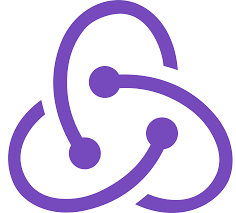
**REDUX**

-Redux is an open-source JavaScript library used to manage application state. React uses Redux for building the user interface. It was first introduced by **Dan Abramov** and **Andrew Clark** in **2015**.

|  |  |
| --- | --- |
| **Redux** | |
|  | |
| [**Original author(s)**](https://en.wikipedia.org/wiki/Programmer) | Dan Abramov and Andrew Clark |
| **Initial release** | June 2, 2015; 9 years ago[[1]](https://en.wikipedia.org/wiki/Redux_(JavaScript_library)#cite_note-githubreleasetag-1) |
| [**Stable release**](https://en.wikipedia.org/wiki/Software_release_life_cycle) | 5.0.1 / December 23, 2023; 11 months ago[[2]](https://en.wikipedia.org/wiki/Redux_(JavaScript_library)#cite_note-ghrelease-2) |
| [**Repository**](https://en.wikipedia.org/wiki/Repository_(version_control)) | * [github.com/reduxjs/redux](https://github.com/reduxjs/redux) [Edit this at Wikidata](https://www.wikidata.org/wiki/Q28957057#P1324) |
| **Written in** | [TypeScript](https://en.wikipedia.org/wiki/TypeScript) |
| [**Platform**](https://en.wikipedia.org/wiki/Computing_platform) | [Cross-platform software](https://en.wikipedia.org/wiki/Cross-platform_software) |
| [**Type**](https://en.wikipedia.org/wiki/Software_categories#Categorization_approaches) | [JavaScript library](https://en.wikipedia.org/wiki/JavaScript_library) |
| [**License**](https://en.wikipedia.org/wiki/Software_license) | [MIT License](https://en.wikipedia.org/wiki/MIT_License) |
| **Website** | [redux.js.org](http://redux.js.org/) |

-Redux is a JavaScript library that helps manage an application's state.

* **What it does**

Redux helps manage the data and variables that determine how an application behaves and what information it shows to the user.

* **Why it's useful**

Redux helps you write applications that:

* + Behave consistently
  + Run in different environments
  + Are easy to test
  + Are easier to understand, debug, and reset
* **How it works**

Redux centralizes an application's state and logic, which makes it easier to:

* + Implement undo/redo
  + Persist state
  + Make state management predictable
* **Features**

Redux includes:

* + A time traveling debugger
  + Live code editing
  + A large ecosystem of add-ons
  + Works with any UI layer

**HISTORY OF REDUX**

**A (Brief) History of Redux**

**2011: JS MVC Frameworks**[**​**](https://redux.js.org/understanding/history-and-design/history-of-redux#2011-js-mvc-frameworks)

Early JavaScript MVC frameworks like AngularJS, Ember, and Backbone had issues. AngularJS tried to enforce separation of "controllers" from templates, but nothing prevented you from writing <div onClick="$ctrl.some.deeply.nested.field = 123"> in a template. Meanwhile, Backbone was based on event emitters - Models, Collections, and Views were all each capable of emitting events. Models might emit a "change:firstName" event, and Views would subscribe to those. But, *any* code could subscribe to those events and run more logic, which could trigger *more* events

That made these frameworks very hard to debug and maintain. It was possible that updating one field in one model could trigger dozens of events and logic running around the app, or that any template could make changes to state at any time, which made it impossible to understand what would happen when you did a state update.

**2014: Flux**[**​**](https://redux.js.org/understanding/history-and-design/history-of-redux#2014-flux)

Back around 2012-2013, when React was first publicly released, Facebook had been using it internally for a couple years. One of the problems they ran into was that they had multiple independent pieces of their UI that needed access to the same data, like "how many unread notifications are there", but they found it hard to keep that logic straight when using Backbone-style code.

Facebook ultimately came up with a pattern called "Flux": create multiple singleton Stores, like a PostsStore and CommentsStore. Each of those Store instances would register with a Dispatcher, and the *only* way to trigger an update in a store was to call Dispatcher.dispatch({type: "somethingHappened"}). That plain object was called an "action". The idea was that all the state update logic would be semi-centralized - you couldn't just have any random part of the app mutate state, and all the state updates would be predictable.

Facebook announced this "Flux Architecture" concept around 2014, but didn't provide a full library that implemented that pattern. That led the React community to build *dozens* of Flux-inspired libraries with variations on the pattern.

**2015: The Birth of Redux**[**​**](https://redux.js.org/understanding/history-and-design/history-of-redux#2015-the-birth-of-redux)

In mid-2015, Dan Abramov began building yet another Flux-inspired library, called Redux. The idea was to demonstrate "time-travel debugging" for a [conference talk](https://youtu.be/xsSnOQynTHs?t=601). The library was designed to use the Flux pattern, but with some functional programming principles applied. Rather than Store *instances*, you could use predictable reducer functions that did immutable updates. This would allow jumping back and forth in time to see how the state looked at various points. It would also make the code more straightforward, testable, and understandable.

Redux came out in 2015, and quickly killed off all the other Flux-inspired libraries. It got early adoption from advanced developers in the React ecosystem, and by 2016, many people began to say that "if you're using React, you *must* be using Redux too". (Frankly, this led to a lot of people using Redux in places they didn't *need* to be using it!)

It's also worth noting that at the time, React only had its *legacy* Context API, which had was basically broken: it couldn't properly pass *updated* values down. So, it was possible to put event emitters into Context and subscribe to them, but you couldn't really use it for plain data. That meant that a lot of people began adopting Redux because it *was* a way to consistently pass updated values around the entire application.

Dan said early on that "Redux is not meant to be the *shortest* way to write code - it's meant to make it predictable and understandable". Part of that is about having a consistent pattern (state updates are done by reducers, so you always look at the reducer logic to see what the state values *can* be, what the possible actions are, and what updates they cause). It's also about moving logic *out* of the component tree, so that the UI mostly just says "this thing happened", and your components are simpler. Along with that, code that is written as "pure functions", like reducers and selectors, are more straightforward to understand: arguments in, result out, nothing else to look at. Finally, Redux's design enabled the Redux DevTools, which show you a readable list of all the actions that were dispatched, what the actions/state contained, and changes occurred for each action.

The early Redux patterns were especially boilerplate-heavy. It was common to have actions/todos.js, reducers/todos.js, and constants/todos.js, just to define a single action type ( const ADD\_TODO = "ADD\_TODO"), action creator function, and reducer case. You also had to hand-write immutable updates with spread operators, which were easy to mess up. People did fetch and cache server state in Redux, but it took a lot of manually-written code to write thunks to do the fetching, dispatch the actions with the fetched data, and manage the cache status in the reducers.

Redux became popular in *spite* of that boilerplate, but it was always the biggest point of concern.

**2017: Ecosystem Competition**[**​**](https://redux.js.org/understanding/history-and-design/history-of-redux#2017-ecosystem-competition)

By 2017-18, things had changed. A lot of the community was now focusing more on "data fetching and caching" rather than "client-side state management", and that's when we saw the rise of libraries like Apollo, React Query, SWR, and Urql for data fetching. At the same time, we also had the *new* React Context API came out, which does properly pass updated values down the component tree.

That meant that Redux wasn't nearly as "required" as it used to be - there were now other tools that solved many of the same problems, with varying amounts of overlap (and often with less code). The frequent complaints about "boilerplate" also caused a lot of concern from folks using Redux.

**2019: Redux Toolkit**[**​**](https://redux.js.org/understanding/history-and-design/history-of-redux#2019-redux-toolkit)

So, in 2019, we built and shipped Redux Toolkit as a simpler way to write the same Redux logic with less code. RTK is still "Redux" (single store, dispatching actions to trigger state updates done in reducers via immutable update logic), but with a simpler API and better built-in default behaviors. That also includes RTK Query, our built-in data fetching and caching library that was inspired by React Query and Apollo.

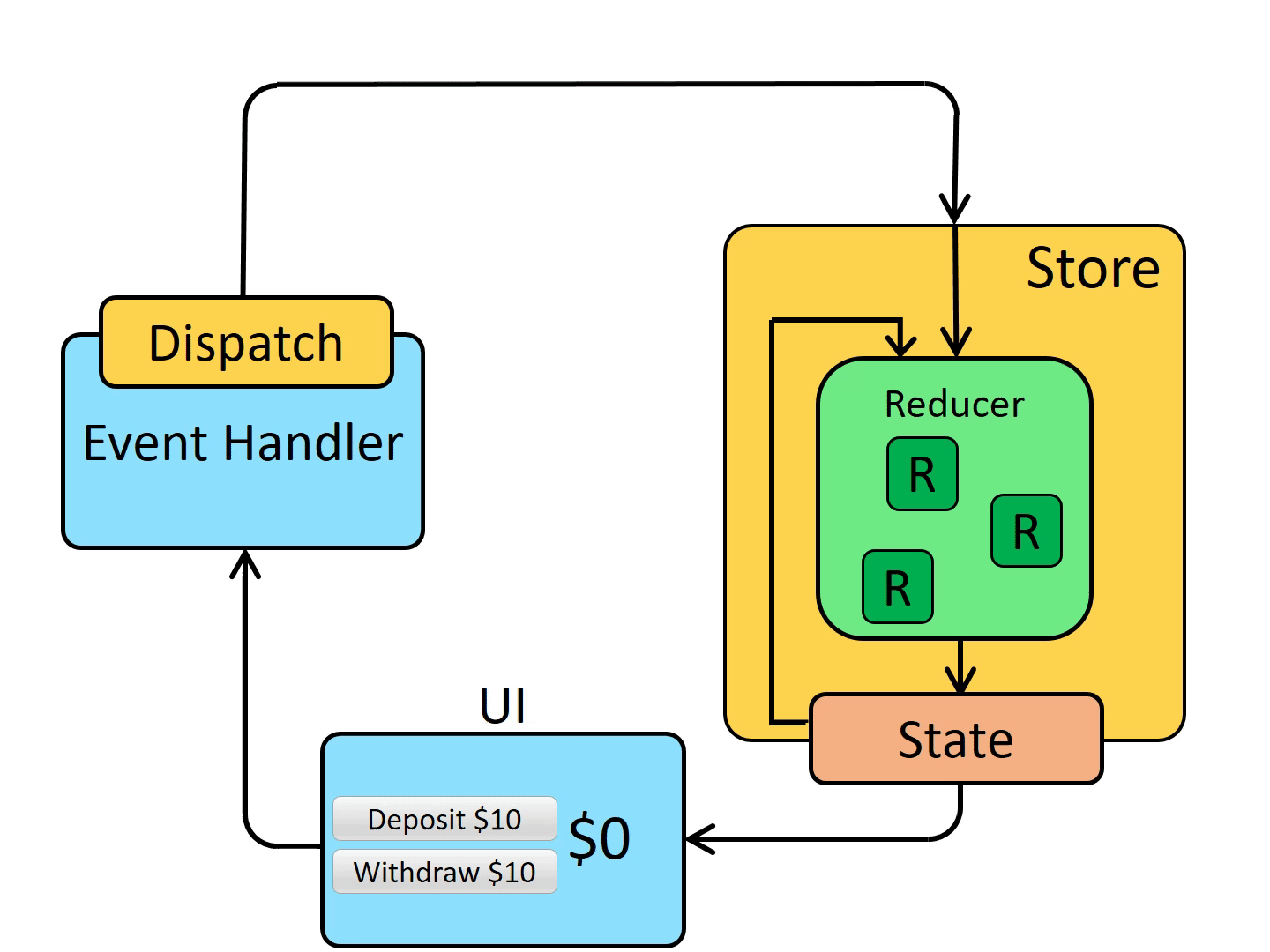
Today, [RTK is the standard way to write Redux logic](https://redux.js.org/introduction/why-rtk-is-redux-today). Like all tools, it has tradeoffs. RTK is probably going to be a bit more code to use than Zustand, but it also provides useful patterns for separating app logic from the UI. Redux isn't the right tool for every app, but it is still the most widely used state management lib with React apps, has excellent documentation, and provides a lot of features to help you build apps with a consistent and predictable structure.

**WHAT IS REDUX**

-Redux is Predictable Javascript State container for Javascript apps.

-Basically Redux is the container where you can store your whole application data. So we call it to state management.

-But Redux state and Component state are the different.



-Redux is a JavaScript library that manages an application's state, or the data and variables that determine the user's experience.

-It's used in both frontend and backend development, and is well-suited for applications with many components that share data.

Redux can help you:

* Write applications that behave consistently
* Run applications in different environments
* Test applications easily
* Understand how the application's state is modified
* Debug issues
* Reset the store to a known initial state

Redux's architecture includes:

* **Centralization**: Centralizes the application's state and logic, which enables features like undo/redo and state persistence
* **Debuggable**: The Redux DevTools make it easy to trace how, when, where, and why the application's state changed
* **Flexibility**: Works with any UI layer, and has a large ecosystem of add-ons

**1]Redux Is Used For Javascript Apps**

-

**2]Redux Is a State Container**

-Redux store the State of the application.

-

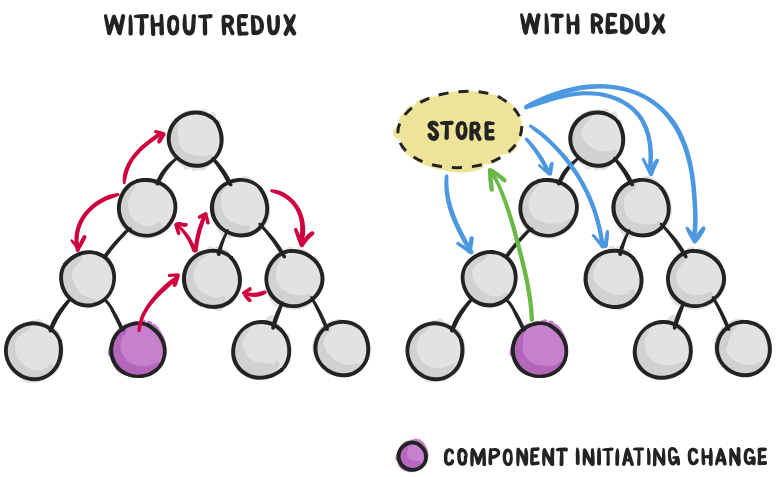
**3] Redux Is Predictable.**

-The state of an application can be changed.

-In Redux a Pattern is Enforced to ensure all State transition are explicit and can be tracked.

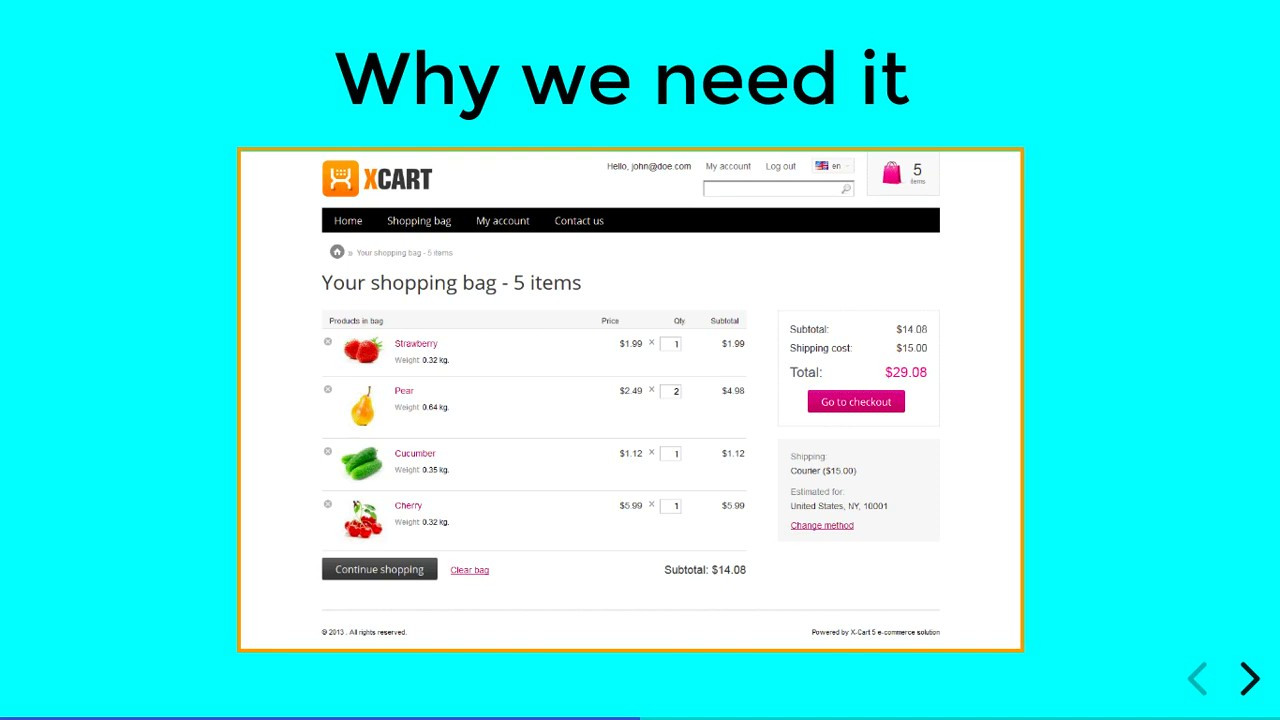
-The Changes to your application state become predictable.

**WHY IS REDUX**



-The main reason to use React Redux are:

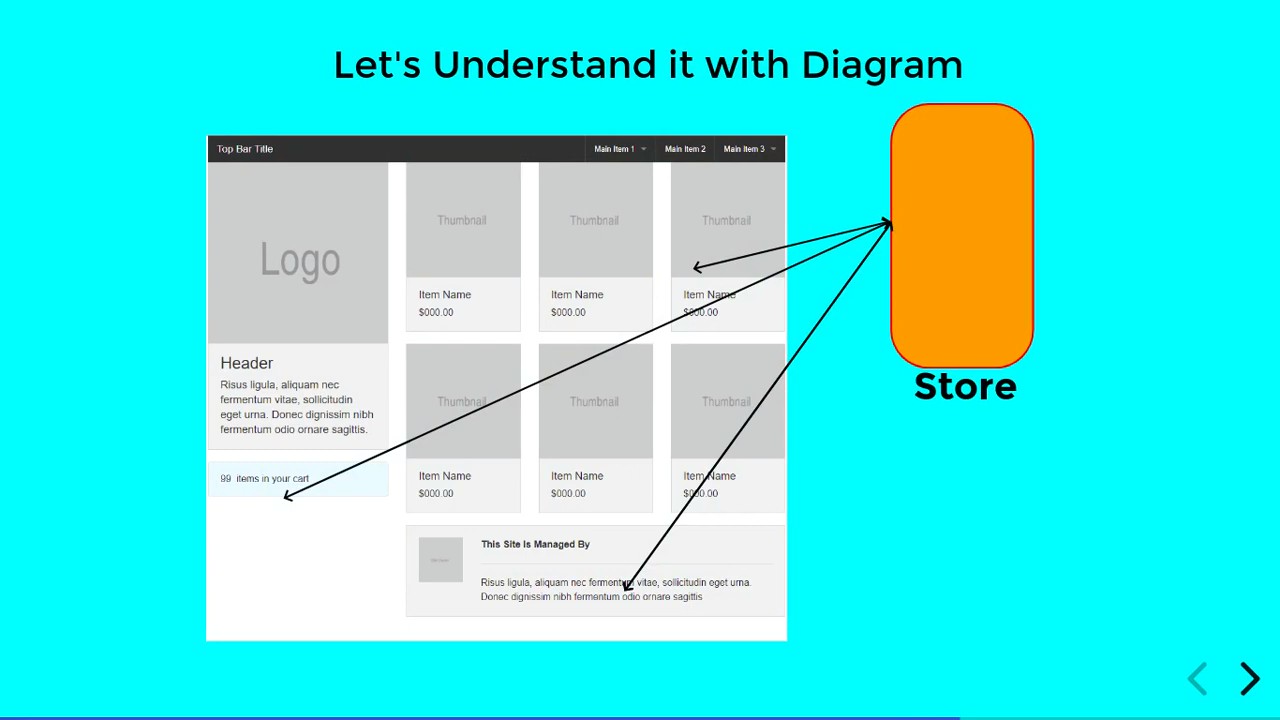
* When we build the application using ReactJS we breaks into components.
* So it is hard to communication one component to other component without Redux.
* React Redux is the official **UI bindings** for react Application. It is kept up-to-date with any API changes to ensure that your React components behave as expected.
* It encourages good 'React' architecture.
* It implements many performance optimizations internally, which allows to components re-render only when it actually needs.



**For Example:**

-In above diagram we are adding the component getting total in the cart.

-So using Redux it is easy to getting value from different component and add in cart which another component.



-Redux is a **predictable state management library** commonly used with JavaScript applications, especially those built with React. It helps manage the state of your application in a consistent and centralized way.

**Key Concepts in Redux:**

1. **Store**:
   * The centralized location where the entire state of the application is stored.
   * The store is immutable, meaning you cannot directly change the state.
2. **Actions**:
   * Plain JavaScript objects that describe what happened in the application.
   * They have a type property (a string constant) and may include additional data (payload).
3. **Reducers**:
   * Pure functions that determine how the state should change based on an action.
   * They take the current state and an action as arguments and return a new state.
4. **Dispatch**:
   * A method used to send actions to the store.
5. **Selectors**:
   * Functions used to extract specific parts of the state from the store.
6. **Middleware** (Optional):
   * Used to extend Redux's capabilities, such as handling asynchronous actions (e.g., with redux-thunk or redux-saga).

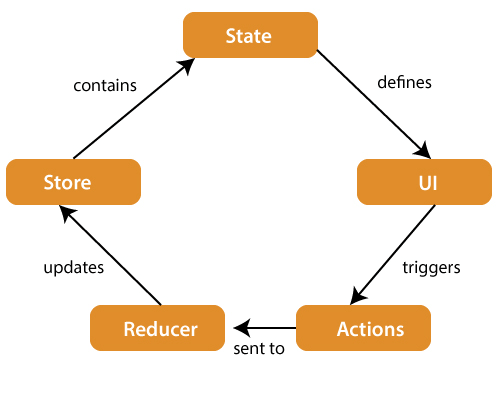
**Redux Workflow:**

1. An **action** is dispatched when an event occurs (e.g., user clicks a button).
2. The **reducer** listens for the action and returns a new state based on it.
3. The **store** is updated with the new state.
4. React components subscribed to the store are re-rendered with the updated state.

**Benefits of Redux:**

* **Predictable State**: The state is predictable due to pure reducers.
* **Centralized State Management**: All application state is in one place, making it easier to debug and test.
* **Easier Debugging**: Tools like Redux DevTools help track state changes over time.
* **Scalability**: Suitable for large-scale applications where state sharing is complex.

**REDUX ARCHITECTURE**



The components of Redux architecture are explained below.

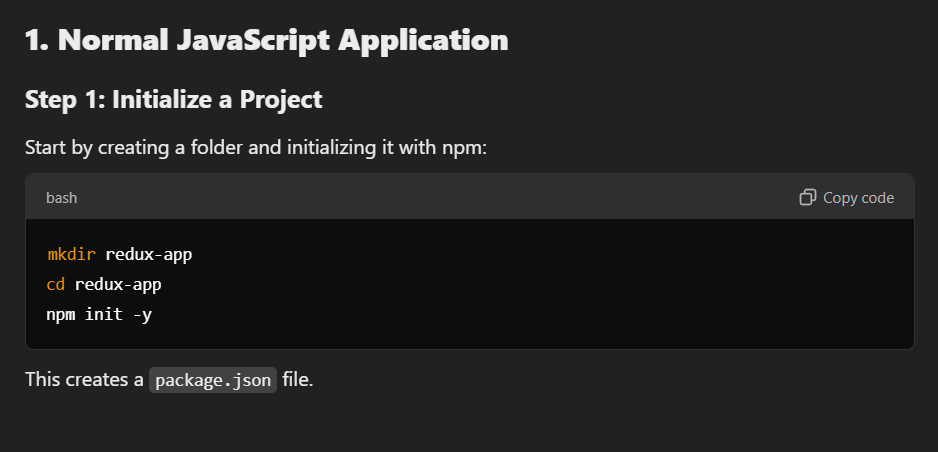
**STORE:** A Store is a place where the entire state of your application lists. It manages the status of the application and has a dispatch(action) function. It is like a brain responsible for all moving parts in Redux.

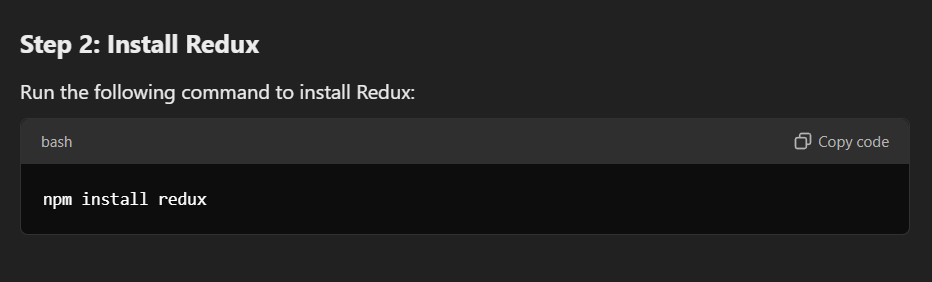
**ACTION:** Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.

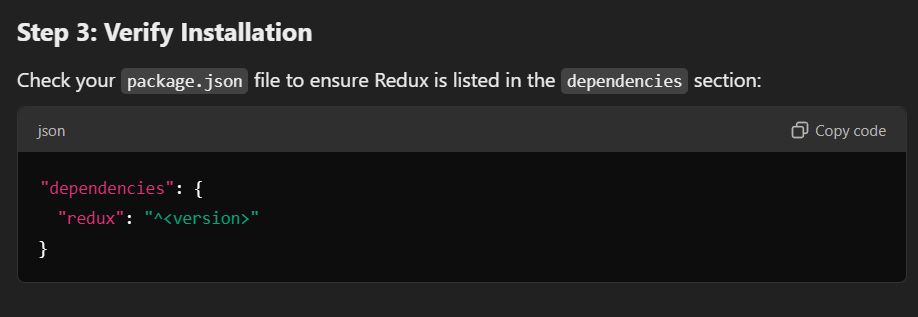
**REDUCER:** Reducer read the payloads from the actions and then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

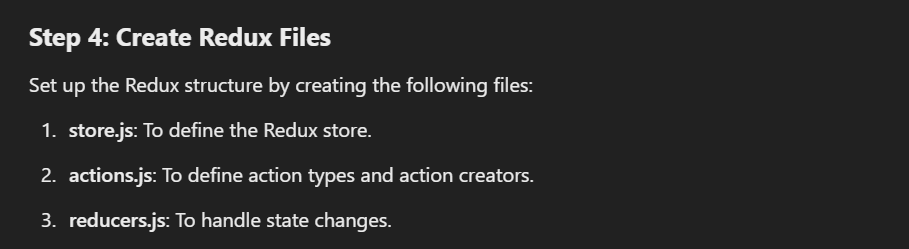
**REDUX INSTALLATION**

-Before installing Redux, **we have to install Nodejs and NPM**. Below are the instructions that will help you install it. You can skip these steps if you already have Nodejs and NPM installed in your device.







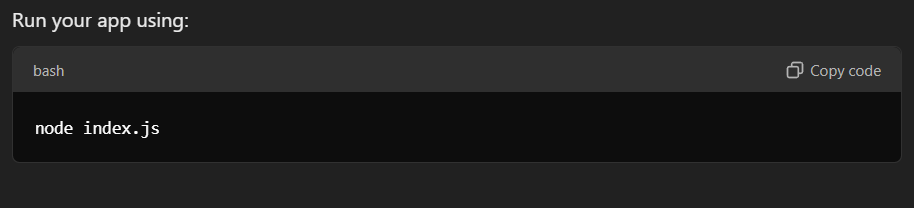


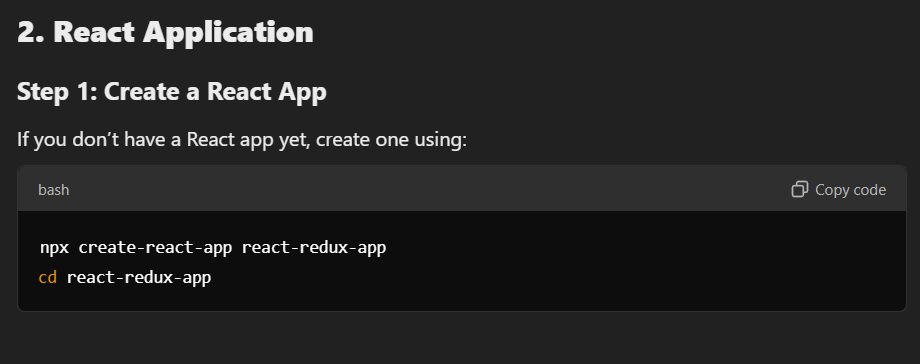




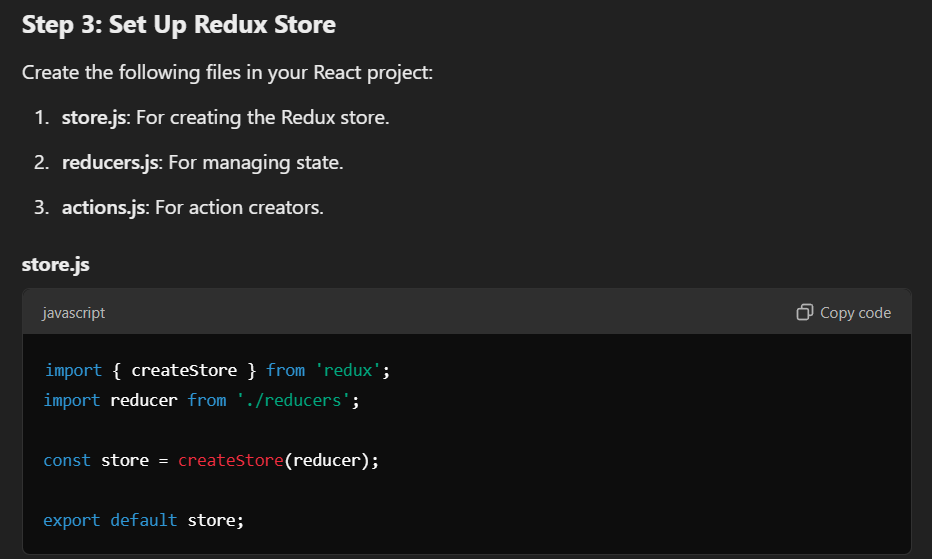




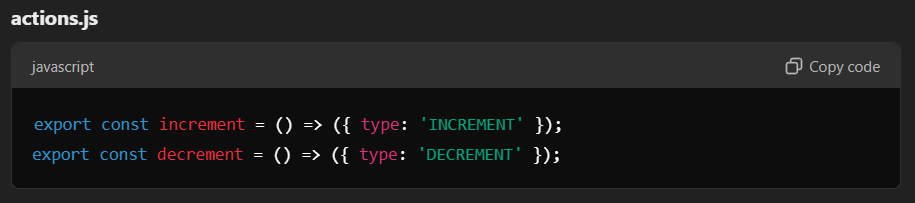


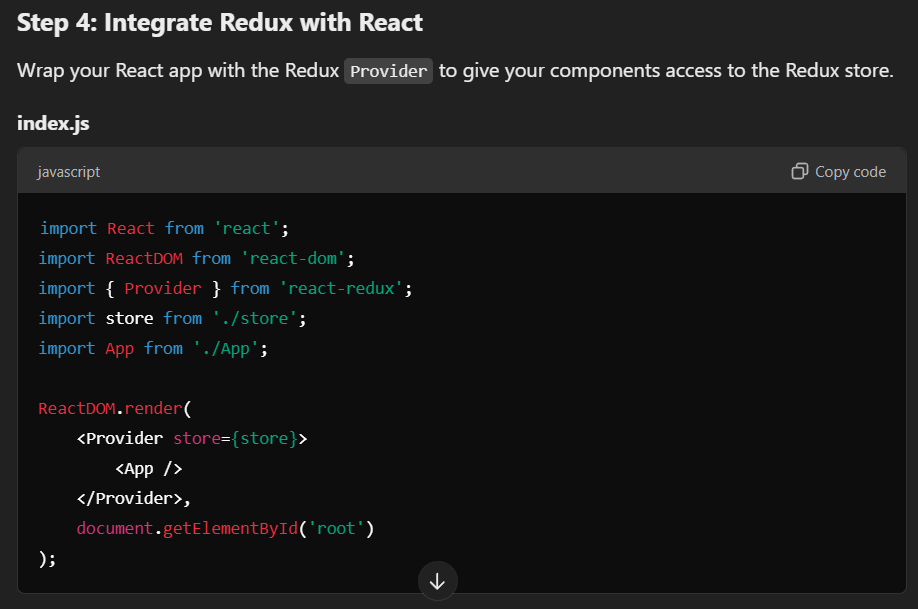


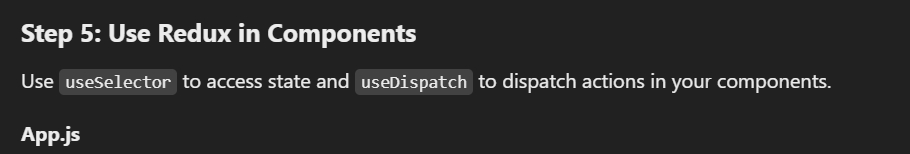


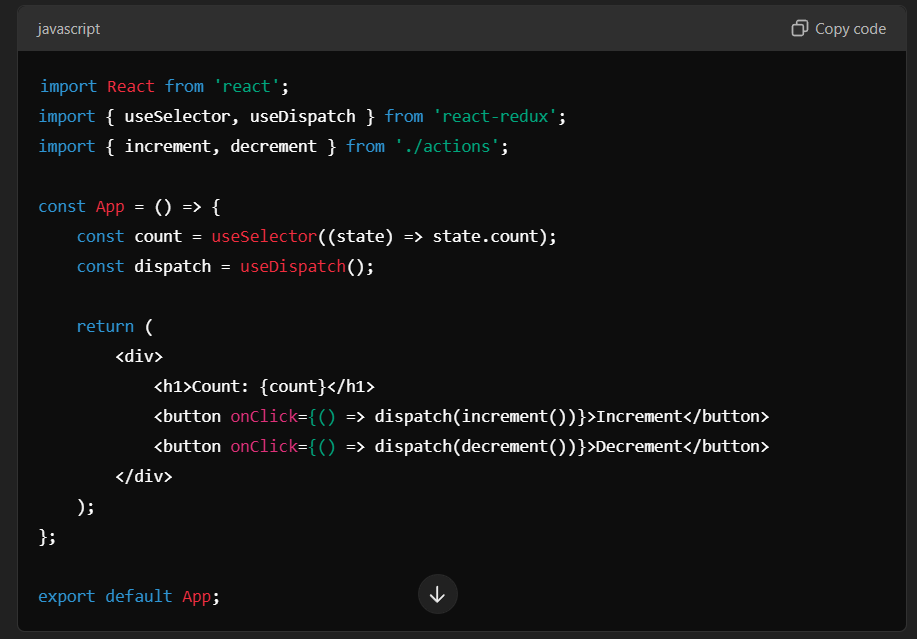










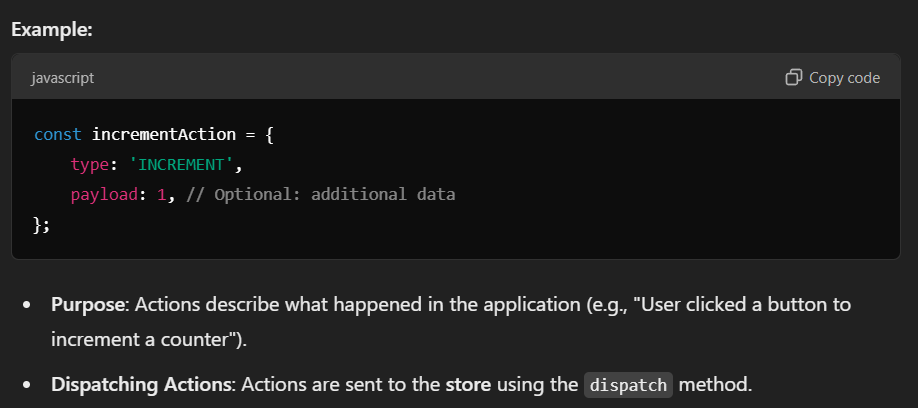


**THREE CORE CONCEPTS**

-In Redux, **Action**, **Reducer**, and **Store** are the core building blocks that work together to manage the application's state in a predictable way. Here's a breakdown of each:

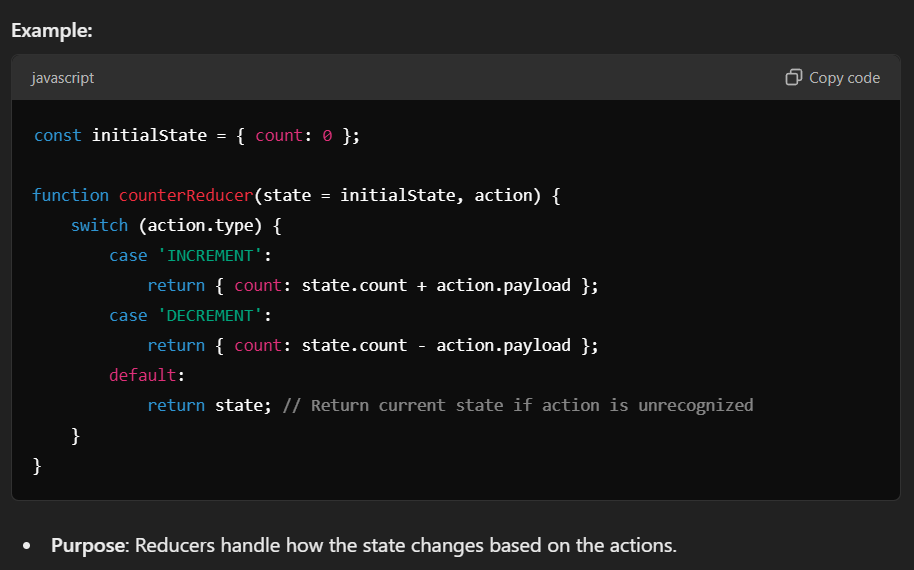
**1. Action:**

* An **action** is a plain JavaScript object that describes an event or intention to change the state.
* It must have a type property (a string that identifies the action) and may include additional data (called a "payload") relevant to the action.
* An action that describes the changes in the state of the applicatiob.



**2. Reducer:**

* A **reducer** is a pure function that determines how the application's state should change in response to an action.
* A reducer that describes the changes in the state of the application.
* It takes two arguments:
  1. The **current state**.
  2. The **action** object.
* It returns the **new state**.



**3. Store:**

* The **store** is the central place where the entire state of the application is stored.
* It is created using the createStore function in Redux or configureStore in Redux Toolkit.
* A store that holds the state of your application.

**Responsibilities of the Store:**

1. **Hold State**: The store contains the application's current state.
2. **Dispatch Actions**: The store allows you to dispatch actions using store.dispatch(action).
3. **Subscribe**: Components can subscribe to the store to get updates when the state changes.
4. **Expose State**: Access the current state using store.getState().



