

Advanced ReactJS

Week 8

Overview

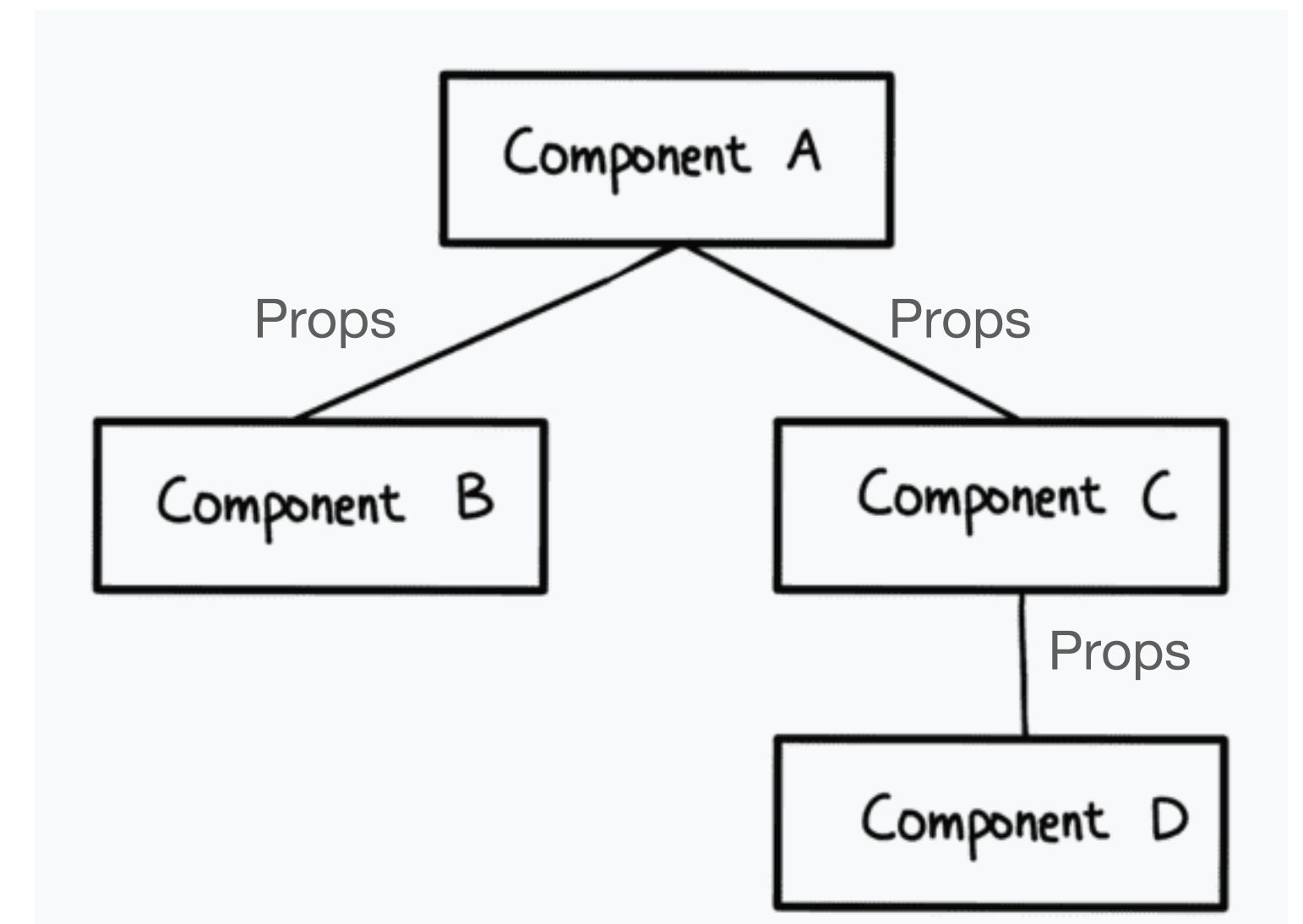
React Context API

Redux

Refresher - React Component Props

Props enable passing values from one component to another

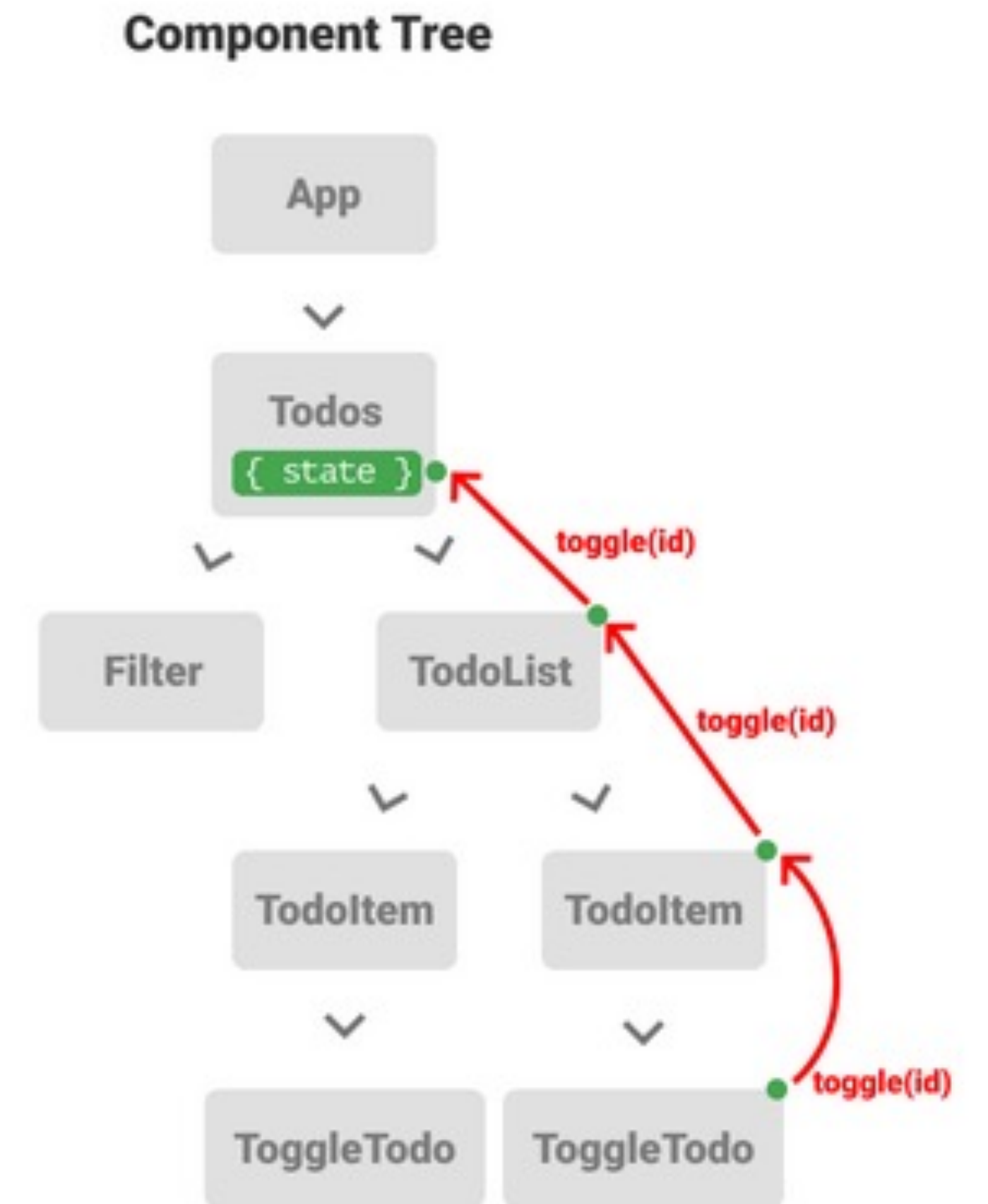
The Parent element can pass a prop and the child element can receive the prop using prop argument in the functional component



The Problem

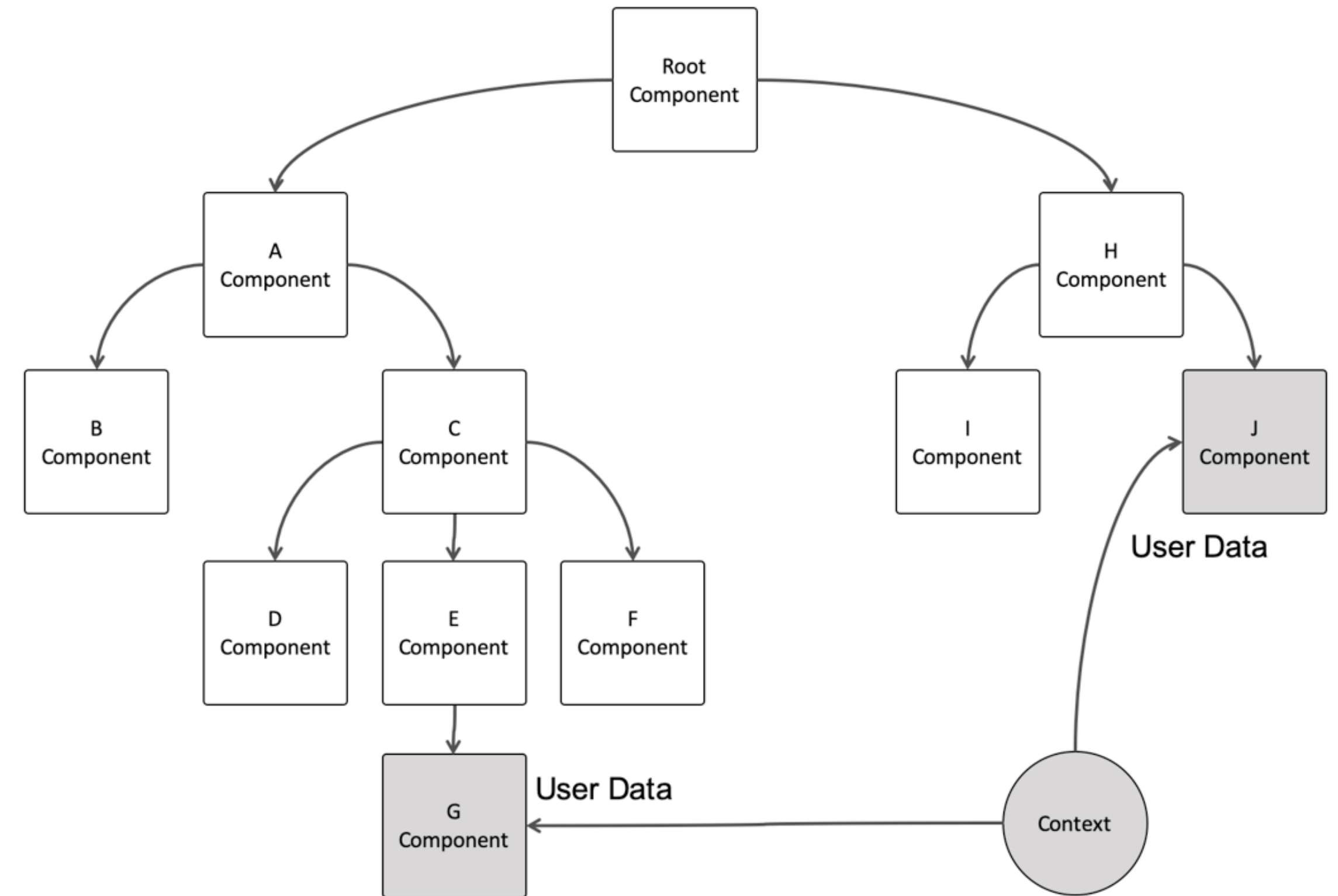
Prop Drilling

When working with highly nested component tree, passing props down the tree through every component can be tedious and may affect the app's performance



React Context API

Context provides a way to pass data through the component tree without having to pass props down manually at every level.



Using Context API in React

- Create Context
- Provide Context
- Consume/Use Context

Creating Context

Using 'createContext' import from React, create a Context Value

The value can also be updated later as a response to user inputs/API response etc



```
const MyContext = React.createContext(defaultValue);
```

Providing Context

To be able to use Context within your component tree, components must be wrapped with the Context Provider

Only components nested within the provider-wrapped component will have access to the Context



```
export default function App() {  
  return (  
    <MyContext.Provider value="some value">  
      <App />  
    </MyContext.Provider>  
  )  
}
```


Using Context

Context values can be accessed using a 'useContext' hook from React



```
function User() {  
  const value = React.useContext(MyContext);  
  
  // prints "some value"  
  return <h1>{value}</h1>;  
}
```

Pitfalls with Context

Everything that consumes a context re-renders everytime that context's state changes.

Redux

Redux is a JavaScript library to create and manage state containers outside the component tree

Redux is most commonly used with React to manage state

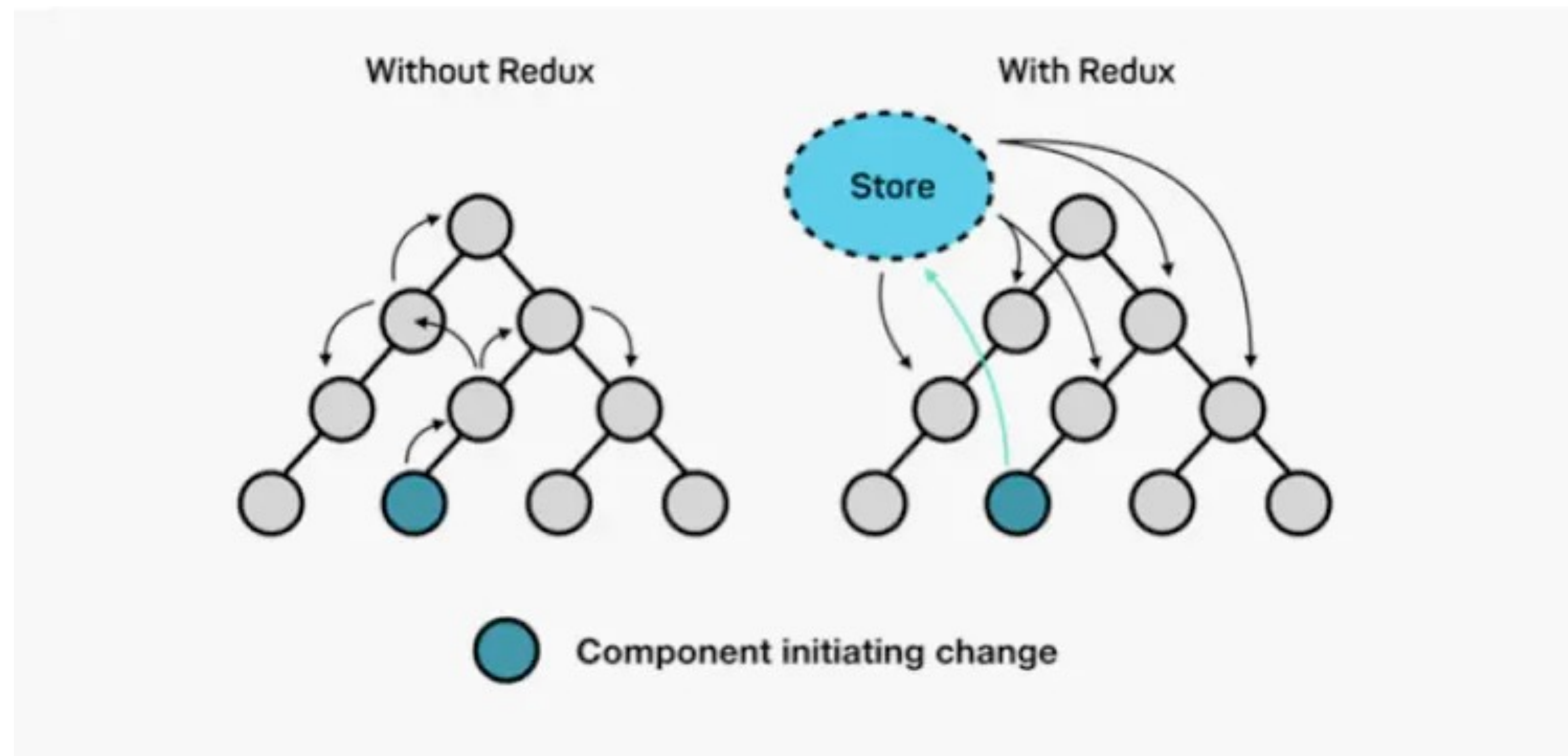
It is also developed by one of the React core team members and maintained by the same community



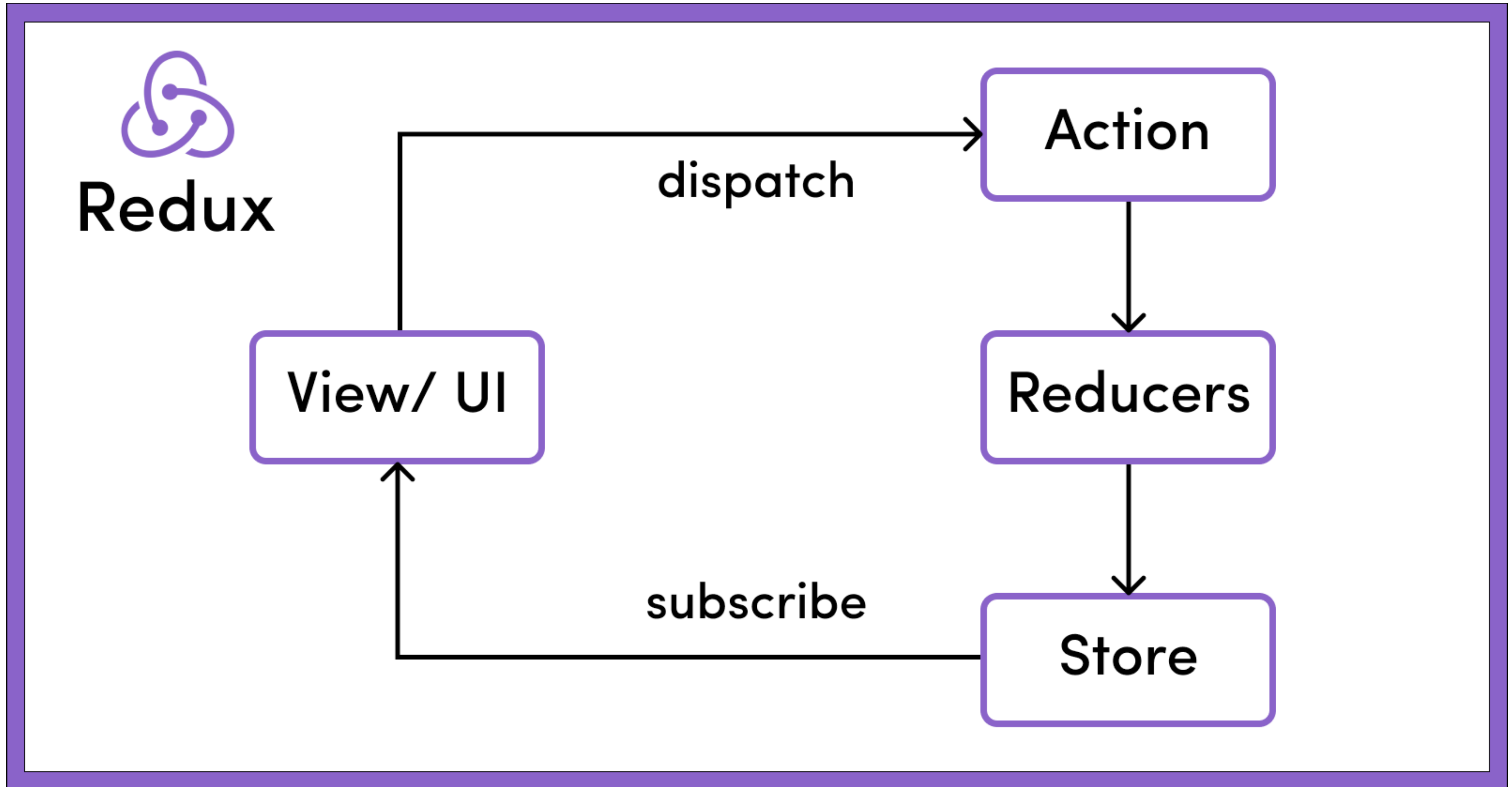
Why use Redux

Apart from enhanced performance over Context, State is predictable in Redux.

To change the state, we need to dispatch an action that are consumed by reducers



Redux Flow

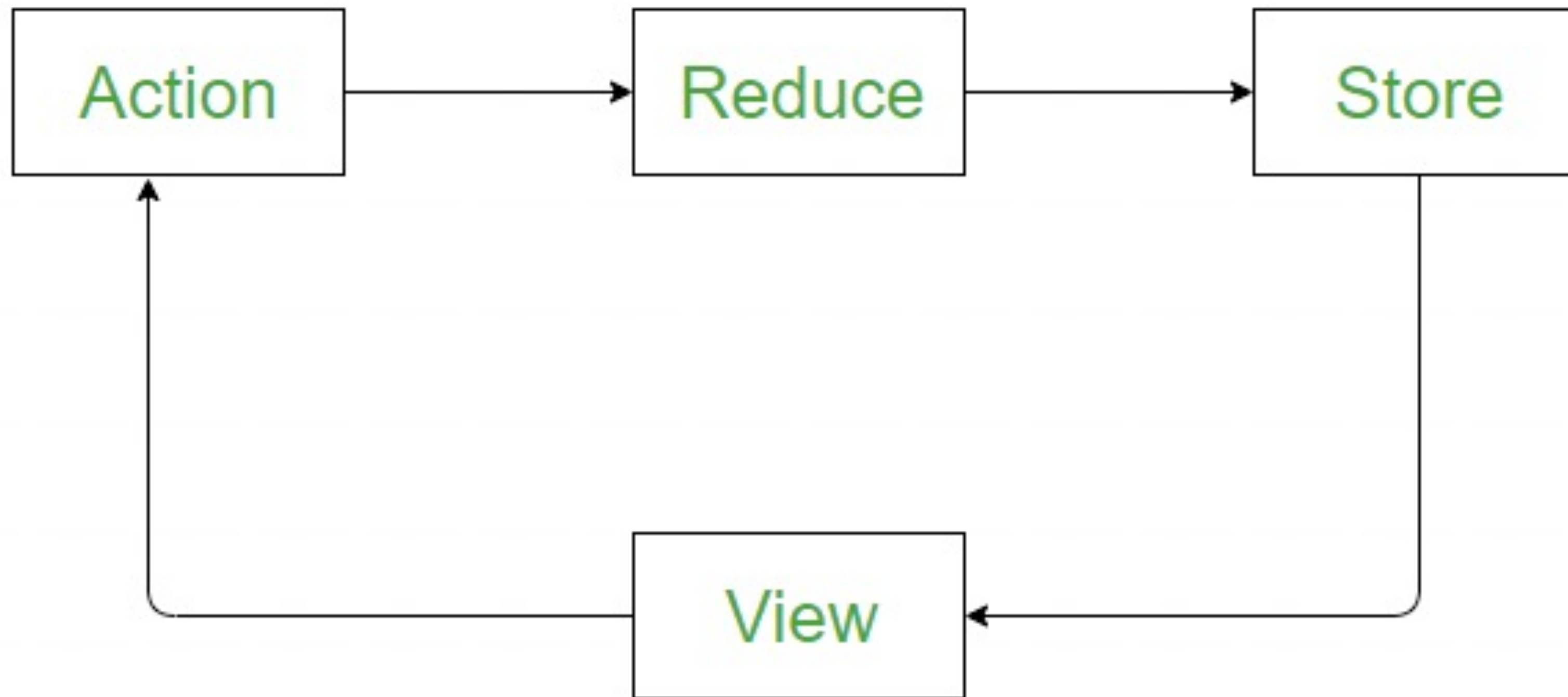


Action-Reducer Flow

Actions: Actions are a plain JavaScript object that contains information. Actions are the only source of information for the store. Actions have a type field that tells what kind of action to perform and all other fields contain information or data

Reducers: Since actions only tell what to do, but they don't tell how to do, so reducers are the pure functions that take the current state and action and return the new state and tell the store how to do.

Action-Reducer Flow



Redux Toolkit

Redux Toolkit (RTK) is an abstraction layer library built on top of Redux to make the Redux workflow easier

RTK is also the officially recommended way of writing Redux logic

```
npm install @reduxjs/toolkit react-redux
```


RTK Glossary

State	State is a value managed by the store
Store	A store is an object that holds the application's state tree. There should only be a single store in a Redux app
Slice	Slices are manageable ‘slices’ of state values and action-reducers
Action	An <i>action</i> is a plain object that represents an intention to change the state. Actions are the only way to get data into the store.
Reducer	Reducer is a function that performs changes to the state value in the store
Dispatch	Dispatcher is a function that accepts an action and performs the state change

RTK - Setup

configureStore

JS store.js

```
1 import { configureStore } from '@reduxjs/toolkit';
2
3 export default configureStore({
4   reducer: {
5   },
6 });
```


RTK - Setup

Provider

```
ReactDOM.render(  
  <Provider store={store}>  
    <App />  
  </Provider>,  
  document.getElementById('root')  
)
```

RTK - Setup

createSlice

```
JS counterSlice.js •   
1  import { createSlice } from '@reduxjs/toolkit'  
2  
3  export const counterSlice = createSlice({  
4    name: 'counter',  
5    initialState: {  
6      value: 0,  
7    },  
8    reducers: {  
9      increment: (state) => {  
10        // Redux Toolkit allows us to write "mutating" logic in reducers. It  
11        // doesn't actually mutate the state because it uses the immer library,  
12        // which detects changes to a "draft state" and produces a brand new  
13        // immutable state based off those changes  
14        state.value += 1  
15      },  
16      decrement: (state) => {  
17        state.value -= 1  
18      },  
19      incrementByAmount: (state, action) => {  
20        state.value += action.payload  
21      },  
22    },  
23  })  
24  
25  export const { increment, decrement, incrementByAmount } = counterSlice.actions  
26  export default counterSlice.reducer  
27
```

RTK - Setup

configureStore reducer

JS store.js x

```
1 import { configureStore } from '@reduxjs/toolkit';
2 import counterReducer from '../features/counter/counterSlice';
3
4 export default configureStore({
5   reducer: {
6     counter: counterReducer,
7   },
8 });
9
```

RTK - Use

useSelector, useDispatch

```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment } from './counterSlice'

export function Counter() {
  const count = useSelector((state) => state.counter.value)
  const dispatch = useDispatch()

  return (
    <div>
      <div>
        <button
          aria-label="Increment value"
          onClick={() => dispatch(increment())}
        >
          Increment
        </button>
        <span>{count}</span>
        <button
          aria-label="Decrement value"
          onClick={() => dispatch(decrement())}
        >
          Decrement
        </button>
      </div>
    </div>
  )
}
```

