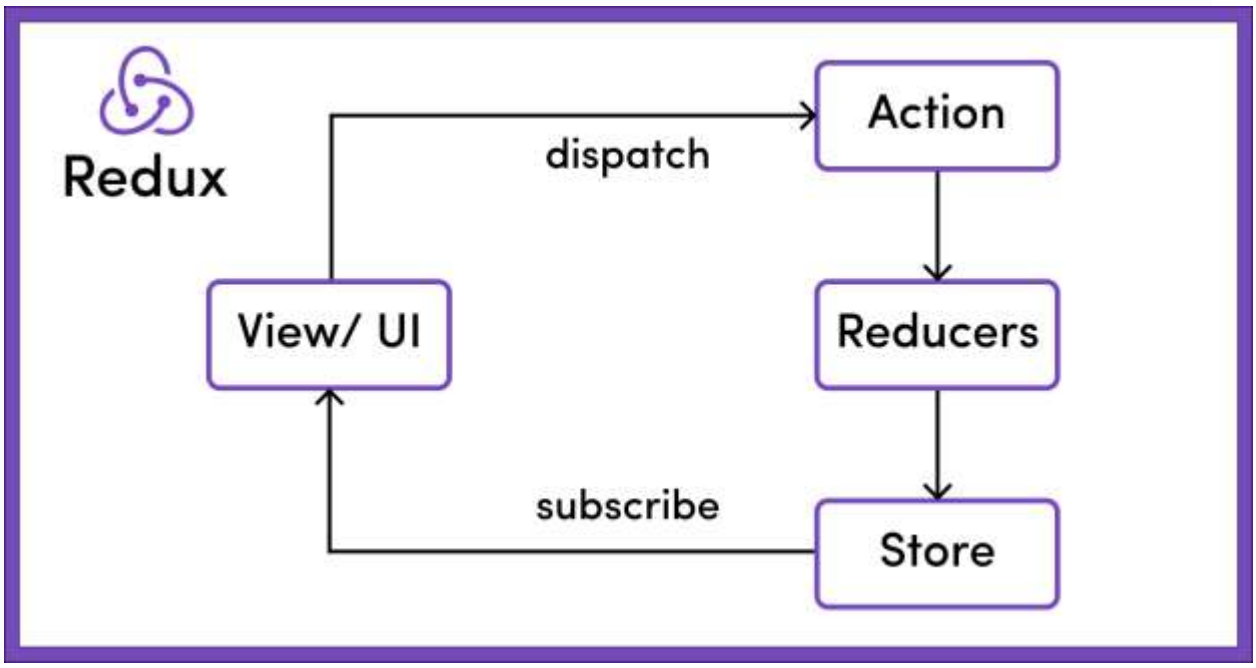
Introduction

Redux is a state management library that is frequently used in conjunction with React to manage an application's state reliably. It adheres to the Flux architecture tenets and is especially helpful for state management in expansive and intricate applications. Let's explore Redux's in-depth features within a React application.

Core Concept of Redux



- **Store:**
  The store is a single source of truth for the state in your application. It holds the entire state tree of your application. You create the store using the 'createStore' function from Redux.

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

const store = createStore(rootReducer);
```

- **Actions:**
  Simple JavaScript objects called actions are used to explain changes made to the application. A type property on them must indicate the 'type' of action

being carried out.

```
const increment = {
  type: 'INCREMENT',
};
```

- **Reducers:**
  Reducers are simple functions that define how an action affects the state of the application. They return to the new state after accepting as arguments the prior state and an action.

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

- **Dispatch:**
  The store uses a method called dispatch to send actions to the store. It is the sole means of inducing a shift in state

.

```
store.dispatch(increment);
```

- **Subscribe:**
  A listener function can be added using the 'subscribe' method, which will trigger every time an action is dispatched and the state may have changed

.

```
const unsubscribe = store.subscribe(() => {
  console.log('State changed:', store.getState());
});
```

- You can unsubscribe to stop listening to state changes.

```
unsubscribe();
```

Example:

Redux Flow (Counter Example)

1. **Store**
   - Think of it like a **big box** that holds your app's state.
   - In your case, the box only has one thing inside:
   - { count: 0 }
2. **Actions**
   - These are just **messages** that describe *what happened*.
   - Example:
   - { type: "INCREMENT" }
   - { type: "DECREMENT" }
3. **Reducer**
   - The **rules** that decide how the state changes when an action happens.
   - Example:
     - If action = "INCREMENT" → add +1 to count.
     - If action = "DECREMENT" → subtract -1.
4. **Dispatch**
   - The **only way** to send an action to the store.
   - Example: when you click the button:
   - dispatch({ type: "INCREMENT" })
5. **Subscribe / useSelector**
   - Your React components **watch the store** (subscribe).
   - When the state changes, React automatically re-renders with the new value.
   - Example:
   - const count = useSelector(state => state.count);

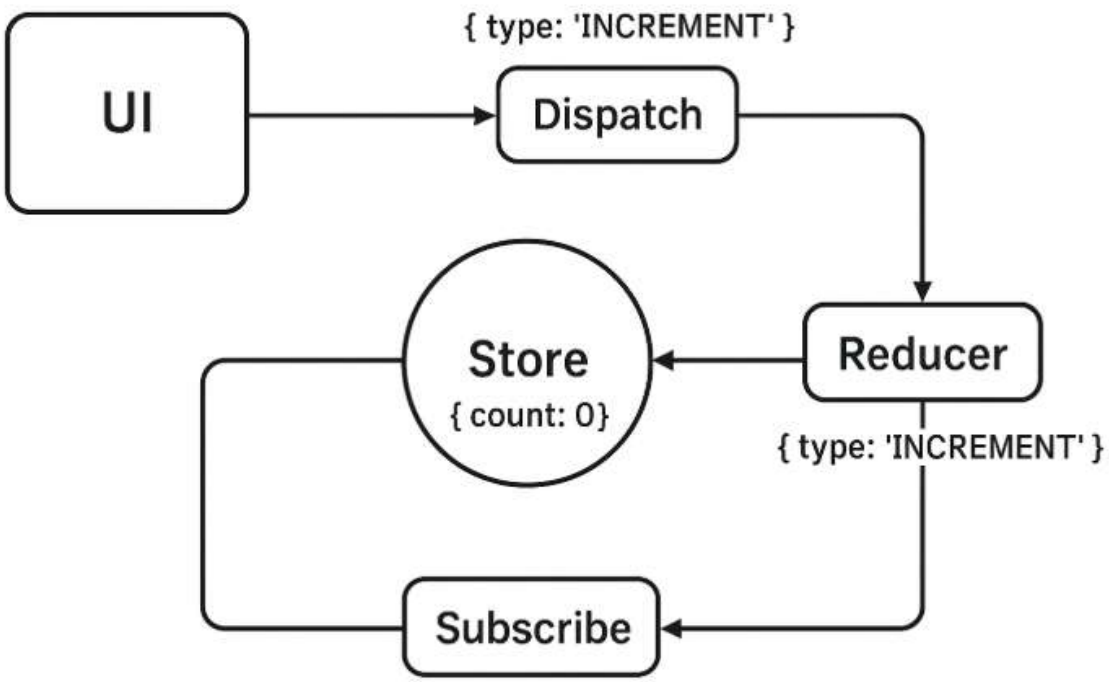
How it works step by step when you click "Increment":

1. When click the button → dispatch({ type: "INCREMENT" }) is called.
2. The **store** sends this action to the **reducer**.
3. The **reducer** looks at the action and updates the state → count + 1.
4. The **store** saves the new state.

5. All components using useSelector notice the change and re-render with the new count.

So the cycle is always:

UI → dispatch(action) → reducer(state, action) → new state → UI updates



The flow works like this:

1. **UI (User Interface)**
   - You click a button in your React app (like *Increment*).
2. **Dispatch**
   - That click sends an **action** (a plain object, e.g. { type: "INCREMENT" }) to the Redux store.
3. **Reducer**
   - The reducer looks at the current state and the action.
   - It decides how the state should change (e.g. increase count by 1).
4. **Store**
   - The store saves this **new state**.

5. **Subscribe (useSelector in React)**
    - React components are “subscribed” to the store.
    - When the state changes, those components re-render with the updated value.

Advantages

Redux is a state management library commonly used with React to manage the state of an application. Here are some advantages of using Redux in React:

- Centralized State Management: Redux centralizes the application state in a single store, making it easier to manage and debug the state of the entire application.
- Predictable State Changes: Redux follows a strict unidirectional data flow, making it easier to predict and understand how the state will change in response to actions.
- Debugging Capabilities: Redux provides powerful debugging tools such as time-travel debugging, which allows you to step forward and backward through state changes, making it easier to identify and fix issues.