Redux Js

Redux is an open-source JavaScript library for managing and centralizing application state. It is most commonly used with libraries such as React or Angular for building user interfaces. Similar to Facebook's Flux architecture, it was created by Dan Abramov and Andrew Clark. Wikipedia

Programming languages: JavaScript, TypeScript

Developer: Dan Abramov

Initial release date: 2 June 2015

License: MIT License

Platform: Cross-platform software

Stable release: 4.1.0 / April 24, 2021

Redux is a state management framework that can be used with a number of different web technologies, including React.

In Redux, there is a single state object that's responsible for the entire state of your application. This means if you had a React app with ten components, and each component had its own local state, the entire state of your app would be defined by a single state object housed in the Redux store.

Contents -

- **What is redux?**
- **Redux principle.**
- **Redux state.**
- **Redux action.**
- **Redux reducer.**
- **Redux store.**
- **Redux data flow.**
- **Redux middleware.**

What is redux?

Redux is a predictable state container for JavaScript apps. As the application grows, it becomes difficult to keep it organized and maintain data flow. Redux solves this problem by managing the application's state with a single global object called Store. Redux fundamental principles help in maintaining consistency throughout your application, which makes debugging and testing easier.

Redux Principle -

Redux has mainly below 4 parts/components -

1. State
2. Action
3. Reducer
4. Store

Principles of Redux

Predictability of Redux is determined by three most important principles as given below -

Single Source of Truth

The state of your whole application is stored in an object tree within a single store. As the whole application state is stored in a single tree, it makes debugging easy, and development faster.

State is Read-only

The only way to change the state is to emit an action, an object describing what happened. This means nobody can directly change the state of your application.

Action -

An action is simply a JavaScript object that contains information about an action event that has occurred. The Redux store receives these action objects, then updates its state accordingly.

```
{
    type : "ADDTOCART",
    payload : "Samsung Galaxy Note 2"
}
```

Reducer -

A reducer is a central place where state modification takes place. Reducer is a function which takes state and action as arguments, and returns a newly updated state. That means reducer takes action and state and updates the state of application as per action type.

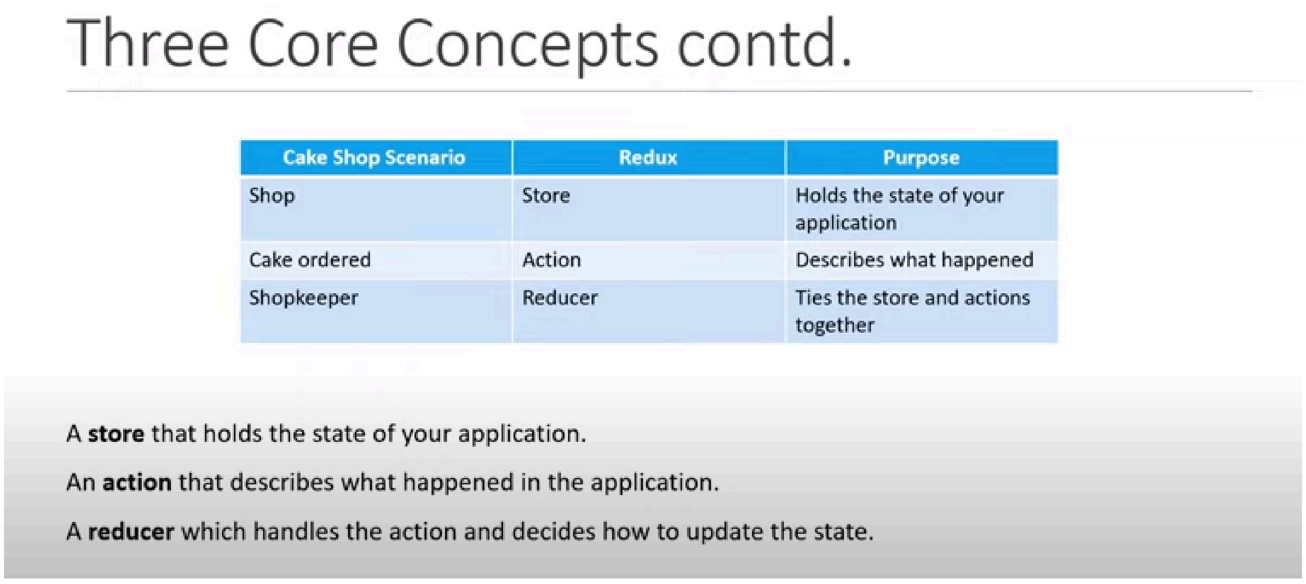
```
const addToCartReducer = (state, action) => {
    //logic
}
```

Store -

The Redux store is an object which holds and manages application state.

const store = Redux.createStore(addToCartReducer)

An example of a Cake seller to understand above -



Three Core Concepts contd.

Cake Shop Scenario	Redux	Purpose
Shop	Store	Holds the state of your application
Cake ordered	Action	Describes what happened
Shopkeeper	Reducer	Ties the store and actions together

A **store** that holds the state of your application.

An **action** that describes what happened in the application.

A **reducer** which handles the action and decides how to update the state.

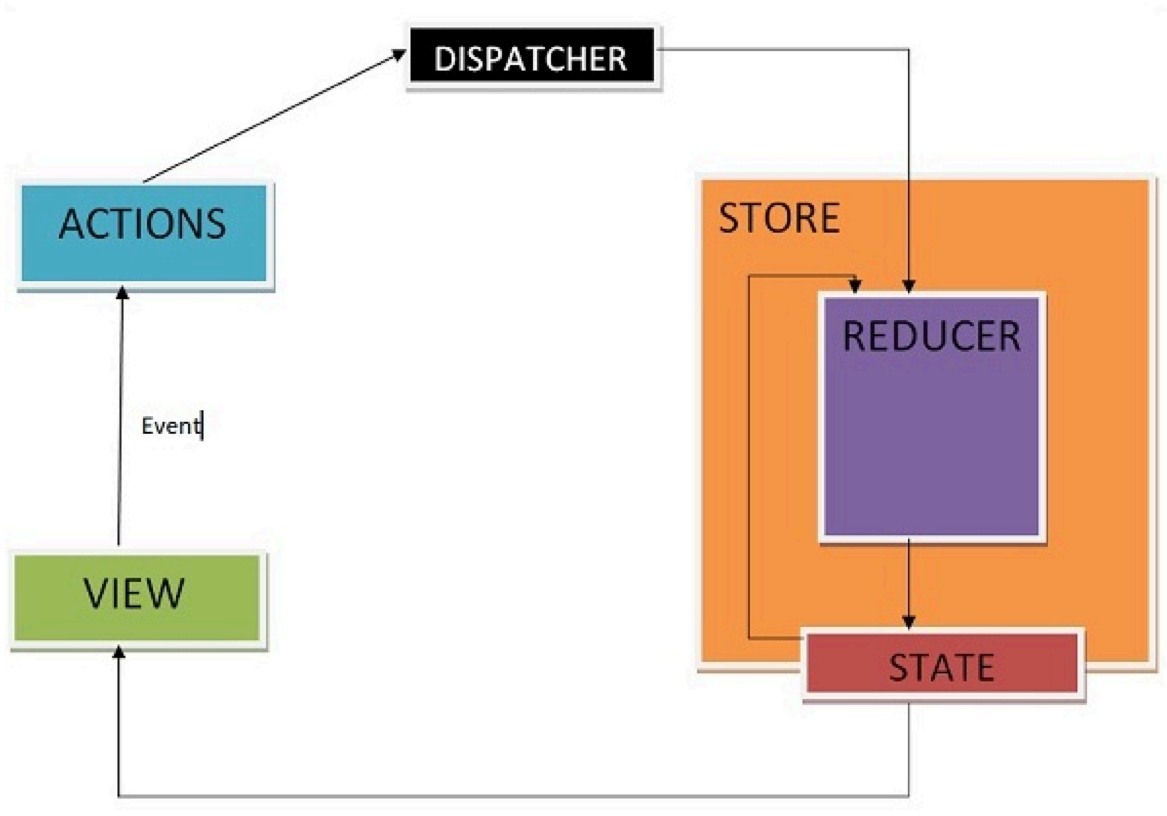
Redux installation -

```
> npm i redux
```

Redux data flow -

Redux follows the unidirectional data flow. It means that your application data will follow a one-way binding data flow. As the application grows & becomes complex, it is hard to reproduce issues and add new features if you have no control over the state of your application.

Redux reduces the complexity of the code, by enforcing the restriction on how and when state updates can happen. This way, managing updated states is easy. We already know about the restrictions as the three principles of Redux. Following diagram will help you understand Redux data flow better –



- An action is dispatched when a user interacts with the application.

- The root reducer function is called with the current state and the dispatched action. The root reducer may divide the task among smaller reducer functions, which ultimately returns a new state.

- The store notifies the view by executing their callback functions.
- The view can retrieve updated state and re-render again.

Redux store -

The Redux store is an object which holds and manages application state. There is a method called createStore() on the Redux object, which you use to create the Redux store. This method takes a reducer function as a required argument. It simply takes state as an argument and returns state.

1.Creating a redux store -

1.Problem statement -

Create a simple redux store.

Solution -

```
const store = Redux.createStore(reducer);
```

Creating a reducer -

```
const reducer = (state, action) => {
  return state;
}
```

Creating a action -

```
const action = {
  type : 'Login',// type
  userName : 'Stew' // payload
}
```

2.Get State from the Redux Store -

The Redux store object provides several methods that allow you to interact with it. For example, you can retrieve the current state held in the Redux store object with the **getState()** method.

Syntax -

`store.getState()`

2.Problem statement -

Declare a store variable and assign it to the createStore() method, passing in the reducer as an argument.

```
const reducer = (state = 5) => {
  return state;
}
```

Solution -

In app.js

```
import {legacy_createStore as createStore} from 'redux';

function App() {
  // creating a reducer.
  const abc = (state = 5, action = {}) => {
    return state;
  }

  // creating a store.
  const store = createStore(abc);

  //predefined redux method to access state value.
  const currentvalue = store.getState();
  console.log("value inside store =", currentvalue);
  return (
    <>
      <h1>Value of Redux store = {currentvalue}</h1>
    </>
  );
}

export default App;
```

3.Define a Redux Action -

Since Redux is a state management framework, updating state is one of its core tasks. In Redux, all state updates are triggered by dispatching actions. An action is simply a JavaScript object that contains information about an action event that has occurred. The Redux store receives these action objects, then updates its state accordingly.

Syntax -

```
const action = {
 type : 'value',
 payload : 'data'
}
```

3.Problem statement -

Writing a Redux action is as simple as declaring an object with a type property. Declare an object action and give it a property type set to the string 'LOGIN'.

Solution -

```
const action = {
  type: 'LOGIN',
  payload : null
}
```

4.Define an Action Creator-

After creating an action, the next step is sending the action to the Redux store so it can update its state. In Redux, you define action creators to accomplish this. An action creator is simply a JavaScript function that returns an action. In other words, action creators create objects that represent action events.

4.Problem statement -

Define a function named actionCreator() that returns the action object when called.

```
const action = {
  type: 'LOGIN'
}
```

Solution -

```
const action = {
  type: 'LOGIN',
  payload : null
}
// Define an action creator here:
function actionCreator() {
  return action
}
```

Or

```
const actionCreator = () => {
    // const action = {
```

```
  //   type: 'LOGIN',
  //   payload : null
  // }
  // return action;
  // shorthand of above commented code.
  return {
    type : 'LOGIN',
    payload: null
  }
};
```

5.Dispatch an Action Event -

dispatch method is what you use to dispatch actions to the Redux store. Calling store.dispatch() and passing the value returned from an action creator sends an action back to the store.

Syntax -

store.dispatch(actionCreator());

6.Handle an Action in the Store -

After an action is created and dispatched, the Redux store needs to know how to respond to that action. This is the job of a reducer function. Reducers in Redux are responsible for the state modifications that take place in response to actions. A reducer takes state and action as arguments, and it always returns a new state.

```
import { createStore } from "redux";
const Initialstate = { // initial values
  name: "John"
}

function App() {
  const nameAction = () => { // action creator
    return {
      type: "LOGIN",
      payload: null
    }
  }
  const nameReducer = (state = Initialstate, action) => { // reducer
    console.log("action type is =", action.type)
    if (action.type === "LOGIN") {
      console.log("inside IF")
      return {
        name: "John Loged In Success"
      }
    } else {
      console.log("inside Else");
      return {
        name: "LoggedIn Failed"
      }
    }
  }

  const store = createStore(nameReducer); // store
  store.dispatch(nameAction()); // dispatch
  console.log("name is =", store.getState()) //getState() use to access
state value from store.
  const nm = store.getState();
  return (
    <div>
      <h1>Name =  {nm.name}</h1>
    </div>
  );
}

export default App;
```

6.Problem statement -

Fill in the body of the reducer function so that if it receives an action of type 'LOGIN' it returns a state object with login set to true. Otherwise, it returns the current state. Note that the current state and the dispatched action are passed to the reducer, so you can access the action's type directly with action.type.

```
const defaultState = {
  login: false
};

const reducer = (state = defaultState, action) => {
  // Change code below this line
};

const store = createStore(reducer);
const loginAction = () => {
  return {
    type: 'LOGIN'
  }
};
```

Solution -

```
import { legacy_createStore as createStore } from 'redux';

function App() {
  // passing default or initial value to redux state.
  const defaultState = {
    login: false
  };

  // action creator is  a function which takes action.
  const loginAction = () => {
    // const action = {
    //   type: 'LOGIN',
    //   payload : null
    // }
    // return action;
    // short hand of above commented code.
    return {
      type : 'LOGIN',
      payload: null
    }
  };

  const reducer = (state = defaultState, action) => {
    if (action.type === 'LOGIN') {
      return {
        login: true
      }
    } else {
      return state;
    }
  };
  // create a redux store
  const store = createStore(reducer);
  let stateValue = store.getState();
  console.log("default State Status =", stateValue);
  // update redux state on button click.
  const handleLogin = () => {
    // dispatch action/action creator to update the state of redux store.
    store.dispatch(loginAction());
    stateValue = store.getState();
    console.log("new status Status =", stateValue)
  }
  return (
    <>
      <div>
        <button onClick={() => handleLogin()}>Update State</button>
      </div>
    </>
  );
}
export default App;
```

Using switch statement for action -

```
  const reducer = (state = defaultState, action) => {
    switch(action.type) {
      case 'LOGIN' :
        return {
          login : true
        }
        default :
        return state;
    }
  };
```

7.Use a Switch Statement to Handle Multiple Actions

7.Problem statement -

You can tell the Redux store how to handle multiple action types. Say you are managing user authentication in your Redux store. You want to have a state representation for when users are logged in and when they are logged out.

```
import { legacy_createStore as createStore } from 'redux';

function App() {
  // passing default or initial value to redux state.
  const defaultState = {
    login: "User Not Active"
  };

  // action creator is  a funtion which takes action.
  const loginUser = () => {
    return {
      type : 'LOGIN',
      payload: null
    }
  };
  const logoutUser = () => {
    return {
      type: 'LOGOUT'
    }
  };
  const reducer = (state = defaultState, action) => {
    switch(action.type) {
      case 'LOGIN' :
        return {
          login : "Logged In Success!!"
        };
        case 'LOGOUT' :
        return {
          login : "Logged Out Success!!"
        }
        default :
        return defaultState;
    }
  };
  // create a redux store
  const store = createStore(reducer);
  let stateValue = store.getState();
  console.log("default State of User ===", stateValue);
  // update redux state on button click.
  const handleLogin = () => {
    // dispatch action/action creator to update the state of redux store
on click.
    store.dispatch(loginUser());
    stateValue = store.getState();
    console.log("status of User ===", stateValue)
  }
  const handleLogout = () => {
    // dispatch action/action creator to update the state of redux store
on click.
    store.dispatch(logoutUser());
    stateValue = store.getState();
    console.log("status of User ===", stateValue)
  }
  return (
    <>
      <div>
        <button onClick={() => handleLogin()}>Login</button>
        <button onClick={() => handleLogout()}>Logout</button>
      </div>
    </>
  );
}
export default App;
```

8.Use const for Action Types

A common practice when working with Redux is to assign action types as read-only constants, then reference these constants wherever they are used. You can refactor the code you're working with to write the action types as const declarations.

8. Problem statement -

Declare LOGIN and LOGOUT as const values and assign them to the strings 'LOGIN' and 'LOGOUT', respectively. Then, edit the authReducer() and the action creators to reference these constants instead of string values.

```
const defaultState = {
  authenticated: false
};
const authReducer = (state = defaultState, action) => {
  switch (action.type) {
    case 'LOGIN':
      return {
        authenticated: true
      }
    case 'LOGOUT':
      return {
        authenticated: false
      }

    default:
      return state;
  }

};

const store = Redux.createStore(authReducer);
const loginUser = () => {
  return {
    type: 'LOGIN'
  }
};

const logoutUser = () => {
  return {
    type: 'LOGOUT'
  }
};
```

Solution -

```
import {legacy_createStore as createStore} from 'redux';

const UseConstForActionTypes = () => {
    const LOGIN = 'LOGIN';
    const LOGOUT = 'LOGOUT';

    const defaultState = {
        authenticated: false
    };
    const authReducer = (state = defaultState, action) => {
        switch (action.type) {
            case LOGIN:
                return {
                    authenticated: true
                }
            case LOGOUT:
                return {
                    authenticated: false
                }

            default:
                return state;
        }
    };
    const loginUser = () => {
        return {
            type: LOGIN
        }
    };
    const logoutUser = () => {
        return {
            type: LOGOUT
        }
    };
    const store = createStore(authReducer,
window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__({
    serialize: true
  }));
    let currentState = store.getState();
    console.log("currentState", currentState);

    const handleLogin = () => {
        store.dispatch(loginUser());
        currentState = store.getState();
        console.log("state when login", currentState);
    }

    const handleLogout = () => {
        store.dispatch(logoutUser());
        currentState = store.getState();
        console.log("state when logout", currentState);
    }
    return (
        <div>
            <button onClick={() => handleLogin()}>Login</button>
            <button onClick={() => handleLogout()}>Logout</button>
        </div>
    )
}
export default UseConstForActionTypes;
```

9&10.Combine Multiple Reducers & Send Action Data to the Store.

When the state of your app begins to grow more complex, it may be tempting to divide the state into multiple pieces. Instead, remember the first principle of Redux: all app state is held in a single state object in the store. Therefore, Redux provides reducer composition as a solution for a complex state model. You define multiple reducers to handle different pieces of your application's state, then compose these reducers together into one root reducer. The root reducer is then passed into the Redux createStore() method.

In order to let us combine multiple reducers together, Redux provides the **combineReducers()** method.

Syntax -

```
const rootReducer = Redux.combineReducers({
  reducer1: reducer1_name,
  reducer2: reducer2_name
});
```

9&10. Problem statement -

There are counterReducer() and authReducer() functions provided in the code editor, along with a Redux store. Finish writing the rootReducer() function using the Redux.combineReducers() method. Assign counterReducer to a key called count and authReducer to a key called auth.

```
const INCREMENT = 'INCREMENT';
const counterReducer = (state = 0, action) => {
  switch(action.type) {
    case INCREMENT:
      return state + 1;
    default:
      return state;
  }
};

const LOGIN = 'LOGIN';

const authReducer = (state = {authenticated: false},
action) => {
  switch(action.type) {
    case LOGIN:
      return {
```

```
    authenticated: true
   }
  Default:
   return state;
 }
};
const rootReducer = // Define the root reducer here
const store = Redux.createStore(rootReducer);
```

Solution -

```
import {createStore, combineReducers} from 'redux';
const CombineReducerComponent = () => {
    // count redux part
    const initialState = {
        count : 0
    }

    const INCREMENT = 'INCREMENT';

    const IncrementCount = () => {
        return {
            type : INCREMENT
        }
    }
    const counterReducer = (state = initialState, action) => {
        switch (action.type) {
            case INCREMENT:
                return {
                    count : state.count + 1,
                }
            default:
                return state;
        }
    };
    // Login redux part =================
    const defaultUserStatus = {
        authenticated : false,
        data : ''
    }
    const LOGIN = 'LOGIN';

    const loginUser = () => {
        return {
            type : LOGIN,
            payload : "Sam Cameron"
        }
    }

    const authReducer = (state = defaultUserStatus, action) => {
        switch (action.type) {
            case LOGIN:
                return {
                    authenticated: true,
                    data : action.payload
                }
            default:
                return state;
        }
    };

    const rootReducer = combineReducers({
        count: counterReducer,
        auth: authReducer
    })

    const store = createStore(rootReducer,
window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__({
    serialize: true
  }));
    console.log('default values -', store.getState());

    const handleCount = () => {
        store.dispatch(IncrementCount());
        console.log('counter value -', store.getState());
    }

    const handleLogin = () => {
        store.dispatch(loginUser());
        console.log('login status -', store.getState());
    }
    return (
       <>
        <h1>combineReducers</h1>
        <button onClick={() => handleCount()}>Click Count</button>
        <button onClick={() => handleLogin()}>Login</button>
       </>
    )
}
export default CombineReducerComponent;
```

11.Use Middleware to Handle Asynchronous Actions

At some point you'll need to do asynchronous calls (like API calls) in your Redux app, so how do you handle these types of requests? Redux provides middleware designed specifically for this purpose, called Redux Thunk middleware. Here's a brief description of how to use this with Redux.

```
> npm i redux-thunk
```

With a plain basic Redux store, you can only do simple synchronous updates by dispatching an action. Middleware extends the store's abilities, and lets you write async logic that interacts with the store.

Thunks are the recommended middleware for basic Redux side effects.

Redux middleware is used for the same purpose for which we use useEffect in React.

The `useEffect` Hook allows you to perform side effects in your components.

Some examples of side effects are: fetching data, directly updating the DOM, and timers.

We use thunk and saga middlewares for the same purpose but in redux application.

To include Redux Thunk middleware, you pass it as an argument to Redux.applyMiddleware(). This statement is then provided as a second optional parameter to the createStore() function. Then, to create an asynchronous action, you return a function in the action creator that takes dispatch as an argument. Within this function, you can dispatch actions and perform asynchronous requests.

In this example, an asynchronous request is simulated. It's common to dispatch an action before initiating any asynchronous behavior so that your application state knows that some data is being requested. Then, once you receive the data, you dispatch another action which carries the data as a payload along with information that the action is completed.

Remember that you're passing dispatch as a parameter to this special action creator. This is what you'll use to dispatch your actions, you simply pass the action directly to dispatch and the middleware takes care of the rest.

```
import { createStore, applyMiddleware } from "redux";
import thunk from "redux-thunk";

function App() {
  const initialState = {
    dataFetchRequest: false,
    data: []
  }
  const REQUESTDATA = "REQUESTDATA";
  const RESPONSEDATA = "RESPONSEDATA";

  const requestForData = () => {
    return {
      type: REQUESTDATA
    }
  }
  const responseData = (userData) => {
    return {
      type: RESPONSEDATA,
      payload: userData
    }
  }

  const ApiReducer = (state = initialState, action) => {
    switch (action.type) {
      case REQUESTDATA:
        return {
          dataFetchRequest: true,
          payload: []
        }
      case RESPONSEDATA:
        return {
          dataFetchRequest: true,
          data: action.payload
        }
      default:
        return state
    }
  }
  const store = createStore(ApiReducer,
    window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__({
      serialize: true
    },
      applyMiddleware(thunk))
  );

  const handleApiCall = () => {
    store.dispatch(requestForData());
    console.log('--->', store.getState());
    // Side Effect
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(resp => resp.json())
      .then(data => {
        console.log('Data -', data);
        store.dispatch(responseData(data)); // passing API response to
responseData Action.
        console.log('====>', store.getState());
      })
      .catch(err => console.log(err));
  }
  return (
    <>
      <h1>Middleware</h1>
      <button onClick={() => handleApiCall()}>Call API</button>
    </>
  );
}

export default App;
```

React and Redux

- **Getting Started with React Redux**
- **Manage State Locally First**
- **Use Provider to Connect Redux to React**
- **useDispatch**
- **useSelector**

1]Getting Started with React Redux

React is a view library that provides UI with data, then it renders the view in an efficient, predictable way. Redux is a state management framework that you can use to simplify the management of your application's state. Typically, in a React Redux app, you create a single Redux store that manages the state of your entire app.

Although React components can manage their own state locally, when you have a complex app, it's generally better to keep the app state in a single location with Redux.

Use Provider to Connect Redux store to React application and useDispatch to dispatch action and useSelector to render redux state data to UI -

To provide React access to the Redux store and the actions it needs to dispatch updates.
React Redux provides its react-redux package to help accomplish these tasks.

React Redux provides a small API with key features:
Provider.
The Provider is a wrapper component from React Redux that wraps your React app.
This wrapper then allows you to access the Redux store and dispatch functions throughout your component tree.
Provider takes two props, the Redux store and the child components of your app.
Defining the Provider for an App component might look like this:
```
<Provider store={store}>
  <App/>
</Provider>
```
useDispatch -
The useDispatch hook is used to update the state of the component and return a new state by dispatching action creator.
Problem statement -
Create a simple app where entering values in the textbox should be shown in Lists as well as redux state.

Solution -

Install react-redux package-
> npm i react-redux

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import ReduxMiddleware from './ReduxMiddleware'; // component
import { Provider } from 'react-redux';
import { createStore, applyMiddleware } from 'redux';
import {apiReducer} from './ReduxMiddleware';
import thunk from 'redux-thunk';

const root = ReactDOM.createRoot(document.getElementById('root'));
// store
const store = createStore(apiReducer, applyMiddleware(thunk));
root.render(
  <div>
    <Provider store={store}>
    <ReduxMiddleware />
    </Provider>
  </div>
);
```

ReduxMiddleware.js

```
import { useDispatch, useSelector } from "react-redux";
// default state
const apiDefaultState = {
    dataFetching: false,
    data: []
}

// STEP -3 action creator
const REQUESTDATA = 'REQUESTDATA';
const requestAPIActionCreator = () => {
    return {
        type: REQUESTDATA,
        payload: []
    }
}
// action creator
const RESPONSEDATA = 'RESPONSEDATA'
const responseDataActionCreator = (apiDataResp) => {
    return {
        type: RESPONSEDATA,
        payload: apiDataResp
    }
}

// STEP -4
export const apiReducer = (state = apiDefaultState, action) => {
    switch (action.type) {
        case REQUESTDATA:
            return {
                dataFetching: false,
                payload: []
            }
        case RESPONSEDATA:
            return {
                dataFetching: true,
                data: action.payload
            }
        default:
            return state
    }
}

const ReduxMiddleware = () => {
    const lists = useSelector(state => state);
    const dispatch = useDispatch();

    const handleAPICall = () => {
        dispatch(requestAPIActionCreator());
        // calling the API
        fetch("https://jsonplaceholder.typicode.com/posts")
            .then(resp => resp.json())
            .then(data => {
                dispatch(responseDataActionCreator(data));// this will
update store with api response data

            })
            .catch(err => console.log(err))
    }
    console.log('Lists -', lists.data);
    // STP -7
    return (
        <div>
            <h1>REDUX THUNK !!</h1>
            <button type="button"
      onClick={() => handleAPICall()}>SHOW DATA</button>
            <div>
                {
           lists.dataFetching === false ? <li>No Record Found...</li> :
               <>
                    {
                        lists.data.map((i) => <li>Title : {i.title}</li>)
                     }
               </>
                }
            </div>
        </div>
    )
}
export default ReduxMiddleware;
```

-Redux interview questions -

Redux is an open-source JavaScript library commonly used with React to manage the state of a web application. It provides a predictable and centralized state container that helps developers manage the data flow in a more organized and maintainable way, especially in larger and more complex applications.

1] What is the redux principle ?

Ans -

Three Principles

1. Single source of truth The global state of your application is stored in an object tree within a single store. ...
2. State is read-only The only way to change the state is to emit an action, an object describing what happened. ...
3. Changes are made with pure functions

2] What is a redux life cycle ?

Ans -

3] What is a redux store ? how to create it?

Ans -

The Redux store is a central repository for managing the state of a JavaScript application. It holds all the data that your application needs to keep track of in a single, immutable object. The store provides methods to access and update this state, and it enforces a predictable and controlled way to manage data flow in your application, especially when combined with actions and reducers.

4] What is redux thunk ?

Ans - Redux Thunk middleware allows you to write action creators that return a function instead of an action.
The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met.
The inner function receives the store methods dispatch and getState as parameters.