Redux – Basic Understanding of the Concepts

What is Redux?

Redux is a predictable state container for JavaScript apps.

State management is a big concern in large applications and Redux solves this problem. Some nice things you can do with Redux are logging, hot reloading, time travel, universal apps, recording, replay, etc.

Three Core Principles of Redux

We will keep it simple for you and let's understand it first without using technical jargon. Let's consider a real-life scenario of banks. You want to withdraw some cash from your bank account. You go to the bank branch with one **intention/action** in your mind i.e. WITHDRAW_MONEY. When you enter the bank you go straight to the counter with the Cashier to make your request. *But….why do you need to talk to the cashier? Why you just don't enter the bank vault to get your money?*

You're aware that there is a process that you need to follow to withdraw your money. When you talk to the cashier, he takes some time, checks some details, enters some commands, and hands over the cash to you. Let's relate this example to Redux and understand some of its terminologies.

1. Consider the **Redux Store** as a bank vault and the **State** of your application is like money. The entire user interface of your application is a function of your state. Just like your money is safe in the bank vault, the state of your application is safe in the Redux Store. Now, this leads to the **first principle** of Redux…

Single source of truth: The state of your whole application is stored in an object tree within a single store.

Let's simplify this statement more. Instead of littering your money everywhere in the bank, keep money in one vault. So in Redux, it is advisable to store the application state in a single object managed by the Redux store.

2. You visit the bank with **action** in your mind i.e WITHDRAW_MONEY. No one is going to give you money if you just roam around here and there. A similar thing happens in Redux. If you want to update the state of your Redux (like you do with setState in React) you need to let Redux know about your action. Just like you follow a process to withdraw money from your bank, Redux also follows a process to change/update the state of your application. This leads to the **second principle** of Redux.

`State is read-only`

Note: The only way to change the state is to emit an action an object describing what happened.

The meaning of the above statement is quite simple. In Redux your action WITHDRAW_MONEY will be represented by an object and it looks something like below…

```
{
 type: "WITHDRAW_MONEY",
 amount: "$10,000"
}
```

The above object is an **action** in the Redux application that has a **type** field describing the action you want to perform. So whenever you need to change/update the state of your Redux application, you need to dispatch an action.

3. Consider your cashier in the bank as a **Reducer** in your Redux application. To WITHDRAW_MONEY from your bank vault, you need to convey your intention/action to the cashier first. Now the cashier will follow some process and it will communicate to the bank vault that holds all the bank's money. A similar thing happens in Redux. To **update** the state of your application you need to convey your **action** to the **reducer.** Now the reducer will take your action, it will perform its job and it will ensure that you get your money. Your Reducer always returns your **new state**. Sending off the action to the reducer is called dispatching an action. This leads to the last or **third principle** of Redux.

To specify how the state tree is transformed by actions, you write pure reducers.

The above 3 principles explained well about three main terminologies of Redux: The **Store**, The **Reducer**, and **Action**.

Setting Up Redux in a React App

1. Store Creation
2. Action Creation
3. Dispatching Actions
4. Reducer Functions
5. Combining Reducers
6. Connecting to React Component

1. Store Creation in Redux

To build a Redux store, developers use the redux library's createStore function and send in a root reducer as an argument. A root reducer is a collection of reducers that describe how the state in an application changes. Here's an illustration of a store built in Redux:

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

// Create the redux store by calling createStore
// and passing in the root reducer
const store = createStore(rootReducer);
```

2. Action Creation in Redux

Redux actions are simple objects that explain changes to the state. Developers define an object with a type property and any other data required to describe the change to make an action. Here's an illustration of how to make an action in Redux:

```
// Define a action creator function that
// takes test as an argument and returns
// an action object.

const addTodo = (text) => {
    return {

        // Describes the action to be taken
        type: 'ADD_TODO',
        text
    };
};
```

3. Dispatching Actions in Redux

To dispatch an action and update the state, developers call the dispatch method on the store and pass in the action as an argument. Here is an example of dispatching an action in Redux:

```
// Dispatch the addTodo action by calling
// store.dispatch and passing in the action
store.dispatch(addTodo('Learn Redux'));
```

4. Reducer Functions in Redux

Redux reducers are pure functions in Redux that accept the current state and an action and return the next state. Here's an example of a Redux reducer function:

```
// Define a reducer function that accepts the
// current state and an action and return the
// next state
```

```
const todoReducer = (state = [], action) => {

    // To handle different action types
    switch (action.type) {

        // For the ADD_TODO action type
        case 'ADD_TODO':
            return [

                // Create a new array with the
                // existing todos
                ...state,
                {
                    // Add a new todo with the text
                    // from the action
                    text: action.text,

                    // Set the completed property
                    // to false
                    completed: false
                }
            ];
        default: // For any other action types
            return state;
    }
};
```

5. Combining Reducers in Redux

If an application has multiple reducers, developers can use the redux library's combineReducers function to combine them into a single root reducer. Here's an example of how to combine reducers in Redux

```
// Combine multiple reducers into a single
// root reducer using combineReducers

import { combineReducers } from 'redux';
const rootReducer = combineReducers({
    todos: todoReducer,
    visibilityFilter: visibilityFilterReducer
});
```

6. Connecting Components to Redux

Developers use the react-redux library's connect function to connect a Redux store to React components. Here's an example of a Redux store being linked to a React component:

```
// Connect a Redux store to a react component
// using the connect
```

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

// Define functional components that accepts
// todos as a prop
const TodoList = ({ todos }) => (
    <ul>
        /* map over the todos array to render
        each todo */
        {todos.map((todo, index) => (
            <li key={index}>{todo.text}</li>
        ))}
    </ul>
);
const mapStateToProps = (state) => {
    return {
        // Specify which properties should
        // be mapped to props
        todos: state.todos
    };
};
const mapDispatchToProps = (dispatch) => {
    return {
        // dispatching plain actions
         todos: ()=> dispatch(todos())
    };
};


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

More info about mapStateToProps & mapDispatchToProps

1. https://react-redux.js.org/using-react-redux/connect-mapstate
2. https://react-redux.js.org/using-react-redux/connect-mapdispatch

Advantages of using Redux with React JS

- **Centralized state management system i.e. Store:** React state is stored locally within a component. To share this state with other components in the application, props are passed to child components, or callbacks are used for parent components. Redux state, on the other hand, is stored globally in the store. All the components of the entire application can easily access the data directly. This centralizes all data and makes it very easy for a component to get the state it requires. So while developing large, complex applications with many components, the Redux store is highly preferred.

- **Performance Optimizations:** By default, whenever a component is updated, React re-renders all the components inside that part of the component tree. In such a case when the data for a given component hasn't changed, these re-renders are wasted (cause the UI output displayed on the screen would remain the same). Redux store helps in improving the performance by skipping such unnecessary re-renders and ensuring that a given component re-renders only when its data has actually changed.

- **Pure reducer functions:** A pure function is defined as any function that doesn't alter input data, doesn't depend on the external state, and can consistently provide the same output for the same input. As opposed to React, Redux depends on such pure functions. It takes a given state (object) and passes it to each reducer in a loop. In case of any data changes, a new object is returned from the reducer (re-rendering takes place). However, the old object is returned if there are no changes (no re-rendering).

- **Storing long-term data:** Since data stored in redux persists until page refresh, it is widely used to store long-term data that is required while the user navigates the application, such as, data loaded from an API, data submitted through a form, etc. On the other hand, React is suitable for storing short-term data that is likely to change quickly (form inputs, toggles, etc.)

- **Time-travel Debugging:** In React, it becomes a tedious task to track the state of the application during the debugging process. Redux makes debugging the application an easy process. Since it represents the entire state of an application at any given point in time, it is widely used for time-travel debugging. It can even send complete error reports to the server!

- **Great supportive community** Since redux has a large community of users, it becomes easier to learn about best practices, get help when stuck, reuse your knowledge across different applications. Also, there are a number of extensions for redux that help in simplifying the code logic and improving the performance.

What's the difference between useContext and Redux ?

In React, useContext and Redux both approaches provide ways to manage and share state across components. Both work and manage global data to share and access from any component present in the application DOM tree, but they have different purposes and use cases.

useContext

useContext is a hook that provides a way to pass data through the component tree without manually passing props down through each nested component. It is designed to share data that can be considered global data for a tree of React components, such as the current authenticated user or theme(e.g. color, paddings, margins, font-sizes).

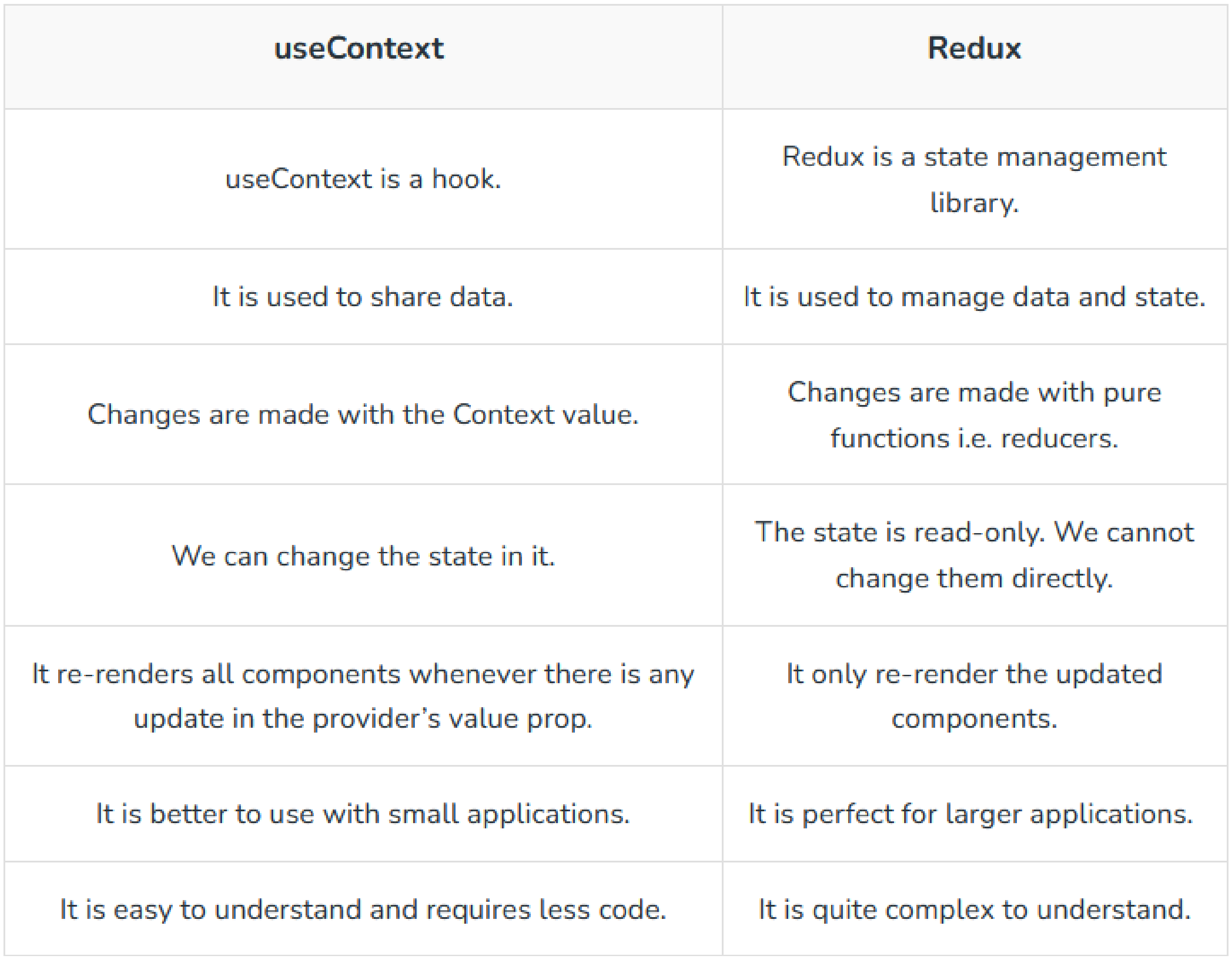
Redux

Redux is a state managing library used in JavaScript apps. It is very popular for React and React-Native. It simply manages the state and data of your application.

Building Parts of Redux

- **Action:** Actions are JavaScript object that contains information. Actions are the only source of information for the store.
- **Reducer: R**educers are the pure functions that contain the logic and calculation that needed to be performed on the state.
- **Store:** Store is an object which provides the state of the application. This object is accessible with help of the provider in the files of the project.

Difference between useContext and Redux



useContext	Redux
useContext is a hook.	Redux is a state management library.
It is used to share data.	It is used to manage data and state.
Changes are made with the Context value.	Changes are made with pure functions i.e. reducers.
We can change the state in it.	The state is read-only. We cannot change them directly.
It re-renders all components whenever there is any update in the provider's value prop.	It only re-render the updated components.
It is better to use with small applications.	It is perfect for larger applications.
It is easy to understand and requires less code.	It is quite complex to understand.

Conclusion:

Using Redux with ReactJS brings several advantages to your application development process. It offers centralized state management, predictable state updates, efficient component communication, scalability, and benefits from a thriving ecosystem and community support. By combining the strengths of ReactJS and Redux, you can build robust, maintainable, and scalable applications with ease.

Redux was developed to help front-end developers write applications for consistent behavior. In addition to this, redux also helps tackle the React performance issues. As a result, Redux is constant when it comes to running in different environments–native, server, and client. Usually, Redux works with React to eliminate issues encountered with the state management in massive applications. Furthermore, since React is troublesome to resume components as it is tightly coupled with the root components, redux helps reduce the complexity. In addition to this, it also offers global accessibility for building easy-to-test applications.

Choosing the right state management solution is crucial for your application’s scalability. So that this comparision helping you make informed decisions based on your project needs.