Hyperiondev

TASK

React - Redux and Global State Management

Visit our website

Introduction

WELCOME TO THE REACT - REDUX AND GLOBAL STATE MANAGEMENT TASK!

In this task we will gain insights into Redux as a global state management system. We'll unpack exactly why Redux is essential for writing efficient, scalable ReactJS applications. We'll explore the concept of Redux and its role in handling state across React components. Discover how Redux solves the challenges of state management, leading to improved maintainability and scalability of your applications. We'll also discuss the benefits of incorporating Redux into your projects and how it elevates the way you manage state.

Moving forward, we'll dive into hands-on practice by setting up a simple React Redux application. We'll guide you through establishing the Redux store, Provider component, and reducers to effectively manage state. In addition, we'll also explore React global state management hooks. By the end of this task, you'll have a solid foundation in Redux and its integration with React, empowering you to create streamlined and effective applications.

WHAT IS REDUX IN GENERAL?

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.

Note: that '`React Redux`' is a library specifically designed to work with React, and it simplifies the integration of Redux into React applications.

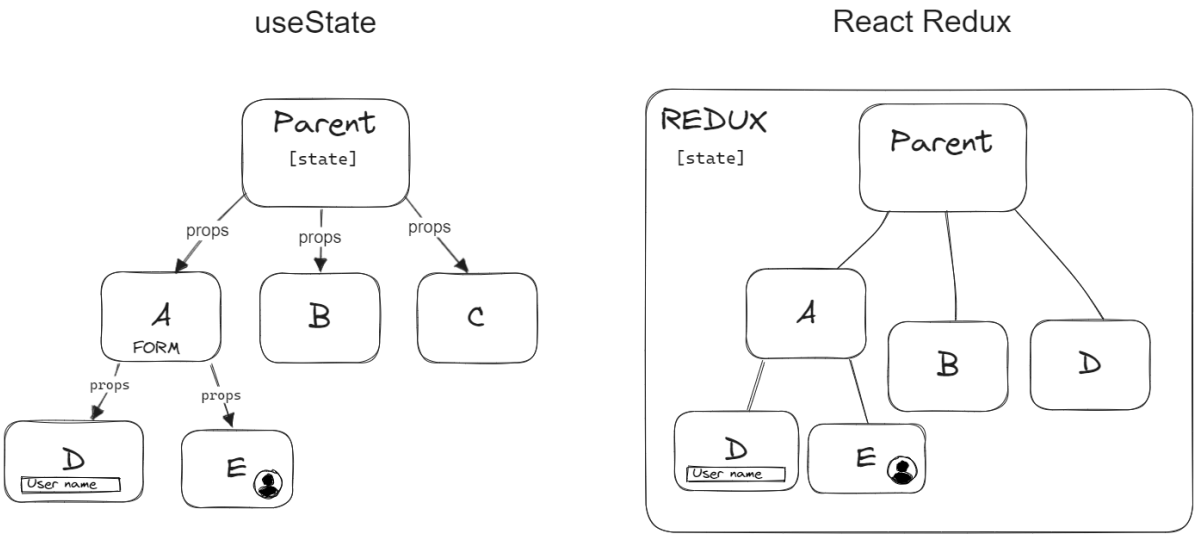
Why should I use Redux?

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 task. To ensure we write efficient, modular Redux implementations, we will use the **Redux toolkit**.

HOW REDUX SOLVES STATE MANAGEMENT



When you needed to pass data to a child component or a deeply nested child component, you had to use props. While this approach works, it becomes increasingly tedious as your application grows and branches into several components. Remember that breaking your React application into smaller components improves maintainability and makes code management easier. However, when passing data through props, also known as 'prop drilling,' keeping track of the data being passed can become complicated. For instance, if we wanted to update Component E by displaying an image we got from the parent, we'd have to pass that data down through intermediate components, meaning we'd have to define extra logic inside Component A to be able to pass that data to Component E.

This is where Redux solves this issue. Redux acts as a global state manager that can be accessed by any component without the need for prop drilling.

HOW WOULD YOU BENEFIT FROM USING REDUX?

Centralised State Management: Redux provides a single source of truth for your application's state. The entire state of your application is stored in a central store, making it easier to manage and maintain data.

Code Reusability and Modularity: By using React Redux, your components can be more reusable and modular. Since they don't directly depend on specific state locations, they can be easily moved to different parts of your application.

Easier State Management for Complex Apps: As your application grows in size and complexity, managing state with React's built-in state management alone can become challenging. React Redux provides a more structured and organised approach, making state management more manageable and scalable.

Popular and Well-Supported: React Redux is widely used and well-supported by the community. Many developers are familiar with its patterns and practices, making it easier to find help and resources when needed.

INSTALL AND SET UP A SIMPLE REACT REDUX APPLICATION

Installing the redux toolkit

NOTE: Redux can be installed on an existing React app or you can create a new React application for this guide.

The **Redux Toolkit** is not a mandatory requirement to use Redux in a React app, but it is highly recommended and will be required in every Redux task from here on.

Redux Toolkit is an official package from the Redux team that aims to simplify and streamline the process of working with Redux. It provides utilities and abstractions that make Redux code more concise, efficient, and easier to maintain.

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

Ensure the command executed is in the correct project folder:

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

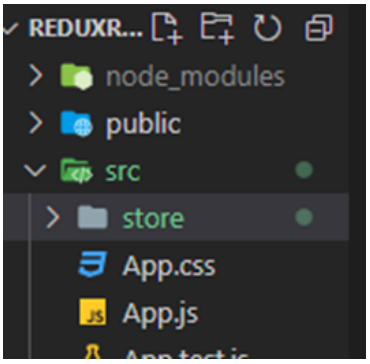
This command will install both Redux and the Redux toolkit into your React application.

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 REDUCERS

Step 1: In your `src` directory, create a folder `store`, 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 logic.

With Redux, the application state is stored in a single JavaScript object called the store. The store holds the entire state tree of your application. The Redux Toolkit has several other built-in functions, for this guide, we are only going to use one called configureStore.

Inside the `store.js`, we will import the `configureStore()` function from `reduxjs/toolkit` and export it with a reducer as indicated below.

```
//store.js
import { configureStore } from "@reduxjs/toolkit";
import counterState from "./counterState";
const store = configureStore({
```

```
  reducer: {
    counter: counterState,
    // ...more reducers can be added here.
},
 // other store option
 middleware: [],
 devTools: process.env.NODE_ENV !== 'production'
});

export default store;
```

The `configureStore({ })` function is a utility function that simplifies the process of creating a Redux store. It provides a convenient way to configure and customise the store with additional middleware and options. It takes in an object with various configuration options and returns a Redux store that can be used in your application. For this guide, we will only use the `reducer:` option.

The `reducer:` property, is a function that specifies how the state of an application changes in response to actions and returns the state defined inside the reducer called `counterReducer`. Later we will define the state logic in a separate file called `counterState.js`.

Step 2: To ensure your application has access to the store, let's go to the `index.js` file and implement the Provider component that will encapsulate our App component. It serves a crucial role in connecting your React application with the Redux store.

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import App from "./App.js";
import reportWebVitals from "./reportWebVitals";
import store from './store/store';
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(

 <Provider store={store}>
  <App />
 </Provider>

);
```

```
reportWebVitals();
```

To allow the React App to access the store, we need to wrap the `<App />` component with the `<Provider />` component. Inside the `Provider` tag, we simply pass the store as props using `store={store}`. This will ensure that the store can be accessed throughout the entire app. So, any component defined inside `App.js` will now be able to access the store effortlessly.

Step 3: Now let's create the actual `state`. For this, we need to create a **slice** of the state. We will define the state and reducers, explained later, inside a new JavaScript file. Let's call it `counterState.js`. You will also notice that we already imported this file into the `store.js` file, earlier in step 1.

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

const counterSlice = createSlice({
    name: 'Counter',
    initialState: {
     value: 0,
     heading: 'Counter App'
    },
    reducers: {
     //This function will increment the state value property by 1
     increment: (state) => {
      state.value += 1;
     },
     //This function will decrement the state value property by 1
     decrement: (state) => {
      state.value -= 1;
     },
      //This function will increment the state value property by an amount
      passed through the action.payload property
     incrementByAmount: (state, action) => {
      state.value += action.payload;
     },
   },
});

// Export the action functions to be used in components
```

```
export const { increment, decrement, incrementByAmount} =
counterSlice.actions
//Export the reducer function to be used in the store
export default counterSlice.reducer
```

In the example, we use the createSlice method that creates a piece of state inside the store. To help you understand this better, think of the `store.js` as a cake, and the `counterState.js` as a piece/slice of the cake. Inside this slice, we will define all the **state values** and **functions** also known as **actions**, to manage the application's state.

There are **three** properties required when creating a slice.

1. `name:` This property's value will be the name of the slice and is defined as a string.
2. `initailState:` This property will contain the entire state object of the application. Similar to how you would use the `useState` to store data or an object with multiple key-value pairs that stores all the data of a program. Currently, our state only has one key value of `value: 0`. You can add more if needed, but for this tutorial, we will only manipulate the `value` property/key.
3. `reducers:` This property will have a set of functions/actions defined as key-value pairs where the key is the function name and the value is a function. A function typically has two arguments, `state` and `action` (optional).

`State` is a reference to the initialState object and allows us to make changes to the state values, for example, if I wanted to change `value: 0`, I would do the following: `state.value = 15`.

`Actions` are mainly used to receive data from a component to make dynamic changes to the state. In other words, that state can be changed through user actions. For example, I could receive the number 15 from a component and update the value property instead of hardcoding the number 15, as follows: `state.value += actions.payload`. Here, 15 will be sent with the payload. Later we will look at how this data is sent from the component to a reducer function.

Note: Reducer functions will initially be triggered from a component, allowing us to also pass data as an argument to a reducer function. Similar to how he would pass data through props and use that data inside a component. This is how we will also update the state or add our own custom values.

Additionally, in our example `reducers`, we defined three different functions. When any of the first two functions are called, they immediately modify the `value` property in the state, either incrementing or decrementing the value by 1. However, the third function relies on a value received from the component.

In the next step will create the component that will allow us to trigger these reducer functions and also displays the state value like the `heading` property on the UI.

Step 4: Let's create a component called `Counter.js` and as a best practice let's place this component inside a folder called components, like this "src/components/Counter.js." After creating the counter component, you will need to import it into the `App.js` component as usual. By now you should be familiar with how to create components in React.

Now we will define the logic for the counter component. This component will have three buttons and one heading tag to display the count value. You're welcome to use your own custom CSS to style your component, but for this tutorial, we will make use of React Bootstrap to make our app more user-friendly and attractive.

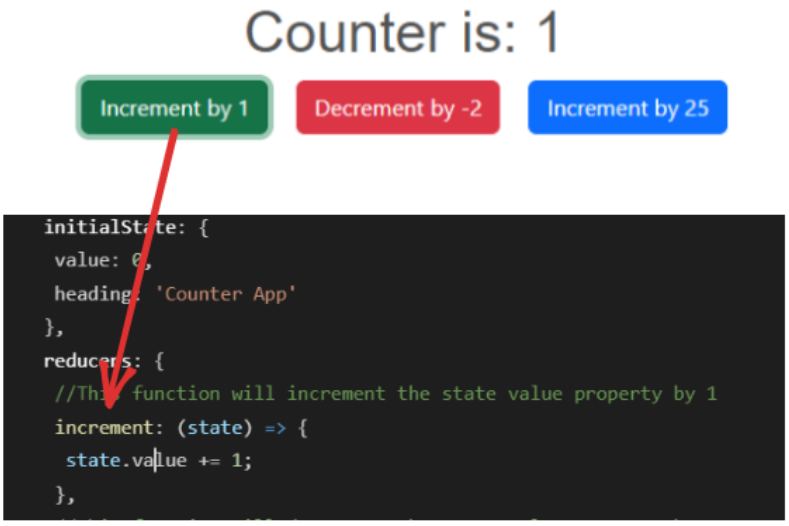
```
// Counter.js
import React from "react";
import { Button } from 'react-bootstrap';
import { useSelector, useDispatch } from "react-redux";
import { decrement, increment, incrementByAmount } from
"../store/counterState";

export default function Counter() {
  // useSelector function to get the state from the store
  const state = useSelector((state) => state.counter);

  // dispatch function to execute reducer functions
  const dispatch = useDispatch();

  return (
    <div className="counter">
      <h1>Counter is: {state.heading}</h1>
       <div className="btn-container">
        <Button variant="success" onClick={() => dispatch(increment())}>
          Increment by 1
        </Button>
```

```
        <Button variant="danger" onClick={() => dispatch(decrement())}>
          Decrement by -2
        </Button>
        <Button variant="primary" onClick={() =>
          dispatch(incrementByAmount(25))
          }>
          Increment by 25
        </Button>
      </div>
    </div>
  );
}
```

In the **`Counter.js`** component above, we have created three buttons, each with an **`onClick`** event. Our aim is to update the Redux initialState - **`value:0`** property by 1 by calling the **`increment`** reducer function when the 'increment by 1' button is clicked.

To achieve this, we need to make use of the **`dispatch()`** function. This function acts like a messenger or postman. Whenever a change needs to be made to the application's state, an action is created, which is essentially a message describing the required changes. Dispatch sends the message to the Redux store. The store receives this action and passes it to the reducers, which then executes their code to modify the state accordingly.

`()=> dispatch(increment())`: Here, when a button is clicked, the dispatch function gets executed and allows us to pass in the name of a reducer function we wish to execute in the store. So, essentially, the dispatch function sends a message to the store saying that an **`increment`** reducer function needs to be executed, updating the state.



Passing values to the state: Earlier when we defined the `incrementByAmount` reducer function in the store, we used the syntax `actions.payload`. The payload essentially allows us to pass any value or object from the Counter component to the store. In the example code, we passed in the number `25` as an argument to the `incrementByAmount` reducer function. Updating the state value by 25:

```
<Button variant="primary" onClick={ () => dispatch(incrementByAmount(25)) }>
        Increment by 25
</Button>
```

Note: You can only pass *one value* as an argument to a reducer function. If you want to pass an array or multiple values, you will have to place your array or objects inside a single value like a local state variable, then pass that state value as an argument.

Displaying state values on the UI: In our state, we also have a property called `` `heading:'Counter App'` ``, to extract this or any values from the state and use it in a component, we need to make use of a function called `useSelector`. Simply said, the `useSelector` allows us to select properties in the state and use them throughout the application:

```
export default function Counter() {
 // useSelector function to get the state from the store
 const state = useSelector((state) => state.counter);

 // dispatch function to execute reducer functions
 const dispatch = useDispatch();
   return (
     <div className="counter">
     <h1>Counter is: {state.heading}</h1>
```

In the example, we wrote `` `const state = useSelector((state) => state.heading)` ``. Here `` `state.heading` `` is a reference to the counter property defined in `store.js` in step 1.

Note: When using the `useSelector`, state properties are read-only, only the reducer functions can be used to change state values.

REACT GLOBAL STATE MANAGEMENT HOOKS

Redux can indeed require some setup and boilerplate code that might make it seem tedious compared to similar state management hooks built-in to React. For large and complex applications with extensive data flow and state management requirements, Redux can be a valuable tool. For smaller and simpler applications, react built-in hooks might be sufficient and be more straightforward to implement, as React has two built-in hooks that allow an application to manage global state in a simplified way by using: **useContext** and **useReducer.** Both hooks serve different purposes for managing the global state in a react application.

USE CONTEXT

This hook allows components to access and use data that is shared and managed by a higher-level component in the component tree. It helps in passing down state to components without the need for prop drilling.

Take note:

In React, "**context**" refers to a feature that allows data to be passed through the component tree without explicitly passing it down through props.

It provides a way to share data between components at different levels in the component tree without using prop drilling.

The **context** itself is **not** the state, rather, the context is a mechanism to share the state or data across components without passing it explicitly through props. It acts as an intermediary, allowing components to access and modify the shared state.

Have a look at this example using the useContext hook:

```
// MyContext.js
import React, { createContext, useState } from 'react';
import App from './App';

// Create the context
const MyContext = createContext();

function App() {
  // Your global state goes here
  const [sharedData, setSharedData] = useState('Hello, I am shared data!');
```

```
  return (
    // Pass the state and the setter function to the context
    <MyContext.Provider value={[sharedData, setSharedData]}>
      <App />
    </MyContext.Provider>
  );
}
export default App;
```

In the above example, we created a new element called `MyContext.Provider` using the **createContext** function imported from React. This component is used to expose the state (sharedData) and the setter function as the `value` prop, making it accessible to all its descendant components, allowing the entire application defined in App, to consume the state we created here.

This is how we can create a global state and use it throughout the entire application. Notice how we are able to pass both the state and the setter function as props. This allows nested components consuming the context to not only access to state but also make updates to the state using the setter of functions.

Now let's see how a child component inside the App.js would be able to use the global state:

```
//ChildOne.js
import React, { useContext } from 'react';
import { MyContext} from './MyContext;

export default function ChildOne() {
  const value = useContext(MyContext);//Import the context object

  let [sharedData, setSharedData] = value;//Destructure the context objects
  return (
    <div>
      <h1>{sharedData}</h1>
    </div>
  )
}
```

To use the global state we first need to import the my `MyContext` component, then extract the `value` prop using the `useContext(MyContext)` hope, passing in the myContext component as an argument. This essentially gives us access to both the state variable and setter function, allowing the current component to modify or use global state values.

USE REDUCER

The basic idea behind the useReducer is to provide a state management pattern similar to Redux, where state is changed by user actions. It takes a reducer function and an initial state as arguments and returns the current state and a dispatch function that can be used to trigger state updates.

Let's take a look at an example of using the useReducer in React.

First, we will create a file where we will define all the reducer functions and initial state values. For this example, we are going to create a simple counter application that only uses one property.

```
// GlobalState.js
export const initialState = {
   count: 0,
 };

export const reducers = (state = initialState, action) => {
   if (action.type === 'INCREMENT') {
       return { ...state, count: state.count + 1 };
     } else if (action.type === 'DECREMENT') {
       return { ...state, count: state.count - 1 };
     }

     // If no action type matches, return the current state
     return state;
 };
```

In the above example, the `reducers` function is responsible for updating the state based on dispatched actions. It takes two parameters: state and action.

`(state = initialState, action)`: The reducer function uses ES6 default parameter syntax to assign initialState as the default value for state. If the state parameter is not provided (i.e., it is undefined), it will default to the initialState

object. This means that when the reducer is initially called, it will use the initial state as the starting state.

The **`action`** parameter represents the action object that is dispatched to the reducer, meaning: the actions taken by the user. The user will define what type of action they want to take and their actions will be compared to the action type inside the if statements.

Now let's take a look at how the user can trigger these actions from a component. We'll create a component called **`Counter.js`**. This component will contain two buttons and also display the count value on the UI, similar to what we did in our Redux app:

```
 // Counter.js
import React, { useReducer } from 'react';
import { initialState , reducers } from './GlobalState';

const Counter = () => {
  //Get access to use and update state
  const [state, dispatch] = useReducer(reducers, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'INCREMENT'
        })}>Increment</button>
      <button onClick={() => dispatch({ type: 'DECREMENT'
        })}>Decrement</button>
    </div>
  );
};

export default Counter;
```

After defining this component, import it into App.js to display its content on the UI.

First, we need to import the **`useReducer`** hook and the **`globalState`** file, in order to have access to the state values and reducers.

```
      const [state, dispatch] = useReducer(reducers, initialState);
```

Let's break this down:

- The useReducer hook uses a **destructuring** syntax similar to `useState`, this means that it returns an array with two values `` `state` `` and `` `dispatch` ``. Similar to how we used dispatch in Redux, it's used in the same way here, where dispatch sends a message to the global state file (store), the message is received by the reducers and executes the logic based on the action type value.
- `` `useReducer(arg1,arg2)` ``, is a built-in React hook that takes two arguments.
- The first argument, the `` `reducers` `` function is the custom function defined in `GlobalState.js` and imported into this file. It is responsible for specifying how the state should be updated based on different actions. This function defined there, takes two arguments: the current state and an action object. It returns the new state based on the action.
- The second argument `` `initialState` `` simply gives us access to all the state values.

This approach also removes the need to create a `` `Provider` `` element that encapsulates the App and its child components.

When a `button` is clicked, it uses the dispatch function to send the action value to the reducers. The `type` value is defined as a string and sent to the reducers that have several `if statements` to determine its response. When the reducers return a result, React immediately triggers a rerender, updating the content of the component.

NOTE: any component that imports reducers and initialState from the GlobalState.js file can use and alter the global state. Each component will have access to the same state and will be able to update it using the dispatch function provided by the useReducer hook.

Keep in mind that the global state is shared among all the components using this approach, so any updates to the state made by one component will be reflected in all other components using the same state. This can be very powerful for managing shared state across the application. However, with power comes responsibility, and you should carefully manage how and when the global state is updated to avoid unexpected behaviour in your application.

Compulsory Task 1

You are going to create a ReactJS application that manipulates the cash balance on an account using either the **useContext** or **useReducer** hook.

- All your data/state should be stored in a global state manager using any of the two hooks required for this task.
- This app should contain 4 descriptive buttons named:
    - **Withdraw**,
    - **Deposit**,
    - **Add Interest**,
    - and **Charges**.
- Each button must be styled using CSS or any React UI liberty – like **React Bootstrap**.
- Each button must be made from a **reusable button component**.
- The app will also contain one input box that you will use to enter the amount that you wish to deposit or withdraw.
- In your global state, create a property called `` `balance: 0` `` as the initial value.
    - Display this value on the UI. This value must also update when an account change is made.
- 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%.
    - Remember that users don't know what's happening in the background. You will have to make your button 'user-friendly' to indicate the amount calculated.
- Your App UI must have a heading with the app's name.

- The content must be centred and attractively styled. You can use custom CSS or any React UI library for styling.

Compulsory Task 2

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

- The initial state should be as follows:

```
const initialTodoState = {
   list: [
       {content1:"Content1", completed: false},
       {content2:"Content2", completed: false},
   ]
}
```

- Your reducers should allow a user to add, delete, and edit a specific to-do/task and change the completed variable.
- When your app starts, the UI should display:
  - A heading with your app name.
  - The 2 initial to-do items
  - A info icon (explained below)
  - A to-do counter (explained below)
  - An add-todo button
- Each to-do/task item rendered should then have four styled buttons.
  - **Add**,
  - **Edit**,
  - **Delete**,
  - and **Completed** (checkbox).
- **Add** or **Edit** a to-do:
  - On the UI using an input: A user should also be able to add a to-do using the 'enter key' or a button (both options should be available). When a to-do is added, clear the input.

- To Edit: you must use a **modal** or a custom HTML popup to edit a to-do. Use any HTML or React UI library **modal**/**popup** for this.
- When the user edits a to-do, fill in the input with the selected to-do's text.
- If a user tries to add a to-do with no value. Show a warning message in the popup.
- **Do not use** any browser methods like **prompt or alert.**

- On your UI, you must have an info icon. When clicked. A popup will display above all other content in the centre of the page with user instructions (**don't forget** to add a close button).

- On your UI, you must have an element to display the number of to-do items in your global state, i.e. a to-do counter.
  - When a to-do is added or deleted, this number should also update.
  - The counter should be in a fixed position on the UI (meaning, if the user scrolls down to other to-dos, the counter should still be visible)
- When a to-do is checked (**true**), the to-do should be faded out.
  - A user should not be able to edit completed to-dos.
- Your App UI content must be centred and attractively styled. You can use custom CSS or any React UI library for styling.

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