, action.payload likely holds the identifier (id) of the item whose quantity needs to be decreased.
  - The function logic:
    - It attempts to find the item in the shopping cart whose identifier (id) matches the identifier provided in the action payload.
    - If the item is found (itemToDecrease is not undefined) and its quantity is greater than 1, it decrements the quantity property of that item by 1.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
1. decreaseItemQuantity(state, action) {
2.       const itemToDecrease = state.cartItems.find(item => item.id === action.payload);
3.       if (itemToDecrease && itemToDecrease.quantity > 1) {
4.         itemToDecrease.quantity -= 1;
5.       }
6.     },
```

Copied!

2. Exporting Action Creators and Reducer:
   - **createSlice** returns an object containing the generated action creators and the reducer function.
   - You can then export these action creators and the reducer function to use in your Redux store setup and throughout your application.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
1. export const {
2.   addItemToCart,
3.   removeItemFromCart,
4.   clearCart,
5.   increaseItemQuantity,
6.   decreaseItemQuantity,
7. } = CartSlice.actions;
8. export default CartSlice.reducer;
```

Copied!

You can include above code in the last of the component **CartSlice**.

Step 6: Create store.js file

1. Next, create a **store.js** file.
2. Select the **src** folder and right-click on the folder. Then select **New File** and write the name as **store.js**.
3. Inside this file perform the following operations:
   - Import **configureStore** and **Reducer**:
     - The code imports the **configureStore** function from @**reduxjs/toolkit**, used to create the Redux store.
     - It also imports the reducer function, **cartReducer**, from the **CartSlice** file, assuming that you have a slice for managing the shopping cart state defined in the file.
   - Store Configuration:
     - **configureStore** is invoked with an object containing the store configuration options.
     - The reducer property is specified as an object where each key represents a slice of state, and each value represents the corresponding reducer function.
     - In this case, the **cartReducer** is associated with the cart slice of state. This means that the state managed by the **cartReducer** will be stored under the cart key in the Redux store.
   - Other Store Configuration Options:
     - Additional store configuration options can be included in the object passed to **configureStore**.
     - For example, you can include middleware, enhancers, or other options such as devtools configuration.
   - Exporting the store:
     - Finally, the configured Redux store (store) is exported from the module, making it available for use throughout the application.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
1. import { configureStore } from '@reduxjs/toolkit';
```

```
2. import cartReducer from './Components/CartSlice';
3. const store = configureStore({
4.   reducer: {
5.     cart: cartReducer,
6.   },
7. });
8.
9. export default store;
```

Copied!

4. Now, to make this data available globally for any component in the application, you need to import the data in **main.jsx** component. For this navigate to **main.jsx** file and paste the below code in the file.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
1. import React from 'react'
2. import ReactDOM from 'react-dom/client'
3. import App from './App.jsx'
4. import './index.css'
5. import { Provider } from 'react-redux'
6. import store from './store.js'
7. ReactDOM.createRoot(document.getElementById('root')).render(
8.   <React.StrictMode>
9.  <Provider store={store}>
10.     <App />
11.     </Provider>
12.   </React.StrictMode>,
13. )
```

Copied!

- In the above code **store.js** file is imported within `<React.StrictMode>`. `<Provider>` from react-redux supplies the Redux store to all components within its hierarchy by passing store as props. This allows components, including `<App />`, to access and interact with the Redux store for state management.

Step 7: Add product and store data globally

1. In the **ProductList** component, now initialize two variables, one named **disabledProducts** using **useState** hook and the other named **dispatch** which will utilize the functionality of **useDispatch()**.

```
1
2
1. const dispatch = useDispatch();
2. const [disabledProducts, setDisabledProducts] = useState([]); // State to store disabled products
```

Copied!

- Ensure you have included the given statement at the top of the component.

```
1
2
3
1. import { useDispatch } from 'react-redux';
2. import { useState } from 'react';
3. import { addItemToCart } from './CartSlice';
```

Copied!

In the above code, **addItemToCart** is used to get the reducer function detail to dispatch which product is added to the cart to **store.js**.

2. In the **ProductList** component, implement the functionality to add the data to the cart and send the data directly in `store.js` using **useDispatch** hook. In the `<button>` tag call one function **handleAddToCart** for **onClick** event.

```
1
2
3
4
5
6
```

```
1. <button
2.     className={`add-to-cart-btn ${disabledProducts.includes(product.id) ? 'disabled' : ''}`}
3.     onClick={() => handleAddToCart(product)}
4.     disabled={disabledProducts.includes(product.id)} // Disable button if product is in disabledProducts >
5.     Add to Cart
6. </button>
```

Copied!

- This button, when clicked, invokes the `handleAddToCart` function with the product as an argument.
  - The button's appearance is dynamically determined by whether the product is included in the `disabledProducts` array, which disables the button if the product is in the array or if the product is added.
  - This functionality prevents adding duplicate items to the cart and provides visual feedback by styling the button as disabled when necessary.

3. Initialize function named as **handleAddToCart**.

```
1
2
3
4
```

```
const handleAddToCart = product => {
    dispatch(addItemToCart(product));
    setDisabledProducts([...disabledProducts, product.id]); // Mark the product as disabled
  };
```

Copied!

▼ Click here for ProductList.jsx code

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
```

```
import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { addItemToCart } from './CartSlice';
import './ProductList.css'; // Import CSS file for component-specific styles

const ProductList = () => {
  const dispatch = useDispatch();
  const [disabledProducts, setDisabledProducts] = useState([]); // State to store disabled products

  const products = [
    { id: 1, name: 'Product A', price: 60 },
    { id: 2, name: 'Product B', price: 75 },
    { id: 3, name: 'Product C', price: 30 },
  ];

  const handleAddToCart = product => {
    dispatch(addItemToCart(product));
    setDisabledProducts([...disabledProducts, product.id]); // Mark the product as disabled
  };

  return (
    <div className="product-list">
      <h2 className="product-list-title">Products</h2>
      <ul className="product-list-items">
        {products.map(product => (
          <li key={product.id} className="product-list-item">
            <span>{product.name} - ${product.price}</span>
            <button
              className={`add-to-cart-btn ${disabledProducts.includes(product.id) ? 'disabled' : ''}`}
              onClick={() => handleAddToCart(product)}
              disabled={disabledProducts.includes(product.id)} // Disable button if product is in disabledProducts
            >
              Add to Cart
            </button>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default ProductList;
```

Copied!

Step 8: Display products in shopping cart

1. Now in **ShoppingCart.jsx** component, you will create logic that shows items that user have added to the shopping cart. It uses a special tool called Redux to manage the cart using redux and keep track of what the user is buying. The component lets the user see the items in the cart, their prices, and how many of

each item the user has added. Users can also remove items from the cart or change the quantity of each item. It's like having a virtual shopping basket that helps the user to keep track of what they will purchase.

2. For this to implement navigate to **ShoppingCart.jsx** component under src folder.

3. The basic structure of this component will be as shown in the screenshot.

4. Implement the given functionalities:-

   - *Import*: The component imports necessary dependencies: React, useDispatch, useSelector from react-redux, and action creators (removeItemFromCart, clearCart, increaseItemQuantity, decreaseItemQuantity) from the CartSlice.

```
1
2
3
import { useDispatch, useSelector } from 'react-redux';
import { removeItemFromCart, clearCart, increaseItemQuantity, decreaseItemQuantity } from './CartSlice'; // Assuming you have action
import './ShoppingCart.css';
```
Copied!

   - *Function Component*: The ShoppingCart component is a functional component declared using the arrow function syntax.

   - *Redux Hooks*: The component uses **useDispatch** and **useSelector** hooks from react-redux to interact with the Redux store. **useDispatch** is used to dispatch actions, and **useSelector** is used to extract data from the Redux store.

   - *State Retrieval*: **cartItems** variable retrieves the array of items from the Redux store's state by selecting **state.cart.cartItems**. **totalAmount** calculates the total amount by iterating through **cartItems** and multiplying each item's price by its quantity, then summing them up.

```
1
2
3
const dispatch = useDispatch();
  const cartItems = useSelector(state => state.cart.cartItems);
  const totalAmount = cartItems.reduce((total, item) => total + item.price * item.quantity, 0);
```
Copied!

     Include above code before return of the fucntion component.

   - *Event Handlers*: **handleRemoveItem** dispatches the **removeItemFromCart** action with the ID of the item to be removed. **handleClearCart** dispatches the **clearCart** action to clear all items from the cart. **handleIncreaseQuantity** dispatches the **increaseItemQuantity** action to increase the quantity of a specific item. **handleDecreaseQuantity** dispatches the **decreaseItemQuantity** action to decrease the quantity of a specific item.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
const handleRemoveItem = itemId => {
    dispatch(removeItemFromCart(itemId));
  };

  const handleClearCart = () => {
    dispatch(clearCart());
  };

  const handleIncreaseQuantity = itemId => {
    dispatch(increaseItemQuantity(itemId));
  };

  const handleDecreaseQuantity = itemId => {
    dispatch(decreaseItemQuantity(itemId));
  };
```
Copied!

   - *Rendering*: The component renders a shopping cart UI, listing each item in the cart along within `<ul>` tag with class name **cart-items** with its name, price, quantity controls (buttons to increase or decrease quantity), and a remove button for each item. The total amount is displayed below the cart if it is greater than 0.

```
1
2
3
4
5
6
7
8
9
10
11
{cartItems.map(item => (
            <li key={item.id} className="cart-item">
              <span>{item.name} - ${item.price}</span>
              <div className="quantity-controls">
                <button className="quantity-control-btn" onClick={() => handleDecreaseQuantity(item.id)}>-</button>
                <span> {item.quantity}</span>
                <button className="quantity-control-btn" onClick={() => handleIncreaseQuantity(item.id)}>+</button>
              </div>
```

```
              <button className="remove-item-btn" onClick={() => handleRemoveItem(item.id)}>Remove</button>
            </li>
          ))}
```
Copied!

- *Button Controls*: Quantity controls (- and + buttons) are provided to decrease or increase the quantity of each item. Clicking the - button invokes **handleDecreaseQuantity** with the item's ID. Clicking the + button invokes **handleIncreaseQuantity** with the item's ID.
- A button labeled **Clear Cart** is provided to remove all items from the cart when clicked. It triggers the handleClearCart function.

```
1
<button className="clear-cart-btn" onClick={handleClearCart}>Clear Cart</button>
```
Copied!

  - if products have been added then only display totalAmount else renders nothing.

```
1
<div>{totalAmount ? <div>'The total amount is {totalAmount}</div> : ''}</div>
```
Copied!

  - Let's break it down:
    - The outermost div element contains an expression inside curly braces {}.
    - Inside the expression, there's a ternary operator (condition ? expression1 : expression2) used for conditional rendering in JSX.
    - If totalAmount is truthy, a nested div element is rendered. This nested div contains a string 'The total amount is {totalAmount}', where {totalAmount} is intended to be the value of the totalAmount variable interpolated into the string.
    - If totalAmount is falsy, an empty string is rendered.
    - The result of the ternary operation is rendered inside the outer div element.

Click below to view code of **CartSlice.jsx**.

▼ Click here for code of CartSlice.jsx

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
```

```
import { createSlice } from '@reduxjs/toolkit';
const initialState = {
    cartItems: [],
  };
const CartSlice = createSlice({
    name: 'cart',
    initialState,
    reducers: {
        addItemToCart(state, action) {
            const existingItem = state.cartItems.find(item => item.id === action.payload.id);
            if (existingItem) {
              existingItem.quantity += 1;
            } else {
              state.cartItems.push({ ...action.payload, quantity: 1 });
            }
          },
```

```
          removeItemFromCart(state, action) {
            state.cartItems = state.cartItems.filter(item => item.id !== action.payload);
          },
          clearCart(state) {
            state.cartItems = [];
          },
          increaseItemQuantity(state, action) {
            const itemToIncrease = state.cartItems.find(item => item.id === action.payload);
            if (itemToIncrease) {
              itemToIncrease.quantity += 1;
            }
          },
          decreaseItemQuantity(state, action) {
            const itemToDecrease = state.cartItems.find(item => item.id === action.payload);
            if (itemToDecrease && itemToDecrease.quantity > 1) {
              itemToDecrease.quantity -= 1;
            }
          },


    }
});
export const {
    addItemToCart,
    removeItemFromCart,
    clearCart,
    increaseItemQuantity,
    decreaseItemQuantity,
  } = CartSlice.actions;
  export default CartSlice.reducer;
```

Copied!

Click below to view code of **ShoppingCart.jsx** component.

▶ Click here for code of ShoppingCart.jsx

Step 9: Check the output

1. Stop the execution of the React application in the terminal by performing `ctrl+c` to quit.
2. Then, write the given command in the terminal and hit Enter.

```
1
npm run preview
```

Copied!

3. To view your React application, refresh the already opened webpage for the React application on your browser. If it is not open, then click the Skills Network icon on the left panel. This action will open the "SKILLS NETWORK TOOLBOX." Next, select "Launch Application". Enter the port number **4173** in "Application Port" and click .
4. The output will display as per the given screenshot after adding products to the cart.
5. Add one more product and you will see the change in the total amount.

> Please note that the `Add to Cart` button can be used only once to add a product. After this, it will be disabled and won't add the same product if you click on it again.

Note- To see the latest changes, you need to execute `npm run preview` again in the terminal.

Congratulations! You have created an E-Commerce Data Rendering React application!

Step 10: Practice Task

1. Now in this practice exercise you will create one more component where you will implement the concept of super coin.
    - Super coins are a form of loyalty or reward points offered by some e-commerce platforms or retailers as part of their customer loyalty programs. To see how much user have earned based on total cart amount you need to create this functionality.
2. For this create one Super Coin Component named `SuperCoin.jsx` by right clicking on the **Components** folder after selecting it.
3. Now initialize **superCoins** variable using **useState** hook along with its corresponding function before retun of the component.

   Hint: use **useState** hook to initialize variable with `0`.

▼ Click here for the sample solution

```
1
```

```
const [superCoins, setSuperCoins] = useState(0);
```

Copied!

4. Now you need to retrieve the cartItems from the cart slice of the Redux store's state to get the total quantity of number of products using the useSelector hook.

   Hint: use **useSelector** hook to get the state of cart items.

▼ Click here for the sample solution

```
1
```

```
const cartItems = useSelector(state => state.cart.cartItems);
```

Copied!

5. Next calculate the total amount by summing the product of the price and quantity for each item in the `cartItems` array using the **reduce** method.

   Hint: use **useSelector** hook to get the state of cart items.

▼ Click here for the sample solution

```
1
```

```
const totalAmount = cartItems.reduce((total, item) => total + item.price * item.quantity, 0);
```

Copied!

6. Now you need to update the `superCoins` state based on the totalAmount: setting it to 10, 20, or 30 coins for different ranges of the total amount, and to 0 if the amount is below 100. This effect runs whenever the totalAmount changes.

   Hint: use **useEffect** hook to update the state of superCoins.

▼ Click here for the sample solution

```
1
2
3
4
5
6
7
8
9
10
11
```

```
useEffect(() => {
  if (totalAmount >= 100 && totalAmount < 200) {
    setSuperCoins(10);
  } else if (totalAmount >= 200 && totalAmount < 300) {
    setSuperCoins(20);
  } else if (totalAmount >= 300) {
    setSuperCoins(30);
  } else {
    setSuperCoins(0);
  }
}, [totalAmount]);
```

Copied!

7. Now create `<div>` within return of function component using jsx syntax and integrate the `superCoins` variable state within `<div>` tag.

   Hint: Use {} to include superCoins variable

▼ Click here for the sample solution

```
1
2
3
4
```

```
<div className="super-coins" style={{textAlign:'center'}}>
  <h2 className="super-coins-title">Super Coins</h2>
  <p className="super-coins-info">You will earn {superCoins} super coins with this purchase.</p>
</div>
```

Copied!

8. Check the output
   - Save the changes and re-run the application.
   - Add the product in the cart and when it reaches to **$100** amount it will display that you have earned **10 Super Coins**.

9. You can increase the amount and depending upon the logic it will also increase supercoins value.

Conclusion

Congratulations! You have created an E-Commerce React application!

In this lab, you have:

- Implemented Redux Toolkit for universal state management across the application.
- Developed a basic e-commerce platform using React and Redux.
- Featured a product list with an “Add to Cart” button for each item.
- Enabled users to view and manage items added to the cart, including removing items.
- Utilized the useDispatch and useSelector hooks to interact with Redux, providing global data accessibility.
- Ensured seamless state management throughout the application, enhancing user experience and scalability.

Author(s)

Richa Arora