



TASK

Redux and Global State Management

Visit our website

Introduction

WELCOME TO THE REDUX AND GLOBAL STATE MANAGEMENT TASK!

In this module, you will learn about Redux as a global state management system. Let's unpack exactly why Redux is essential for writing efficient, scalable ReactJS applications.



Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at <https://discord.com/invite/hyperdev>, where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!



WHAT IS REDUX?

Redux is a predictable state container designed to help you write JavaScript apps that behave consistently across client, server, and native environments, making it easier to test. While it's mainly used as a state management tool with React, you can use it with other JavaScript frameworks or libraries, like AngularJS and VueJS.

Why should I use Redux?

1. All states of your application are saved in one predictable state container.
2. It lets you avoid unnecessary prop drilling from one component to the next.
3. It prevents the re-rendering of components.
4. It's useful in server-side rendering.
5. Redux optimises the performance of your ReactJS application.

There are a multitude of other libraries or frameworks that you can use: like Context API or Apollo Client. You're more than likely going to be exposed to Redux in your future career, so we will concentrate on Redux for this exercise. To ensure we write efficient, modular Redux implementations, we will use the **Redux toolkit**.

A REDUX APPLICATION IS MADE UP OF THE FOLLOWING COMPONENTS

Actions

You can think of an action as an event that describes something in the application. More technically, an action is a plain JavaScript object with a type field.

The type field should be a string that gives this action a descriptive name. We usually write that type string like: "domain/eventName," where the first part is the **feature** or **category** that this action belongs to, and the second part is the specific thing that happened.

An action object can have other fields with additional information about what happened. By convention, we put that information in a **payload field**.

Reducers

A reducer is a function that receives the current state and an action object and decides how to update the state if necessary, and then returns the **new** state:

$(state, action) \Rightarrow newState.$

You can think of a reducer as an event listener which handles events based on the received action (event) type.

Reducers must always follow specific rules:

- They should only calculate the new state value based on the state and action arguments.
- They are not allowed to modify the existing state. Instead, they must make immutable updates by copying the current state and changing the copied values.
- They must not do any asynchronous logic, calculate random values, or cause any other "side effects".

Reducers can use any logic to decide the new state: *if/else, switch, loops, etc.*

Store

The current Redux application state lives in an object called the store. The store is created by passing in a reducer and has a method called **getState** that returns the current state value:

The Redux store has a method called **dispatch**. So the only way to update the state is to call the **store.dispatch()** and pass in an action object. The store will then run its reducer function and save the new state value, and we can call **getState()** to retrieve the updated value.

You can think of dispatching actions as “triggering” an event in the application. Something happened, and we want the store to know about it. Reducers act like event listeners, and when they hear an action, they are interested in how to update the state in response.

Selectors

Selectors are functions that know how to extract specific pieces of information from a store state value. As an application grows more significant, this can help avoid repeating logic as different parts of the application need to read the same data.

REACT HOOKS VS THE CONVENTIONAL CONNECT FUNCTION

React Hooks, which were added to React in version 16.8, allow developers to use state and other React features without writing a class component. In addition, react hooks allow developers additional functionality when using redux to write modular, efficient code in a functional component using the **useDispatch()** and

useSelector() functions. Hooks also offer the added advantage of separating concerns of a UI component and logical components.

useSelector() is an alternative to Connect's **mapStateToProps()** function. You pass in a function that takes the Redux store state and returns the pieces of state your component requires to function properly.

useDispatch() replaces Connect's **mapDispatchToProps()** function but is lighter weight. In addition, it returns your store's dispatch method so you can manually dispatch actions to avoid binding action creators to an event or method inside a functional component, which is a conventional implementation in class components.

Connect() function from Redux

The **connect()** function connects a React component to a Redux store. It provides its connected component with the pieces of data it needs from the store and the functions it can use to dispatch actions to the store. It does not modify the component class passed to it; instead, it returns a new, connected component class.

The Connect function accepts four different parameters, all optional. By convention, they are called:

1. mapStateToProps?: Function
2. mapDispatchToProps?: Function | Object
3. mergeProps?: Function
4. options?: Object

All of which can be simplified when using Redux hooks.

Have a look at this example using Connect:

```
import React from "react";
import { connect } from "react-redux";
import { addCount } from "../store/counter/actions";
export const Count = ({ count, addCount }) => {
  return (
    <div>
      <div>Count: {count}</div>
      <button onClick={addCount}>Count</button>
    </div>
  );
};
```

```
const mapStateToProps = state => ({
  count: state.counter.count
});
const mapDispatchToProps = { addCount };
export default connect(mapStateToProps, mapDispatchToProps)(Count);
```

Now let's see what we can do with Redux Hooks:

```
import React from "react";
import { useDispatch, useSelector } from "react-redux";
import { addCount } from "../store/counter/actions";
export const Count = () => {
  const count = useSelector(state => state.counter.count);
  const dispatch = useDispatch();
  return (
    <div>
      <div>Count: {count}</div>
      <button onClick={() => dispatch(addCount())}>
        Count
      </button>
    </div>
  );
};
```

The code is more precise, and fewer lines mean better performance. It's easier to read, easier to understand, and of course, easier to test.

Please use Redux hooks when implementing Redux in an application to ensure module-efficient code. This will ensure maintainability and scalability for future development.

INSTALLATION

First, we must ensure we have installed all the applicable applications/libraries on our local machines.

To install NodeJS, follow this next link:

<https://nodejs.org/en/download/>

Alternatively, if NodeJS is already installed on your local laptop or workstation, you can follow the instructions below to update your current version of NodeJS and

NPM to the **latest** versions. Ensure you have admin access when executing the following commands in your terminal.

Updating Node Package Manager:

```
npm install -g npm@latest
```

Updating Nodejs:

```
npm install -g node@latest
```

CREATING THE REACTJS APPLICATION

We are going to use the **create-react-app** library. If you haven't installed create-react-app on your local machine, you can follow the next link below for more information on how to do so:

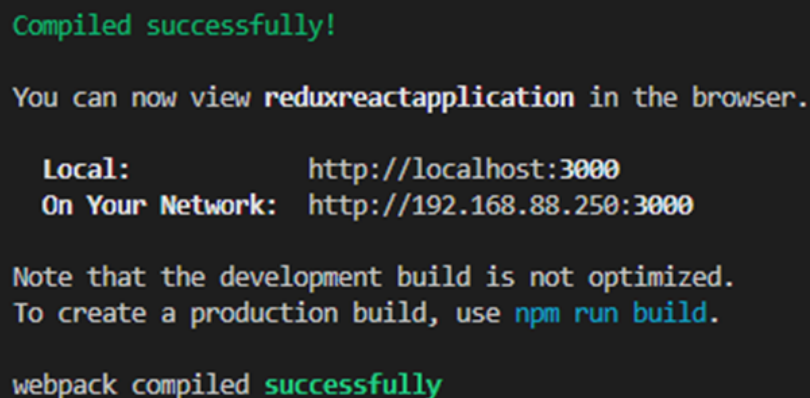
<https://www.npmjs.com/package/create-react-app>

Here's the command to run in the folder where you would like to save your new Redux application.

```
create-react-app reduxreactapplication
```

If you are unable to run create-react-app in your terminal, ensure your **npm** folder is added to the advanced settings of your local machine. Then ensure your environment variables / PATH settings are edited accordingly.

Let's ensure our application is up and running:



```
Compiled successfully!

You can now view reduxreactapplication in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.88.250:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

INSTALLATION OF REDUX TOOLKIT AND OTHER APPLICABLE NODE MODULES

Let's begin by installing the libraries we require to implement Redux into our ReactJS application.

Ensure the command executed is in the correct project folder:

```
npm install --save @reduxjs/toolkit react-redux
```

Ensure the npm_modules were installed correctly.

```
added 11 packages, and audited 1448 packages in 33s

205 packages are looking for funding
  run `npm fund` for details

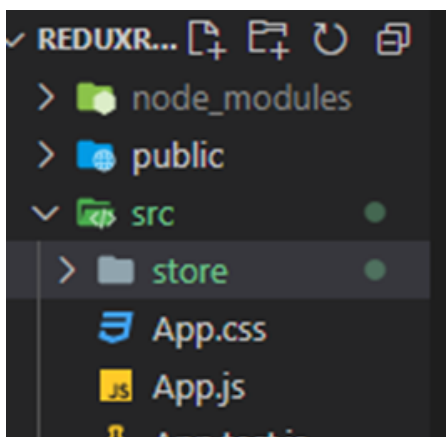
6 high severity vulnerabilities

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

SETTING UP OF THE REDUX STORE, PROVIDER COMPONENT, AND REDUCER

Step 1: create a folder inside the **src** (or source directory), as indicated in the following picture:



Inside the store directory, create a new JavaScript file called **store.js**. In this file, we will start implementing our Redux store.

For this implementation, we will import **configureStore** from **reduxjs/toolkit** and export it with an empty reducer as indicated below.

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counter";

// The configureStore function will automatically set up an empty store for you
// with the relevant settings you will need in the future.
export default configureStore({
  reducer: {

  },
});
```

Step 2: to ensure our application has access to the store, we will go to the **index.js** file and implement the **Provider** component that will encapsulate our **App** component. With the **Provider** component, we will need to pass our newly initiated empty store as a prop into our **Provider** component:

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import App from "./App";
// Importing the Provider component from react-redux, which we will require to
// set up our Redux store. To ensure the whole application has access
// to the relevant slices of state, each component will require it to function
// correctly.
import { Provider } from "react-redux";
// Importing the newly created store implementation we have just created using
// the configureStore function.
import store from "./src/store/store";
import reportWebVitals from "./reportWebVitals";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  // Implementing the Provider component and passing our store as one of its
  // props to ensure the store is correctly implemented and initiated.
  <Provider store={store}>
    <App />
  </Provider>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Step 3: We need to create a **slice** of state. Now with any ReactJS application, you will have multiple slices of state that different components will require when operating your application. Next, let's create a **counter.js** file inside our store directory, which we created earlier in this tutorial:

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

export const counterSlice = createSlice({
  // This is the name of the slice of state that we will implement in our
  // empty store.
  name: "counter",
  // This is the initial state for your slice of state. This can be
  // anything from an empty array or other data structure that your
  // application requires to avoid prop drilling.
  // For this example, we use a number.

  initialState: {
    value: 0,
  },
  // As indicated before. The reducer is used to manipulate the initial
  // state or current state.
  reducers: {
    // This is one example of an action type the reducer will identify
    // how to manipulate this slice of state.
    // In our case, we want the increment action type to add one to our
    // value, which is now set to zero in our initial state.
    increment: (state) => {
      state.value += 1;
    },
    // Second example, our action type will reduce the initial state or
    // current state by decrementing by 1.
    decrement: (state) => {
      state.value -= 1;
    },
    // The third example is probably the most important as this function
    // passes the action value or payload to our reducer to increment or
    // decrement the value of the state depending on what the user enters
    // or submits.
    incrementByAmount: (state, action) => {
      state.value += action.payload;
    },
  },
});

// Action creators are generated for each case reducer function.
```

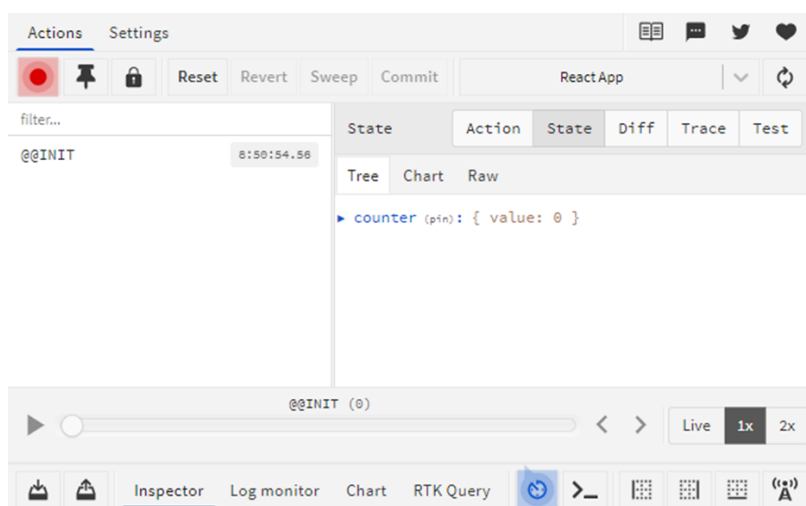
```
export const { increment, decrement, incrementByAmount, resetValueToZero } =
  counterSlice.actions;
// Exporting the reducer function we will implement into our empty
// store, previously created.
export default counterSlice.reducer;
```

Step 4: With our reducer now implemented, we want to ensure our store can access the reducer and the actions applicable to this slice of the state by importing our `counter.js` file into our `store.js` file, which contains our newly created empty store.

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counter";

// The configureStore function will automatically setup an empty store for you
// with the relevant settings you are going to need in the future.
export default configureStore({
  reducer: {
    counter: counterReducer,
  },
});
```

To ensure your Redux implementation is working as expected, you can install [Redux DevTools](#) as an extension onto your Google Chrome/Firefox or Internet Explorer/Edge. Before we start manipulating this slice of the state, check if everything is up and running:



SETTING UP THE APP.JS FILE

The fun begins when we implement the required JSX / HTML into our **App.js** file to manipulate our newly created slice of state with the actions we have initiated in our counter reducer. Let's modify the **App.js** file to do this:

```
// Importing the useSelector and useDispatch functions to select the required
// slices of state.
import { useSelector, useDispatch } from "react-redux";
// Importing the action creators we've implemented in our counter reducer.
import { decrement, increment } from "./store/counter";
// Importing the App.css file to add styling to our App component
import "./App.css";

function App() {
  // The useSelector function allows us to find the needed slices of state we
  // require. The useDispatch function will dispatch all the necessary actions to
  // the reducer to enable us to modify the state.
  const count = useSelector((state) => state.counter.value);
  // Initiating the dispatch variable using the useDispatch function.
  const dispatch = useDispatch();
  return (
    <div>
      <div className="App">
        {/* Displaying the state or count variable we have initiated earlier
        using the useSelector function*/}
        <h1>{count}</h1>
      </div>
      <div className="Buttons">
        {/* Each time any of the buttons below are clicked upon, the state will
        increment or decrease depending on the button.*/}
        <button onClick={() => dispatch(increment())}>Increment</button>
        <button onClick={() => dispatch(decrement())}>Decrement</button>
      </div>
    </div>
  );
}

export default App;
```

IMPLEMENTING CSS RULES FOR A USER-FRIENDLY APPLICATION

Below you'll find a bit of helpful CSS to style your application:

```

.App {
  display: flex;
  align-items: center;
  justify-content: center;
}

h1 {
  font-size: 6.5rem;
}

.Buttons {
  display: flex;
  padding: 3rem;
  justify-content: space-around;
}

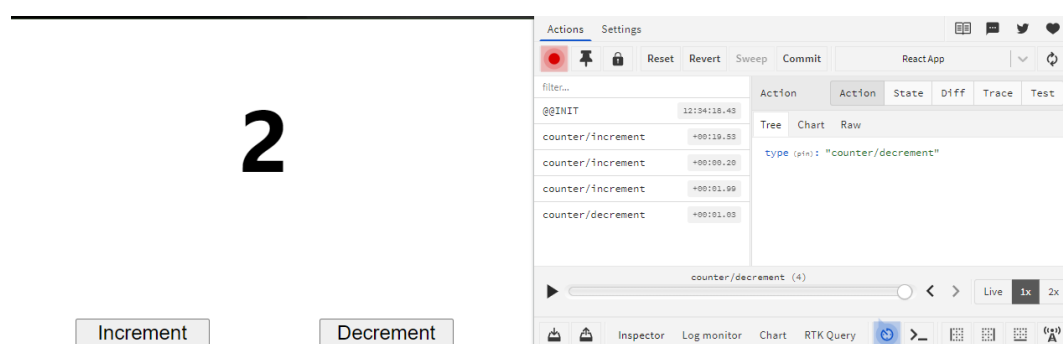
.Buttons > button {
  font-size: 1.5rem;
  margin: 2rem;
  width: 10rem;
}

```

TESTING YOUR IMPLEMENTATION

If all went according to plan, you will have an application where you can click on two buttons to increment and decrement the slice of state we have implemented.

Check out this example:



Let's implement the third and final action from our reducer function. The **incrementByAmount** function takes a parameter or payload variable to modify the slice of state by the amount a user has requested. We will implement a form element inside our **App.js** file for this test, using the **useState** React hook. Please ensure you pass the correct data type to the reducer to avoid runtime errors or other incorrect data manipulations.

```

return (
  <div>
    <div className="App">
      {/* Displaying the state or count variable we have initiated earlier
using the useSeletor function*/}
      <h1>{count}</h1>
    </div>
    <div className="Buttons">
      {/* Now each time any of the buttons are clicked on, the state will
increment or decrement depending
on which button is clicked.*/}
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
    </div>

    <form className="Form" onClick={handleSubmit}>
      <label>
        Value:
        <input
          type="text"
          name="value"
          onChange={(e) => setUserInput(e.target.value)}
          value={userInput}
        />
      </label>
      <button type="submit">Submit</button>
    </form>

  </div>
);

```

useState Implementation: a snippet of our App.js file:

```

import { useState } from "react";

function App() {
  const [userInput, setUserInput] = useState(0);

```

Let's not forget our **handleSubmit** function:

```

const handleSubmit = (event) => {
  event.preventDefault();
  dispatch(incrementByAmount(Number(userInput)));
  setUserInput(0)
};

```

Oh, and don't forget to adjust the original import of our actions creators:

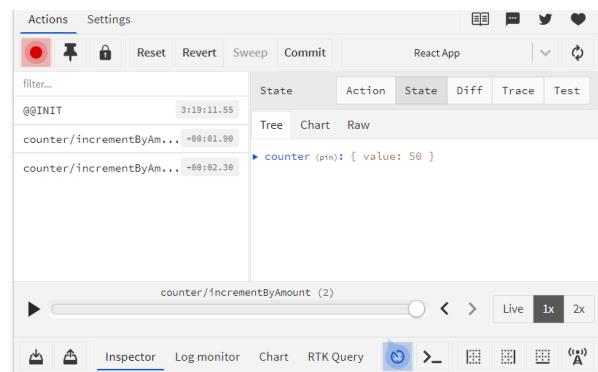
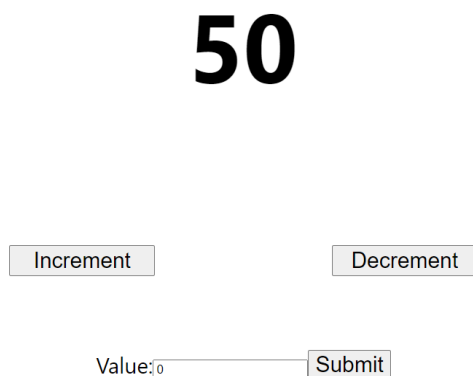
```
import { decrement, increment, incrementByAmount } from
"./store/counter"
```

And some styling for the newly implemented form component:

```
.Form {
  display: flex;
  align-items: center;
  justify-content: center;
  font-size: 1.5rem;
}

.Form > button {
  font-size: 1.5rem;
}
```

Now you should have a successful ReactJS application using Redux as a state management system to ensure data consistency!



INSTRUCTIONS:

Compulsory Task 1

You are going to create a ReactJS application that manipulates a cash balance on an account using Redux.

- This app should contain 4 buttons named:
 - **Withdraw**,
 - **Deposit**,
 - **Add Interest**,
 - and **Charges**.
- It must also include one input box that you will use to enter the amount that you wish to deposit or withdraw.
 - Include the space to view your balance as it gets updated.
- The store will store the current balance of the account.
- If the **Deposit** button is clicked, the balance amount should increase by the input value from the input box.
- If the **Withdraw** button is clicked, the balance amount should decrease by the input value from the input box.
- If the **Add Interest** button is clicked, the balance amount should increase by 5%.
- If the **Charges** button is clicked, the balance amount should decrease by 15%.

Compulsory Task 2

Next up you are going to create a **to-do** application using React and Redux.

- The initial state of your reducer should be as follows:


```
const initialState = {
  nextId: 2,
  data:
  {
    1: {
      content: 'Content 1',
      completed: false
    }
  }
}
```

- The reducer of your store should be able to add, delete, and edit a specific to-do/task and change the completed variable to: **true**.
- Each to-do/task rendered in your application should then have three buttons.
 - **Edit**,
 - **Delete**,
 - and **Completed**.
- Ensure you have a form component implemented **to add additional tasks**.
- You can use any library to style your application.

If you are having any difficulties, please feel free to contact our specialist team [on Discord](#) for support.



Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

[Click here](#) to share your thoughts anonymously.



REFERENCE(S)

Getting Started with React Redux. (2022, April 16). React Redux.

<https://react-redux.js.org/introduction/getting-started>