- ⏱ 3 hours

Grocery Store App

In this project, you'll connect Redux to a React application. You''ll configure a Redux store, create Redux reducers and action creators, and use them in a Grocery Store application to add produce to a shopping cart. You'll also learn how to normalize data in a Redux store.

When you finish today's project, your application should have the following features:

- Display all the produce in a list
- Add produce to a cart
- Increment/decrement the quantity of produce in the cart
- Checkout the cart

Phase 0: Familiarize yourself with the code

Clone the starter repository at https://github.com/appacademy-starters/grocery-store.
Run `npm install` then `npm start` to start the development server.
Navigate to http://localhost:3000. You should see the text "Grocery Store". If you click on the "Checkout" button in the navigation bar, it should open a side menu. Since there is no produce listed, you cannot add items to the shopping cart yet.
The `Cart`, `ProduceList` and `ProduceDetails` are the components you need to connect to the Redux store. Take a look at the code there to familiarize yourself with it.
The `src/mockData/produce.json` is the mock data that you will use to populate the Redux store with. The file has a JSON array of objects. Each object has the following structure:

```
{
  id: ID,
  name: String,
  liked: Boolean
}
```

In your preview of the code, make sure to get understand how the components are all connected (parent-child relationships of all the components, the component tree).

Phase 1: Set Up

Install dependencies

`npm install` the following dependencies:

- `redux` - the Redux package
- `react-redux` - for connecting Redux to React

`npm install -D` the following dev-dependencies:

- `redux-logger` - a debugging tool for logging all actions dispatched

Configure the Redux store

First thing you need to do is set up and configure the Redux store.

Create a folder called `store` inside of `src`. Inside of that folder, create a file called `index.js`. In this file, you will define the root reducer and a function that will return a Redux store.
Import `createStore`, `combineReducers`, `applyMiddleware`, and `compose` from redux.

```
// ./src/store/index.js
import { createStore, combineReducers, applyMiddleware, compose }
from "redux";
```

- `createStore` - creates a Redux store
- `combineReducers` - creates one reducer from multiple reducers as slices of state
- `applyMiddleware` - a store enhancer that will allow you to attach middlewares (a middleware is a function called before any action hits the root reducer)
- `compose` - another store enhancer that will allow you to use more than one store enhancer

Next, define the `rootReducer` which will be the root reducer for the Redux store.
The `rootReducer` will just be the return of `combineReducers` invoked with an empty object for now.

```
// ./src/store/index.js
// ...
const rootReducer = combineReducers({

});
```

Now, you are going to create a store enhancer that will be set only when your application is in development. When in development, you want to add the `redux-logger` middleware and the Redux dev tools extension store enhancer to your store.
For `redux-logger`, you will use `applyMiddleware` which returns a store enhancer. The store enhancer for the Redux dev tools extension is set by the extension to a global

property, `window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__`. This store enhancer combines other store enhancers into one, just like `compose` from `redux` does.
Set the `enhancer` variable when in development to use these enhancers like so:

```
// ./src/store/index.js
// ...
let enhancer;

if (process.env.NODE_ENV !== "production") {
  const logger = require("redux-logger").default;
  const composeEnhancers =
    window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;
  enhancer = composeEnhancers(applyMiddleware(logger));
}
```

Let's break this down.

The `logger` variable is set to the `default` export of the imported `redux-logger` package using `require` instead of ES6 module imports. Why? ES6 module imports need to be defined at the very top of the file in the outermost scope and will be loaded in all node environments. However, `redux-logger` is a development dependency and cannot be loaded in production. Loading it with `require` is needed if you only want it in a certain node environment.
The `composeEnhancers` variable is set to the Redux dev tools extension's store enhancer, `window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__` or the `compose` function from `redux` if the extension fails to load or if you don't have the extension installed in your browser.
The Redux `compose` function takes in store enhancers as its arguments and combines them to create a single store enhancer. The only other store enhancer that you need is the `applyMiddleware` invoked with the loaded `logger` `enhancer` is set to the combined store enhancer.
Finally, define a function called `configureStore` that will take in a `preloadedState` and return the result of `createStore` invoked with the `rootReducer`, the `preloadedState`, and the `enhancer`. Export the `configureStore` function as the default export.

```
// ./src/store/index.js
// ...
const configureStore = (preloadedState) => {
  return createStore(rootReducer, preloadedState, enhancer);
};

export default configureStore;
```

Provide the Redux store

Now, you need to wrap the React application with the Redux store provider.

Import the `configureStore` function into the entry file, `src/index.js`. Import the `Provider` component from `react-redux`.
Initialize a variable called `store` and set it to the return of `configureStore` (a `preloadedState` does not need to be passed in, it can be `undefined`).
In the `Root` function component, wrap the `BrowserRouter` component with the `Provider` component. Pass the prop of `store` with the value of `store` into the `Provider`.

Your entry file should look like this:

```
// ./src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-router-dom';
import { Provider } from 'react-redux';
import configureStore from './store';
import './index.css';
import App from './App';

const store = configureStore();

function Root() {
  return (
    <Provider store={store}>
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </Provider>
  );
}

ReactDOM.render(
  <React.StrictMode>
    <Root />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Test your setup

Let's test your setup!

Navigate to http://localhost:3000 and open up the browser's dev tools. In the console, you should see a message about an invalid reducer. That's because the `rootReducer` isn't combining any reducers yet. Ignore this warning for now.

Open the "Redux" tab in your browser's dev tools. You should not see the message "No store found".

In your entry file, expose the `store` variable on the `window` **only in development** for testing purposes.

```
// ./src/index.js
// ...
if (process.env.NODE_ENV !== "production") {
  window.store = store;
}
```

Navigate to http://localhost:3000 and refresh. You should see the `store` on the `window` in your browser's dev tools console. If you call `store.getState()` then you should just get back an empty object.

If you are having issues while testing, check your syntax in your store and entry files.

Commit your setup code!

Phase 2: Displaying a list of Produce

To display a list of produce, you need to first store the list of produce in the Redux store. Then you need to extract the produce from the store for display in a React component.

Produce Reducer

First, you need to create space in the Redux store for the produce. To do this, you will add a key in the Redux store for handling the produce information. This key will be the produce slice of state.

In your store file, `src/store/index.js`, add a key of `produce` to the `rootReducer` with the value of `produceReducer`:

```
// ./src/store/index.js
// ...
const rootReducer = combineReducers({
  produce: produceReducer
});
```

The `produceReducer` is responsible for handling the produce information in the store, or the produce slice of state.
Now, you need to define the `produceReducer`. Create a file in the `store` folder called `produce.js`. Define a function called `produceReducer` with `state` and `action` as parameters. The `state` should default to an empty object. Add a `switch/case` statement on `action.type` inside of the function and return the state as the `default` case. Export the `produceReducer` as default.

```js
// ./src/store/produce.js
export default function produceReducer(state = {}, action) {
  switch (action.type) {
    default:
      return state;
  }
}
```

This is the structure of how all reducer functions should look like by convention. The reducer should return the old state or a new state depending on the `type` of the `action` that gets dispatched. The `state` does not have to be an object. It could be an array, boolean, etc.

The produce slice of state should be an object with produce id's as keys. The values should be objects of produce information.

Import the `produceReducer` into the store file, `store/index.js`.

Test the reducer

Now, test the reducer by refreshing at http://localhost:3000. The state in the Redux dev tools extension should show a key of `produce` with the value of an empty object! Also, if you call `store.getState()` in the console, you'll get an object with a key of produce with an empty object as the value.

Populate the produce slice of state

You will use the mock data in the `src/mockData/produce.json` file to add produce to the produce slice of state. To populate the produce slice of state, you need to trigger a change to the Redux store state. This can only be done by dispatching an action.
In the `store/produce.js` file, create and export an action creator called `populateProduce`. This function should return an object with a unique `type` key.
Create a constant called `POPULATE` and set it to a string literal of `produce/POPULATE`. Set the key of `type` in the return of `populateProduce` to this constant.
Import the `produce.json` as `produceData`. Set a key of `produce` in the return of `populateProduce` to `produceData`. The key of `produce` is a payload key.

If you try dispatching the `populateProduce` action right now, the Redux store state will not change. Whenever you create a new action type, you need to add a case for it in reducer that handles the slice of state that needs to be updated.
In the `produceReducer` add a case for the `action.type` of `POPULATE`. This case should turn the `action.produce` array into an object and return the object. `action.produce` is an array of produce data objects. The object returned should have keys set to the id's of the produce data objects with their values as the respective produce data objects.
The `action.produce` array should look something like this:

```
[
  { id: 1, name: "Longos - Greek Salad", liked: false },
  { id: 2, name: "Juice - Lagoon Mango", liked: false },
  { id: 3, name: "Lamb - Whole, Frozen", liked: false },
  // ...
]
```

The object created from the array should look something like this:

```
{
  1: { id: 1, name: "Longos - Greek Salad", liked: false },
  2: { id: 2, name: "Juice - Lagoon Mango", liked: false },
  3: { id: 3, name: "Lamb - Whole, Frozen", liked: false },
  // ...
}
```

Try attempting this before looking below.

```
// ./src/store/produce.js
import produceData from '../mockData/produce.json';

const POPULATE = 'produce/POPULATE';

export const populateProduce = () => {
  return {
    type: POPULATE,
    produce: produceData
  };
};

export default function produceReducer(state = {}, action) {
  switch (action.type) {
    case POPULATE:
      const newState = {};
      action.produce.forEach(produce => {
        newState[produce.id] = produce;
      });
      return newState;
    default:
      return state;
  }
}
```

This conversion from an array into this object structure is known as **normalizing data**. It's faster to search for a produce by its id in this object structure than in an array. Normalizing data is a common practice even outside of Redux, so practice getting comfortable with it!

Testing `populateProduce` action

To test the `populateProduce` action, import it into the entry file and attach it to the `window` just like you did with the `store`.

```
// ./src/index.js
// ...
import { populateProduce } from './store/produce';
// ...
if (process.env.NODE_ENV !== "production") {
  window.store = store;
  window.populateProduce = populateProduce;
}
```

Go to http://localhost:3000 and use `store.dispatch` on the browser's dev tools console to dispatch the action from the `populateProduce` action creator on the `window`.

```
store.dispatch(populateProduce());
```

As a response to dispatching this action, the `redux-logger` middleware should aesthetically display the previous state, action, and the next state in the console. The Redux dev tools extension should now show the action dispatched and the changed state.

The new state should look something like this:

```
{
  produce: {
    1: { id: 1, name: "Longos - Greek Salad", liked: false },
    2: { id: 2, name: "Juice - Lagoon Mango", liked: false },
    3: { id: 3, name: "Lamb - Whole, Frozen", liked: false },
    // ...
  }
}
```

To debug the action, you can add `console.log`'s and/or `debugger`'s in the action creator and reducer. To hit them, try refreshing and dispatching the action in the console again. Even if your state looks the way you expect it to look, try adding `console.log` and/or `debugger` into the action creator and reducer to examine how and when they are invoked.

Programmatically dispatch `populateProduce`

To programmatically dispatch the `populateProduce` action instead of doing it through the browser console, you will dispatch this action when the `App` component gets loaded. To get the `dispatch` method in a React component, import and use the `useDispatch` hook from `react-redux` in the `App` component.
Import the `populateProduce` action creator into the `App` component file.
Use the `useEffect` hook to dispatch the action when the `App` component first loads.
Your `App` component should look something like this:

```
// ./src/App.js
// ...
import { useDispatch } from 'react-redux';
import { populateProduce } from './store/produce';

function App() {
  const dispatch = useDispatch();
  useEffect(() => {
    dispatch(populateProduce());
  }, []);
  // ...
}
```
Reload http://localhost:3000. You should see the Redux store state updated automatically with the populate produce action!
You may have noticed a warning in the React server logs or in the browser's console. "React Hook useEffect has a missing dependency: 'dispatch'."
Add `dispatch` into `useEffect`'s dependency array. The `dispatch` function should never change across re-renders so `useEffect` should only dispatch the action once.

Display the produce in a list

So far, you added the produce mock data into the Redux store state but nothing changed on your actual React application. To display the produce, you need the `ProduceList` component to access the produce slice of state from the Redux store.
In the `ProduceList` component file, import `useSelector` from `react-redux`. `useSelector` accepts a function as a parameter and will pass the updated state into the function whenever the state gets updated. The return value of the function will be the return of the `useSelector` function. In the component, set the `produce` variable to the return of `useSelector` and pass in a function that returns the `produce` slice of state from the updated state.
```
const produce = useSelector(state => state.produce);
```
The `ProduceList` component will turn the products into an array of `ProduceDetails` components to be rendered.
If you refresh the http://localhost:3000 now, the produce should be displayed under "All Produce"!

Commit your code!

Phase 3: Add produce to the cart

When the "Checkout" button in the navigation bar is clicked, the side bar containing the shopping cart should open. Let's try adding produce to the shopping cart.

The shopping cart is a separate entity from the produce list. The shopping cart needs to know which produce and how much of it was added to the cart. You need a cart slice of state that will hold an object with produce id's as its keys and objects with a `count` property as its values.

Cart Reducer

Create a `cart.js` file in `src/store` and define and export a `cartReducer` from this file. Add the `cart` slice of state with this as the reducer in the store file (`src/store/index.js`). Use the `produceReducer` as an example if you get stuck. Test the reducer in the same way as the way you tested the `produceReducer`.

The store state should now look something like this:

```
{
  cart: {},
  produce: {
    // ...
  }
}
```

Add to cart action

Create the necessary Redux entities for adding produce to the cart.

The Redux store state should look something like this when a produce with an id of 3 is added.

```
{
  cart: {
    3: {
      id: 3,
      count: 1
    }
  },
  produce: {
    // ...
  }
}
```

Define an action creator for adding an item with a specified id to the cart. The action creator should have the produce's `id` as the parameter and should return it as a payload key.
In the reducer, add a case for the action type of adding an item with the id in the action payload. The state returned should be a new object that includes all the keys of the previous state plus with the desired item id as a new key. The value of that key should be an object with properties of `id` and `count` (set to 1).

Hint: To create a new object that includes all the keys of another object with additional keys:

```
const newObj = {
  ...oldObj,
  newKey: { property: "value" }
};
```

Test the action creator in the same way that you tested the `populateProduce` action.

Connect the plus button to the add to cart action

In the `ProduceList` component, all the produce in the Redux store is mapped to a array of `ProduceDetails` components. The `ProduceDetails` component has a button element with a `className` of "plus-button". When that button is clicked, dispatch the add to cart action with the id of the produce in that component.
Test this out by navigating to http://localhost:3000 and clicking the plus button on any of the produce in the list. Check to see if the new Redux store state has that produce in the cart slice of state.

Note: If you open the cart by clicking the "Checkout" button right now, you will not see the produce that you added in there. That's because you didn't connect the cart to read the cart slice of state yet.

Display the cart items

Display the produce in the cart slice of state in the cart side bar.

The `Cart` component renders the list of items in the cart side bar. Set the `cart` variable to the extracted cart slice of state in the Redux store state (take a look at the `ProduceList` component to refresh yourself on how to access information from the Redux store).
If you refresh http://localhost:3000 and add a produce to the cart, you should see something in the cart now!

There is something in the cart, but the item can't be identified. The name of the produce should be displayed next to the count. The `name` of the produce is stored in the `produce` slice of state, but the `Cart` component only has the `cart` slice of state. Extract the `produce` from the state and set it to the `produce` variable.
The `cartItems` array should return an array of produce items with information combined from both the cart and the produce slices of state.

```
  const cartItems = Object.values(cart)
    .map(item => {
      return {
        ...item,
        ...produce[item.id]
      };
    });
```

Now if you refresh the page, you should see the cart items with names. Hooray! We're almost done.

This is also a good time to commit. Every time you finish a phase, or a chunk of working code, don't forget to commit!

Phase 4: Remove produce from the cart

Add the functionality to remove produce from the cart when the "Remove" button is clicked.

Phase 5: Update the number of a produce in the cart

When the plus button next to a cart item is clicked, increment the count of the cart item by 1. When the minus button next to a cart item is clicked, decrement the count of the cart item by 1.

When the number in the input field next to a cart item is changed and clicked off of, it should update the count for that cart item in the Redux store state.

If clicking the minus button or changing the number in the input field ever updates to a count lower than 1, remove the cart item.

When the plus button next to a produce in the produce list is clicked, the item should be added to the cart if it's not in the cart. If it is already in the cart, increment the count of the respective cart item by 1.

Phase 6: Purchase produce

When the "Purchase" button is clicked in the cart, it should empty the cart.

Phase 7: Like/Unlike a Produce

When the like button next to a produce in the produce list is clicked, then the boolean of `liked` on the produce in the Redux store produce slice of state should be toggled to `true`/`false`.

Phase 8: Redux selectors

Instead of creating a different selector function every time you use the hook, `useSelector`, for the same purpose (e.g. extracting the `produce` slice of state), you can define a single function to be used anytime you need the same information from the Redux store state.
For example, to get an array of all the produce in the Redux store state, you can use the following selector function in the `src/store/produce.js` file:

```
// ./src/store/produce.js
export const getAllProduce = (state) =>
Object.values(state.produce);
```

Import this function into the `ProduceList` component and remove the following lines:

```
  const produce = useSelector(state => state.produce);

  const produceArr = Object.values(produce);
```

Replace it with this line:

```
const produceArr = useSelector(getAllProduce);
```

This may seem like it's much more work than just defining the selector function inside the component, but it will save you the time from having to refactor all your code in all the components if you change the structure of your state. Instead, you can just refactor the one file that holds all your Redux actions, reducer, and selector functions.

You don't need to use selector functions. It's up to you whether or not you want to use them.

Practice defining your own selector functions by refactoring the code to use selector functions only.

Note: For the `Cart` component, try making a single selector function that will extract the cart items with the cart and produce information (including the `name`).

Bonus Phase: Keep the order of added produce to the cart

Reconfigure your cart reducer to maintain the order of the added produce to the cart. For example, if you add produce with an id of 3 then produce with an id of 1 to the shopping cart, the produce with the id of 3 should come *before* the produce with the id of 1 in the shopping cart.

Hint: Store an object and an array in your cart slice of state. If you did the selector functions correctly, you only need to update the Redux files. No need to change any of the React components.

Bonus Phase: Programmatically open the cart side bar

For an added bonus, try to configure the application to open the cart side bar whenever there is a new item added to the cart.

Did you find this lesson helpful?
No
Yes