International Journal of Trend in Scientific Research and Development (IJTSRD)
Volume 6 Issue 7, November-December 2022 Available Online: www.ijtsrd.com e-ISSN: 2456 – 6470

Redux State Management System - A Comprehensive Review

Krutika Patil

Master of Science in Computer Science, The University of Texas at Dallas, Tracy, CA, USA

ABSTRACT

Reactivity is a common trait of JavaScript applications and comes with efficient state management. Any JavaScript application has multiple ways of managing state; for example, in React, we can use "useState" and "useReducer" Hooks. However, another third-party library called Redux has grown in popularity to be an efficient state management tool in JavaScript applications. This paper makes an in-depth review of the Redux state management system. It is famously used with React as a state-management tool and by other JavaScript frameworks as well. Redux is most suitable for applications with frequent updates to the state since Redux has better efficiency in a flux-like setup than React's Context API.

KEYWORDS: *ReactJs, Redux, state management, web development, JavaScript frameworks, React context, cross-component state, app-wide state, local state*

How to cite this paper: Krutika Patil "Redux State Management System - A Comprehensive Review" Published in International Journal of Trend in Scientific Research and Development (ijtsrd), ISSN: 2456-6470, Volume-6 | Issue-7, December 2022, pp.1021-1027, URL: www.ijtsrd.com/papers/ijtsrd52530.pdf

IJTSRD52530

INTRODUCTION

Redux is a third-party JavaScript library. It has quickly gained recognition in the front-end world as the most efficient state-management system in an application with frequent state updates. It is a state-management system for cross-component and app-wide states. To understand the concept of various states, let us classify the different types of states.

There are three states in a JavaScript application.

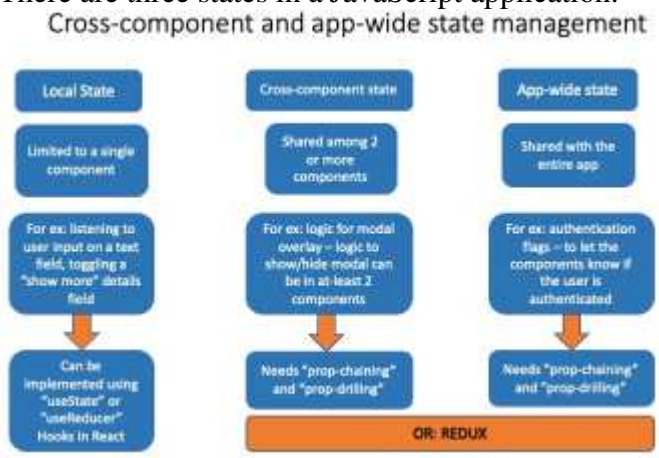

Figure 1: Classification of States in a JavaScript application

- **Local state** - a state limited to a single component. For example, listening to a click event for a button or a user input for a text field. Usually handled in React applications by the "**useState**" or the "**useReducer**" hook.
- **Cross-component state** - a state that affects multiple components. For example, opening and closing a modal overlay. In the case of a cross-component state, multiple components share the state. We achieve this state management using **prop chains** and **prop drilling**.
- **App-wide state** - affects the entire application or, at least, most components. One example of this could be the authentication status. Whether authenticated or not, the components might choose to hide, show, or update the content they render. We can use **prop drilling** and **prop chains** to achieve this state management.

In the case of Cross-component and App-wide states, the "prop-drilling" and "prop-chaining" can get cumbersome since the props get passed around component hierarchies which might increase the chances of introducing defects in the logic. We can instead use the Redux library to manage the state for us in these cases. Redux maintains a central data store that the components can subscribe to and then

dispatch specific actions if the state needs to be updated.

These concepts are depicted diagrammatically in **Figure 1**.

CORE CONCEPTS OF REDUX

A. Central Data Store

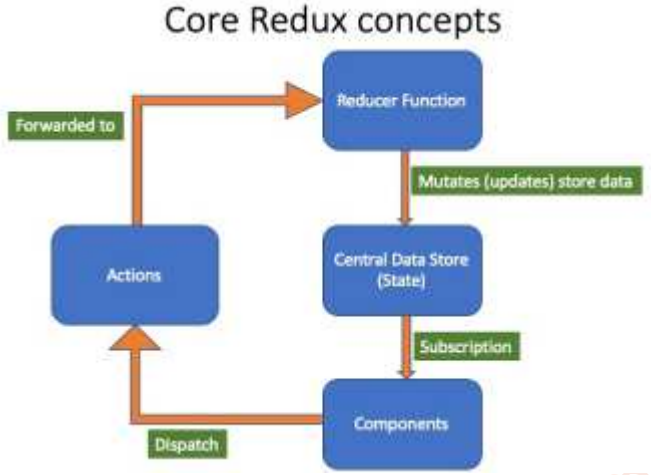

Figure 2: Core Concepts of Redux

Redux has a single central data store.

Components can subscribe to a store to get any updates to the state or a part of the state. The components would also need a way to change the state. One important thing is that components never directly change or mutate a store. Instead, we use the Reducer function.

B. Reducers

A reducer is a programming concept. It is a function that is responsible for mutating or updating the store. A reducer is a pure function, meaning it should always produce the same output for the same input. We should not add side-effect logic (API calls or reading and writing to local storage) in the reducer functions. The components must use the reducers if they need to change or update the store. A reducer function always needs to be triggered by a component—for example, clicking a button or a user input on a text field. Hence, we need a way for components to do that. The concept of actions comes into play here.

C. Actions

The components dispatch or trigger actions, and the actions are JavaScript objects which include the description of the action and optionally contain a payload. The components dispatch these actions that contain a description of the intended action. The redux then forwards these actions to the reducer. The reducer reads the action description that describes the action and updates or mutates the state accordingly. We need to note here that components DO NOT directly update the store. Instead, the components dispatch actions, which get forwarded to the reducers. The reducer then spits a new state that replaces the existing state in the central data store. When the central state updates, the components that have subscribed to the store receive notifications instantly so that the components can update their UI. **Figure 2** above describes in detail the concepts we discussed in this section.

BASIC REDUX EXAMPLE IN JAVASCRIPT

A. Setting up the central data store

Consider the below script in **Figure 3**.

```
const counterReducer = (state = { flag: 0 }, action) => {
  if (action.type === "increment") {
    return {
      flag: state.flag + 1,
    };
  } else if (action.type === "decrement") {
    return {
      flag: state.flag - 1,
    };
  } else if (action.type === "increment by") {
    return {
      flag: state.flag + action.payload,
    };
  }
  return state;
};

const store = redux.legacy_createStore(counterReducer);
// console.log(store.getState());

const counterSubscriber = () => {
  const latestState = store.getState();
  console.log(latestState);
};

store.subscribe(counterSubscriber);
store.dispatch({ type: "increment" });
store.dispatch({ type: "decrement" });
store.dispatch({ type: "increment by", payload: 50 });
```

Figure 3 Basic redux script in JavaScript

Let us see the core concepts of Redux discussed in the previous section in action using the example above in **Figure 3**. First off, we import redux using the statement below in **Figure 4**.

```
const redux = require("redux");
```

Figure 4 System to import redux in a JavaScript application

The ***require("redux")*** expression returns a redux const that will be our Hook to the central data store.

Next, let us create the central store on which the redux state management functionality depends. We can create a store using the statement below in **Figure 5**.

```
const store = redux.legacy_createStore(counterReducer);
```

Figure 5: Syntax to create a central data store in a JavaScript application

The function ***legacy_createStore*** returns a store object. The components **subscribe** to this store. It also takes in an argument which is our reducer. The **reducer** function mutates the store by spitting out a new state snapshot that replaces the previous one. Hence, this becomes the input to the ***legacy_createStore*** function.

Next, let us look at the way to create a reducer. Consider the code below in **Figure 6**. A reducer is a JavaScript function. It takes in two arguments: state and action. The state is the current state snapshot and is forwarded to the reducer by redux. The second argument, "**action**", is passed to the reducer by the components. This argument action contains a property type that describes the type of action. Based on the type of action, a particular state snapshot gets returned by the reducer. This state snapshot replaces the existing state in the central store.

```
const flagReducer = (state = { flag: 0 }, action) => {
  if (action.type === "increment") {
    return {
      flag: state.flag + 1,
    };
  } else if (action.type === "decrement") {
    return {
      flag: state.flag - 1,
    };
  }
  return state;
};
```

Figure 6 A typical redux reducer function

Components can subscribe to the store using the function below in Figure 7.

```
const flagSubscriber = () => {
  const latestState = store.getState();
};
```

Figure 7 Subscriber function to a Redux store

In the example above in **Figure 7**, the subscriber is a function that executes the logic to get the updates from the store using ***the store.getState()***. As the state in the central store changes, the components subscribed to the store get notified instantly to update the UI.

The components can update the state by dispatching specific actions using the logic in **Figure 8**.

```
store.disptach({ type: "increment" });
```

Figure 8 Logic to dispatch a Redux action

We dispatch the action to **increment** the flag state in the above code. The **dispatch** function takes in an argument that is a JavaScript object and contains the **type** property. This property describes the type of action. We may also send a payload if we update the state to a specific value.

```
const counterReducer = (state = { flag: 0 }, action) => {
  if (action.type === "increment") {
    return {
      flag: state.flag + 1,
    };
  } else if (action.type === "decrement") {
    return {
      flag: state.flag - 1,
    };
  } else if (action.type === "increment by") {
    return {
      flag: state.flag + action.payload,
    };
  }
  return state;
};
```

Figure 9 A Redux Reducer function in JavaScript

The dispatched action is forwarded to the reducer function by redux. An example of that is depicted above in **Figure 9**.

We dispatch three actions in our script.

- ***store.dispatch({type: 'increment'});***
- ***store.dispatch({type: 'increment'});***
- ***store.dispatch({type: 'increment', payload: 50});***

Hence, our output would be as depicted in **Figure 10**. The first output is for the action increment that increases the flag value to 1 from its initial value of 0. The second output is for the dispatched action decrement that decrements the flag value to 0 from 1. Furthermore, the third value is due to the action "***incrementBy***", which also contained a payload of 50. The action increments the flag by 50 from 0.

```
{ flag: 1 }
{ flag: 0 }
{ flag: 50 }
```

Figure 10 Output of the script depicted in Figure 3

USING REDUX IN A REACT APPLICATION

Consider a simple react project below in **Figure 11.** It prints out the value of the flag that is initially 0. There are four buttons to perform various operations that we will cover soon.

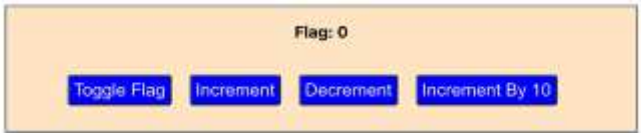

Figure 11 A simple React App

Let us now install **redux** into the application. To do that, we need to get the ***reduxjs/toolkit***, a third-party package that makes working with redux very easy. The command to install redux is depicted below in **Figure 12**.

```
npm install @reduxjs/toolkit
```

Figure 12 Command to install reduxjs/toolkit library in a React Application.

Next, we will install **react-redux**, another package that gives us access to libraries responsible for dispatching actions and subscribing to a store. The syntax is depicted below in **Figure 13**.

```
npm install react-redux
```

Figure 13 Command to install react-redux in a React Application.

Once done, let us go ahead and create our central store that stores the state. A vital feature given by the redux toolkit is that we can have slices of the state. We can split the state into two slices for our current React application in **Figure 11**. A **UI Slice** manages the state to display or hide the flag; a **Flag Slice** could maintain the flag state.

Let us start by creating the UI Slice. We must import the ***createSlice*** library from the redux toolkit to create our slice script. **Figure 14** depicts the syntax.

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

Figure 14 Syntax to import createSlice hook function from reduxjs/toolkit

The **createSlice** is a function. It takes in an object with the following three properties: a **name** property that uniquely identifies the state slice, an **initialState** property with which the state gets initialized, and the **reducers** property where we can list the various actions supported by this slice.

The logic to create a state slice looks as below in **Figure 15**.

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

const initialUIState = {
  showFlag: true,
};

const uiSlice = createSlice({
  name: "ui",
  initialState: initialUIState,
  reducers: {
    toggleFlag(state) {
      state.showFlag = !state.showFlag;
    },
  },
});

export default uiSlice.reducer;
export const uiActions = uiSlice.actions;
```

Figure 15 Logic to create a state slice to toggle the flag visibility

Here, the **reducers** property is, in turn, an object with various actions that are supported. We see that **toggleFlag** modifies the state by flipping the previous value. We notice here that the initial state is **true**.

Similarly, let us also create the **flagSlice**. We will have three actions in the **flagSlice** as supported by the reducer. The export **flagSlice.reducer** gives us access to this reducer. The components use the export "**flagActions**" to dispatch various actions.

Our next step is to configure these reducers in our store. Let us create a new script file and import ***configureStore*** from ***reduxjs/toolkit***. The syntax is depicted below in **Figure 16**.

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

Figure 16 Syntax to import configureStore from reduxjs/toolkit

This function **configureStore** takes in an argument which is a JavaScript object. This JavaScript object needs to list out the reducers. We can achieve this by adding an object with property ***reducers***, and its value could be another object with any key that the user prefers and its value being a reducer. Below in **Figure 17**, we have configured both the reducers into the store.

```
const store = configureStore({
  reducer: {
    ui: UIReducer,
    flag: FlagReducer,
  },
});

export default store;
```

Figure 17 Configuring a Redux store in React by combining multiple state slices.

The store needs to be made available to the entire application. We can do this in the index.js file in the root folder. Import the store from the store folder. We also need to import Provider from the react-redux library. The **Provider** is a React element, and by wrapping it around our main App component, we will make the store available to the app. The **Provider** element takes in a prop, the store object imported from the redux script. The code looks like the one below in **Figure 18**.

```
import { Provider } from "react-redux";
import store from "./store/index";

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

Figure 18 Making the Redux store available to the entire application

Following the above steps, we successfully set up our React application with Redux. The following section discusses how to dispatch an action on the store and subscribe to the store.

B. Dispatching Actions to the central data store

The components need to import the "*useDispatch*" Hook from react-redux. This Hook, when executed, gives us a dispatch constant that we can use to trigger our actions. The logic to import *useDispatch* Hook is as below in Figure 19.

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

Figure 19 Syntax to import the useDispatch Hook from react-redux library

Execute the useDispatch Hook inside the component to retrieve a dispatch constant, as depicted in Figure 20.

```
const dispatch = useDispatch();
```

Figure 20 Executing useDispatch Hook to retrieve a dispatch constant.

Next, we would like to dispatch specific actions when clicking the buttons. For example, the **Toggle Flag** button will show or hide the flag. The **Increment** button will increment the flag. The **Decrement** button will decrement the flag. Furthermore, we need to increment the flag by a certain number upon clicking the "**Increment By 10**" button. For this, let us import the actions from our reducer scripts. The syntax is depicted below in **Figure 21**.

```
import { uiActions } from "../store/uiActions";
import { flagActions } from "../store/flagActions";
```

Figure 21 Importing the Reducer actions from the Reducer scripts

Let us consider clicking on the button **Toggle Flag**. We want to show or hide the flag when clicking this button. We essentially dispatch the action "**toggleFlag**" in this case which gets forwarded to the **Reducer** by **Redux**. The syntax is depicted below in **Figure 22**.

```
const toggleFlagHandler = () => {
  dispatch(uiActions.toggleFlag());
};
```

Figure 22 Dispatching a Redux Action from a component

The function **toggleFlagHandler** dispatches the **toggleFlag** action, and as we know, the **toggleFlag** action in the **uiActions** reducer flips the **showFlag** state.

Similarly, consider incrementing the flag by clicking on the **Increment** button. The button's **onClick** handler executes the **incrementHandler** function that dispatches the **incrementFlag** action. We have depicted the same in **Figure 23**.

```
const incrementHandler = () => {
  dispatch(flagActions.incrementFlag());
};

const decrementHandler = () => {
  dispatch(flagActions.decrementFlag());
};

const incrementByHandler = () => {
  dispatch(flagActions.incrementby(10));
};

return (
  <>
    <button onClick={toggleFlagHandler}>Toggle Flag</button>
    <button onClick={incrementHandler}>Increment</button>
    <button onClick={decrementHandler}>Decrement</button>
    <button onClick={incrementByHandler}>Increment By 10</button>
  </>
);
```

Figure 23 Dispatching increment Flag, decrement Flag and increment By actions

C. Subscribing to the store

The components need to import the "**useSelector**" Hook from the react-redux library to get updates from the store. The import syntax is depicted below in **Figure 24**. When the central state in the store updates, the subscribing components get updated instantly.

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

Figure 24 Syntax to import "useSelector" Hook from 'react-redux' library

We need to execute the "**useSelector**" Hook inside the components only. It takes in an argument that is a state and returns the specific state that we need. The logic looks like below. When we set up our store, we added keys to the specific reducers **ui** and **flag**. Hence, while retrieving the state, we need to use these keys to refer to a state of our choice. We have depicted the same in **Figure 25**.

```
const showFlag = useSelector((state) => state.ui.showFlag);
```

Figure 25 Syntax to retrieve the state from the store

The retrieved state ***showFlag*** updates whenever the ***state.ui.showFlag*** changes, thereby re-rendering the component. The exact mechanism applies to the other buttons that trigger their actions, resulting in reducers performing appropriate state updates based on the action types.

RESULTS

The initial state of the UI is depicted below in **Figure 26**.

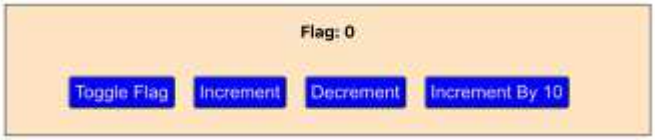

Figure 26 Initial state of the UI

Below are the results on the UI due to clicking on **Increment**, **Decrement**, **Increment By 10**, and **Toggle Flag** buttons, respectively. These are depicted respectively in **Figure 27 - 30**.

The first action triggered due to a click event on the "**Increment**" button is the "**incrementFlag**" action. The initial state 0 of the "**flag**" got updated to 1.

The second action triggered due to a click event on the "**Decrement**" button is the "**decrementFlag**" action. The initial state 1 of the "**flag**" got updated to 0.

The third action triggered due to a click event on the "**Increment By 10**" button is the "**incrementBy**" action. The initial state 0 of the "flag" got updated to 50.

The fourth action triggered due to a click event on the "**ToggleFlag**" button is the "**toggleFlag**" action. With the initial state of the "showFlag" state in the store being "**true**", it was updated to "**false**".

Since the components have subscribed to the store, the updates are reflected on the UI instantaneously.

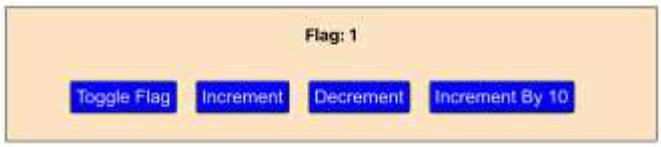

Figure 27 UI State after clicking on the "Increment" button

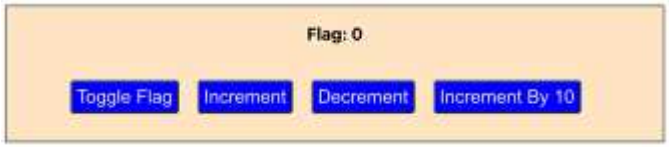

Figure 28 UI State after clicking on the "Decrement" button

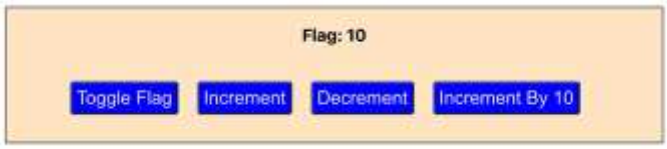

Figure 29 UI State after clicking on the "Increment By 10" button

Figure 30 UI State after clicking on the "Toggle Flag" button

CONCLUSION

This paper attempts to dive deep into the features of the **Redux** state management mechanism. We started with understanding managing the state in an application and looked at various aspects of Redux, like the **central data store**, **actions**, and **reducers**. Also, we discussed the **subscription** mechanism of the components to the store to get instant updates. We looked at how we can split the state into multiple slices that serve a specific purpose. The reducers can update the slices only, and behind the scenes, the Redux will combine the state snapshot into the overall state. The feature of Redux discussed in this paper helps us de-clutter the logic into respective scripts and helps make the code easy to maintain. The Redux statement system can be a good option in a flux-like system with frequent updates to the state.

REFERENCES

[1] Krutika Patil, Sanath Dhananjayamurty Javagal, "React state management and side-effects – A Review of Hooks", IRJET Journal, volume 9, 2022, https://www.irjet.net/archives/V9/i12/IRJET-V9I1225.pdf.

[2] https://www.udemy.com/course/react-the-complete-guide-incl-redux/learn/lecture/25599228?start=405#overview

[3] Shravan G V, Anitha Sandeep. "COMPREHENSIVE ANALYSIS OF REACT-REDUX DEVELOPMENT FRAMEWORK", International Journal of Creative Research Thoughts (IJCRT), ISSN:2320-2882, Vol.8, Issue 4, pp.4230-4233, April 2020, URL: http://www.ijcrt.org/IJCRT2004607

[4] Banks, Alex, and Eve Porcello. Learning React: functional web development with React and Redux. " O'Reilly Media, Inc.", 2017.

[5] Caspers, Matthias Kevin. "React and redux." Rich Internet Applications w/HTML and Javascript 11 (2017).

[6] Chinnathambi, Kirupa. Learning React: A Hands-on Guide to Building Web Applications Using React and Redux. Addison-Wesley Professional, 2018.

[7] McFarlane, Timo. "Managing State in React Applications with Redux." (2019).

[8] Roldan, Carlos Santana. React Cookbook: Create dynamic web apps with React using Redux, Webpack, Node. js, and GraphQL. Packt Publishing Ltd, 2018.

[9] Bugl, Daniel. Learning Redux. Packt Publishing Ltd, 2017.

[10] Pronina, Daria, and Iryna Kyrychenko. "Comparison of Redux and React Hooks Methods in Terms of Performance." Proceedings http://ceur-ws. org ISSN 1613 (2022): 0073