Overview

Software developers tend to use many tools and libraries to make the development of an application smooth and easy. However, in this process state management is one of the biggest things to be taken care of. State management is the process of maintaining the knowledge of application inputs to understand the condition of an application at a particular moment across a transaction. As we talk about state management, redux is the library that should come to a developer's mind. Now what is redux and why is it used is the main question. Redux is one such state management library that is used to manage the state of an application by storing the global state of a component in a redux store.

What is Redux?

Redux is a very popular open-source library that facilitates state management in applications. It is a cross-platform and open-source library and has taken its inspiration from Facebook's Flux architecture. It has eliminated the unnecessary complexities that existed in the Flux architecture.

Redux is commonly used with React using React Redux but can also be used with other UI frameworks such as Angular, Vue.js, and vanilla Javascript. However, both react and redux are used together but it is worth noting that they are independent of each other.

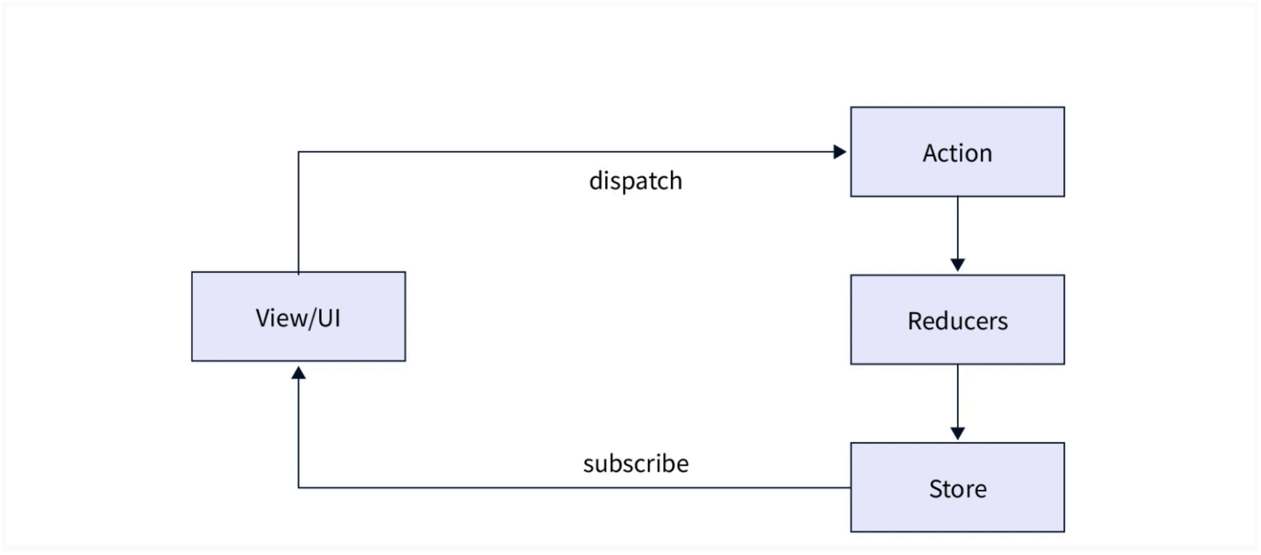
The main feature of Redux is to manage and update the state of an application. However, it is easier to manage the state of an application with a few components but as the components increase and the application becomes larger, it becomes very difficult the maintain the state of each component of your application.

Therefore, Redux comes to the rescue for such large applications in which it creates a container or a store that would have the state of the application, and whichever components need it can take it from there without passing down props from one component to another.

Redux uses three main components- a store, an action, and a reducer. A store is used to store an application's state. An action is an event that occurs when an application state is changed and sent to the redux store whereas reducers are pure functions that take the current state of the application, perform an action and return a new state.

We use React Redux as the official UI binding library for React which provided a lot of performance optimizations so that a component re-renders only when it is needed.

Therefore, with redux, you can easily track down when, where, and why a component has changed.



When to Use Redux?

Many people get confused about when to use Redux in their projects. Well, that depends on you whether you need it or not! The use of Redux is needed when your application grows and managing the app's state becomes cumbersome. This is the time when the developers tend to search for tools and libraries that can scale their applications easily and manage the state of each component.

Redux is used for maintaining and updating the state of multiple components to share in your application while the components remain independent of each other. However, without using redux, you need to make data-dependent of components by passing them through different components to where it is needed.

Moreover, redux is also used when the application state is changing frequently, that is, where there is a lot of user interaction with the website or the app such as an e-commerce website, in a payment process.

Advantages of Using Redux

The advantages of Redux are as follows-

- Redux makes the state predictable - In redux applications, the state is always predictable because the reducers are pure functions therefore, they will return the same result if you give them the same action and state. Moreover, the state is immutable which leads to simple debugging and programming as the data that never changes is easier to reason than the one that changes randomly throughout the app. It also has the ability to retrieve the previous states and get the results in real time.

- Redux is maintainable - Redux is quite strict about its structure, that is, how the code should be organized in an application. This feature of redux makes it easier for developers who have in-depth knowledge of redux to understand the structure of any Redux application. This structure constraint makes redux highly maintainable and helps to separate business logic from the component tree. However, for large applications that need frequent updations, redux is quite feasible.
- Debugging is easy in Redux - Redux provides excellent Dev Tools that are used for debugging purposes. In the case of large applications, debugging takes more time than the development of the application, therefore, DevTools of Redux gives an added advantage. Moreover, logging actions and states also helps to identify coding errors or any kind of bug in the application.
- Performance benefits - People might think that keeping an application's state global and keeping it in the redux store would degrade the performance of the application. However, that is not the case internally. The UI binding library of react known as React Redux implements several performance optimizations so that a component or a connected component re-renders only when it is needed.
- Ease of testing - Redux applications can be tested very easily as the functions can be used to change the state of pure functions.
- State persistence - You can save the state of redux applications in the local storage and retrieve them after a refresh.
- Server-side rendering - Service-side rendering is also possible with redux. It handles the initial rendering of the application by sending the state to the server along with the response to the server request. This feature of redux makes it preferable to the developers of other state management libraries.

Most Frequently Identified Drawbacks of Redux

Although, Redux has a lot of advantages that make it preferable to the developers, however, it also has some disadvantages that need to be addressed.

- Increased Complexity - However, redux has a lot of benefits but it has an additional complexity in layering while using actions and reducers.
- Restricted Design - It has a restricted design and has very minimum alternatives.
- Lack of Encapsulation - Redux does not encapsulate any data therefore, the security issues increase as an application becomes larger.
- Excessive Memory Use - As the state is immutable therefore when a state is updated, every time the reducer has to return a new state which leads to excessive memory usage in the long run. Therefore, the bigger the redux store, the higher the memory usage.
- Time-Consuming - Redux is very suitable for larger applications, however, for small or medium-level applications, redux can be very time-consuming because it requires more inputs of the boilerplate code beforehand.

Conclusion

- Redux is a state management library used for maintaining and updating the state of an application.
- Redux is very much suitable for large applications where the components keep on increasing which leads to difficulty in managing the state of an application.
- Redux uses three main components to manage the state - Redux store, action, and reducers.
- A redux store is used to store the state of a component globally. Whenever any component wants to retrieve the state, it directly accesses the redux store.
- An action in redux is an event that occurs when an application's state is changed.
- A reducer is a pure function that is used to retrieve the current state and return a new state by performing an action.
- Redux is maintainable, makes debugging easy, performs server-side rendering, has ease of testing, and has state persistence which makes it preferable for the developers.
- However, redux also has some disadvantages that it is time-consuming, lacks encapsulation, has excessive memory usage, increased complexity, and has a restricted design.

Overview

Redux is a powerful state management library for JavaScript applications that plays a crucial role in maintaining the flow of data within an application using a centralized data store. It is built on three fundamental principles that guide its design and usage. Understanding these principles is important for developers seeking to use Redux in their projects. Let us explore the core principles of Redux for managing state in complex applications.

Pre-requisites

Before understanding the principles of Redux, it's essential to have a basic understanding of the following technologies:

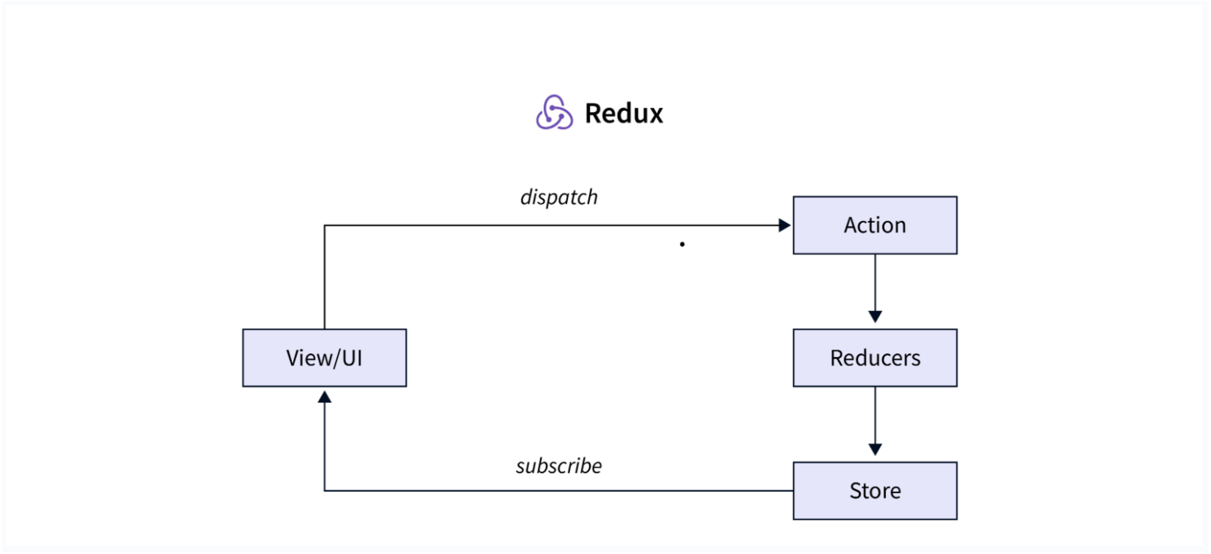
- Node.js:
  A JavaScript runtime that allows developers to run JavaScript on the server. Understanding Node.js is crucial for setting up a development environment, managing dependencies, and building client or server-side applications that are powered by Redux.
- React:
  React is a library in javascript for building user interfaces. Redux is commonly used along with React to manage the state of applications more predictably and efficiently. Hence a solid understanding of React is essential to comprehend Redux principles.
- React-Redux:
  Official React binding for Redux and it provides a set of helper functions that allow React components to interact with the Redux store seamlessly. Understanding of the React-Redux and flow of data between React components and the Redux store is important for implementing and understanding the three principles of Redux.

In addition to Node.js, React, and React-Redux, a broader understanding of the JavaScript ecosystem, including package managers like npm or Yarn. Familiarity with tools like Webpack for bundling, Babel for transpiling and ensuring compatibility across different environments, and ESLint for code linting contributes to setting up redux applications to handle dependencies, manage assets, and ensure that the codebase adheres to the latest ECMAScript standards.

Three Principles that Redux Follows

A Redux principle provides a structured approach to state management, promoting predictability, maintainability, and scalability in complex web applications. The principles are:

- Redux is a Single Source of Truth
- The State is a Read-only State
- Modifications are Done with Pure Functions



Redux is a Single Source of Truth

This principle is a foundational concept that emphasizes the centralization of an application's state in a single, well-defined location called the Redux store. This principle simplifying data management, debugging, and ensuring a consistent view of the application's data.

- In Redux, the entire state of an application is encapsulated within a single JavaScript object known as the Redux store.
- The store is created using the createStore function from the Redux library and is configured with a reducer, which determines how the state changes in response to dispatched actions.
- This store serves as the sole source of truth for the application's data. This means that every piece of data needed by the application is stored in this single location. The store

simplifies the debugging process and enhances predictability by keeping all data in one accessible location.

Let's consider a simple real-life example of a counter application.

Redux Actions:

```
// Redux Actions
// Defines action types for increment and decrement

// Redux Actions
export const INCREMENT = 'INCREMENT';
export const DECREMENT = 'DECREMENT';
```

Explanation:

This module defines action types that represent the events triggering state changes. In this example, we have two actions: INCREMENT and DECREMENT. These constants are exported to be used in other modules, ensuring consistency in action-type references.

Redux Reducer:

```
// Redux Reducer
// Manages the state of the counter within the Redux store

// Redux Reducer with initial state
export const counterReducer = (state = { value: 0 }, action) => {
  switch (action.type) {
    case INCREMENT:
      return { value: state.value + 1 };
    case DECREMENT:
      return { value: state.value - 1 };
    default:
      return state;
  }
};
```

Explanation:

This module specifies how the application's state changes in response to actions. The counterReducer takes the current state and an action as parameters, and based on the action type, it returns a new state. The initial state is set to { value: 0 }, representing the initial count value.

Redux Store Configuration:

```
// Creates the Redux store and exports it

// Redux Store
import { createStore } from 'redux';
import { counterReducer } from './ReduxReducer';

// create Redux store
const store = createStore(counterReducer);

// export store object
export default store;
```

Explanation:

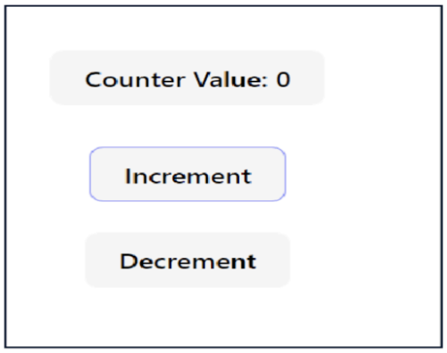
- This module configures the Redux store using the createStore function. The Redux store object has methods and properties that allow interaction with the application's state.
- It imports the counterReducer and initializes the store with this reducer, which is responsible for specifying how the application state changes in response to actions.
- The store includes methods like dispatch to dispatch actions, getState to retrieve the current state, subscribe to add listeners for state changes, and more.

React Component:

```
// React Component
// Defines the React component for the counter
// React Component
import React from 'react';
import { connect } from 'react-redux';
import { INCREMENT, DECREMENT } from './ReduxActions';

class CounterApp extends React.Component {
  render() {
    return (
      <div>
        <p>Counter Value: {this.props.value}</p>
        <button onClick={() =>
this.props.increment()}>Increment</button>
        <button onClick={() =>
this.props.decrement()}>Decrement</button>
      </div>
    );
  }
}

// React-Redux Connection
const mapStateToProps = state => ({
  value: state.value
});

const mapDispatchToProps = dispatch => ({
  increment: () => dispatch({ type: INCREMENT }),
  decrement: () => dispatch({ type: DECREMENT })
});

export const ConnectedCounterApp = connect(mapStateToProps,
mapDispatchToProps)(CounterApp);
```

Explanation:

- This module defines a React component responsible for rendering the counter UI. It uses the connect function from react-redux to connect the component to the Redux store.
- The mapStateToProps function maps the Redux state to the component's props, specifically mapping the value property from the Redux state to the value prop of the CounterApp component.
- The mapDispatchToProps maps the action creators to props. In this case, it maps the increment and decrement functions, which dispatch actions of type INCREMENT and DECREMENT respectively.



The code follows Single Source of Truth by the following:

- Redux Store:
  The counterReducer manages the state of the counter within the Redux store. The state is a single JavaScript object, where the value represents the current count.
- Redux Actions:
  Actions like INCREMENT and DECREMENT are dispatched to modify the state. These actions are the only way to initiate changes to the state, ensuring a controlled and predictable state transition.
- React Component:
  The React component is connected to the Redux store, and its props are mapped to the state and dispatch actions. This ensures that the component directly relies on the single source of truth (Redux store) for its data and actions.

The State is Read-only State

In Redux, the state after it is defined once is considered immutable, meaning that it cannot be changed directly. Instead, any modifications to the state must occur through the dispatching of actions, which are processed by reducers to create a new state. Immutability or ready-only state is important in Redux as it ensures predictability in state changes which helps in debugging and understanding how the application's state evolves over time, easier to understand about how actions impact the application's data flow as original state remains unchanged.

Let's consider an example of a to-do list application where the state, representing the list of tasks, follows the read-only principle.

Redux Actions

```
// Redux Actions
// Defines action types for adding tasks

// Redux Actions
export const ADD_TASK = 'ADD_TASK';
```

Redux Reducer:

```
// Manages the state of the tasks within the Redux store

// Redux Reducer
export const tasksReducer = (state = { tasks: [] }, action) => {
  switch (action.type) {
    case ADD_TASK:
      return { tasks: [...state.tasks, action.payload] };
    default:
      return state;
  }
};
```

Explanation:

The Redux reducer focuses on managing the state of tasks. The tasksReducer takes the current state and an action as parameters, and based on the action type (ADD_TASK), it returns a new state with the updated list of tasks.

Redux Store

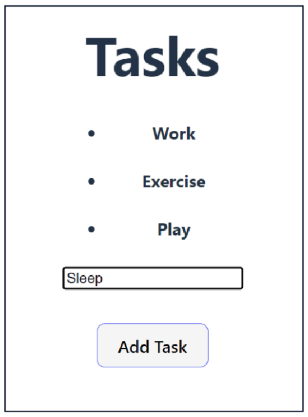
```
// Creates the Redux store and exports it

// Redux Store
import { createStore } from 'redux';
import { tasksReducer } from './ReduxReducer';

const store = createStore(tasksReducer);

export default store;
```

React Component:

```
// Defines the React component for the todo application

import React from 'react';
import { connect } from 'react-redux';
import { ADD_TASK } from './ReduxActions';

// class component
class TodoApp extends React.Component {
  constructor() {
    super();
    // initialize task
    this.state = { newTask: '' };
  }
  // update input state
  handleInputChange = (event) => {
    this.setState({ newTask: event.target.value });
  };

  handleAddTask = () => {
    // get current state
    const newTask = this.state.newTask;
    // dispatch action to update tasks
    this.props.addTask(newTask);
    this.setState({ newTask: '' });
  };

  render() {
    return (
      <div>
          <h1> Tasks </h1>
        <ul>
          {this.props.tasks.map((task, index) => (
            <li key={index}>{task}</li>
          ))}
        </ul>
        <input
          type="text"
          value={this.state.newTask}
          onChange={this.handleInputChange}
        />
        <button onClick={this.handleAddTask}>Add Task</button>
      </div>
    );
  }
}

// React-Redux Connection
const mapStateToProps = state => ({
  tasks: state.tasks
});

const mapDispatchToProps = dispatch => ({
  addTask: newTask => dispatch({ type: ADD_TASK, payload: newTask
})
});

export const ConnectedTodoApp = connect(mapStateToProps,
mapDispatchToProps)(TodoApp);
```

Explanation:

The code uses the connect function from react-redux to connect the component to the Redux store. The mapStateToProps function maps the Redux state to the component's props, and mapDispatchToProps maps the action creator (addTask) to props.



The code follows The State is Read-only by using the following:

- Redux Reducer:
  The tasksReducer defines how the state should be modified in response to actions. It follows the read-only principle by returning a new state object rather than modifying the existing one directly.

- Redux Store and Dispatch:
  The Redux store is created with the tasksReducer. When a new task needs to be added, the ADD_TASK action is dispatched, carrying the payload (new task). This action is processed by the reducer, and a new state is created.
- Rendering:
  The tasks are displayed in the component based on the current state obtained from the Redux store. The read-only nature ensures that the UI reflects the most recent state without direct manipulation.

The Modifications are Done with Pure Functions

Redux relies on pure functions, known as reducers, to specify how the application's state changes in response to actions. A pure function is a function that, given the same input, will always return a predictable output without causing any side effects. Pure functions ensures that the state changes are consistent and reproducible and do not produce side effects, such as modifying external variables or interacting with the DOM.

Let's consider a simple counter application to exemplify the use of pure functions (reducers) to manage state changes in Redux.

```
// Redux Actions
const INCREMENT = 'INCREMENT';
const DECREMENT = 'DECREMENT';

// Redux Reducer
const cReducer = (state = { count_num: 0 }, action) => {
  switch (action.type) {
    case INCREMENT:
      return { count_num: state.count_num + 1 };
    case DECREMENT:
      return { count_num: state.count_num - 1 };
    default:
      return state;
  }
};

// Redux Store
const { createStore } = Redux;
const store = createStore(cReducer);

// Subscribe to Redux Store Changes
store.subscribe(() => {
  console.log('Current Count:', store.getState().count_num);
});

// Dispatch Actions
store.dispatch({ type: INCREMENT });
store.dispatch({ type: INCREMENT });
store.dispatch({ type: DECREMENT });
```

Explanation:

- The counterReducer is a pure function that takes the current state and an action as parameters. Depending on the action type, it returns a new state. In this case, for INCREMENT, it increments the count by 1, and for DECREMENT, it decrements the count by 1.
- The Redux store is created and the store.subscribe() method is then used to subscribe to changes in the Redux store. Whenever an action is dispatched and the state is modified, the callback function within subscribe is executed. In this case, it logs the current count to the console.
- Actions are dispatched to the Redux store using the store.dispatch() method. As each action is dispatched, the cReducer is called to calculate the new state based on the current state and the action type. The store subscription ensures that the updated count is logged to the console after each action, providing visibility into the state changes.

The code follows Modifications with Pure Functions by using the following:

- Redux Reducer:
  The counterReducer is a pure function that takes the current state and an action as parameters and returns a new state without mutating the original state. It follows the principle of immutability, ensuring predictability and traceability.
- Redux Store and Dispatch:
  The Redux store is created with the cReducer. Actions (INCREMENT and DECREMENT) are dispatched to the store, triggering the reducer to create a new state based on the dispatched actions.
- Subscribe to Store Changes:
  The store.subscribe() method is used to listen to changes in the Redux store. Upon each action dispatch, the new count is logged to the console, showcasing the updated state without direct modifications.

The predictability of pure functions simplifies testing and debugging. Developers can isolate and test reducers independently, ensuring that state changes are precisely as expected, contributing to a more robust and predictable codebase.

Output:

Executing the provided code will result in the following output in the console:

```
Current Count: 1
Current Count: 2
Current Count: 1
```

The output shows the sequential modifications to the count state using pure functions (reducers) in Redux. Each action dispatch triggers the reducer, creating a new state based on the previous state and the action type, without mutating the original state.

Conclusion

- Redux is a widely used state management library in JavaScript, often integrated with React for efficient state handling in applications. Familiarity with Node.js, React, React-Redux, and tools like Webpack and Babel, are all essential for working with redux in applications.
- The three key principles of Redux include concepts to ensure a centralized and accessible store, enforce immutability to achieve predictability, and utilize pure functions (reducers) to transition between states.
- The Single Source of Truth principle promotes a unified state in a central store or Redux store for simplifying debugging and testing processes in Redux-powered applications.
- Immutability in Redux ensures that the state of an application cannot be modified directly. This concept is crucial for predictability, debugging, and maintaining a consistent state transition.
- The State is read-only principle enforces immutability and prevents side effects by preventing direct state modifications. This enhances predictability and traceability in complex applications.
- Pure functions or reducers in Redux, play an important role in ensuring predictable state changes. These functions are deterministic, producing the same output for the same input, and they have no side effects.
- The Modifications are done with the pure functions principle involves using reducers to create newer states from older states, ensuring consistent and predictable state transitions in Redux applications.

Getting Started with Redux in React

Let’s take a look at what Redux in React is.

Redux is a State Management tool used for JavaScript applications. This is a very well-known tool for managing states using a centralized store known as Redux Store. This can be easily used with React apps.

These three main tenets form the foundation of Redux:

- A single state object known as the state or state tree houses all of the app's data.
- States in Redux is read-only. Only sending an action to the store will modify it.
- A function that accepts the current state and the desired action must be written in order to explain state changes. A newly updated state, not a tweaked version of the current state, must be returned by the function. Reducers are the term for these operations.

Installation

Let's take a look at how the installation of Redux in React is done. First, we will take a look at how it is done in a completely new React project, then we shall see how it is done in an existing React project.

Creating a new React project and installing Redux in React

First, we need to create a new React project using the following line of code:

```
npx create-react-app mynewreactapp
```

-

Then enter the folder using cd as follows:

```
cd mynewreactapp
```

-

Then install redux and react-redux as follows:

```
npm install redux react-redux
```

-

Note: If there is any error installing the two together then install them one by one as follows:

```
npm i redux
```

Once, the installation of redux is complete, then go for the installation of react-redux

```
npm i react-redux
```

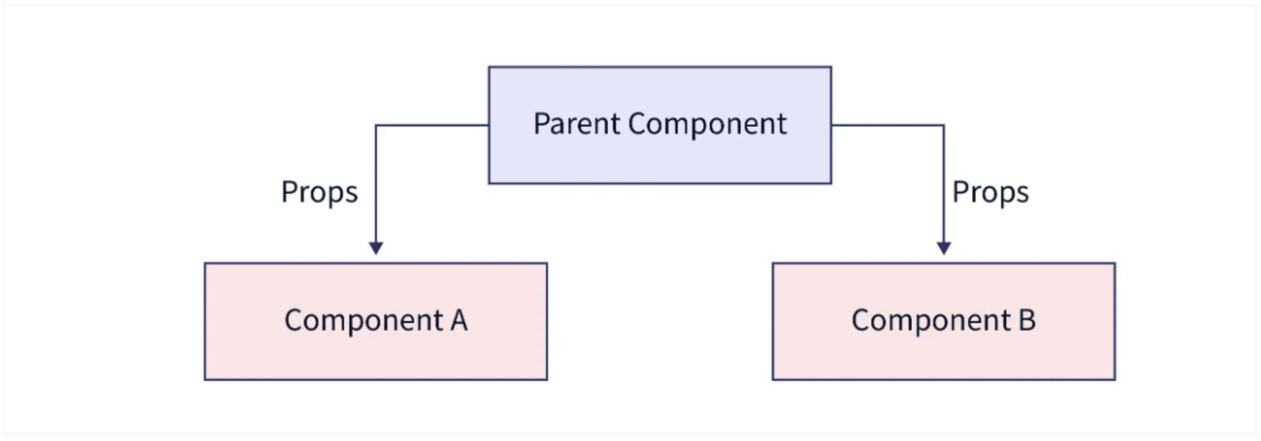
We are now good to get started with a brand new project which supports Redux in React!

Installing Redux in an existing React project

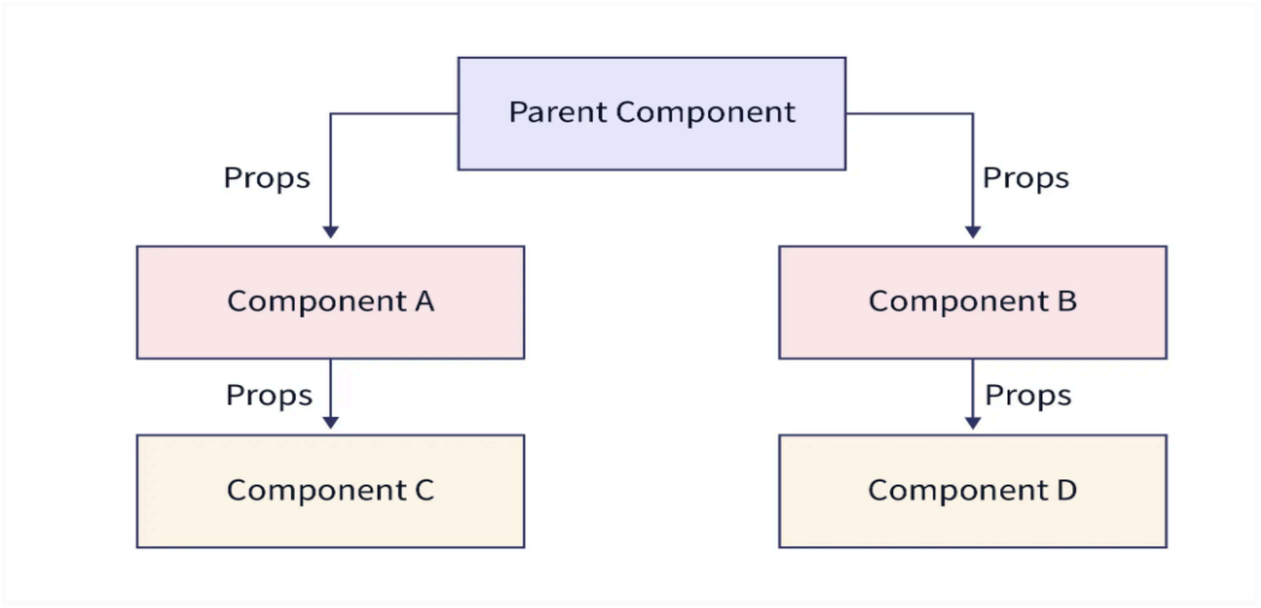
You might have guessed it. If the React app is already created then we can simply get into the project folder and install redux using step 3 as above! Simple, right?

Why Use Redux in React?

State management in React could be easily done with props as you might know. The technique is known as prop drilling. Let’s take a simple example first.



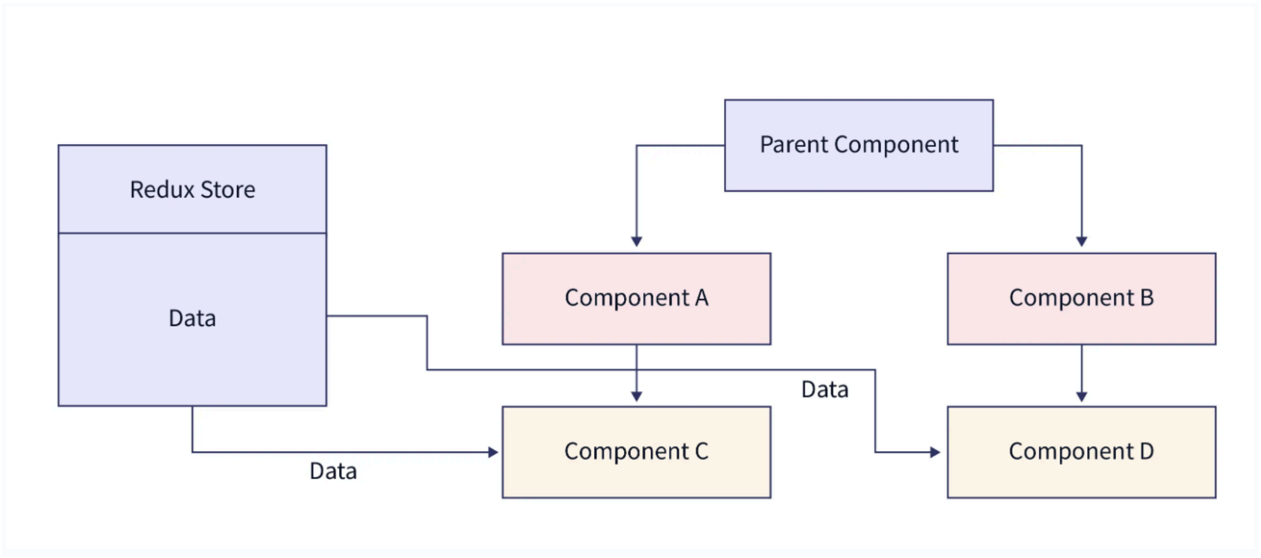
A good illustration of React's unidirectional data flow is this. Through read-only characteristics known as "props," the parent component in this illustration gives the child components a snapshot of its current state. The data given down from the "Parent Component" is shared by "Component A" and "Component B." The next instance demonstrates where the problems start.



What direction is this moving in? Not even necessarily huge projects using React frequently experience this. If Component C and Component D require the same data, you must send it from Parent Component, which is also their closest common ancestor. The data passes through several tiers of intermediary parts before arriving at some of them, which might not even require the data.

This method of data transmission can cause a lot of problems. You are unable to freely move any component in this structure since every component is currently closely connected. The performance of your app may also be impacted because every state modification necessitates a re-rendering of all child components. It is time-consuming and frequently frustrating to have to rewrite the code of numerous components to set up the data flow from top to bottom of the tree if two components need the same data.

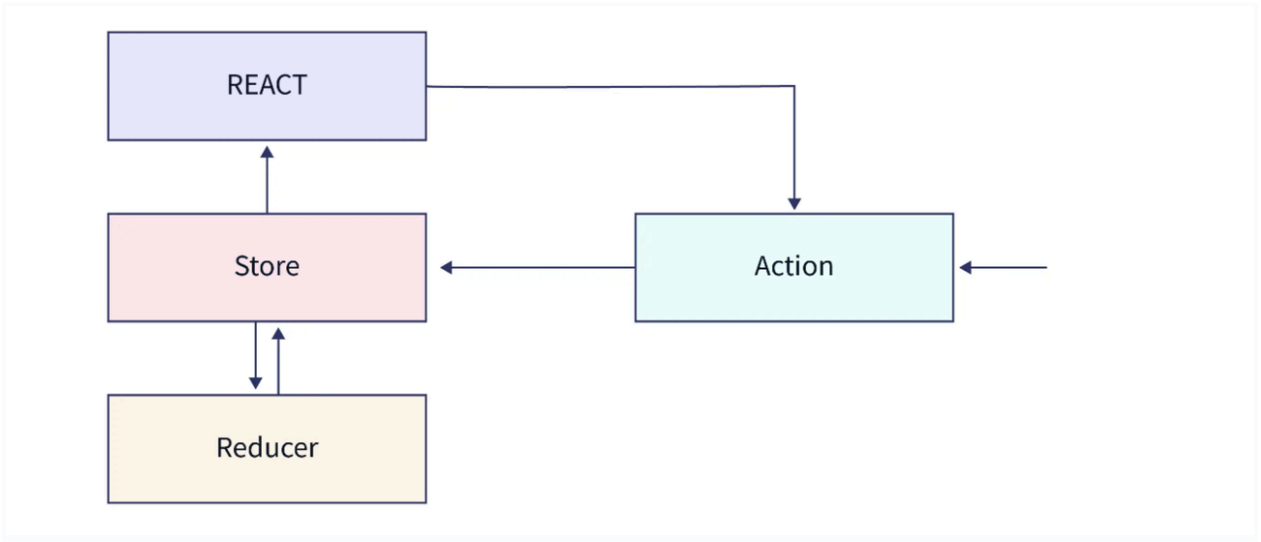
Here Redux comes into play. For JavaScript applications, Redux is a predictable state container. You may use it to create applications that function consistently, work in client, server, and native contexts, and are simple to test. Redux uses a single Store global object to control an application's state.



- Redux is a tool for managing states.
- Any JavaScript framework or library can utilize Redux.
- Redux saves the application's state, and its components may access it through a state store.

Redux Architecture

MVC, which stands for Model, View, and Controller, is now one of the most well-liked and often-used design patterns. In MVC, the application model is built using the raw data. The logic takes the raw input and transforms it into functions that manage the application's state. The Redux architecture conforms to the flowchart below based on the mechanism.



The flutter architecture is virtually identical to redux, and both are derived from it. Redux, a library, functions as a state container and aids in controlling an application's data flow. All React components may subscribe to Redux's store, which is one of the two biggest enhancements it provides. The modifications to the state of your application will be determined by a reducer inside that store.

Creating a New ReactJS Based Project and Adding Redux to it

In this example, we will make a simple Counter, where we will have a value that we can increase or decrease using buttons.

Installations

First things first, let's create a new React Project using the create-react-app command. After that is done we will enter the directory and install React Redux as follows.

```
npm install redux react-redux
```

You will require these three items to use Redux:

- Actions: These objects should have two properties: one indicating the sort of action to be taken, and the other specifying the changes that should be made to the app state.
- Reducers: These are the functions that put the actions' behavior into practice. Based on the description of the action and the description of the state change, they alter the app's state.
- Store: There is just one store, and it connects the actions and reducers, holding and altering the state for the entire program.

Project Structure

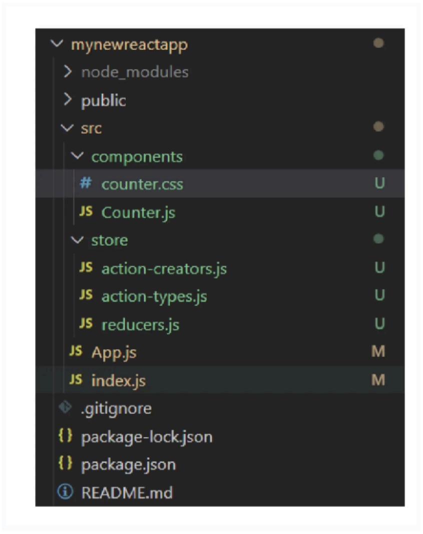
Let’s arrange the Project Structure properly so that it is easy to maintain. Inside the src, we will add two new folders:

- components folder
- store folder

Inside components we will have all our components for the application.

Inside store we will have three files:

- action-types.js: this will contain all the possible types of actions that can take place in our application.
- action-creators.js: we’ll define our action creators. As you may remember, actions are plain JavaScript objects that describe what happens in our application.
- reducers.js: we define our app’s initial state and our reducer.



```
mynewreactapp
  node_modules
  public
  src
    components
      counter.css          U
      Counter.js           U
    store
      action-creators.js   U
      action-types.js      U
      reducers.js          U
    App.js                 M
    index.js               M
  .gitignore
  package-lock.json
  package.json
  README.md
```

This is how things will look inside the project directory once we are done setting it up.

Now, let’s get started with some codes! Time to get your hands dirty!

Action

action-types.js:

First, let’s define the actions that can occur in action-types.js. In our case, we will have only two, increase and decrease.

```
export const TYPE_DECREMENT = 'DECREMENT'
export const TYPE_INCREMENT = 'INCREMENT'
```

action-creator.js:

Now, let’s define our action creators.

```
import { TYPE_DECREMENT, TYPE_INCREMENT } from "./action-types";

export const decrement = () => ({
    type: TYPE_DECREMENT,
});

export const increment = () => ({
    type: TYPE_INCREMENT,
});
```

As you can see, these functions are typical ones that produce objects (actions) having a type attribute. Depending on the action types in our reducers, we'll take different actions. Only the type attribute is required and is sufficient for our example, but actions would contain extra data in more complex applications.

Reducers

We will be defining our reducers along with our app’s initial state here:

```
import { TYPE_DECREMENT, TYPE_INCREMENT } from "./action-types";

const initialState = {
    counter: 0,
};

export const counterReducer = (state = initialState, action) => {
    switch (action.type) {
        case TYPE_DECREMENT:
            return {
                ...state,
                counter: state.counter - 1,
            };
        case TYPE_INCREMENT:
            return {
                    ...state,
                    counter: state.counter + 1,
               };
          default:
               return state;
     }
};
```

Every time an action is sent in a Redux application, redux calls each reducer, sending the current state as the first parameter and the most recent action as the second parameter. One reducer will be used in our straightforward app. Here, we take three alternative actions based on the type of action:

- Bring back a new state with a higher counter value.
- New state with a lower counter value returned
- Restore the previous state.

Simple, yes? Observe that the only language used thus far in these files is JavaScript; neither Redux nor React-Redux have been imported.

Building the Counter Component

Let us now build the counter component in the counter.js file.

```
import React from "react";
import { connect } from "react-redux";

import { increment, decrement } from "../store/action-creators";
import "./counter.css";

export const CounterComponent = ({
     counter,
     handleIncrement,
     handleDecrement,
}) => {
     return (
          <>
               Counter: {counter}
               <button onClick={handleIncrement}>+</button>
               <button onClick={handleDecrement}>-</button>
          </>
```

```
    );
};

const mapStateToProps = ({ counter }) => ({
    counter,
});

const mapDispatchToProps = {
    handleIncrement: increment,
    handleDecrement: decrement,
};

export const Counter = connect(
    mapStateToProps,
    mapDispatchToProps
)(CounterComponent);
```

Presentational component CounterComponent calls handleIncrement or handleDecrement on button clicks and displays the current counter.

The function mapStateToProps accepts the parameter state. We get "state.counter" using ES6 destructuring, and the object we provide translates the required piece of state to the properties of the component.

Our action creators are mapped to the props of the component using an object called mapDispatchToProps.

Next, we send our CounterComponent and the linked component to connect, which produces a new function that will accept both of them. Observe how to connect handles access to the store for us behind the scenes so that our component doesn't have direct access to it.

Connecting Redux and React

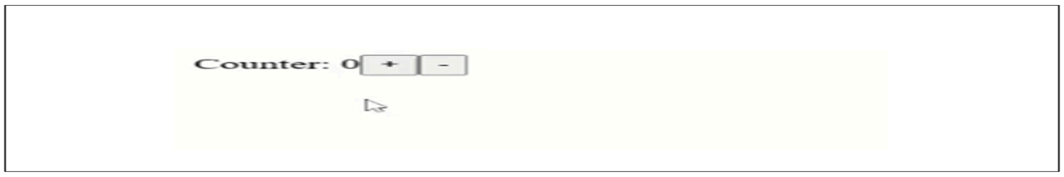
Now is the moment to merge Redux with React. Go to src/index.js and add the following code to its existing content:

```
import React from "react";
import ReactDOM from "react-dom";

import App from "./App";
import { legacy_createStore as createStore } from "redux";
import { Provider } from "react-redux";
import { counterReducer } from "./store/reducers";

const store = createStore(
    counterReducer,
    window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__()
);

ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById("root")
);
```

Our app's entrance point is this. There are some adjustments we're making. By providing our reducer to the createStore method, we can create our store. The App component is then enclosed in a Provider component, making the store accessible to our React app.

And of course, you are free to use your own creativity in the counter.css file to stylize your Counter app using Redux in React.

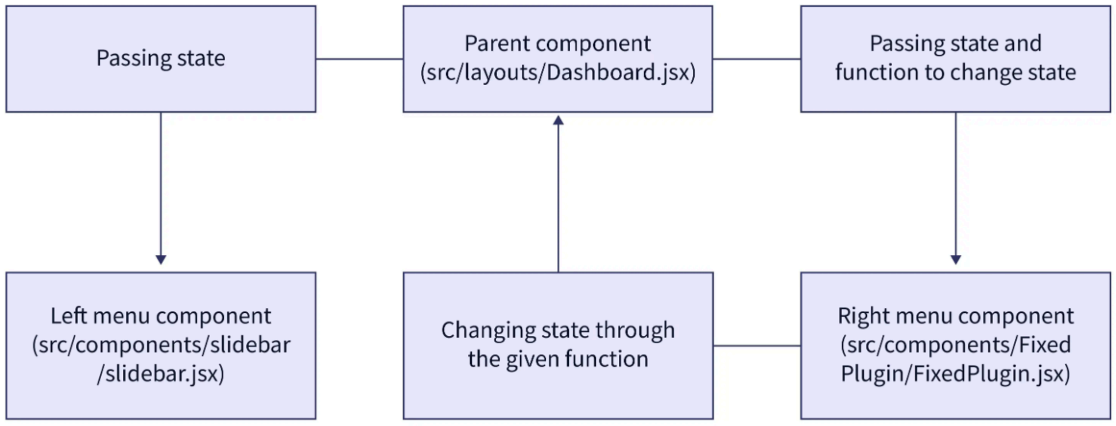
Final Output



That's it, we have learned how to use Redux in React using this simple Counter application as an example. Let's now jump into something very real.

A Real-Life example with Paper Dashboard React

We will be utilizing the right menu to alter the colors of the left menu, as you can see in the gif image above. To accomplish this, component states are used, and the state is sent from a parent component to two menus and a few methods to modify the state.


Passing state
Parent component (src/layouts/Dashboard.jsx)
Passing state and function to change state
Left menu component (src/components/slidebar/slidebar.jsx)
Changing state through the given function
Right menu component (src/components/FixedPlugin/FixedPlugin.jsx)

Let us take this product and replace the component states with Redux in React.

Cloning the Existing Git Repository

We will start by cloning this git repository using the following command:

```
git clone
https://github.com/creativetimofficial/paper-dashboard-react.git
```

Installation

Let us cd into the project directory and install the Redux as we have done in the previous example.

```
npm install redux react-redux
```

Let's get started with the coding part. We will be making a few additions and a few changes to the existing code.

Actions

We then need to create the actions. We need 7 actions, or 2 actions creators because the right menu has 2 colors that set the backdrop of the left menu and 5 colors that modify the color of the links. We have chosen the latter option because it requires a little less code to produce.

Let's create the action folder inside src, then inside the action folder, we create two more files for the two different actions:

- setBgAction.js
- setColorAction.js

Let's create each of the two actions now:

setBgAction.js

```
const setBgAction = (payload) => {
  return {
    type: "bgChange",
    payload
  }
}
export default setBgAction;
```

setColorAction.js

```
const setColorAction = (payload) => {
  return {
    type: "colorChange",
    payload
  }
}
export default setColorAction;
```

Reducers

Now, as we did in the first example, we need reducers here as well.

Let’s create a folder named reducers inside src and create a file rootReducer.js inside it.

Let’s code the rootReducer.js now.

```
export default (state, action) => {
  switch (action.type) {
    case "bgChange":
      return {
        ...state,
        bgColor: action.payload
      };
    case "colorChange":
      return {
        ...state,
        activeColor: action.payload
      };
    default:
      return state;
  }
};
```

Store

Let’s create the store.js as well now inside src.

```
import { createStore } from "redux";
import rootReducer from "reducers/rootReducer";

function configureStore(state = { bgColor: "black", activeColor:
"info" }) {
  return createStore(rootReducer,state);
}
export default configureStore;
```

Changes to index.js

As all of you know, this is the starting point of our app. So to add Redux to the existing file we need to make the necessary imports first.

```
// new imports start
import { Provider } from "react-redux";
import configureStore from "store";
// new imports stop
```

Once importing new packages are done, we will have to make some changes to the render function as well. The changed render function should look like this:

```
ReactDOM.render(
  <Provider store={configureStore()}>
    <Router history={hist}>
      <Switch>
        {indexRoutes.map((prop, key) => {
          return <Route path={prop.path} key={key}
component={prop.component} />;
        })}
      </Switch>
    </Router>
  </Provider>,
  document.getElementById("root")
);
```

Once these are done our index.js is ready! Let’s take a look at the entire index.js file.

```
import React from "react";
import ReactDOM from "react-dom";
import { createBrowserHistory } from "history";
import { Router, Route, Switch } from "react-router-dom";
// new imports start
import { Provider } from "react-redux";

import configureStore from "store";
// new imports stop

import "bootstrap/dist/css/bootstrap.css";
import "assets/scss/paper-dashboard.scss";
import "assets/demo/demo.css";

import indexRoutes from "routes/index.jsx";

const hist = createBrowserHistory();

ReactDOM.render(
    <Provider store={configureStore()}>
        <Router history={hist}>
            <Switch>
                {indexRoutes.map((prop, key) => {
                    return (
                        <Route
                            path={prop.path}
                            key={key}
                            component={prop.component}
                        />
                    );
                })}
            </Switch>
        </Router>
    </Provider>,
    document.getElementById("root")
);
```

Dashboard.jsx

The src/layouts/Dashboard/Dashboard.jsx file has to be modified right now. Both the state and the functions that alter the state must be deleted. So feel free to REMOVE the following code fragments:

Between lines 16 and 22 [The constructor]:

```
constructor(props){
  super(props);
  this.state = {
    backgroundColor: "black",
    activeColor: "info",
  }
}
```

Between lines 41 and 46 [The state functions]:

```
handleActiveClick = (color) => {
    this.setState({ activeColor: color });
  }
handleBgClick = (color) => {
  this.setState({ backgroundColor: color });
}
```

Lines 53 and 54 [The sidebar bgColor and activeColor props]:

```
bgColor={this.state.backgroundColor}
activeColor={this.state.activeColor}
```

Between lines 59 and 62 [FixedPlugin props]:

```
bgColor={this.state.backgroundColor}
activeColor={this.state.activeColor}
handleActiveClick={this.handleActiveClick}
handleBgClick={this.handleBgClick}
```

Let’s take a look at the remaining code in Dashboard.jsx after all the removals.

```
import React from "react";
// javascript plugin used to create scrollbars on windows
import PerfectScrollbar from "perfect-scrollbar";
import { Route, Switch, Redirect } from "react-router-dom";

import Header from "components/Header/Header.jsx";
import Footer from "components/Footer/Footer.jsx";
import Sidebar from "components/Sidebar/Sidebar.jsx";
import FixedPlugin from "components/FixedPlugin/FixedPlugin.jsx";

import dashboardRoutes from "routes/dashboard.jsx";

var ps;

class Dashboard extends React.Component {
    componentDidMount() {
        if (navigator.platform.indexOf("Win") > -1) {
            ps = new PerfectScrollbar(this.refs.mainPanel);
            document.body.classList.toggle("perfect-scrollbar-on");
        }
    }
    componentWillUnmount() {
        if (navigator.platform.indexOf("Win") > -1) {
            ps.destroy();
            document.body.classList.toggle("perfect-scrollbar-on");
        }
    }
    componentDidUpdate(e) {
        if (e.history.action === "PUSH") {
            this.refs.mainPanel.scrollTop = 0;
            document.scrollingElement.scrollTop = 0;
        }
    }
    render() {
        return (
            <div className="wrapper">
                <Sidebar {...this.props} routes={dashboardRoutes}
/>
                <div className="main-panel" ref="mainPanel">
                    <Header {...this.props} />
                    <Switch>
                        {dashboardRoutes.map((prop, key) => {
                            if (prop.pro) {
                                return null;
                            }
                            if (prop.redirect) {
                                return (
                                    <Redirect
                                        from={prop.path}
                                        to={prop.pathTo}
                                        key={key}
                                    />
                                );
                            }
                            return (
                                <Route
                                    path={prop.path}
                                    component={prop.component}
                                    key={key}
                                />
                            );
                        })}
                    </Switch>
                    <Footer fluid />
                </div>
                <FixedPlugin />
            </div>
        );
    }
}

export default Dashboard;
```

Connecting Sidebar and FixedPligin Components to Store

Sidebar.jsx

We need to import connect from react-redux.

```
import { connect } from "react-redux";
```

Change the export to:

```
const mapStateToProps = state => ({
  ...state
});

export default connect(mapStateToProps)(Sidebar);
```

FixedPlugin.jsx

```
import { connect } from "react-redux";
import setBgAction from "actions/setBgAction";
import setColorAction from "actions/setColorAction";
```

And this is what the export should now be:

```
const mapStateToProps = state => ({
  ...state
});

const mapDispatchToProps = dispatch => ({
  setBgAction: (payload) => dispatch(setBgAction(payload)),
  setColorAction: (payload) => dispatch(setColorAction(payload))
});

export default connect(mapStateToProps,
mapDispatchToProps)(FixedPlugin);
```

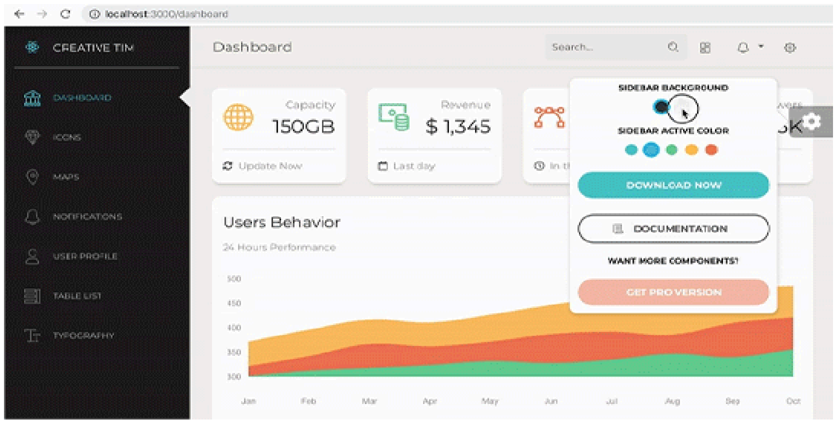
The following changes will occur:

- You must replace any instances of handleBgClick with setBgAction.
- You must replace any instances of handleActiveClick with setColorAction.

As a result, the FixedPlugin component should now appear as follows:

```
import React, { Component } from "react";

import { connect } from "react-redux";
import setBgAction from "actions/setBgAction";
import setColorAction from "actions/setColorAction";

import Button from "components/CustomButton/CustomButton.jsx";

class FixedPlugin extends Component {
    constructor(props) {
        super(props);
        this.state = {
            classes: "dropdown show",
        };
        this.handleClick = this.handleClick.bind(this);
    }
    handleClick() {
        if (this.state.classes === "dropdown") {
            this.setState({ classes: "dropdown show" });
        } else {
            this.setState({ classes: "dropdown" });
        }
    }
    render() {
        return (
            <div className="fixed-plugin">
                <div className={this.state.classes}>
                    <div onClick={this.handleClick}>
                        <i className="fa fa-cog fa-2x" />
                    </div>
                    <ul className="dropdown-menu show">
                        <li className="header-title">SIDEBAR
BACKGROUND</li>
                        <li className="adjustments-line">
                            <div className="badge-colors
text-center">
                                <span
                                    className={
                                        this.props.bgColor ===
"black"
                                            ? "badge filter
badge-dark active"
                                            : "badge filter
badge-dark"
                                    }
                                    data-color="black"
                                    onClick={() => {

this.props.setBgAction("black");
                                    }}
                                />
                                <span
                                    className={
                                        this.props.bgColor ===
"white"
                                            ? "badge filter
badge-light active"
                                            : "badge filter
badge-light"
                                    }
                                    data-color="white"
                                    onClick={() => {

this.props.setBgAction("white");
                                    }}
                                />
                            </div>
                        </li>
                        <li className="header-title">SIDEBAR ACTIVE
COLOR</li>
                        <li className="adjustments-line">
                            <div className="badge-colors
text-center">
                                <span
                                    className={
                                        this.props.activeColor ===
"primary"
                                            ? "badge filter
badge-primary active"
                                            : "badge filter
badge-primary"
                                    }
                                    data-color="primary"
                                    onClick={() => {

this.props.setColorAction("primary");
                                    }}
                                />
                                <span
                                    className={
                                        this.props.activeColor ===
"info"
                                            ? "badge filter
badge-info active"
                                            : "badge filter
badge-info"
                                    }
                                    data-color="info"
                                    onClick={() => {

this.props.setColorAction("info");
                                    }}
                                />
                                <span
                                    className={
                                        this.props.activeColor ===
"success"
                                            ? "badge filter
badge-success active"
                                            : "badge filter
badge-success"
                                    }
                                    data-color="success"
                                    onClick={() => {

this.props.setColorAction("success");
                                    }}
                                />
                                <span
                                    className={
                                        this.props.activeColor ===
"warning"

                                            ? "badge filter
badge-warning active"

                                            : "badge filter
badge-warning"
                                    }
                                    data-color="warning"
                                    onClick={() => {

this.props.setColorAction("warning");
                                    }}
                                />
                                <span
                                    className={
                                        this.props.activeColor ===
"danger"

                                            ? "badge filter
badge-danger active"

                                            : "badge filter
badge-danger"
                                    }
                                    data-color="danger"
                                    onClick={() => {

this.props.setColorAction("danger");
                                    }}
                                />
                            </div>
                        </li>
                        <li className="button-container">
                            <Button

href="https://www.creative-tim.com/product/paper-dashboard-react"
                                color="primary"
                                block
                                round
                            >
                                Download now
                            </Button>
                        </li>
                        <li className="button-container">
                            <Button

href="https://www.creative-tim.com/product/paper-dashboard-react/#/
documentation/tutorial"
                                color="default"
                                block
                                round
                                outline
                            >
                                <i className="nc-icon
nc-paper"></i>{" "}
                                Documentation
                            </Button>
                        </li>
                        <li className="header-title">Want more
components?</li>
                        <li className="button-container">
                            <Button

href="https://www.creative-tim.com/product/paper-dashboard-pro-reac
t"
                                color="danger"
                                block
                                round
                                disabled
                            >
                                Get pro version
                            </Button>
                        </li>
                    </ul>
                </div>
            </div>
        );
    }
}

const mapStateToProps = (state) => ({
    ...state,
});

const mapDispatchToProps = (dispatch) => ({
    setBgAction: (payload) => dispatch(setBgAction(payload)),
    setColorAction: (payload) => dispatch(setColorAction(payload)),
});

export default connect(mapStateToProps,
mapDispatchToProps)(FixedPlugin);
```

We're done, so you may start the project and check that everything functions as intended:



Multiple Reducers

There can be as many reducers as there are actions and vice versa. The only need is that you combine them; we'll learn how to achieve this further down.

Reducers

Create two new reducers for our application, one for the setBgAction and one for the setColorAction:

bgReducer.js:

```
export default (state = {}, action) => {
  switch (action.type) {
    case "bgChange":
      return {
        ...state,
        bgColor: action.payload
      };
    default:
      return state;
  }
};
```

colorReducer.js:

```
export default (state = {} , action) => {
  switch (action.type) {
    case "colorChange":
      return {
        ...state,
        activeColor: action.payload
      };
    default:
      return state;
  }
};
```

Code Explanation

You must include a default state for each of your reducers that will be merged when dealing with combined reducers. In our situation, we've selected an empty object, state = {};

Following this, our rootReducer will merge these two:

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

import bgReducer from 'reducers/bgReducer';
import colorReducer from 'reducers/colorReducer';

export default combineReducers({
  activeState: colorReducer,
  bgState: bgReducer
});
```

Code Explanation

Therefore, we indicate that the colorReducer should be referenced by the app state's activeState prop, and the bgReducer should be referred to by the app state's bgState prop.

States

Since, we have two different reducers now, instead of one. So, the way we represent our states will also change.

This is what it used to look like:

```
state = {
  activeColor: "color1",
  bgColor: "color2"
}
```

It will now look like this:

```
state = {
  activeState: {
    activeColor: "color1"
  },
  bgState: {
    bgColor: "color2"
  }
}
```

Redux Store

We now need to alter our store because we modified our reducers and merged them into simply one reducer likewise.

store.js:

```
import { createStore } from "redux";
import rootReducer from "reducers/rootReducer";

// we need to pass the initial state with the new look
function configureStore(state = { bgState: {bgColor: "black"},
activeState: {activeColor: "info"} }) {
  return createStore(rootReducer,state);
}
export default configureStore;
```

Changes To Existing Components

We now need to update the props within the Sidebar and FixedPlugin components to the new state object because we altered the state's appearance.

Sidebar.jsx (src/components/Sidebar):

amend line 36 from

```
<div className="sidebar" data-color={this.props.bgColor}
data-active-color={this.props.activeColor}>
```

To

```
<div className="sidebar" data-color={this.props.bgState.bgColor}
data-active-color={this.props.activeState.activeColor}>
```

FixedPlugin.jsx:

Now, we need to alter all occurrences of this.props.bgColor to this.props.bgState.bgColor. And all those of this.props.activeColor to this.props.activeState.activeColor.

Therefore, the new code should look something like this:

```
import React, { Component } from "react";

import Button from "components/CustomButton/CustomButton.jsx";

import { connect } from "react-redux";
import setBgAction from "actions/setBgAction";
import setColorAction from "actions/setColorAction";

class FixedPlugin extends Component {
  constructor(props) {
    super(props);
    this.state = {
      classes: "dropdown show"
    };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    if (this.state.classes === "dropdown") {
      this.setState({ classes: "dropdown show" });
    } else {
      this.setState({ classes: "dropdown" });
    }
  }
  render() {
    return (
      <div className="fixed-plugin">
        <div className={this.state.classes}>
          <div onClick={this.handleClick}>
            <i className="fa fa-cog fa-2x" />
          </div>
          <ul className="dropdown-menu show">
            <li className="header-title">SIDEBAR BACKGROUND</li>
            <li className="adjustments-line">
              <div className="badge-colors text-center">
                <span
                  className={
                    this.props.bgState.bgColor === "black"
                      ? "badge filter badge-dark active"
                      : "badge filter badge-dark"
                  }
                  data-color="black"
                  onClick={() => {
                    this.props.setBgAction("black");
                  }}
                />
                <span
                  className={
                    this.props.bgState.bgColor === "white"
                      ? "badge filter badge-light active"
                      : "badge filter badge-light"
                  }
                  data-color="white"
                  onClick={() => {
                    this.props.setBgAction("white");
                  }}
                />
              </div>
            </li>
            <li className="header-title">SIDEBAR ACTIVE COLOR</li>
            <li className="adjustments-line">
              <div className="badge-colors text-center">
                <span
                  className={
                    this.props.activeState.activeColor ===
"primary"
                      ? "badge filter badge-primary active"
                      : "badge filter badge-primary"
                  }
                  data-color="primary"
                  onClick={() => {
                    this.props.setColorAction("primary");
                  }}
                />
                <span
                  className={
                    this.props.activeState.activeColor === "info"
                      ? "badge filter badge-info active"
                      : "badge filter badge-info"
                  }
                  data-color="info"
                  onClick={() => {
                    this.props.setColorAction("info");
                  }}
                />
                <span
                  className={
                    this.props.activeState.activeColor ===
"success"
```

```
                      ? "badge filter badge-success active"
                      : "badge filter badge-success"
                  }
                  data-color="success"
                  onClick={() => {
                    this.props.setColorAction("success");
                  }}
                />
                <span
                  className={
                    this.props.activeState.activeColor ===
"warning"
                      ? "badge filter badge-warning active"
                      : "badge filter badge-warning"
                  }
                  data-color="warning"
                  onClick={() => {
                    this.props.setColorAction("warning");
                  }}
                />
                <span
                  className={
                    this.props.activeState.activeColor === "danger"
                      ? "badge filter badge-danger active"
                      : "badge filter badge-danger"
                  }
                  data-color="danger"
                  onClick={() => {
                    this.props.setColorAction("danger");
                  }}
                />
              </div>
            </li>
            <li className="button-container">
              <Button

href="https://www.creative-tim.com/product/paper-dashboard-react"
                color="primary"
                block
                round
              >
                Download now
              </Button>
            </li>
            <li className="button-container">
              <Button

href="https://www.creative-tim.com/product/paper-dashboard-react/#/
documentation/tutorial"
                color="default"
                block
                round
                outline
              >
                <i className="nc-icon nc-paper"></i> Documentation
              </Button>
            </li>
            <li className="header-title">Want more components?</li>
            <li className="button-container">
              <Button

href="https://www.creative-tim.com/product/paper-dashboard-pro-reac
t"
                color="danger"
                block
                round
                disabled
              >
                Get pro version
              </Button>
            </li>
          </ul>
        </div>
      </div>
    );
  }
}

const mapStateToProps = state => ({
  ...state
});

const mapDispatchToProps = dispatch => ({
  setBgAction: (payload) => dispatch(setBgAction(payload)),
  setColorAction: (payload) => dispatch(setColorAction(payload))
});
```

```
export default connect(mapStateToProps,
mapDispatchToProps)(FixedPlugin);
```

Code Explanation

We have combined the two bgReducer and colorReducer into one rootReducer. This makes our life simpler by allowing us to write multiple reducer logic separately, so it is easier to debug and also combine them into a single rootReducer for better usability.

Conclusion

Now that we have been through everything about Redux in React, let us take a quick summary of everything we have seen so far.

- Redux is a State Management tool used for JavaScript applications. This is a very well-known tool for managing states using a centralized store known as Redux Store. This can be easily used with React apps.
- Reasons for using Redux:
    - Avoid prop drilling thereby increasing the efficiency.
    - Works in client, server, and native contexts.
    - Simple to test.
- Redux Architecture is based on three main components. The React Components subscribe to the Redux Store, whenever there is any Action the Reducer is triggered, and the states in the Redux Store are updated.
- We can write multiple reducers for multiple actions in Redux. This makes it more of a logical coding approach and we can combine them into a single reducer using the combineReducers function from redux.

Global store and Imp concepts of redux

Overview

Redux is a JavaScript Library that is used as a global store of application states for state management in a JavaScript application. It is the most popular and lightweight library for managing state in React, the bigger the application the more beneficial it is to use Redux. Reducer in Redux is the main component that handles state management efficiently.

Introduction to Redux Core Concepts

Redux is a State Management tool used for JavaScript applications. This is a very well-known tool for managing states using a centralized store known as the Redux Store. This can be easily used with React apps.

These three main tenets form the foundation of Redux.

- A single state object known as the state or state tree houses all of the app's data.
- States in Redux are read-only. Only sending an action to the store will modify it.
- A function that accepts the current state and the desired action must be written to explain state changes. A newly updated state, not a tweaked version of the current state, must be returned by the function. Reducers are the term for these operations.

For using Redux in your application, first, you need to install it. The installation can be done using NPM or YARN as follows:

```
# NPM
$ npm install redux

OR

# YARN
$ yarn install redux
```

How Redux Works?

Redux has a straightforward interface. The complete state of the application is kept in a central store. Using Redux there will not be any need for prop drilling. Any component in the application can access the states from a global store.

Redux is made up of three essential parts: actions, reducers, and stores. Let's briefly go through each of their job descriptions to understand to working of Redux. The login component from before will be implemented again, but this time in Redux.

The term "state" describes the entity that houses the shared application data between components.

Let’s take a look at the various essential parts in detail now.

What is Redux Action?

Actions are events that are sent to the Redux Store to modify or update the states. Using actions is the only way to send data from the application to the Redux Store. This action can be any user interaction, form submission, or API call.

Actions are sent to the Redux Store in the form of a JavaScript object. There are mainly two fields in a Redux Action:

- A type field: Indicates the type of the action that is carried out, this may be for example: SIGNIN.
- A payload field: This stores the data that will be used in the Redux store to update or change the state.

Let's take a look at an example of action in Redux:

```
{
  type: "SIGN IN",
  payload: {
    userid: "ABC",
    password: "password"
  }
}
```

Actions are created using action creators, which are simple user-defined functions that return the action object. Let's take a look at the action creator for the above-mentioned action.

```
const setSigninnStatus = (userid, password) => {
  return {
    type: "SIGN IN",
    payload: {
     user-idd, // "abc"
      password // "Password"
    }
  }
}
```

The store. dispatch() method is used to execute the actions. This method is responsible for sending the action to the store.

What is Reducer in Redux?

Reducer in Redux is a pure function that takes the previous state and the action, performs the needed changes to the previous state, and returns the new state. Whenever an action is triggered the reducer is called.

Pure functions are the ones that return the same data with the same given parameters, i.e. these functions do not depend on any external data.

Explanation

Let’s take a look at a reducer function.

```
const SigninComponent = (state = initialState, action) => {
    switch (action. type) {
      case "SIGN IN":
          return state.map(user => {
              if (user.userid !== action.userid) {
                  return user;
              }

              if (user.password == action.password) {
                  return {
                      ...user,
                      login_status: "SIGNED IN"
                  }
              }
          });
       default:
          return state;
      }
};
```

However, a reducer in Redux should never directly modify the entire application state. As an alternative, they can duplicate the necessary portion of the state, make the necessary changes, and then copy it back.

Reducers are not permitted to do asynchronous actions like making API requests.

Each action type returns its updated state, and we switch between them using a conditional expression.

Remember that the size of the application affects how many reducer functions are used.

What is Redux Store?

We keep talking about the mysterious Redux store, but we haven't yet explained what it is.

The object that unifies actions and reducers in Redux (which reflects what occurred and changes the state accordingly) is referred to as the Redux store. There is only ONE store in a Redux application.

The shop has numerous responsibilities:

- Permit state access with getState().
- Permit dispatch to update the state (action).
- Holds all application state information.
- Subscribes listeners and registers them (listener).
- Uses the method provided by subscribing to deregister listeners (listener).

Reducer in Redux is all we need to construct a shop. To combine many reducers into one, we mentioned combineReducers. Now, we'll import combineReducers and send it to create a store to make a store:

```
import { createStore } from 'redux';
import reducers from './ReduxReducers';

const store = createStore(reducers);
```

Data Flow in Redux

All the data in the application follows the same data flow direction. That is why Redux is said to have a unidirectional/one-way data flow. This unidirectional flow of data makes it easier to understand and predictable for the developer! That is one of the many benefits of Redux.

Let’s take a look at the four major steps of data flow in Redux:

- Calls to stores are initiated by events that occur within your app dispatch(actionCreator(payload)).
- The root reducer in Redux is called by the Redux store together with the action's current state.
- A single state tree is created by the root reducer by combining the output of many reducers.
- The whole state tree of the root reducer in Redux, which was returned, is saved in the Redux store. Currently, your app's nextState is the new state tree.

We may further split down these stages for Redux specifically:

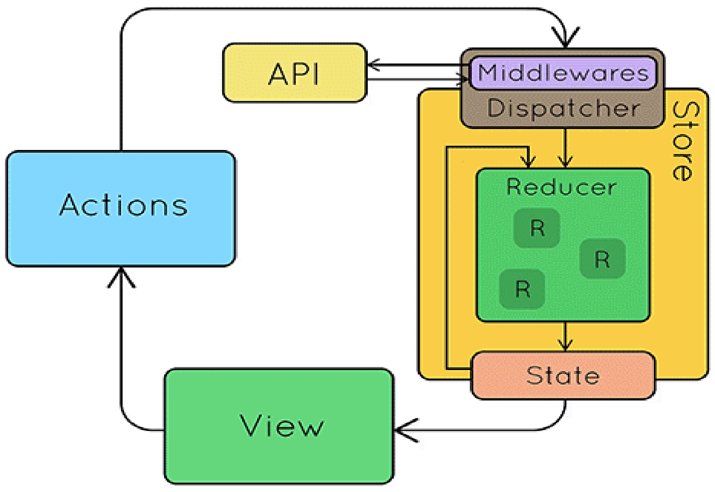
Initial Configuration

- A root reducer function is used to establish a Redux store.
- The store makes a single call to the root reducer and keeps the result value as its starting point.
- The Redux store's current state is accessed by the UI components when the UI is first displayed, and they make rendering decisions based on that information. Additionally, customers sign up for future shop updates so they may be informed if the situation has changed.

Updation Process Using Redux

- The software experiences a change, such as when a user clicks a button.
- The app's code sends an action, such as dispatch(type: counter/incremented), to the Redux store.
- With the previous state and the current action, the store executes the reducer function once more, saving the result value as the new state.
- The store notifies all subscribing areas of the user interface that it has updated.
- Every UI element that requires information from the store checks to see whether the state components they require have changed.
- To update what is displayed on the screen, any component that notices that its data has changed causes a re-render with the updated data.

Here is a graphic representation of that data flow:



Conclusion

Let us take a quick look into what we have learned in this article on Redux:

- Redux is a JavaScript Library that is used as a global store of application states for state management in a JavaScript application. Working of Redux:
  - A single store houses the global app state.
  - The states in the Redux store are in read-only mode to the rest of the application.
  - In response to events, the state is updated using reducer functions.
- Actions are events that are sent to the Redux Store to modify or update the states.
- Reducer in Redux is a pure function that takes the previous state and the action, performs the needed changes to the previous state, and returns the new state.
- Redux one-way data flow:
  - When a user interacts with the application, an action is dispatched.
  - With the dispatched action and the current state, the root reducer function is invoked. The work may be divided up by the root reducer into smaller reducer functions, which finally produce a new state.
  - The execution of the callback routines by the store alerts the view.
  - The view has the ability to get the most recent state and redraw.

Async Tasks and Redux-Thunk

Overview

Redux actions are submitted asynchronously. This is a significant problem for non-trivial applications that need to interface with application programming interfaces or APIs to run external effects in parallel. Redux can be thought of as middleware that handles distribution and reaches between actions and reducers. Redux actions are dispatched asynchronously, so two middleware libraries serve this purpose. Redux Saga and Redux Thunk.

What is Redux-Thunk?

Redux provides an easy way to update the application state in response to synchronous actions. However, it lacks for handling asynchronous code. This is where thunks come into play.

Redux Thunk middleware allows you to create action builders that return functions instead of actions. Thunks can be used to display sending an action, or to send it only when certain conditions are met. The inner function receives the store method dispatch and getState as parameters.

Redux thunks are used to manage the state of React applications.

Redux-thunk is not a replacement for Redux, but it extends the functionality of Redux and acts as a middleware that allows you to keep your reducer code clean from API calls, and once you have the API response, you can dispatch events without issue. Redux Thunk works when you have an API call or an asynchronous task running.

Redux Thunk is a tool in the State Management Library category of a tech stack.

Why Do I Need This?

A Plain simple Redux store can only do simple synchronous updates by performing actions. Middleware extends the functionality of the store and allows you to write asynchronous logic that interacts with the store.

Thunk is the recommended middleware for basic Redux side effect logic, including complex synchronous logic that requires access to memory, and simple asynchronous logic such as AJAX requests.

Asynchronous Calls In Redux Without Middlewares

I am trying to explain how to implement asynchronous action calls in Redux without using middleware.

Let's start by creating a simple react project using "create-react-app"

Also, in addition to Redux, we use React-Redux to make life a little easier.

To not overcomplicate things, we will only implement two API calls.

Create a new file name Api.js. This is the file that stores polling calls to endpoints.

```
export const getPostsById = id =>
fetch(`https://jsonplaceholder.typicode.com/Posts/${id}`);

export const getPostsBulk = () =>
fetch("https://jsonplaceholder.typicode.com/posts");
```

There are three basic actions associated with each API call. That is, REQUEST, SUCCESS, and FAIL. Each API is always in one of these three states. Depending on these states, you can decide how to display your UI. You can display a loader in your UI when in the REQUEST state and a custom UI when in the FAIL state to let the user know something went wrong.

So for each API call, we're going to make, we'll create three constants: REQUEST, SUCCESS, and FAIL. In this case, the Constant.js file would look like this:

```
export const GET_POSTS_BY_ID_REQUEST = "getpostsbyidrequest";
export const GET_POSTS_BY_ID_SUCCESS = "getpostsbyidsuccess";
export const GET_POSTS_BY_ID_FAIL = "getpostsbyfail";
export const GET_POSTS_BULK_REQUEST = "getpostsbulkrequest";
export const GET_POSTS_BULK_SUCCESS = "getpostsbulksuccess";
export const GET_POSTS_BULK_FAIL = "getpostsbulkfail";
```

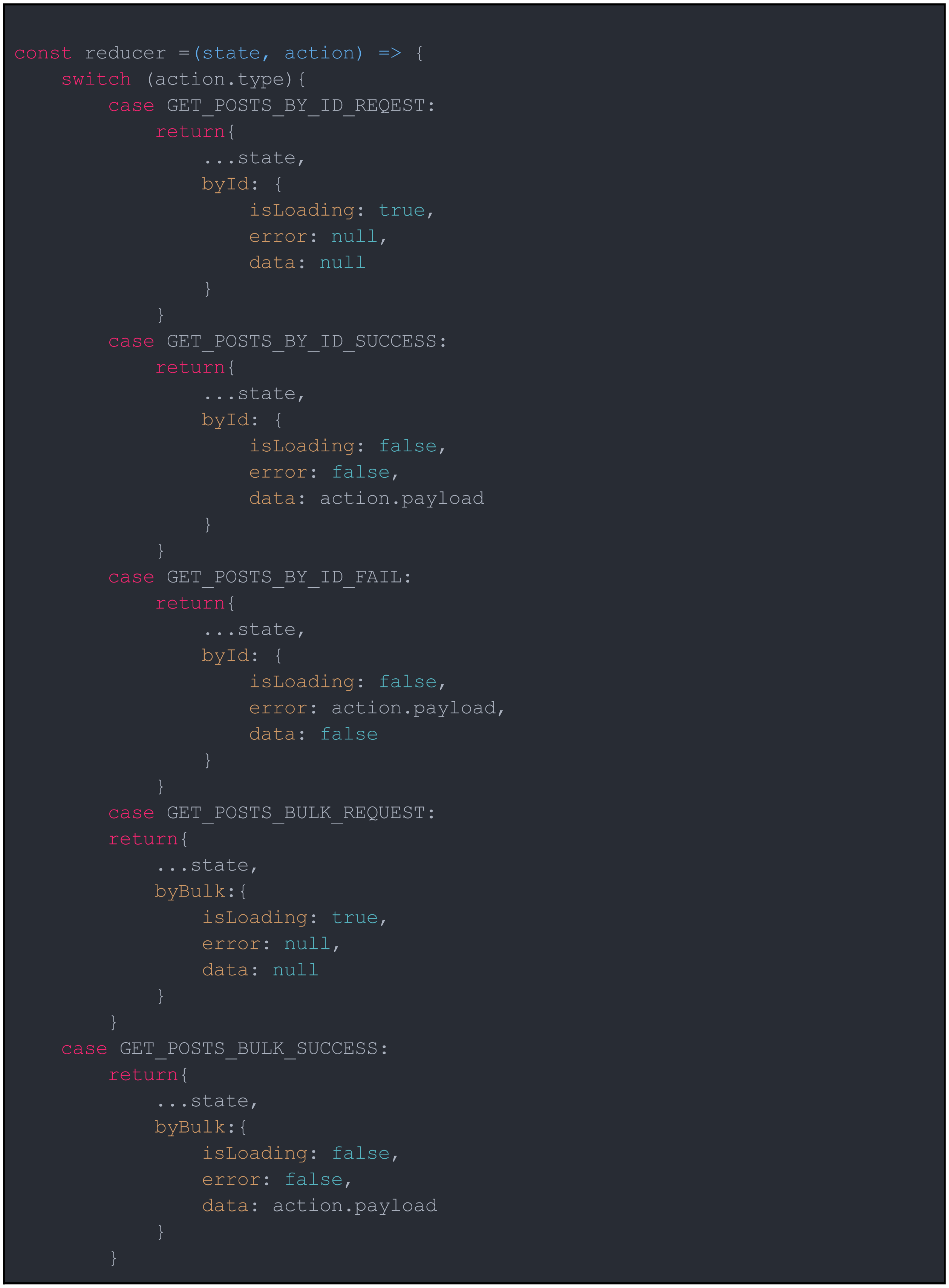
Here is the Store.js file initialState of the application:

```
import {createStore} from 'redux';
import reducer from './reducers';

const initialState ={
    byId:{
        isLoding: null,
        error: null,
        data: null
    },
    byBulk:{
        isLoading: null,
        error: null,
        data: null
    }
}

const store = createStore(reducer, initialState,
window.__REDUX_DEVTOOLS_EXTENSION__&& window.__REDUX_DEVTOOLS_EXTENSION());

export default store;
```

As you can see from the code above, all API data resides in one object, the state object. Keys isLoading indicates whether the API is in the REQUEST state.

Now that we've defined our store, let's see how we can manipulate the state at different stages where there are API calls. Below is the Reducer.js file.

```
import{
    GET_POSTS_BY_ID_REQUEST,
    GET_POSTS_BY_ID_SUCCESS,
    GET_POSTS_BY_ID_FAIL,

    GET_POSTS_BULK_REQUEST,
    GET_POSTS_BULK_SUCCESS,
    GET_POSTS_BULK_FAIL
} from './constants';
```



```
const reducer =(state, action) => {
    switch (action.type){
        case GET_POSTS_BY_ID_REQEST:
            return{
                ...state,
                byId: {
                    isLoading: true,
                    error: null,
                    data: null
                }
            }
        case GET_POSTS_BY_ID_SUCCESS:
            return{
                ...state,
                byId: {
                    isLoading: false,
                    error: false,
                    data: action.payload
                }
            }
        case GET_POSTS_BY_ID_FAIL:
            return{
                ...state,
                byId: {
                    isLoading: false,
                    error: action.payload,
                    data: false
                }
            }
        case GET_POSTS_BULK_REQUEST:
        return{
            ...state,
            byBulk:{
                isLoading: true,
                error: null,
                data: null
            }
        }
    case GET_POSTS_BULK_SUCCESS:
        return{
            ...state,
            byBulk:{
                isLoading: false,
                error: false,
                data: action.payload
            }
        }
    case GET_POSTS_BULK_FAIL:
        return{
            ...state,
            byBulk:{
                isLoading: false,
                error: action.payload,
                data: false
            }
        }
    default: return state;
    }
}
export default reducer;
```

By giving each API call its own variable to indicate the loading phase, it's now easier to implement things like multiple loaders on the same screen, depending on which API call is in which phase. rice field.

To actually implement asynchronous behavior in action, all you need is a regular JavaScript function passing dispatch as the first argument. The function forwards the action to the store, so we pass shipping to the function. Normally the component has access to the shipping, but I want the external function to control the shipping, so I need it.

```
const getPostById = async(dispatch, id) =>{
    dispatch({type: GET_POSTS_BY_ID_REQUEST});

    try{
        const response = await getPostsById(id);
        const res = await response.json();
        dispatch({type: GET_POSTS_BY_ID_SUCCESS, payload: res});
    }catch(e){
        dispatch({type: GET_POSTS_BY_ID_FAIL, payload: e});
    }
};
```

And a function to enable sending in the range of function above:

```
export const getPostByIdFunc = dispatch => {
    return id => getPostById(dispatch, id);
}
```

So the complete action.js file looks like this:

```
import{
    GET_POSTS_BY_ID_REQUEST,
    GET_POSTS_BY_ID_SUCCESS,
    GET_POSTS_BY_ID_FAIL,

    GET_POSTS_BULK_REQUEST,
    GET_POSTS_BULK_SUCCESS,
    GET_POSTS_BULK_FAIL
} from './constants';

import{
    getPostsById,
    getPostsBulk
} from './api';

const getPostById = async(dispatch, id) =>{
    dispatch({type:GET_POST_BY_ID_REQUEST});

    try{
    const response = await getPostsById(id);
    const res = await response.json();
    dispatch({type: GET_POSTS_BY_ID_SCCESS, payload:res});
    }catch(e){
    dispatch({type:GET_POSTS_BY_ID_FAIL, payload:e});
    }
};

const getPostBulk = async dispatch =>{
    dispatch({type:GET_POSTS_BULK_REQUEST});

    try{
        const response = await getPostsBulk();
        const res = await response.json();
        dispatch({type:GET_POSTS_BULK_SUCCESS, payload:res});
    }catch(e){
        dispatch({type: GET_POSTS_BULK_FAIL, paylad:e});
    }
};

export const getPostByIdFunc = dispatch =>{
    return id => getPostById(dispatch, id);
}

export const getPostsBulkFunc = dispatch =>{
    return() =>getPostBulk(dispatch);
}
```

Once that's done, all that's left to do is pass these functions to the connected component's mapDispatchToProps.

```
const mapDispatchToProps = dispatch => {
    return{
        getPostById: getPostByIdFunc(dispatch),
        getPostBulk: getPostsBulkFunc(dispatch)
    }
};
```

App.js

```
import React,{component} from 'react';
import './App.css';

import {connect} from 'react-redux';
import {getPostByIdFunc, getPostsBulkFunc} from './actions';

class App extends Component{
    render(){
        console.log(this.props);
        return(
            <div className = "App">
                <button onClick={()=>{this.props.getPostById(1)}}>By
Id</button>
                <button onClick ={()=>{this.props.getPostBulk()}}>In
bulk</button>
                </div>
        );
    }
}

const mapStateToProps = state => {
    return{
    state
    };
}

const mapDispatchToProps = dispatch => {
    return{
        getPostById: getPostByIdFunc(dispatch),
        getPostBulk: getPostsBulkFunc(dispatch)
    }
};
```

This is how Redux makes asynchronous calls without middleware.

Async Function Middleware

A basic rule when creating reducers in React + Redux applications is that they should not have side effects. A side effect is something that can change something outside the scope of the function. Making an HTTP call is an example of a side effect.

HTTP calls are side effects, but they are an important part of the web applications. You can't make HTTP calls with reducers, but you can write middleware for that. To handle a situation like this, you'll need to create middleware for your asynchronous functions.

You can write middleware for asynchronous functions, React already provides that. Redux Thunk is just the middleware of asynchronous functions.

Installation

To use redux-thunk, you need to install it using Node Package Manager(NPM).

```
npm install redux-thunk
```

Once installed, you need to import the thunkMiddleware from redux-thunk into your store file. Note that the store must be able to pass thunk middleware to the dispatch function. For this, we need to use applyMiddleware

```
import {createStore,applyMiddleware} from 'redux';
import {composeWithDevTools} from 'redux-devtools-exrension';
import thunkMiddleware from 'redux-thunk';
const enhancer = composeWithDevTools(applyMiddleware(thunkMiddleware));
const store = createStore(reducer, enhancer);
```

This is how we can apply "thunkMiddleware" to the store. Now let's see how we can write a thunk function.

```
export async function getData(dispatch, getState){

const res =  await client.get('/getdata')
```

```
    dispatch({type: 'RES_DATA', payload: res.data})
}
```

getData is a thunk middleware function. It has two parameters, Dispatch, and GetState. The last line of the function sends the desired action on the dispatch.

So what happened here? The middleware function created above handles asynchronous HTTP calls that provide data.

Where would I write this logic if there is no redux-thunk? You can't write it directly in reducers or actions because it has no side effects. That's why we have Redux-thunk middleware that allows us to write asynchronous logic.

Remember how earlier we gave the store the ability to pass a middleware function to the shipping function? Notice the following line of code:

```
store.dispatch(getData);
```

The getData function is called when the above code is executed anywhere in the application. No action is taken at this point. So it's perfectly fine to make an HTTP call in the getData function. When a call is made and a response is received, the "dispatch" functionality provided by redux-thunk can perform the necessary actions. From here you can continue to update your state with the normal Redux flow.

Redux-thunk is, therefore, useful in reactive applications that use Redux to manage state. Thunk middleware is neither an action nor a reducer, so it can have side effects. Additionally, it provides a Dispatch and GetState function that can be used to send actions and the access status, respectively.

First, use your terminal to navigate to your project directory and install the redux-thunk package into your project.

```
$ npm install redux-thunk
```

Now apply middleware when creating an app store using Redux's applyMiddleware. For a React application using Redux and React-Redux, the index.js file looks like this:

index.js

```
import React from 'react';
import ReactDom from 'react-dom';
import {Provider} from 'react-redux';
import {createStore, applyMiddleware} from 'redux';
import thunk from 'redux-thunk';
import './index.css';
import rootReducer from './reducers';
import App from './App';
import * as serviceWorker from './serviceWorker';

const store = createStore(rootReducer, applyMiddleware(thunk));

ReactDom.render(
    <Provider store ={store}>
    <App/>
    </Provider>,
    document.getElementById('root')
);
```

Redux Thunk is imported and applied to your application.

Composition

The return value of the inner function can be used as the return value of dispatch itself. This is useful for coordinating the asynchronous flow of control using thunk action builders that return promises that send to each other and wait for the complete.

```
import {createStore, applyMiddleware} from 'redux';
import thunk from 'redux-thunk';
import rootReduer from './reducer';

const store = createStore(rootReduer, applyMiddleware(thunk))

function fetchWhiteSugar(){
    return fetch('https://www.google.com/search?q=white+sugar')
}

function makeCoffee(forPerson, whiteSugar){
    return{
    type: 'MAKE_COFFEE',
    forPerson,
    whiteSugar
    }
}

function apologize(fromPerson, toPerson, error){
    return{
    type: 'APOLOGIZE',
    fromPerson,
    toPerson,
    error
    }
}

function withdrawMoney(amount){
    return{
    type: 'WITHDRAW',
    amount
    }
}

store.dispatch(withdrawMoney(100))

function makeCoffeeWithWhiteSugar(forPerson){
    return function (dispatch){
    sugar => dispatch(makeCoffee(forPerson, sugar)),
    error => dispatch(apologize('The Coffee Shop' forPerson, error))
    }
}

store.dispatch(makeCoffeeWithWhiteSugar('Me'))

store.dispatch(makeCoffeeWithWhiteSugar('My partner')).then(()=>{
    console.log('Done!')
})

function makeCoffeeForEveryBody(){
    return funtion(dispatch, getState){
    if(!getState().coffee.isShopOpen){

    return Promise.resolve()
    }
    return dispatch(makeCoffeeWithWhiteSugar('My Friends'))
        .then(()=>
        Promise.all([
            dispatch(makeCoffeeWithWhiteSugar('Me')),
            dispatch(makeCoffeeWithWhiteSugar('My Mom'))
        ])
        )
        .then(()=>dispatch(getState().myMoney > 50 ? wthdrawMoney(50) :
apologize('Me', 'The Coffee Shop')))
    }
}

store.dispatch(makeCoffeeForEverybody())
.then(()=>
    response.send(ReactDOMServer.renderToString(<MyApp store={store}/>))
)
```

```
import {connect} from 'react-redux';
import {Component} from 'react';

class CoffeeShop extends Component{
    componentDidMount(){
    this.props.dispatch(makeCoffeeWithWhiteSugar(this.props.forPerson))
    }
    componentDidUpdate(prevProps){
    if(prevProps.forPerson !== this.props.forPerson){
        this.props.dispatch(makeCoffeeWithWhiteSugar(this.props.forPerson))
    }
    }
    render(){
        return <div>{this.props.coffee.join('milk')}</div>
    }
}

export default connect(state =>({
    coffees: state.coffees
}))(Coffee)
```

Using Redux Thunk in a Sample Application

The most common use case for Redux Thunk is asynchronous communication with external APIs to retrieve or store data. Redux Thunk makes it easy to send actions to an external API that follows the request lifecycle.

Normally, to create a new todo item, you first send an action to indicate that the creation of the todo item has started. If the todo item is successfully created and returned from the external server, the new todo item triggers another action. If an error occurs and the task cannot be saved to the server, you can take action on the error instead.

Let's see how we can achieve this using Redux Thunk.

Import the action into a container component and send it.

AddTodo.js

```
import {connect} from 'react-redux';
import {addTodo} from '../actions';
import NewTodo from '../components/NewTodo';
```

```
const mapDispatchToProps = dispatch =>{
    return{
        onAddTodo: todo=>{
            dispatch(addTodo(todo));
        }
    };
};
export default connect(null, mapDispatchToProps)(NewTodo);
```

The action uses Axios to send a Post request to the JSONPlaceholder's endpoint.

index.js

```
import{
    ADD_TODO_SUCCESS,
    ADD_TODO_FAILURE,
    ADD_TODO_STARTED,
    DELETE_TODO
} from './types';

import axios from 'axios';

export const addTodo =({title,userId}) =>{
    return dispatch =>{
        dispatch(addTodoStarted());

        axios
            .post(`https://jsonplaceholder.typicode.com/todos`,{title,
            userId,
            completed: false
            })
            .then(res =>{
            dispatch(addTodoSuccess(res.data));
            })
            .catch(err =>{
            dispatch(addTodoFailure(err.message));
            });
    };
};

const addTodoSuccess = todo =>({
    type: ADD_TODO_SUCCESS,
    payload: {
    ...todo
    }
});

const addTodoStarted = () =>({
    type:ADD_TODO_STARTED
});

const addTodoFailure = error => ({
    type: ADD_TODO_FAILURE,
    payload: {
        error
    }
});
```

Notice that the action builder addTodo returns a function instead of a regular action object. This function gets the shipping method from the store.

Within the body of the function, we first send an immediate sync action to the store to indicate that we have started saving the task using the external API. Then use AXIOS to make the actual POST request to the server. A successful response from the server uses the data obtained from the response to send a successful synchronous action, while an error response uses the error message to send another synchronous action.

Using an external API like JSONPlaceholder in this case will allow you to see the actual network latency. However, when using a local backend server, network responses may be too fast to experience the network delays that real users experience, so you can add artificial delays during development.

index.js

```
export const addTodo = ({title,userId}) =>{
    return dispatch =>{
        dispatch(addTodoStarted());

        axios
            .post(ENDPOINT,{
                title,
                userId,
                completed: false
            })
            .then(res => {
                setTimeout(() => {
                    dispatch(addTodoSuccess(res.data));
                }, 3000);
            })
            .catch(err => {
                dispatch(addTodoFailure(err.message));
            });
        };
    };
```

To test error scenarios, you manually throw an error.

index.js

```
export const addTodo =({title, userId}) =>{
    return dispatch =>{
    dispatch(addTodoStarted());

    axios
        .post(ENDPOINT, {
            title,
            userId,
            completed: false
        })
        .then(res => {
            throw new Error('addTodo error!');
            // dispatch(addTodoSuccess(res.data));
        })
        .catch(err =>{
            dispatch(addTodoFailure(err.message));
        });
    };
};
```

For the sake of completeness, here's an example todo reducer that handles the entire request lifecycle:

todoReducer.js

```
import{
    ADD_TODO_SUCCESS,
    ADD_TODO_FAILURE,
    ADD_TODO_STARTED,
    DELETED_TODO
} from '../actions/types';

const initialState ={
    loading: false,
    todos: [],
    error: null
};

export dafault function todoReducer(state = initialState, action) {
    switch (action.type){
        case ADD_TODO_STARTED:
            return {
                ...state,
                loading: true
            };
            case ADD_TODO_SUCCESS:
                return{
                ...state,
                loading: false,
                error: null,
                todos: [...state.todos, action.payload]
                };
                case ADD_TODO_FAILURE:
                    return{
                    ...state,
                    loading: false,
                    error: action.payload.error
                    };
                    default:
                        return state;
    }
}
```

Redux-Thunk vs Redux-SagaConclusion

- This is very useful concept that helps you deal with side effects efficiently.
- Redux allows React applications to submit actions synchronously and rely on external APIs for tight call integration of data while creating and consuming stores.
- Finally, thunks are an effective solution for applications with simple asynchronous requirements.
- Redux Thunk is middleware that allows you to call action builders that return functions instead of action objects.
- Redux Thunk uses a pattern that facilitates the abstraction of storage logic from component to services, action builders, and actions. Component doesn't care what happens to the data store. You just need to send the logic service.
- Writing middleware for Redux is a powerful tool. Redux is one of the most commonly used middleware for asynchronous actions. Thunk is also the standard asynchronous middleware for Redux Toolkit and RTK Query.

Redux Thunk	Redux Saga
● Action Builder may contain too many asynchronous logic functions	● Action builders remains pure functions
● Contains less boilerplate codes	● Contains more boilerplate codes than Redux-Thunk
● Difficult to scale up codes	● Easy to scale codes as compared to redux-thunk codes
● Difficult to taste async functions	● Easy to test as all logic remains together
● As compared to redux-saga, easy to understand logic, functions, concepts	● Due to multiple concepts to learn like generator functions and redux-saga, etc. It is difficult to understand

Conclusion

- This is very useful concept that helps you deal with side effects efficiently.
- Redux allows React applications to submit actions synchronously and rely on external APIs for tight call integration of data while creating and consuming stores.
- Finally, thunks are an effective solution for applications with simple asynchronous requirements.
- Redux Thunk is middleware that allows you to call action builders that return functions instead of action objects.
- Redux Thunk uses a pattern that facilitates the abstraction of storage logic from component to services, action builders, and actions. Component doesn't care what happens to the data store. You just need to send the logic service.
- Writing middleware for Redux is a powerful tool. Redux is one of the most commonly used middleware for asynchronous actions. Thunk is also the standard asynchronous middleware for Redux Toolkit and RTK Query.