Sagar Gautam

Redux Explained in Detail with an Example

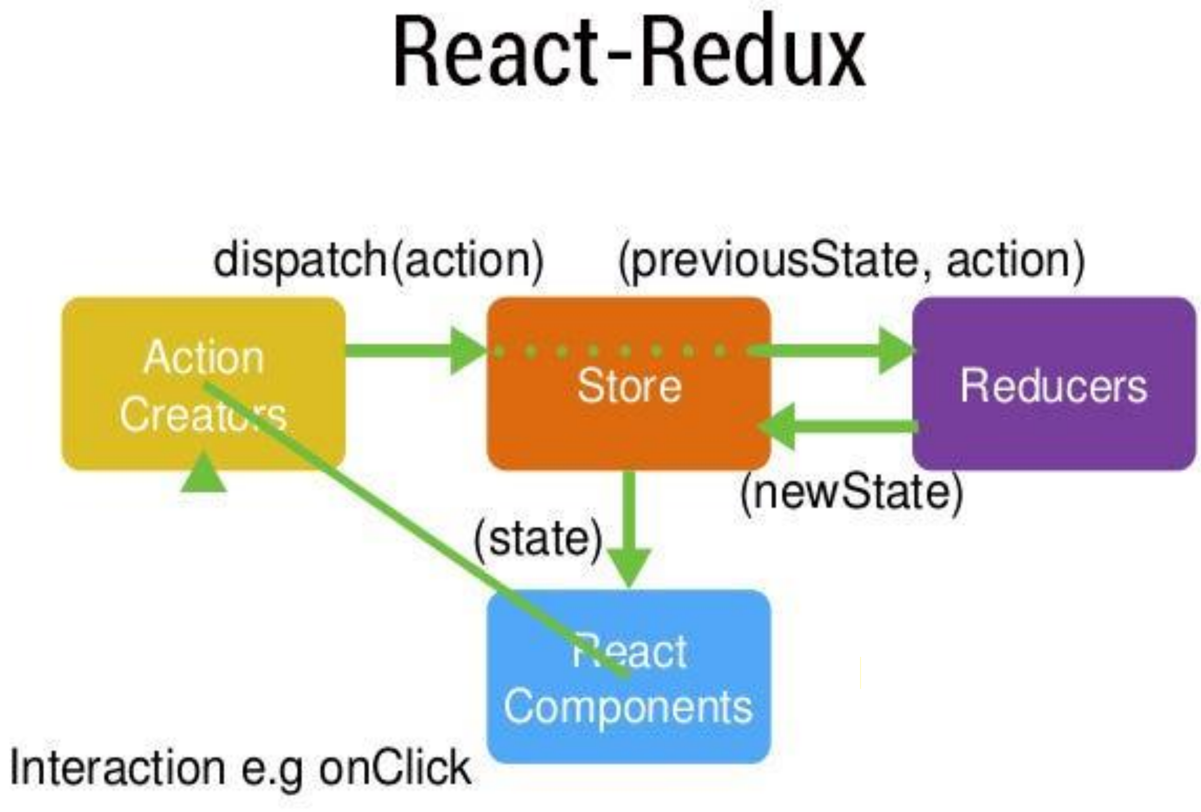
Redux is a predictable state management library for JavaScript applications, often used with React but also compatible with other frameworks or vanilla JavaScript. It helps manage the global state of an application in a centralized and predictable way, making it easier to debug, test, and maintain.

Why Use Redux?

1. **Centralized State** : Instead of managing state in multiple components, Redux stores all the state in a single place called the "store."
2. **Predictable Updates** : Redux enforces strict rules for how and when the state can be updated.
3. **Debugging Made Easy** : Tools like Redux DevTools allow you to track every state change and action in your app.
4. **Scalability** : Redux is particularly useful for large applications where state management can become complex.

Core Concepts of Redux:

1. **Store**: The single source of truth that holds the entire application state.
2. **Actions**: Plain JavaScript objects that describe **what happened** (e.g., a button click).
3. **Reducers**: Pure functions that take the current state and an action, and return a **new state**.
4. **Dispatch**: The method used to send actions to the store to update the state.
5. **Subscribe(State)**: A method to listen for state changes in the store.



How Redux Works

The flow of data in Redux follows these steps:

1. **Dispatch an Action** : When something happens in the app (e.g., a button click), an action is dispatched.
2. **Reducer Processes the Action** : The reducer function receives the current state and the action, then returns a new state based on the action type.
3. **Update the Store** : The store updates its state with the new value returned by the reducer.
4. **React to State Changes** : Components subscribed to the store re-render with the updated state.

Example: A Simple Counter App

Let’s build a simple counter app using Redux to demonstrate how it works.

Step 1: Set Up the Store

The store is the central place where the state lives. We create it using **createStore** from Redux.

………

Javascript:

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

// Initial state

const initialState = {

  count: 0,

};


// Reducer function

function counterReducer(state = initialState, action) {

  switch (action.type) {

    case 'INCREMENT':

      return { ...state, count: state.count + 1 };

    case 'DECREMENT':

      return { ...state, count: state.count - 1 };

    default:

      return state;

  }

}


// Create the store

const store = createStore(counterReducer);
```

....

- **Explanation** :
    - **initialState** defines the starting state of the app (**count: 0**).
    - **counterReducer** is a pure function that takes the current state and an action, and returns a new state based on the action type.
    - **createStore** creates the Redux store with the reducer.

Step 2: Define Actions

Actions are plain objects that describe what happened. They must have a **type** property.

....

Javascript:

```
// Action creators
function increment() {
  return { type: 'INCREMENT' };
}

function decrement() {
  return { type: 'DECREMENT' };
}
```

- **Explanation** :
    - **increment** and **decrement** are action creators that return action objects with specific types.

Step 3: Dispatch Actions

To update the state, we dispatch actions to the store.

....

Javascript:

```
console.log('Initial State:', store.getState()); // { count: 0 }


// Dispatch an action to increment the count
store.dispatch(increment());
console.log('After Increment:', store.getState()); // { count: 1 }


// Dispatch an action to decrement the count
store.dispatch(decrement());
console.log('After Decrement:', store.getState()); // { count: 0 }
```

- **Explanation** :
  - **store.dispatch(action)** sends the action to the reducer.
  - The reducer processes the action and updates the state.

Step 4: Subscribe to State Changes

You can subscribe to the store to react to state changes.

javascript

```
// Subscribe to state changes
store.subscribe(() => {
  console.log('State Updated:', store.getState());
});


// Dispatch actions again to see the subscription in action
store.dispatch(increment()); // Logs: State Updated: { count: 1 }
store.dispatch(decrement()); // Logs: State Updated: { count: 0 }
```

………………………………………………………………………………………………………

- **Explanation** :
    - **store.subscribe(callback)** registers a listener that gets called whenever the state changes.

Step 5: Integrate with React (Optional)

If you’re using React, you can connect Redux to your components using **react-redux**.

1. **Install react-redux:**

   Bash:

   ```
   npm install react-redux
   ```

2. **Wrap your app with the Provider component:**

   ……………………………………………………………………………………………

Javascript:

```
import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import App from './App';

import store from './store';


ReactDOM.render(

  <Provider store={store}>

    <App />

  </Provider>,

  document.getElementById('root')

);
```

3. **Use useSelector and useDispatch hooks in your components:**

Javascript:

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';

function Counter() {
  const count = useSelector((state) => state.count); // Access state
  const dispatch = useDispatch(); // Dispatch actions

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
    </div>
  );
}

export default Counter;
```

- **Explanation** :
  - **useSelector** extracts the state from the Redux store.
  - **useDispatch** allows you to dispatch actions.

Key Takeaways

1. **Store** : Holds the entire state of the app.
2. **Actions** : Describe what happened (e.g., "Increment the counter").
3. **Reducers** : Specify how the state changes in response to actions.
4. **Flow** : Action → Reducer → New State → Update UI.

By following this structure, Redux ensures that your app's state is predictable and easy to manage, even as it grows in complexity.

Final Notes

While Redux is powerful, it’s not always necessary for small apps. For simpler use cases, React’s built-in **useState** and **useReducer** hooks might suffice. However, for larger applications with complex state management needs, Redux shines.