Course › Redux › Modul... › Modul...

Audit Access Expires Jun 20, 2020
You lose all access to this course, including your progress, on Jun 20, 2020.

Module 2 Tutorial Lab

Restaurant Management System using React Redux part 1

In this tutorial lab, we will be state logic behind a restaurant management system using Redux. We will the tie the state logic to React components in Module 3's tutorial lab. The finished application should look like this:

ACTUAL FINISHED: https://codesandbox.io/s/lr8nn59loz REDUX FINISHED: https://codesandbox.io/s/934rx9q00p

Here are a list of features that our application will have:

- User can select a table. If the user selects an already selected table, it will deselect it.
- User can check in/ check out a selected table. Checking out should clear the table items and add the bill total to the total money earned value.
- User can add items to the selected table if it is checked in
- User can delete items from selected table if it is checked in

The user should also be able to see:

- Currently selected table
- Total number of tables available to be checked in.
- Which tables are checked in / checked out
- All the ordered items from the selected table

- The total bill for the selected table
- The total amount of money earned from checked out tables

Step 1. Setting up your project

In this step, we will set up the project structure so that you can keep your project organized. A completed example of this step can be found here : https://codesandbox.io/s/zw3jzp9y54

1. Go to https://codesandbox.io/s/new to create a new React project.
2. Fork the project so that you can save it to your account.
3. Give the project a title such as “Restaurant Management System”
4. Add `redux` as a dependency.
5. Create a new folder inside the `src` folder. Name it `actions`.
6. Create a new folder inside the `src` folder. Name it `reducers`.
7. Create a new folder inside the `src` folder. Name it `constants`.

Step 2. Designing the State tree

In this step, we will figure out how the state of our application should look.

In order to implement all the features, we will need to keep track of the following state:

- Currently selected table
- Status of all tables (checked in or checked out)
- What orders have been placed for each of the tables
- Total money earned from checked out tables

We can model this using the following state tree:

```
{
  selectedTable: id,
  tableStatusData: array,
  tableData: 2D array,
  moneyEarned: num
}
```

Step 3. Designing the Actions

In this step, we will design the actions for our application. A completed example of this step can be found here : https://codesandbox.io/s/zw3jzp9y54

Here is a list of allthe actions we will need to create based off our application features and our state tree:

- SELECT_TABLE - Select a table
- TOGGLE_TABLE - Check in/ check out a table
- INCREMENT_MONEY_EARNED - Add money to the total amount of money earned
- ADD_TABLE_ITEM - Add item to the selected table
- DELETE_TABLE_ITEM - Delete items from the selected table

1. Inside the `constants` folder, create a new file named `constants.js`.
2. Copy the following inside `constants.js`:

```
const SELECT_TABLE = "SELECT_TABLE";
const TOGGLE_TABLE = "TOGGLE_TABLE";
const INCREMENT_MONEY_EARNED = "INCREMENT_MONEY_EARNED";
const ADD_TABLE_ITEM = "ADD_TABLE_ITEM";
const DELETE_TABLE_ITEM = "DELETE_TABLE_ITEM";

module.exports = {
  SELECT_TABLE,
  TOGGLE_TABLE,
  INCREMENT_MONEY_EARNED,
  ADD_TABLE_ITEM,
  DELETE_TABLE_ITEM
};
```

This file contains all the constants that will be used in our actions.

3. Inside the `actions` folder, create a new file named `addTableItem.js`.
4. Copy the following inside `addTableItem.js`:

```
import { ADD_TABLE_ITEM } from "../constants/constants.js";

const addTableItem = (name, price, id) => {
  return {
    type: ADD_TABLE_ITEM,
    tableId: id,
    item: {
      name: name,
      price: price
    }
  };
};

export default addTableItem;
```

The addTableItem action creator will create a ADD_TABLE_ITEM action that contains a tableId to indicate which table to add to and an item object with a name and a price to indicate what to add.

5. Inside the `actions` folder, create a new file named `deleteTableItem.js`.
6. Copy the following inside `deleteTableItem.js`:

```
import { DELETE_TABLE_ITEM } from "../constants/constants.js";

const deleteTableItem = (tableId, id) => {
  return {
    type: DELETE_TABLE_ITEM,
    tableId: tableId,
    id: id
  };
};

export default deleteTableItem;
```

The deleteTableItem action creator will create a DELETE_TABLE_ITEM action that contains a tableId to indicate which table to delete from and an id to indicate which item to delete.

7. Inside the `actions` folder, create a new file named `incrementMoneyEarned.js`.
8. Copy the following inside `incrementMoneyEarned.js`:

```
import { INCREMENT_MONEY_EARNED } from "../constants/constants.js";

const incrementMoneyEarned = amount => {
  return {
    type: INCREMENT_MONEY_EARNED,
    amount: amount
  };
};

export default incrementMoneyEarned;
```

The incrementMoneyEarned action creator will create an INCREMENT_MONEY_EARNED action that contains the amount to add to the total amount of money earned.

9. Inside the `actions` folder, create a new file named `selectTable.js`.
10. Copy the following inside `selectTable.js`:

```
import { SELECT_TABLE } from "../constants/constants.js";

const selectTable = id => {
  return {
    type: SELECT_TABLE,
    id: id
  };
};

export default selectTable;
```

The selectTable action creator will create a SELECT_TABLE action that contains the id of the table to select.

11. Inside the `actions` folder, create a new file named `toggleTable.js`.
12. Copy the following inside `toggleTable.js`:

```
import { TOGGLE_TABLE } from "../constants/constants.js";

const toggleTable = id => {
  return {
    type: TOGGLE_TABLE,
    id: id
  };
};

export default toggleTable;
```

The toggleTable action creator will create a TOGGLE_TABLE action that contains the id of the table to check in/check out.

Step 4. Designing the Reducers

In this step, we will create the Reducers for our application. A completed example of this step can be found here : https://codesandbox.io/s/ovoo232n06

We will create 4 separate reducers since we have 4 attributes in our state tree. Here are the reducers that we will create:

- selectedTable
- tableStatusData
- tableData
- moneyEarned

1. Inside the `reducers` folder, create a new file named `selectedTable.js`.
2. Copy the following inside `selectedTable.js`:

```
import { SELECT_TABLE } from "../constants/constants.js";

const selectedTable = (state = null, action) => {
  switch (action.type) {
    case SELECT_TABLE:
      if (state === action.id) return null;
      else return action.id;
    default:
      return state;
  }
};

export default selectedTable;
```

The selectedTable reducer is in charge of handling the state changes to the selectedTable attribute of the overall state. The selectedTable state should just be a single integer that represents the id of the selected table. The state is initialized to null. The reducer only handles the SELECT_TABLE action. In the SELECT_TABLE action logic, if the dispatched action contains an id that is already selected, we return null to deselect the table. Otherwise, we return the action id to select the table.

3. Inside the `reducers` folder, create a new file named `tableStatusData.js`.
4. Copy the following inside `tableStatusData.js`:

```
import { TOGGLE_TABLE } from "../constants/constants.js";

var initialTableData = [];

for (let i = 0; i < 16; i++) {
  initialTableData.push([]);
}

const tableStatusData = (state = initialTableStatusData, action) =>
  switch (action.type) {
    case TOGGLE_TABLE:
      var stateCopy = state.slice();
      stateCopy[action.id] = !stateCopy[action.id];
      return stateCopy;
    default:
      return state;
  }
};

export default tableStatusData;
```

The tableStatusData reducer is in charge of handling the state changes to the tableStatusData attribute of the overall state. The tableStatusData state should be an boolean array of size 16 that represents whether each of the 16 tables is checked in or not. The state array is initialized to contain all false values. The reducer only handles the TOGGLE_TABLE action. In the TOGGLE_TABLE action logic, we make a copy of the state using `slice()` and then toggle the boolean value of the targeted index before returning the copy.

5. Inside the `reducers` folder, create a new file named `tableData.js`.
6. Copy the following inside `tableData.js`:

```
import {
  ADD_TABLE_ITEM,
  DELETE_TABLE_ITEM,
  TOGGLE_TABLE
} from "../constants/constants.js";

var initialTableData = [];

for (let i = 0; i < 16; i++) {
  initialTableData.push([]);
}

const tableData = (state = initialTableData, action) => {
  switch (action.type) {
    case ADD_TABLE_ITEM:
      var stateCopy = [];
      for (let i = 0; i < 16; i++) {
        stateCopy.push(state[i].slice());
      }
      stateCopy[action.tableId].push(action.item);
      return stateCopy;
    case DELETE_TABLE_ITEM:
      var stateCopy = [];
      for (let i = 0; i < 16; i++) {
        stateCopy.push(state[i].slice());
      }
      stateCopy[action.tableId].splice(action.id, 1);
      return stateCopy;
    case TOGGLE_TABLE:
      var stateCopy = [];
      for (let i = 0; i < 16; i++) {
        stateCopy.push(state[i].slice());
      }
      stateCopy[action.id] = [];
      return stateCopy;
    default:
      return state;
  }
};

export default tableData;
```

The tableData reducer is in charge of handling the state changes to the tableData attribute of the overall state. The tableData state should be a 2D array that contains 16 arrays that each contain the ordered items for the table that they represent. The state initialized to be an array of 16 empty arrays.

The reducer handles the ADD_TABLE_ITEM, DELETE_TABLE_ITEM and TOGGLE_TABLE actions:

ADD_TABLE_ITEM - the reducer makes a copy of the 2D array, finds the array representing the tableId from the dispatched action, and pushes the dispatched action item onto it.

DELETE_TABLE_ITEM - the reducer makes a copy of the 2D array, finds the array representing the tableId from the dispatched action, and deletes the dispatched action index from it.

TOGGLE_TABLE - the reducer makes a copy of the 2D array, finds the array representing the tableId from the dispatched action, and empties the array. (Both checking in and out result in empty arrays)

7. Inside the `reducers` folder, create a new file named `moneyEarned.js`.
8. Copy the following inside `moneyEarned.js`:

```
import { INCREMENT_MONEY_EARNED } from "../constants/constants.js";

const moneyEarned = (state = 0, action) => {
  switch (action.type) {
    case INCREMENT_MONEY_EARNED:
      return state + action.amount;
    default:
      return state;
  }
};

export default moneyEarned;
```

The moneyEarned reducer is in charge of handling the state changes to the moneyEarned attribute of the overall state. The moneyEarned state should just be an integer. The state is initialized to 0. The reducer only handles the INCREMENT_MONEY_EARNED action. In the INCREMENT_MONEY_EARNED action logic, the action increment amount is added to the previous state.

9. Inside the `reducers` folder, create a new file named `reducer.js`.
10. Copy the following inside `reducer.js`:

```
import { combineReducers } from "redux";

import selectedTable from "./selectedTable.js";
import tableStatusData from "./tableStatusData.js";
import tableData from "./tableData.js";
import moneyEarned from "./moneyEarned.js";

const reducer = combineReducers({
  selectedTable,
  tableStatusData,
  tableData,
  moneyEarned
});

export default reducer;
```

In this file, we combine all of our 4 reducers together using the `combineReducers()` method from the redux library.

Step 5. Creating the Store and testing it out

In this step, we will create a store using our reducer and test the logic out by dispatching a few actions! A completed example of this step can be found here: https://codesandbox.io/s/934rx9q00p

1. Inside `index.js`, import the reducer and create a store. Also import all of the action creators:

```
import React from "react";
import ReactDOM from "react-dom";
import { createStore } from "redux";
import reducer from "./reducers/reducer.js";
import addTableItem from "./actions/addTableItem.js";
import deleteTableItem from "./actions/deleteTableItem.js";
import incrementMoneyEarned from "./actions/incrementMoneyEarned.js"
import selectTable from "./actions/selectTable.js";
import toggleTable from "./actions/toggleTable.js";

import "./styles.css";

var store = createStore(reducer);


function App() {
  return (
    <div className="App">
      <h1>Hello CodeSandbox</h1>
      <h2>Start editing to see some magic happen!</h2>
    </div>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

Make sure to import `createStore` from the redux library. Then use it to create a new store with `createStore(reducer)`.

2. Now test out the logic by dispatching a few actions! Copy the code below into `index.js` to run a few tests:

```
import React from "react";
import ReactDOM from "react-dom";
import { createStore } from "redux";
import reducer from "./reducers/reducer.js";
import addTableItem from "./actions/addTableItem.js";
import deleteTableItem from "./actions/deleteTableItem.js";
import incrementMoneyEarned from "./actions/incrementMoneyEarned.js"
import selectTable from "./actions/selectTable.js";
import toggleTable from "./actions/toggleTable.js";

import "./styles.css";

var store = createStore(reducer);

const getSelectedTable = () => {
  var state = store.getState();
  return state.selectedTable;
};

const getAvailableTables = () => {
  var state = store.getState();
  var tablesAvailable = 0;

  for (let i = 0; i < state.tableStatusData.length; i++) {
    if (state.tableStatusData[i] === true) tablesAvailable++;
  }
  return tablesAvailable;
};
const getTableStatusData = () => {
  var state = store.getState();
  return state.tableStatusData;
};

const getTableItems = () => {
  var state = store.getState();

  if (getSelectedTable() === null) var tableItems = [];
  else var tableItems = state.tableData[getSelectedTable()];
  return tableItems;
};
```

```
const getTotalBill = () => {
  var tableItems = getTableItems();
  var totalBill = 0;
  for (let i = 0; i < tableItems.length; i++) {
    totalBill += tableItems[i].price;
  }
  return totalBill;
};

const getMoneyEarned = () => {
  var state = store.getState();
  return state.moneyEarned;
};

const checkOut = () => {
  var totalBill = getTotalBill();
  store.dispatch(incrementMoneyEarned(totalBill));
  store.dispatch(toggleTable(getSelectedTable()));
};

store.subscribe(() => {
  console.log(`Selected Table: ${getSelectedTable()}`);
  console.log(`Tables Available: ${getAvailableTables()} / 16`);
  console.log(`Table Availability Status: ${getTableStatusData()}`);
  console.log(`Selected Table Items List:`, getTableItems());
  console.log(`Selected Table Bill: $${getTotalBill()}`);
  console.log(`Total Money Earned: $${getMoneyEarned()}`);
  console.log();
});

//test dispatched actions

//select and toggle table 0, add items
//select and toggle table 1, add items
//check out current selected table
//select table 0
store.dispatch(selectTable(0));
store.dispatch(toggleTable(0));
store.dispatch(addTableItem("apples", 2, 0));
store.dispatch(addTableItem("bananas", 3, 0));
store.dispatch(deleteTableItem(0 , 1));
store.dispatch(selectTable(1));
store.dispatch(toggleTable(1));
store.dispatch(addTableItem("apples", 2, 1));
store.dispatch(addTableItem("bananas", 3, 1));
store.dispatch(addTableItem("carrots", 4, 1));
checkOut();
store.dispatch(selectTable(0));

function App() {
  return (
    <div className="App">
      <h1>Hello CodeSandbox</h1>
      <h2>Start editing to see some magic happen!</h2>
    </div>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

Since there is no UI to display our store state, we will have to log our store state to the console. I've added a few self explanatory helper methods that get parts of the store state. I also subscribed an event listener that logs what the user should be able to see once we have our UI created.

The checkout() method is necessary because the moneyEarned reducer cant access state from the tableData reducer. Its purpose is to get the total bill from the selected table, add that bill to the total money earned, and then check out the selected table.

I've added a test that does the following:

1. selects and toggles table 0 and then adds a few items and then deletes an item
2. selects and toggles table 1 and then adds a few items
3. checks out the selected table (table 1)
4. reselects table 0

You can see how the state changes with every dispatched action in the console.

Feel free to play around and add your own tests.

Again, here is the completed example of the tutorial:
https://codesandbox.io/s/934rx9q00p