**What is Redux?**

Redux is a **state management library** for JavaScript applications (often used with React).  
It helps manage the **global state** of your app in a predictable way.

Think of Redux as a **single central storage (store)** where all your app’s state lives.

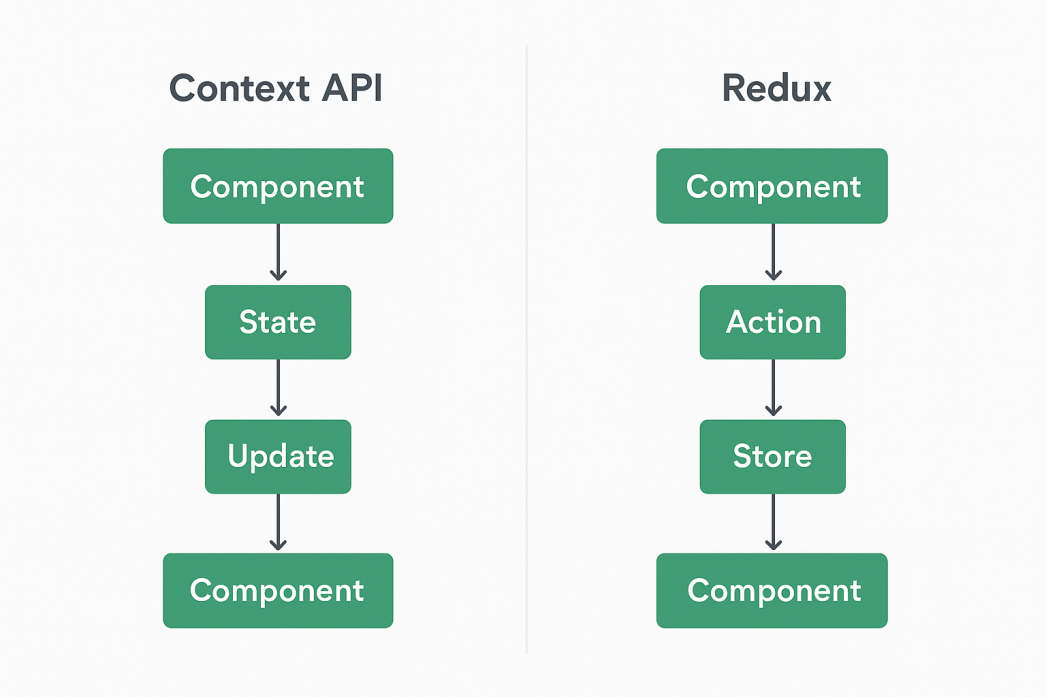
**Why Redux?**

**Without Redux:**

* Data is managed separately in each component.
* Passing props between components (prop drilling) becomes messy.
* Debugging state changes is difficult.

**With Redux:**

* All state is kept in **one store**.
* Components can access the store **anywhere** without prop drilling.
* State changes are predictable (via actions and reducers).



npm install @reduxjs/toolkit react-redux

// CounterContext.js

import React, { createContext, useState } from "react";

* **Imports** React so you can write JSX.
* **createContext**: A React function to create a Context object for sharing data.
* **useState:** React hook to create state inside a functional component.

export const CounterContext = createContext();

* Creates a new context called **CounterContext.**
* This is like creating an empty box that will hold your shared data.
* **export** makes it available to other files.

export const CounterProvider = ({ children }) => {

* Defines a component **CounterProvider** that will wrap other components and give them access to the shared state.
* **({ children }) means it will render whatever components are inside it when used.**

const [count, setCount] = useState(0);

* Creates a state variable count with an initial value of 0.
* **setCount** is the function to update it.

const increment = () => setCount((prev) => prev + 1);

  const decrement = () => setCount((prev) => prev - 1);

* **increment:** increases count by 1.
* **decrement:** decreases count by 1.
* **(prev) => prev + 1** ensures we use the latest value of count before updating.

return (

    <CounterContext.Provider value={{ count, increment, decrement }}>

      {children}

    </CounterContext.Provider>

  );

* **<CounterContext.Provider>:** Special component that provides the context value to all its child components.
* **value={{ count, increment, decrement** }}: The shared data and functions you want children to use.
* **{children}:** Renders whatever is wrapped inside **CounterProvider** in App.js.
* This **CounterProvider** is where the Context API is actually applied — it’s the point that makes **useContext(CounterContext)** in other components possible.

// App.js

import React, { useContext } from "react";

import { CounterProvider, CounterContext } from "./CounterContext";

* Importing **React** so **JSX** can work.
* Importing **useContext** hook (needed to read values from Context).
* Importing:
  + **CounterProvider** → the component that *provides* state & functions.
  + **CounterContext** → the context object we created earlier to share data.

const CounterDisplay = () => {

  const { count } = useContext(CounterContext);

  return <h1>Count: {count}</h1>;

};

* **CounterDisplay** is a small component.
* Inside it, we call **useContext(CounterContext)** → this pulls data from the Provider.
* From the Provider, we destructure **{ count }.**
* Displays the value in **<h1>.**

**Whenever the count changes, this component automatically re-renders with the new value.**

const CounterButtons = () => {

  const { increment, decrement } = useContext(CounterContext);

  return (

    <>

      <button onClick={increment}>+</button>

      <button onClick={decrement}>-</button>

    </>

  );

};

* **CounterButtons** is another component.
* Again, it uses **useContext(CounterContext)** → this time we take the functions **{ increment, decrement }.**
* Two buttons:
  + + calls **increment()** → increases the count.
  + - calls **decrement()** → decreases the count.

**Clicking a button updates the state in the Provider, which causes all subscribed components (like CounterDisplay**) **to update.**

export default function App() {

  return (

    <CounterProvider>

      <CounterDisplay />

      <CounterButtons />

    </CounterProvider>

  );

* **App** is the root component.
* It wraps everything in **CounterProvider.**
* This makes the context **(count, increment, decrement)** available to *all child components*.
* Inside it, we render:
  + **CounterDisplay** → shows the current count.
  + **CounterButtons** → provides controls to change the count.

**In summary:**

* **CounterProvider** = **the brain** (holds state + functions).
* **CounterContext** = **the pipe** (carries data/functions to components).
* **CounterDisplay** = **the eyes** (shows count).
* **CounterButtons** = **the hands** (change the count).

// store.js

import { configureStore, createSlice } from "@reduxjs/toolkit";

* Importing functions from **Redux** Toolkit (a simpler way to write Redux).
* **createSlice** → bundles state, reducers (logic), and actions in one place.
* **configureStore** → creates the Redux store (where all state lives).

const counterSlice = createSlice({

  name: "counter",

  initialState: { value: 0 },

  reducers: {

    increment: (state) => { state.value += 1; },

    decrement: (state) => { state.value -= 1; }

  }

});

* **createSlice** makes a mini "slice" of the Redux store for handling one feature: the counter.
* **name:** "counter" → slice name, used internally by **Redux**.
* **initialState: { value: 0 }** → sets the starting state of the counter (like useState(0)).
* **reducers** → functions that describe how the state changes when an action happens:
* **increment** → increases state.value by 1.
* **decrement** → decreases state.value by 1.

**Redux Toolkit uses Immer.js** under the hood, so you can write **state.value += 1** (it looks like mutation, but it’s safe & immutable internally).

export const { increment, decrement } = counterSlice.actions;

* **createSlic**e automatically generates action creators for each reducer (increment, decrement).
* These are simple functions you can **dispatch()** from components.
* Example: **dispatch(increment())** → triggers the increment reducer.

export const store = configureStore({

  reducer: { counter: counterSlice.reducer }

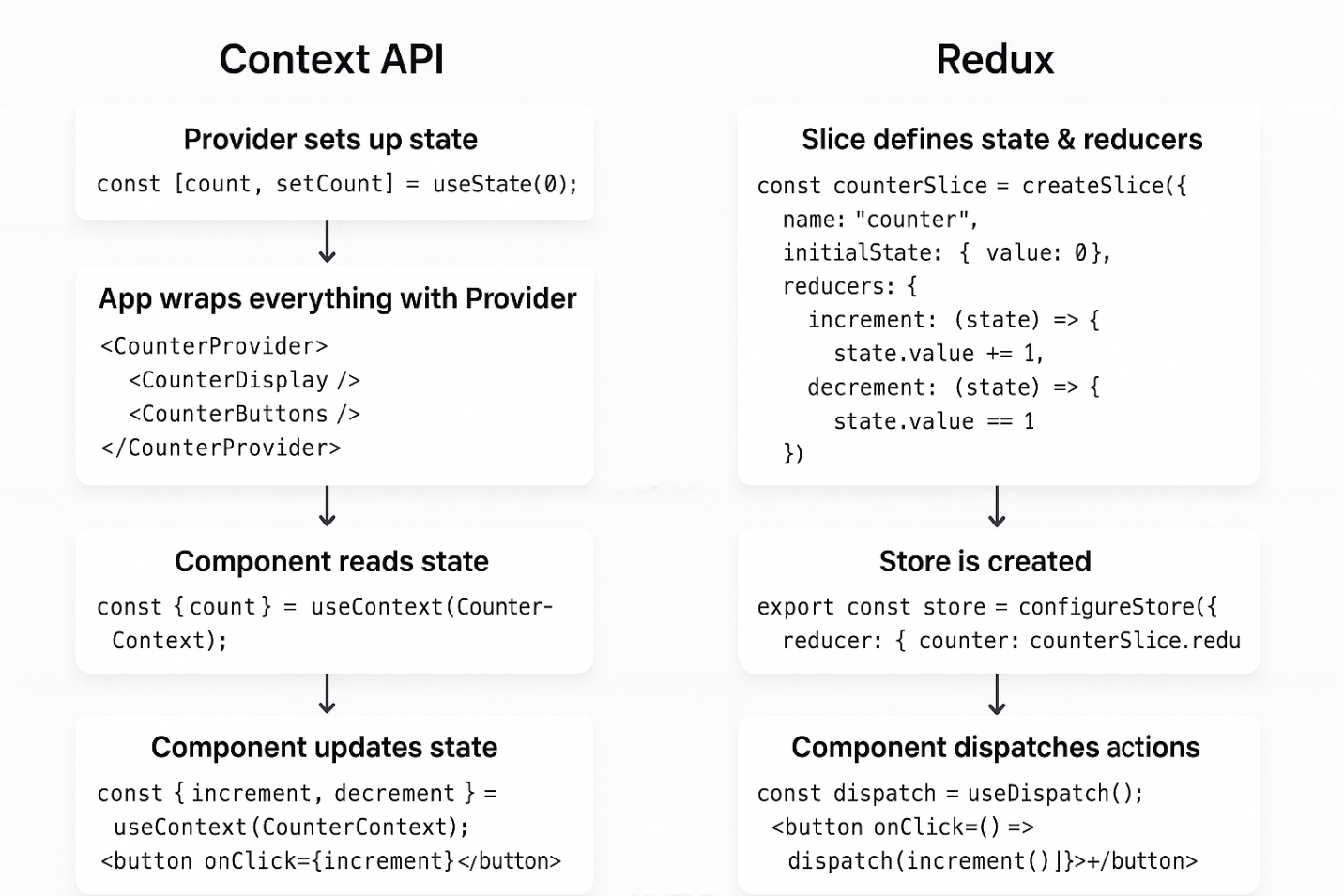
});

* Creates the Redux store.
* **reducer: { counter: counterSlice.reducer }** → adds our counter slice reducer under the key counter.
* Final state structure in the store will look like:

**{**

**counter: { value: 0 }**

**}**



**Summary:**

* **createSlice** = defines state + reducers + actions (all in one).
* **counterSlice.reducer** = updates the state.
* **counterSlice.actions** = action functions you dispatch from UI.
* **configureStore** = combines everything into a single store your app uses.