**Redux**

**Redux** is a **state management library** used mostly with **any** **JavaScript frameworks like React, Angular, Vue or even vanilla JS and also in** Plain JavaScript (Node.js). It helps manage and centralize the **application state** in a predictable way.

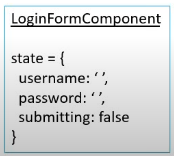
**💡 Simple Definition:**

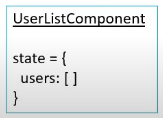
**Redux** is like a **central data store** for your app. Instead of components managing their own data separately, Redux stores all the data in **one place** and updates it in a controlled, predictable way.

### ✅ ****State ante enti?****

* **State** **of component** ante component lo unde **data** (samacharam).

Example:

* + If we have a LoginForm component → then **state of component** is simple an **object** with **username**, **password**, **submitting** properties  
    



Usually ee data ni component lo useState tho manage chestam. Kaani, pedda apps lo **chaala components ki common data kavali**. Appudu confusion avutundi.

* **State of an app is the state represented by all the individual components of that app. This includes the data and UI logic.**



Redux will store and manage the **application state**.

✅ **Predictable in what way?**

* In any JS application, the state of application can change.

**Ex:** In **TodoList** application, a **Todo item** can go from being in **a** **state of pending** to **a state of completed.**

* **In Redux all such state transitions are explicit and it is possible to keep track of them. In other words changes to your application’s state become predictable.**

**👩‍💻 Example:**

👍 Normal JS lo (**Implicit**):

****Ikkada todo object lo state (completed) maarindi, **kani ekkada enduku maarindhi ani manaku clarity ledu** (not traceable easily).

👍 Redux lo (**explicit**):

****

Ikkada state change ki **oka action type** untundi: "MARK\_COMPLETED".

Ah action dispatch aindi anedi clear ga telustundi.

**DevTools lo history track cheyyachu:**

PENDING → MARK\_COMPLETED → COMPLETED

**Hence, Redux** is **predictable state container.**

**Redux will help you, if you want to manage the state of your application in predictable way.**

✅ **React is UI library, Redux is state management library.**

**They both work independently of each other. Which means, the core redux library is not tied to React or any other frontend framework like Angular, Vue etc.**

**Hence, to directly use Redux in React application is bit difficult and confusing.  
This is the reason Redux has a separate react-redux package specifically for React application.**

**For React + Redux application, you should install:**

npm install redux react-redux

redux – The core Redux state management library

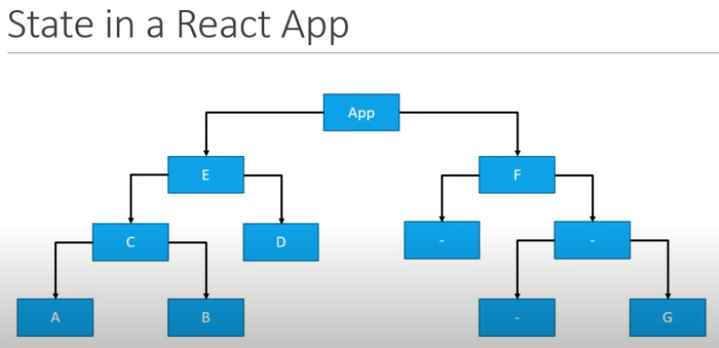
react-redux – The official react bindings for Redux. Which means, it provides useful tools/functions which helps you bind **React** and **Redux** together in React application.

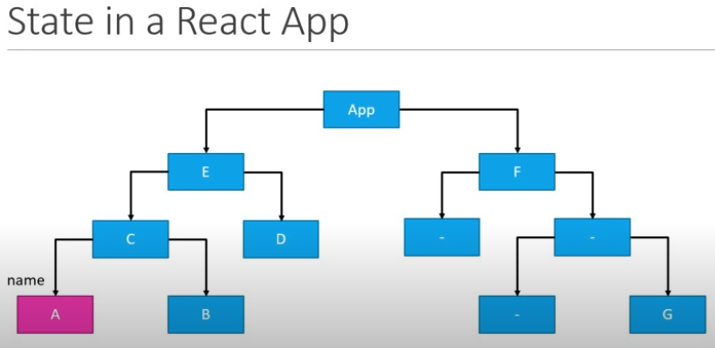
### 🎯 ****Why would we want to use Redux in React application? When React Components have their****

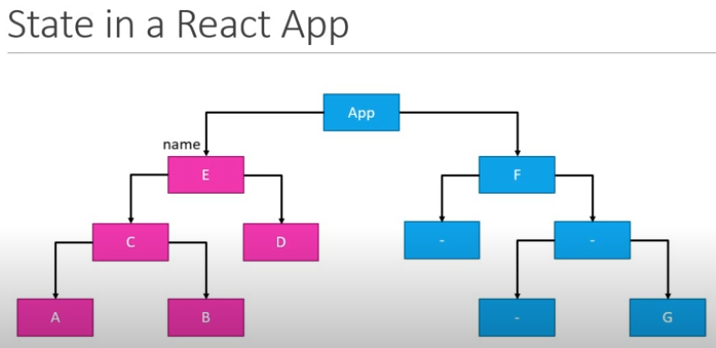
### ****own state then why do we need another tool to help manage that state.****

**In medium to large apps:**

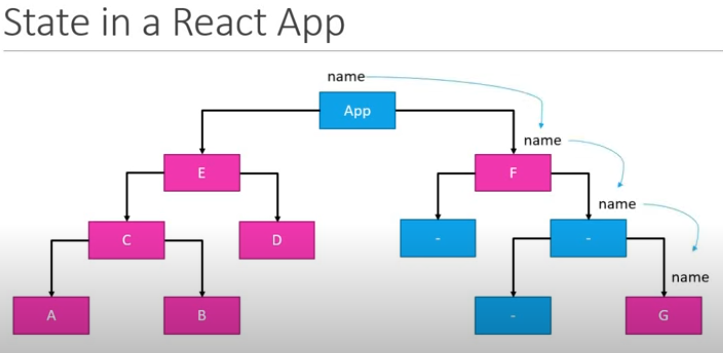
* Data needs to be shared between components. React lo components chala levels lo untay (parent → child → child…). Data ni okka component nunchi inko component ki pampadam kastam avutundi.
* Components might become messy with too much state.
* Updating and debugging becomes harder.

  
**Fig:** React application with nested components.

  
**Fig:** Assume Component **A**  has an input field to accept the user’s name, which is stored locally within the **component State.**

****

**Fig:** Suppose the siblings components **B**, **D** needs to display the user’s name, we have to lift the component **A** state to component **C** and then tocomponent **E** in a React’s `**props**` context way.

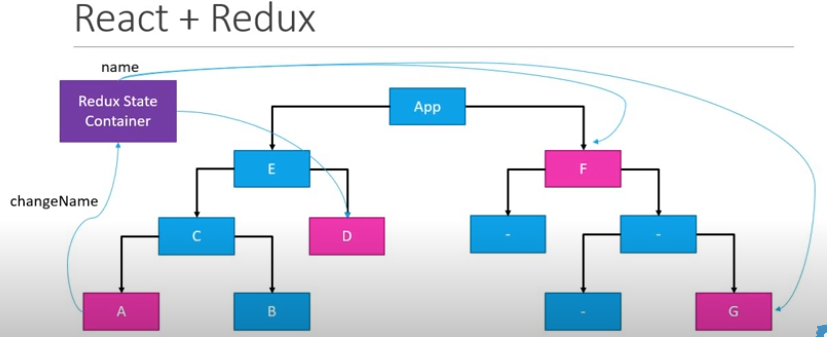


**Fig:** Parent to child.

**NOTE:** Username avasarm leni sibling (parent/child) component kuba still have to aware of username to pass the value to its siblings.

**Redux solves this** by giving:

* **Centralized State,** Redux valla data **okkate place lo** untundi (central ga store avutundi). Andaru akkadinunchi teesukovachu, update cheyyavachu.



* **Predictable State Updates**
* **Time-travel Debugging (via Redux DevTools)**

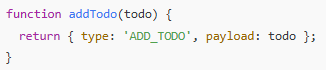
**Three core concepts of Redux – Store, Action, and Reducer:**

1. The **store** is a **JavaScript object** that holds the **entire state** of your application. It’s created using createStore() function from the **Redux** library.  
   
2. An **action** is a **plain JavaScript object** that describes **changes happened** in the state of app. It must have a type property defined as string constants.

****

You can also add a payload for passing data:  
****

Functions that return action objects which you define in code:

****

### ✅ Purpose:

* Tell the store **what to do.**
* Dispatched using store.dispatch(action)

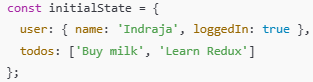
1. A **reducer** is a **pure user defined function** that **takes in** the **current state** and an **action**, and **returns a new state**.

## 🔄 How They Work Together:

1. You call store.dispatch(action)
2. Redux calls the **reducer** with the current state and the action
3. The reducer returns a **new state**
4. The **store** updates with this new state

**Three principles:**

Redux is built on **three core principles** that make state management predictable and manageable, especially for large applications.

1. **1st principle - Redux tells to store/maintain your entire application state in a single JS object which would be managed by the Redux Store.  
   **
2. **2nd principle: State is Read-Only -** The **only way to change the state** is to **dispatch/emit an action**, which is an object describing what happened. It must have a type property defined as string constants.

**To update the state of app, you need to let Redux know about that with an action.  
**

You are not allowed to directly update the state object.  


1. **3rd principle: Changes are Made with Pure Functions (Reducers)** - To specify how the state tree is transformed by actions, you write **pure reducers**.

**Redux Store:**

The **Redux store** is the **central place** where your entire application state lives.

**One store for the entire application  
Holds application state.**

****

**Following are the functions that Redux store has:**

**getState()** is a method provided by the **Redux store** that lets you access the **current state** of the application at any time.

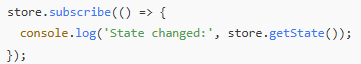


dispatch(action) is a method provided by the **Redux store** and is used to **send an action to the Redux store**.



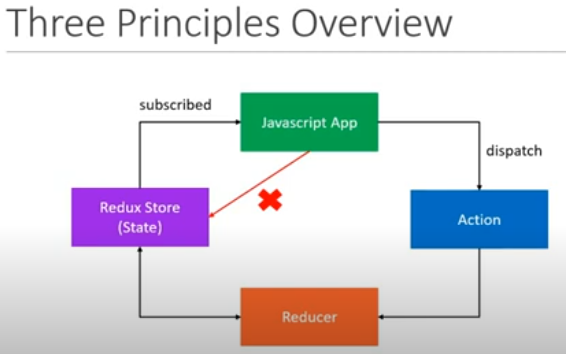
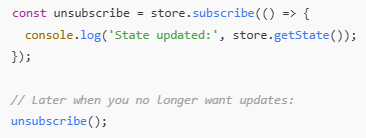
Redux forwards this action to the **reducer.** The reducer returns a **new state.** The store updates with that new state.

subscribe(listener) - This **registers a listener function** that gets called **every time the store's state changes**. It’s useful when you want to run some code **whenever the state changes** (like updating UI or logging)



Here, Inside the callback, you're logging the updated state using store.getState().

unsubscribe() is a function that **stops your listener** from getting called when the Redux store state changes. It is the method returned by the store.subscribe() method.

****

**Fig: How Redux principles work with React**

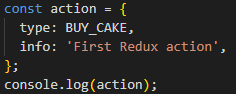
* 1. **The state of react app is maintained separately in Redux store.**
  2. **Our React app always subscribe to this Redux store which holds the React App state.**
  3. **App can not directly update its state stored in Redux store.**
  4. **If App wants to update its state, it has to emit/dispatch an action.**
  5. **Once an action has been dispatched, the reducer then handles that action and updates the current state stored in Redux store.**
  6. **As soon as the state is updated, the value is then passed on to the application because the App is subscribed to the store.**

**Project Creation:**

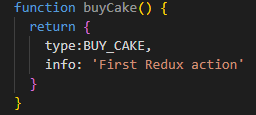
**Node.js project: You either find in local system or Git repo**

1. Prerequisites – Node.js
2. Assume we are building a Cake Shop Application
3. npm init -y – creates package.json
4. npm install react
5. Create an **action** for cake shop application in index.js:

  
Defining a **string constant** that indicates the **type of the action**.



Defining an **action**, which is an object that has `**type**` property.

**Recommended way to create an action** – defining a function which returns an action:  


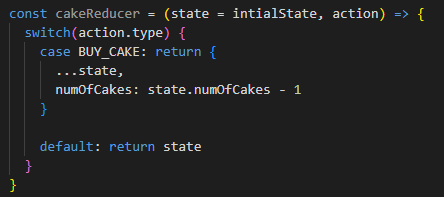
**Here,**

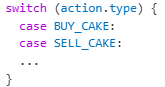
✅ type - Required in **every Redux action**. It tells the reducer **what kind of action** is being dispatched. Reducers use it to decide how to update the state.

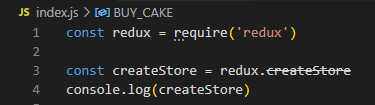
📝 info - Optional. It’s just **extra metadata or comments** for developers or for debugging/logging. It is **not used by the reducer**, unless you specifically write logic to use it.

1. Now defining reducer function which tells Redux **how to update the state** based on the action received.  
   **Syntax:**

  
**What our cake shop application state looks like?**  
As I shopkeeper, all I want to keep track of the number of cakes on the shelf.   
Hence, our app state is to be a simple numeric. **But remember** according to Redux principle, app state has to be represented by a single object.  


  
Let's break this **reducer** function **line by line** to clearly understand:

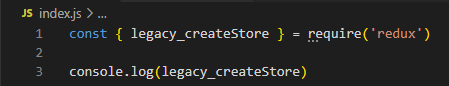
1. **cakeReducer** – constant that holds the Redux reducer function
2. **state** - Current state of that part of the Redux store.
3. **action** - Object that was dispatched, e.g. { type: 'BUY\_CAKE' }.
4. **switch** - A control structure (like multiple if...else if) used to compare action.type against many possible action types.  
     
   **action.type -** A string that describes what the action wants to do.   
   Example: 'BUY\_CAKE', 'ADD\_TODO', 'LOGOUT\_USER'.  
   If the action’s type is 'BUY\_CAKE', then this block will execute. We return a **new state object** here.
5. return { ...state, numCakes: state.numCakes - 1 }  
   ...state - Copies everything from the previous state.  
   numCakes: state.numCakes – 1 - Overrides numOfCakes with one less cake.  
   **NOTE:** ✅ Redux always expects a **new state object**, not a mutated one.
6. default: return state  
   If the action’s type doesn't match any case, we **must return the current state as-is**.  
   This prevents Redux from losing the state when unrelated actions are dispatched.
7. **Create redux store** to store/hold the application state.  
   We already define const variable called **initalState** which sets the application state.  
   **Now,** create redux store which will hold **initialState** value.  
   Hence we required to import `**redux**` library which provides a method called **createStore()** for creating the store.





**NOTE: createStore is deprecated.**

**Instead import,**

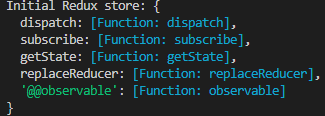




Here **legacy\_createStore()** takes **reducer** method as input.



**Output:**



**NOTE:**

1. **Redux in plain Node.js environment:**

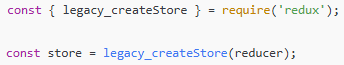
**createStore is deprecated** because the createStore method from the redux package is now considered **legacy**. The Redux team strongly recommends using configureStore from @reduxjs/toolkit (RTK), which simplifies store setup and helps avoid common mistakes.



* + `redux` code package come with this Redux tool kit, You should not need to be using the redux core package by itself today, except for learning purposes.

****

* For more details, please read this Redux docs page: <https://redux.js.org/introduction/why-rtk-is-redux-today>
* The createStore method from the core redux package **will not be removed** - just marked as deprecated, but we encourage all users to migrate to using **Redux Toolkit** for all Redux code.
* If you want to keep using createStore (for learning/testing in Node.js), update your import like this:

****

This way, the deprecation warning will not show up in editors like VSCode.

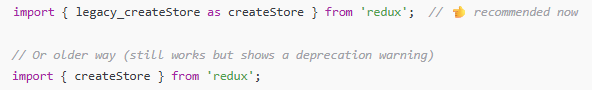
1. **React+Redux:**

****

The redux library **only provides named exports**, like createStore, combineReducers, etc.  
'redux' does not contain a default export or it **does not export a default object**.

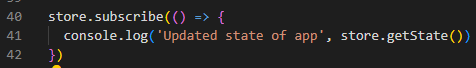
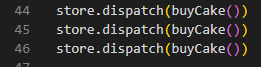
✅ Correct Way to Import for **React+Redux** application:

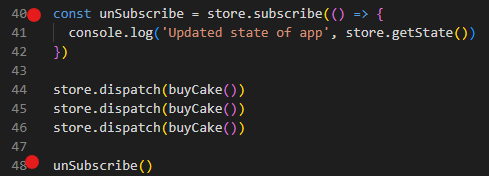
Instead of importing the whole **redux** module as an object, just import what you need directly:

****

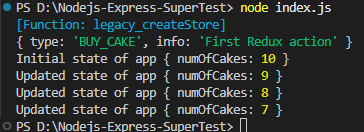
✅ But, If you're working with React or a modern frontend stack, Redux team **recommends using Redux Toolkit (RTK)** instead.

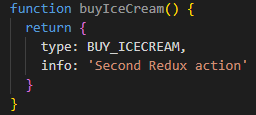
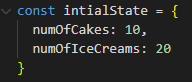
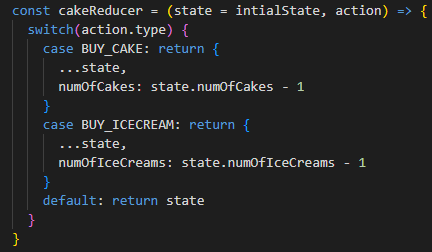


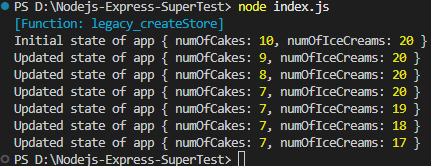
1. Create Redux store has a **getState()** method which lets us access current state of application at any time.  
   
2. Created Redux store has subscribe(listener) method which lets you **listen for state changes** in the Redux store. It’s useful when you want to run some code **whenever the state changes** (like updating UI or logging)  
   
3. Created Redux store has dispatch(action) method which is used to **send an action to the Redux store**.  
   
4. unsubscribe() is a function that **stops your listener** from getting called when the Redux store state changes. It is the method returned by the store.subscribe() method.



1. when I run **index.js**, the output which I got is:

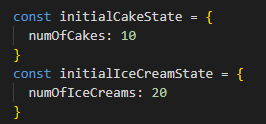


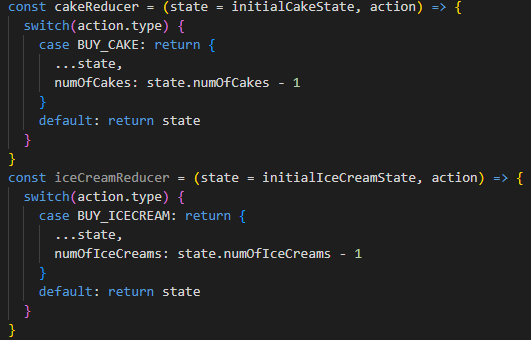
1. Now, I want to enhance the **cake shop application** to handle Ice creams also.  
   **Hence**, **state of application** is now has **No. of cakes** and **No. of ice-creams.  
   However,** in future application may also need to handle cookies, desserts etc.  
   Best practice is to create multiple **reducer** to handle each item in an application.  
     
     
     
   

  
When you run the **index.js**:  
  
Here,  
When user but a cake only No. of cakes has decreased.  
When user buy an ice-cream only No. of Ice-creams got decreased.

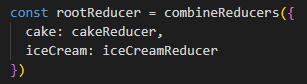
This is due to single **reducer** handling all items in a cake shop application

Let’s create multiple **reducers** to handle each item:

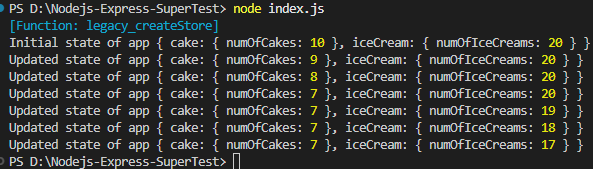
First maintain separate initial states for cake and ice-cream  


Now, create separate reducer functions for each action  
  
**NOTE:** **Redux store accepts only one reducer** (i.e., **const store = legacy\_createStore(reducer)**)

So, We need to combine multiple reducer into single reducer.

**combineReducers()** from **core redux** accepts an object with reducer functions.  
  
Before passing **reducer** to **legacy\_createStore(),** combine multiple reducers using **redux**’s **combineReducers()** method:  




When I run **index.js**:  


**Another Redux concepts:**

**Middleware  
A way to extend Redux with custom functionality.  
Middleware in Redux** is a function that intercepts every action before it reaches the reducer.  
Use **Middleware** for logging, crash reporting, performing asynchronous tasks etc.

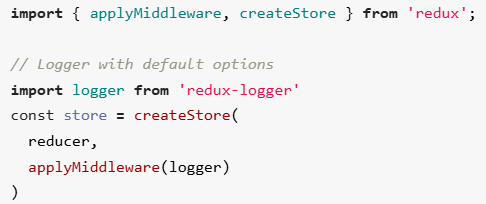
**applyMiddleware** is a **function (method)** provided by the **redux** package. It is used to **add middleware to a Redux store.**

**📦 redux-logger Middleware Package:**

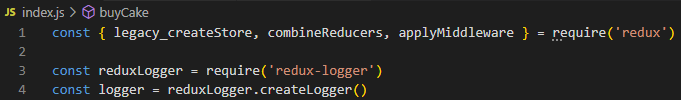
[redux-logger](https://github.com/LogRocket/redux-logger) is a popular middleware that **automatically logs** Redux actions and the resulting state changes to the console — great for debugging!

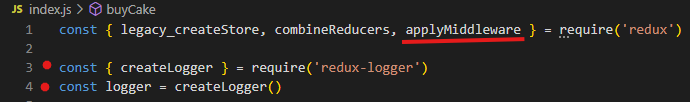


Usage:

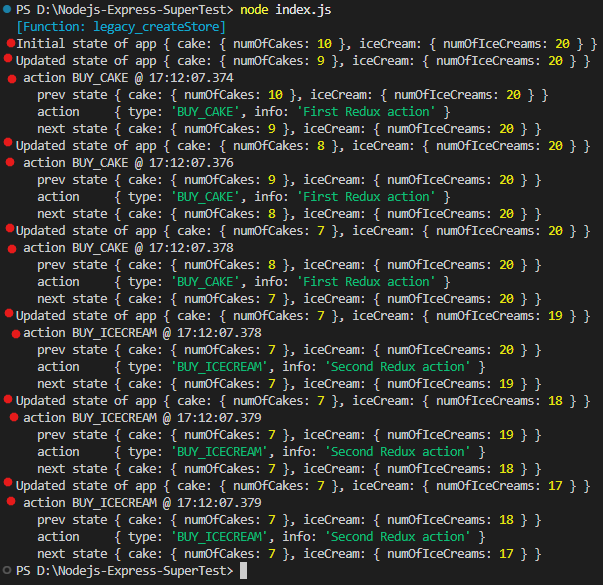


you can create your own logger with custom [**options**](https://github.com/evgenyrodionov/redux-logger#options):

  
 or







**Async Actions:**

By **default**, **Redux actions are synchronous**.   
**Synchronous actions** are **plain JavaScript objects**, dispatched immediately, and the reducer updates the state **right away**.

Example:  
****  
 Here, The **state of app** is updated **synchronously** after this dispatch.

To handle **async logic** (Ex: async API calls to fetch data from an endpoint and use that data in your application), Redux needs **middleware** — usually:

* + **redux-thunk** (most common used middleware)
  + **redux-saga**
  + **redux-observable**

**Assume,** our user application has to fetch a list of user from an API end point and stores it in the redux store.

For this scenario,

* **State of application for data fetching would be:**  
  state = {  
   loading: true, // property to indicate whether data is currently being fetched or not  
   // While build UI for a component this flag would help to show

loading spinner for data being fetch.

data: [ ], // property to store users after fetch

error: ‘ ‘ // property to store fetch error message if API call fail. This would

help to show the error in UI

}

* **Actions in our application would be:**
  + **FETCH\_USER\_REQUEST -** Fetch list of user from API endpoint
  + **FETCH\_USER\_SUCCESS -** Fetched successfully. This action depends

on above fetch user request action.

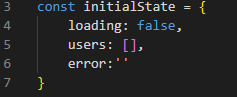
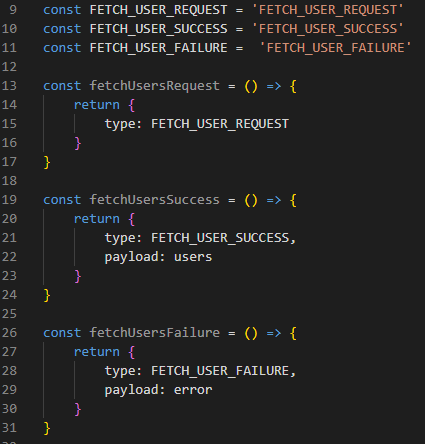
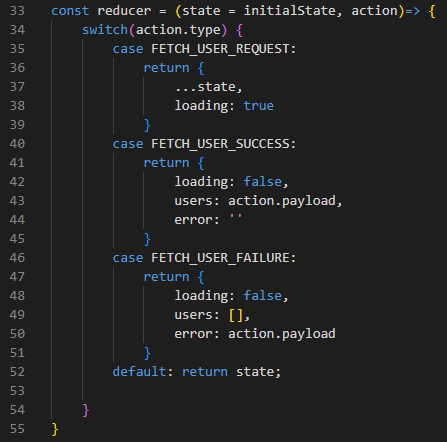
* + **FETCH\_USER\_SUCCESS -** Error fetching the data. This action depends

on above fetch user request action.

* **Reducers of our application would be:**



**Code Implementation:**

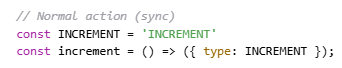
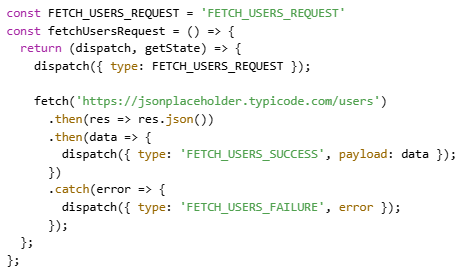
1. **Create** asyncActions.js
2. Defining **initial state of app  
   **
3. Defining **Actions  
   **
4. Defining reducer function  
   ****
5. Importing **legacy\_createStore** from **redux** library to create a Redux **store  
   **
6. Creating a Redux **store** to hold the entire state of app.  
   ****

So far, we set up our Node.JS application with Redux **state**, **actions** and **reducer**.  
Now, let’s see how to use **action** creators **asynchronously** with **network requests**(i.e., making an API call when working with Redux).

1. Install and import **axios** **package** which can help to make request to an API endpoint.



****

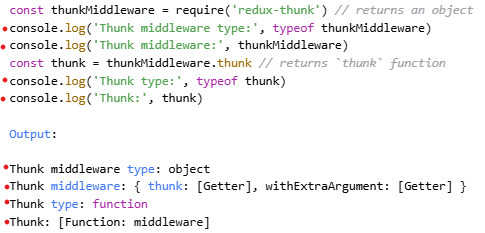
1. Install and import **redux-thunk** **middleware** which helps to define an **async action creators** in your application.   
   **🧠 redux-thunk**:  
   **redux-thunk** is a **middleware** package that lets you **write action creators function that return a function instead of an action(i.e., plain object)**.   
   These return functions called `**thunks**` which can handle **async logic**, such as fetching data from API or performing delayed actions, and then dispatch regular action objects once the asynchronous operation is complete.  
   This return function, the "**thunk**," receives dispatch and getState as arguments.   
   Example:  
   1. Way to define **synchronous Increment action**  
     
   You can **only return plain objects**, no async logic allowed.  
   2. Way to define **Asynchronous API action**
2. **applyMiddleware** is a **function (method)** provided by the **redux** package. It is used to **add middleware to a Redux store.**

****

1. **Install and Import thunk from a `redux-thunk` library**

****

But when you use require() (CommonJS) to import an ES module, the result is:

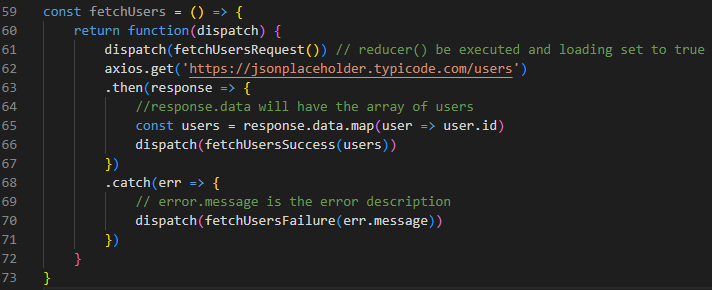
****

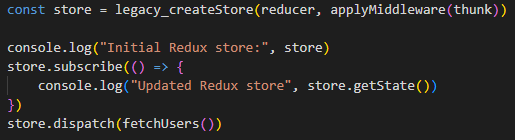
**Or**

****

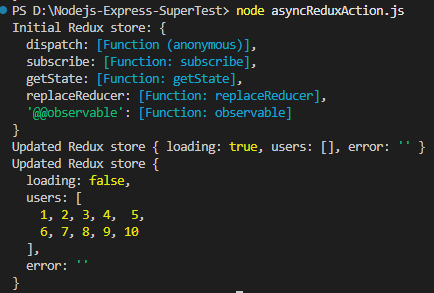
1. Now, pass`thunk` function to `applyMiddleware()` to apply to Redux store.



1. Now, creating action creator function that returns a **function/thunk** that handle async login.  
    This **return function** has `**dispatch**` method as its argument so we can use this **dispatch()** to  
    dispatch the action.  
    **NOTE:**   
    <https://jsonplaceholder.typicode.com/users> - this endpoint gives the random user data.  
    
2. **Subscribed to the store and dispatching** the action which returns thunk (i.e., a function()).



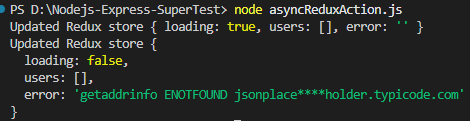
**Output:**



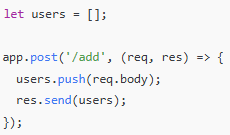
1. If we are passing invalid URL to axios like:



then output be like:



**Finally Remember:**

* Node.js **does not have built-in state like React**.
* But you **can define variables** (in memory) to hold state:  
  ****  
  Here, users is an **in-memory state** in Node, but it's:
  + **Volatile** (resets if the server restarts),
  + **Not structured** like Redux store.
  + You **don’t need Redux** in Node to define or manage basic state.
  + You might use Redux in Node when You want to **share Redux logic between frontend and backend.**

**React+Redux project:**

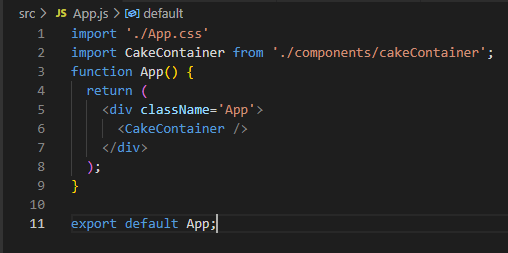
1. Created a branch (**React+Redux**) from Git Hub React project Repo **IndrajaBrishRepo**
2. npm install @reduxjs/toolkit  
   **NOTE:** **core redux** library already included with **Redux Toolkit(RTK)**
3. 

Fig: App.js

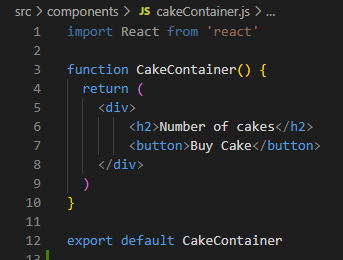
1. 

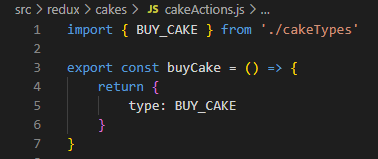
Fig: **CakeContainer** component

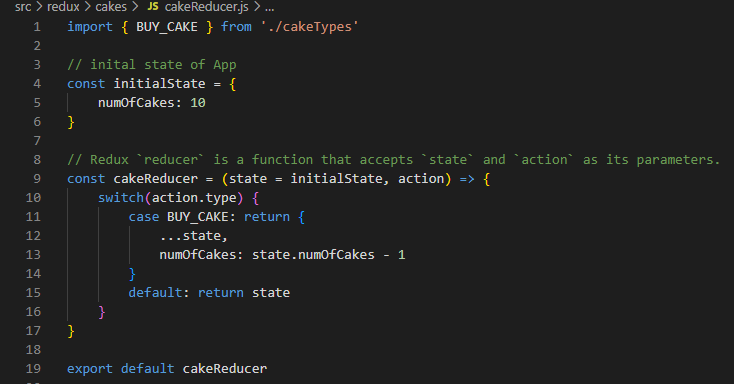
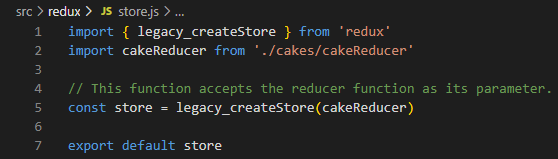
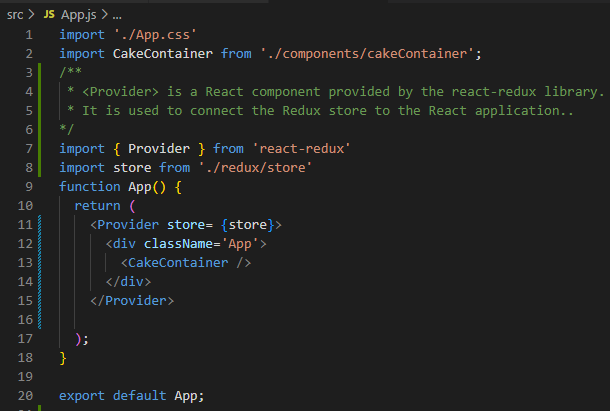
**Steps for creating a redux store and providing it to our React application:**

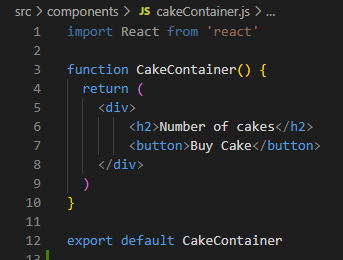
1. We are **defining** all action types in `**cakeTypes.js**`

  
Fig: cakeTypes.js

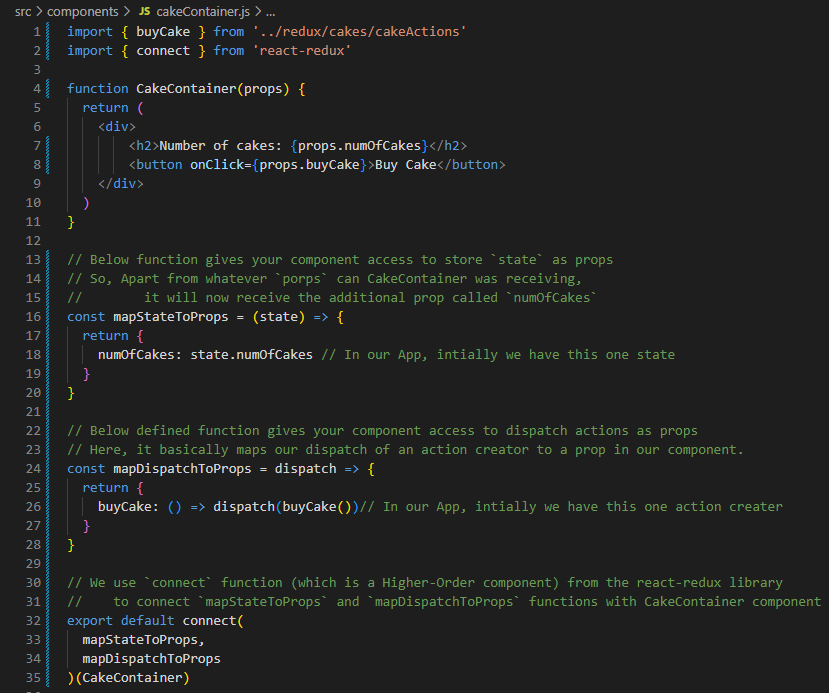
1. We are **creating** each action for each action type in `**cakeActions.js**`



1. Defining **Redux Reducer** **function** and **initial state of App  
   **
2. Within this file, we create our store.  
   
3. **<Provider>** is a **React component** provided by the **react-redux** library.  
   It is used to **connect the Redux store to the React application**.  
   **✅ Why do we need it?**  
   Redux is a **state management library**, but it doesn't know anything about React directly. To make Redux's store available to all React components in your app, we wrap the root of our component tree with <Provider> and **pass the store to it**:
4. In React, how do we hold the redux state? And how to dispatch an action from within a React component?

 **Here** in **cakecontainer** component, we are just showing text and button.

Now, let’s show that when `**Buy Cake**` button click happen an action should be dispatched and currnet number of cakes should be dispalyed.

****

**Explanation:**

connect() :

* is a **Higher-Order Component (HOC)/** Function from **react-redux** to connect the **CakeContainer** component to the **Redux store.**
* **connect() takes two functions as two arguments.**
* **Syntax:**

**connect(arg1, arg2)**

**arg1 -** tells the component what part of the Redux state it needs.

**arg2 -** tells the component what actions it can dispatch.

**mapDispatchProps**

* Defined a function that gets the redux **state** as parameter and returns an object
* Here the **state** from the **Redux store** is mapped to our component **props**  
  We are mapping state.numOfCakes to numOfCakes as a prop to CakeContainer Component
* So, Apart from whatever `porps` can CakeContainer was receiving, it will now receive the additional prop called `numOfCakes`

**mapDispatchToProps**

* Defining a function that gets the dispatch as a parameter and returns an object.
* Here, it basically maps our dispatch of an action creator to a prop in our component.

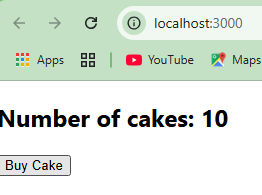
**Connect** from **react-redux**

* Function(which is also a Higher-order component) that connects **Redux** with **React**
* Takes a component as input
* Returns a new component with additional functionality
* In above case, connect() takes your component and gives it access to: Redux **state** and Redux **dispatch**
* You don’t need to call React Hooks useSelector() or useDispatch() manually — connect() gives everything as props.  
  But,

connect() - Good for class components

useSelector(), useDispatch() – Best for modern function components.

**Output when we run npm start:**

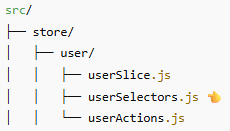
****

**Important:**

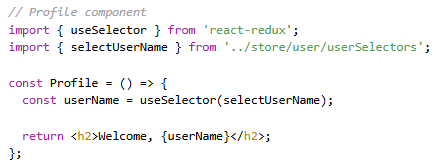
If we take a look at official **react-redux** documentation, they maintain a separate file called `selectors.js`.

Just like we maintaining, file for all **actions** and file for all **reducers**, there would be **selectors.**

A **selector** is a plain JS **function** that takes the **Redux state** and returns **some part of** state information from the Redux **store.**

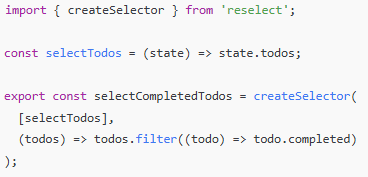






ఈ function లో state అనేది మొత్తం redux state, state.user అంటే అందులో ఉన్న user slice.

✅ Why do we use selectors:

* Avoid repetition - ప్రతి చోటా state.user.name లాంటి కోడ్ రాయకుండా ఒకేసారి select చెయ్యడం కోసం.
* **State structure మారినా**, component లో changes చేయాల్సిన అవసరం ఉండదు.
* **Performance** ( **Memoization** - **performance optimization technique**) కోసం, reselect లాంటి libraries తో selectors ను optimize చేయవచ్చు.  
    
    
  **Memoization** వల్ల unnecessary renders తగ్గుతాయి.  
  meaning : ఒక function ఒకే inputs తో మళ్లీ మళ్లీ పిలవబడితే, ఆ function రిజల్ట్‌ను గుర్తుపెట్టి (cache చేసి), తర్వాత direct గా ఫలితాన్ని ఇచ్చేయడం. అంటే function calculations మళ్లీ చేయాల్సిన అవసరం లేదు → వేగంగా (fast) జరుగుతుంది.

**Remember,**

**React-Redux pattern be like:**

* Action creators, reducers, provide the store and connect the components.
* Components can access state and dispatch actions.
* Now, react-redux **also offers set of APIs as an alternative to the** connect **Higher- Order component**

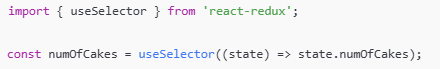
**useSelector + useDispatch**:

**useSelector** and **useDispatch**, which are **React-Redux Hooks** introduced to simplify Redux usage in **function components**.

They are used **inside function components only** and are the modern replacement for the connect() **Higher- Order component(**HOC**)**.

📌 Syntax and Usage:

useSelector – Read **state** from store (Redux store నుండి state ని చదవడానికి ఉపయోగించే Hook)

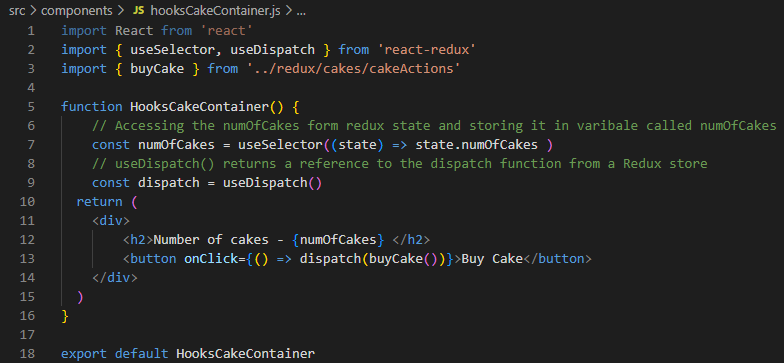


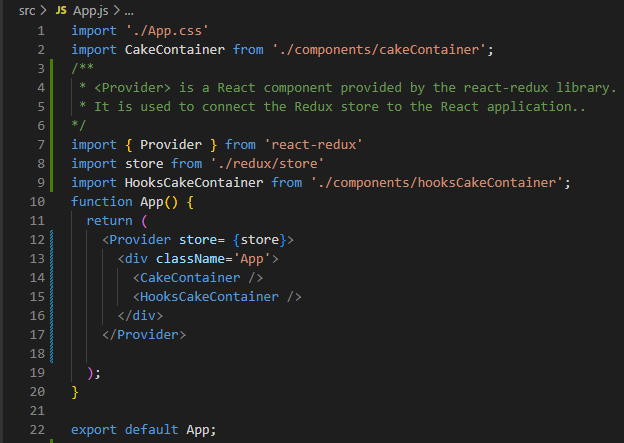
* This reads numOfCakes from the Redux **store** and stores it in a variable.

useDispatch – Send actions to store (Redux store కి actions పంపడానికి (dispatch చేయడానికి) ఉపయోగించే Hook)



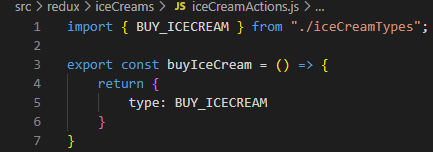
* This allows your component to **dispatch** an action (like buying a cake).

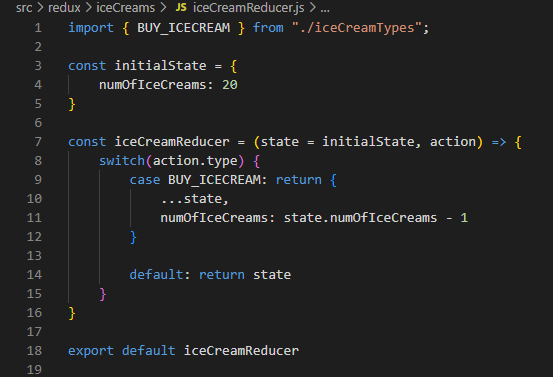




**Assume App has Ice-cream:**

****

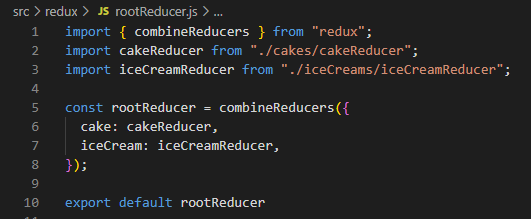
****

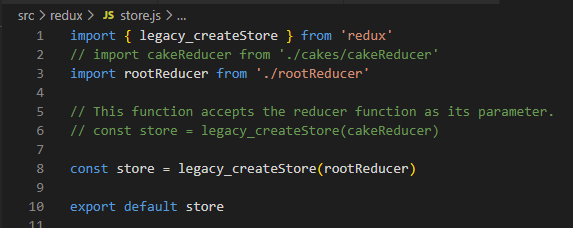
****This ice-cream reducer to **perform state transitions** based on the **action.**

**Now,** lets make sure our redux store aware about this ice-cream reducer.

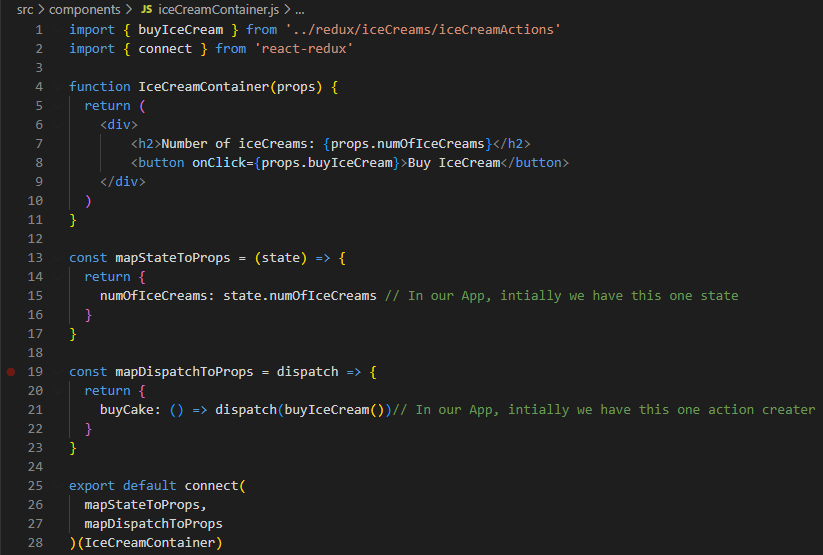
**Remember,** legacy\_createStore()function accepts only one reducer function.

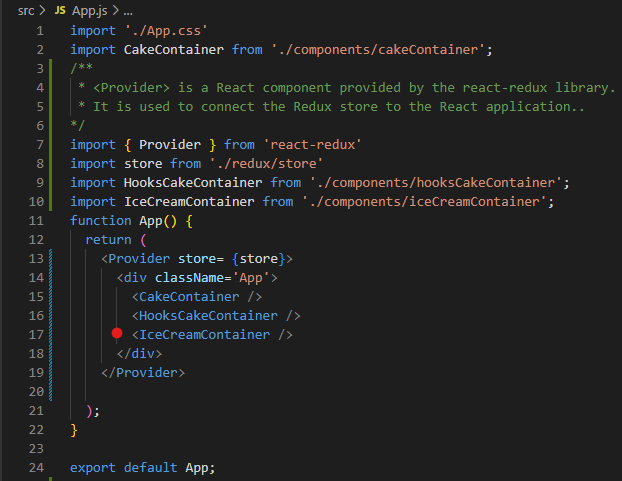
However, if your application has **multiple slices of state** (like customers, policies, and claims in your project), you can **combine them into a single root reducer** using combineReducers.

****

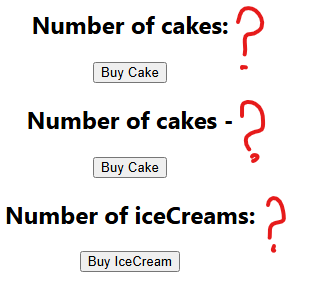
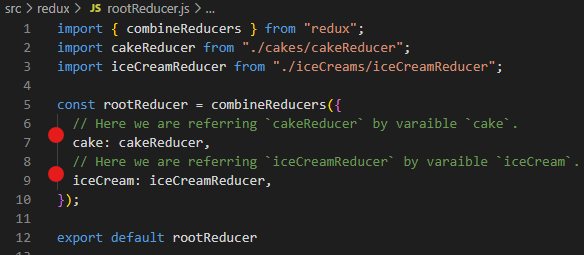
****

**Now,** Let’screate **IceCreamContainer** component

****

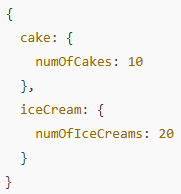
****

**When you run npm start the output you get:**

**  
This is because**, we have split **global state** into **state that is individually managed by separate reducers.  
**

Here, **cakeReducer** is referred by **cake** and **iceCreamReducer** by **iceCream**.

Hence, the **Redux state structure** becomes:



Hence, Each **key** (cake, iceCream) refers to its own **slice of state** managed by its respective reducer.

This means the **state** object that we have defined separately in the **cake** and **iceCream** **reducer files** also need to be accessed differently.

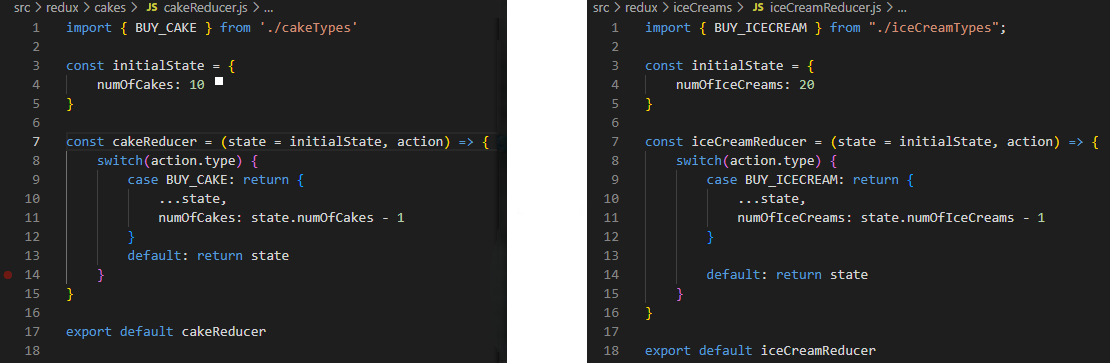
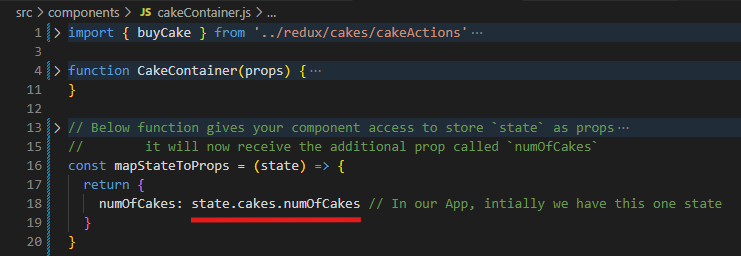
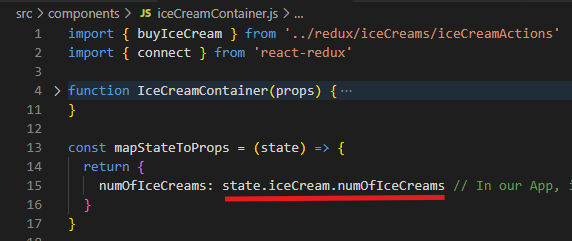
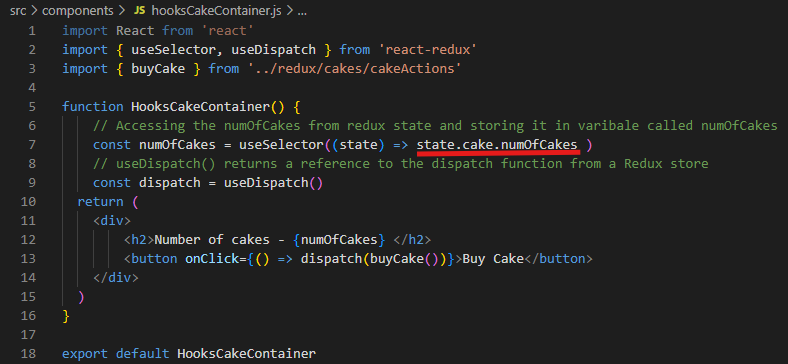


Fig: **cake** and **iceCream** **reducer files**







In **state.cake.numOfCakes**, the .**cake** refers to the **key** we specified in **rootReducer.**

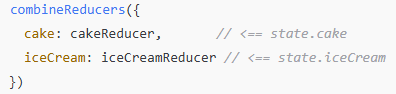
**Means:** You're saying: "Go into the cake slice of state, and get numOfCakes from it."

In **state.iceCream.numOfIceCreams**, the .**iceCream** refers to the **key** we specified in **rootReducer**

**Means:** You're saying: "Go into the iceCream slice of state, and get numOfIceCreams from it."

✅ Why it is required:

Because **combineReducers() nests each reducer's state under the key you gave**.

****

If you used only one reducer directly like this:

****

Then you can access:

****

But when combining reducers:

****

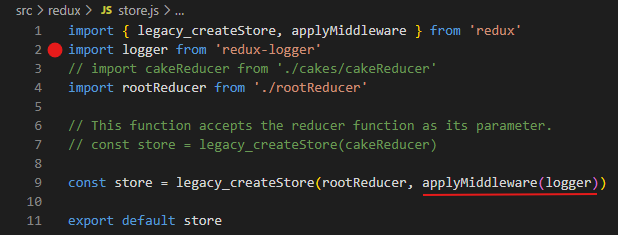
**When you run npm start you could see:**

****

**Middleware  
A way to extend Redux with custom functionality.  
Middleware in Redux** is a function that intercepts every action before it reaches the reducer.  
Use **Middleware** for logging, crash reporting, performing asynchronous tasks etc.

**applyMiddleware** is a **function (method)** provided by the **redux** package. It is used to **add middleware to a Redux store.**

****

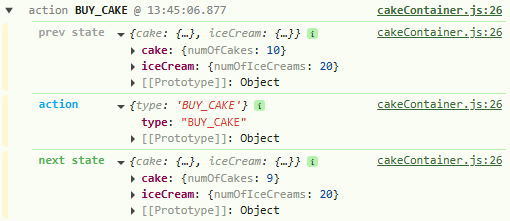
****

**NOTE:**

**`logger` middleware basically logs information related to the Redux store.**

**To view the logs run the React application and inspect the page and open console in browser.**

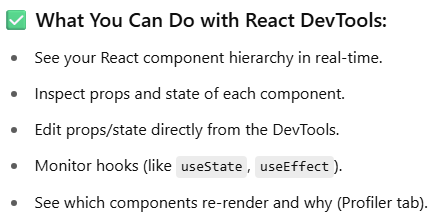
**When you click the button like Buy Cake or Buy IceCream, you see logs in browser console like:**

****

**React Developer Tools extension:**

It is an official tool by the React team that helps you inspect and debug React component trees and their state/props in your browser.

**Remember,** **DevTools only work in development mode**



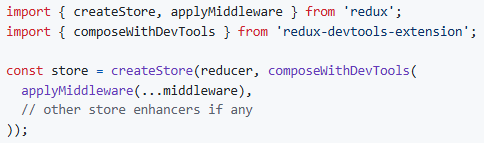
**💻 How to Install Redux DevTools Extension for chrome:**

1. **Open this link:** [React Developer Tools - Chrome Web Store](Redux%20library.docx)
2. Click **“Add to Chrome”**
3. Browser **window** lo **extension** icon lo **redux** **tool** untundi, **pin** it to appear on window like:



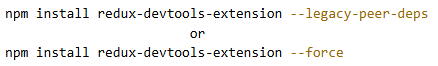
**💻 How to Install Redux DevTools extension package in your React+Redux Application**

1. **Open the link:** [**https://github.com/zalmoxisus/redux-devtools-extension**](https://github.com/zalmoxisus/redux-devtools-extension)
2. **At Usage section, only focus on:Installation:**

 **Usage:  
**

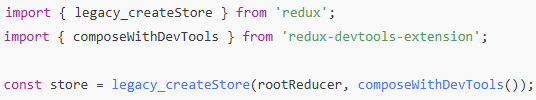
**NOTE:**redux-devtools-extension@2.13.9 only supports **Redux v3 or v4.**

1. **If you are using Redux V5.0.1, either**

**install,  
**

**This tells npm: “**I understand there's a peer version mismatch, but it's okay — I know what I'm doing.”

**Then use it like this in your** store.js:

****

**or**

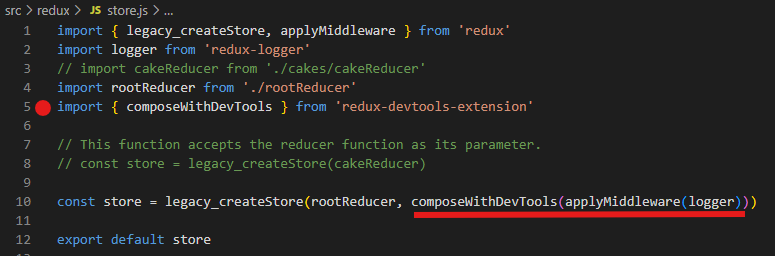
**Use Redux DevTools Without Package** (Best Practice for Redux v5).

Redux v5 already works well with DevTools without needing to install redux-devtools-extension.

****✅ This connects your store to the React DevTools without needing the external npm package.

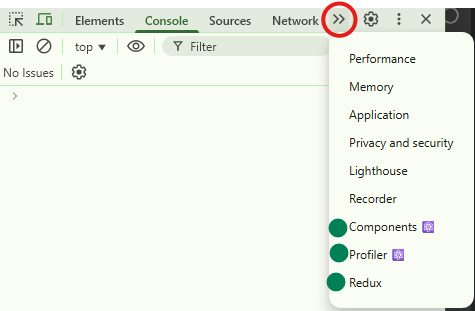
**Let’s implement Redux DevTools Extension in our Application:**

****

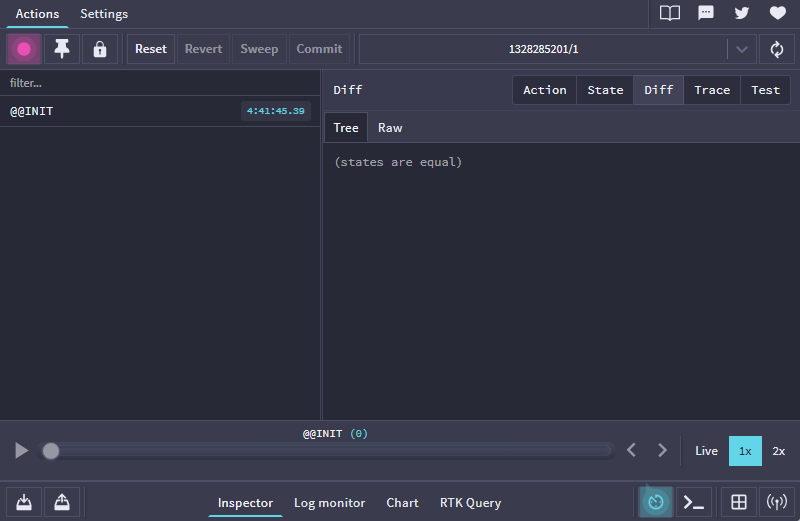
****

**Now, Run npm start,**

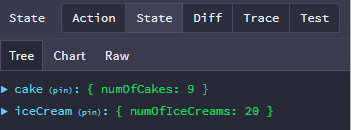
****

****

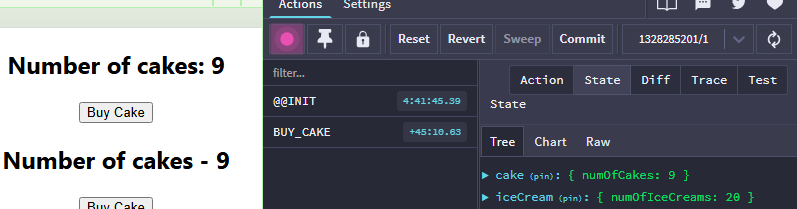
**When you click on `Redux` option, you will see a panel:**

****

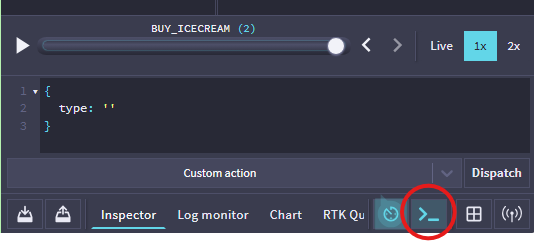
* **This panel is very useful for Redux Application for debugging.**
* **State button displays Redux state structure**

****

* **When you click on Buy Cake button, a dispatched action and updated state will be shown**

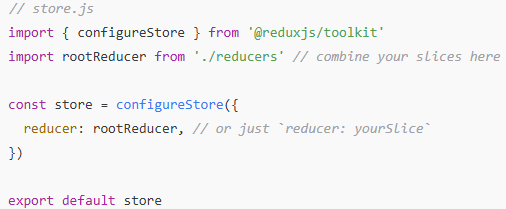
****

* + **With this option I can dispatch an action without the use of UI element.**

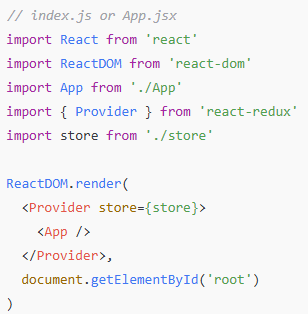
****

**Important:**

* To use **Redux DevTools Extension** in your React application with Redux Toolkit, you **don’t need to install anything extra manually**. Redux Toolkit already comes preconfigured to work with Redux DevTools out of the box when you use configureStore.
* **Create your store using configureStore:** It automatically enables Redux DevTools.

****

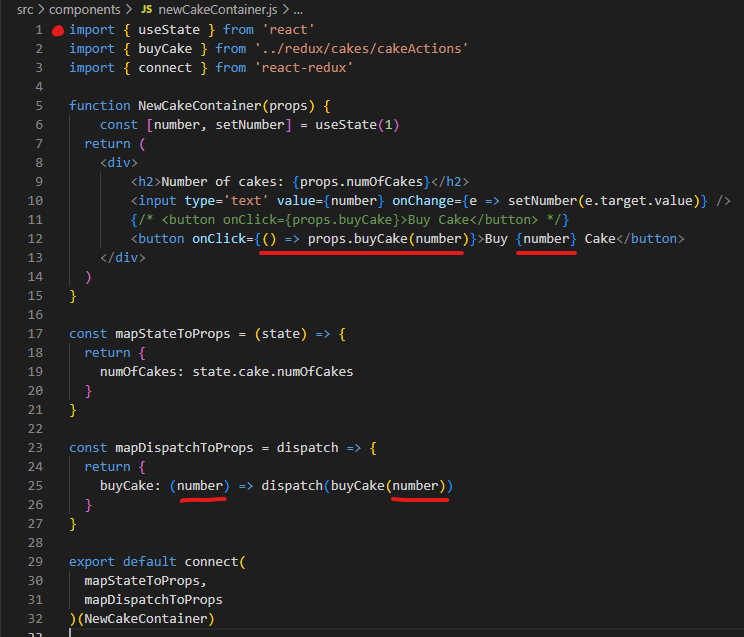
* **Wrap your app with <Provider>:**

****

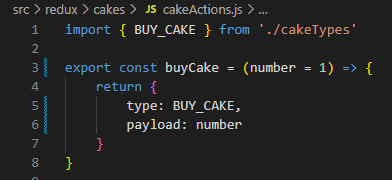
**Handling/Managing Async Actions in React Application with Redux:**

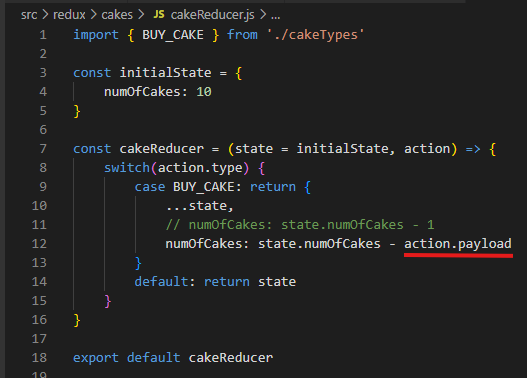
**Let’s see how to add a payload(UI input value) to your Redux action creator.  
So far, we built buttons (Buy Cake and Buy Icecream) which allow user to buy only 1 item.**

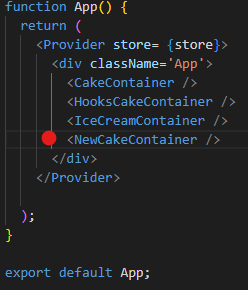
**Now lets implement an Input element to allow user to buy bunch of items.**

**  
Fig: NewCakeContainer component**

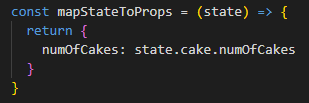
**The dispatch() takes action creator as parameter. The action creator buyCake is expecting a value, hence we need to pass a value to buyCake().**

**  
If you remember we use this buyCake in other components. Hence, we define its default value as number=1 to ensure that existing functionality of other components doesn’t break.**

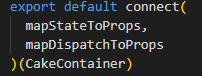
****

****

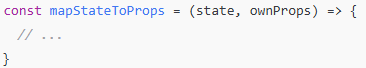
**Important:**

****

* **Here, you see this `mapStateToProps`, which means, this is a user defined function which can receive entire Redux state from the store as parameter. Using that state object we access its small part i.e., `numOfCakes` to map it to component `props`.**
* **Simply,** mapStateToProps is a function used in **React-Redux** when you're connecting a **component to the Redux store** using the connect() function from react-redux.

****

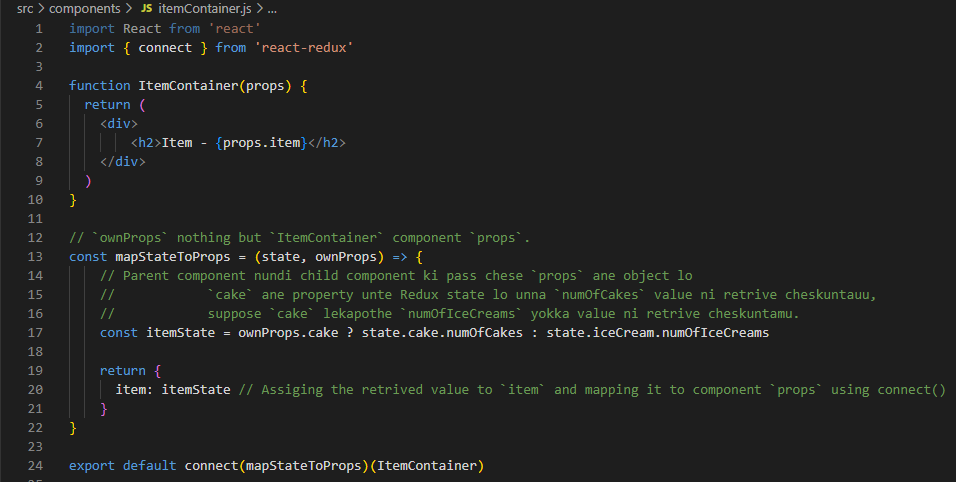
* **Actually, mapStateToProps has two parameters:**

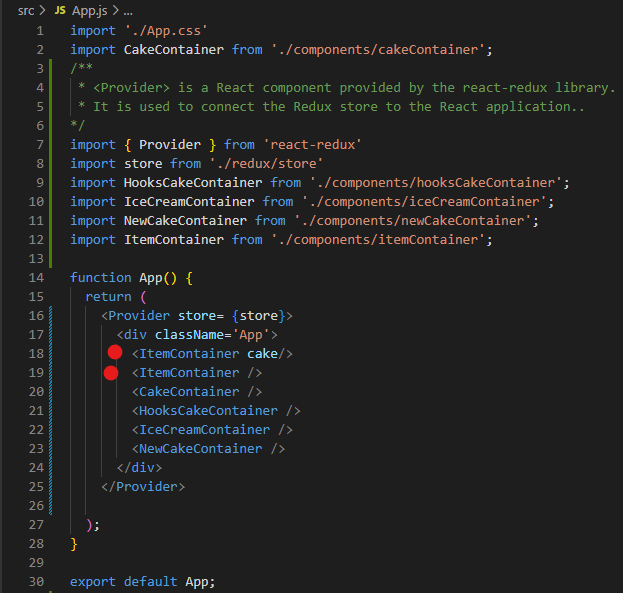
****

state: The **entire Redux state** from the store. You use this to extract the data your component needs to map it with component **props**.

ownProps: The **props passed to the connected component from its parent**. This allows your mapStateToProps to make decisions based on component-specific props.

* **To understand this better let’s create a component. Through this component what we want to achieve is to display either the number of cakes or the number of ice-creams based on the prop that is passed from the parent component.**

****

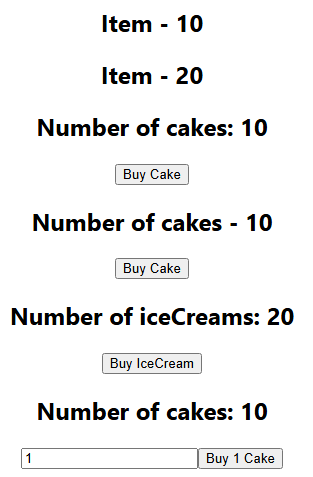
****

**Here,**

**For one child component ItemContainer placeholder, we are passing `cake` as prop from parent component.**

**For another child component ItemContainer placeholder, we are passing nothing as prop from parent component.**

**When we run `npm start`,**

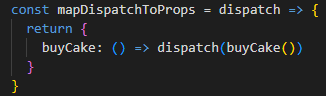
****

**Here,**

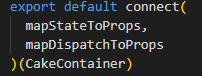
**In Item – 10 , 10 specifies the numOfCakes in Redux state.**

**In Item – 20, 20 specifies the numOfIceCreams in Redux state.**

**Now, Let’s understand about:**

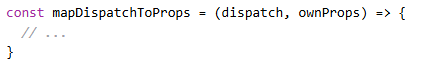
****

**Here,** mapDispatchToProps is a function used with connect() from **React-Redux** to bind **action creators** to the component as props, so your component can **dispatch actions** to the Redux store.



* dispatch is the Redux dispatch() function passed automatically by connect()
* buyCake: () => dispatch(buyCake()) creates a prop called buyCake in your component.
* When buyCake() is called in the component, it **dispatches the buyCake action** to the store.

If you want to access **props passed from the parent component in child component**, you can define mapDispatchToProps with **two parameters**:

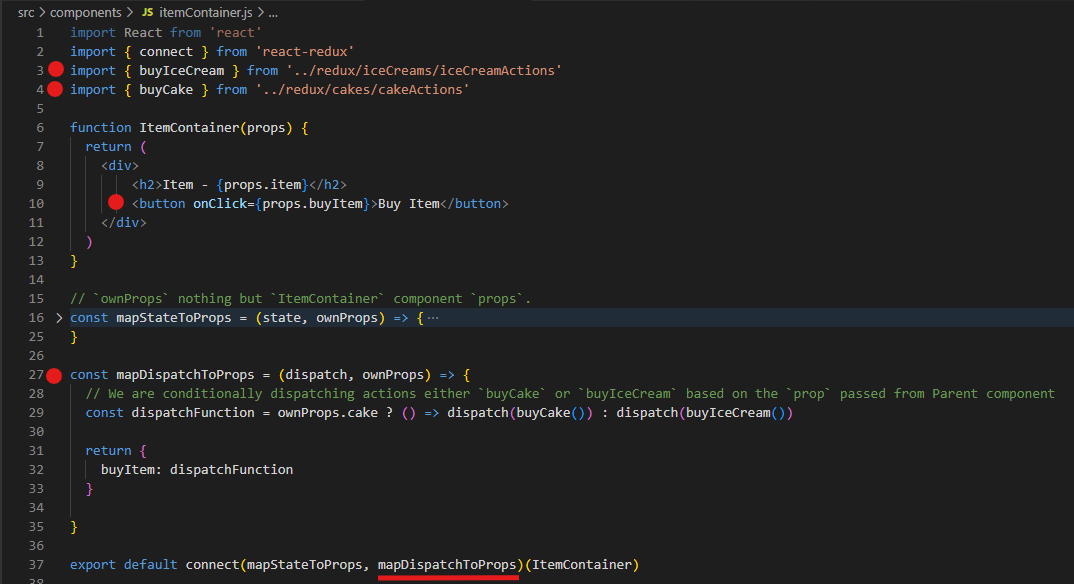


Here,

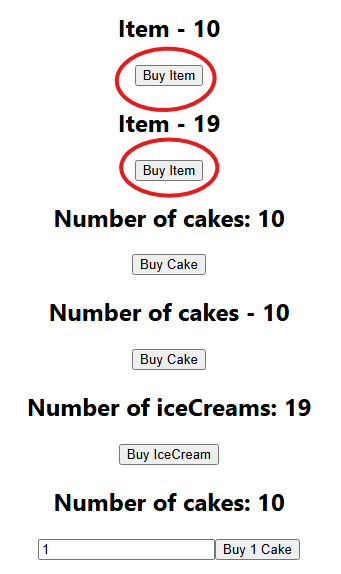
dispatch: The Redux dispatch function.

ownProps: The props passed from the parent to the connected component (child).

**To understand this better let’s implement in component.**



When you run `**npm start**`:

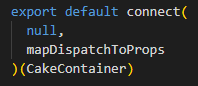


**NOTE:**

* connect() is a **Higher-Order Component (HOC)**/function from **React-Redux** to connect the CakeContainer component to the **Redux store**.
* It takes **two arguments**:

**connect(mapStateToProps, mapDispatchToProps)**

* mapStateToProps – tells the component what part of the Redux state it needs.
* mapDispatchToProps – tells the component what actions it can dispatch.
* When your component **doesn't care about reading Redux state**, but only **needs to dispatch actions**. Then, connect() should be like:



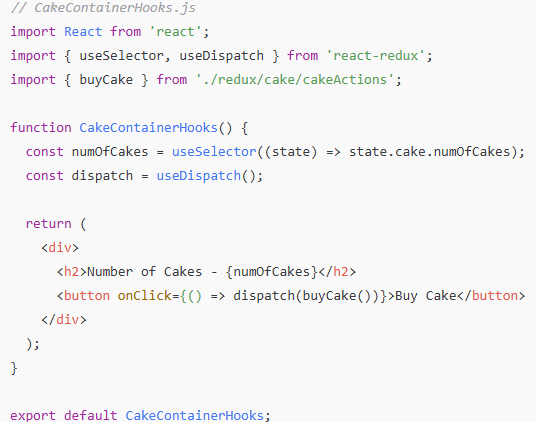
* Meaning:

❌ “This component doesn't need any state from Redux.”

✅ “But I want to give it some **action dispatch functions** as props (from

mapDispatchToProps).”

**We can use React-Redux Hooks like useSelector and useDispatch to build same logic that we implemented using connect().**

****

**Async Actions with React and Redux:**

**Synchronous Actions:**

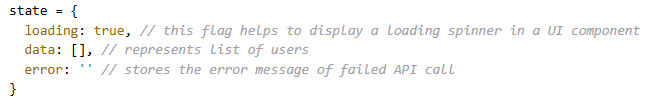
* As soon as an action was dispatched, the state was immediately updated.
* Suppose, if you dispatch the **BUY\_CAKE** action, the **numOfCakes** was right away decremented by 1.

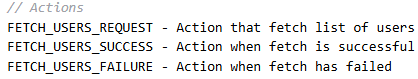
Async Actions:

* You wait for a task to complete before dispatching an action.
* **Use case:** Async API calls to fetch data from an end point and use that data in your application. Making an API call and based on response we need to dispatch an action.

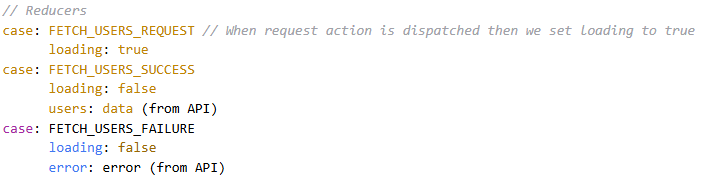
**Let’s Implement this in our Application:**

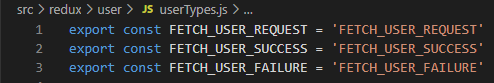
* Code to fetch a list of users from an API end point and stores it in the redux store.
* So How our Application with Redux **state**, **actions** and **reducers** should be like:

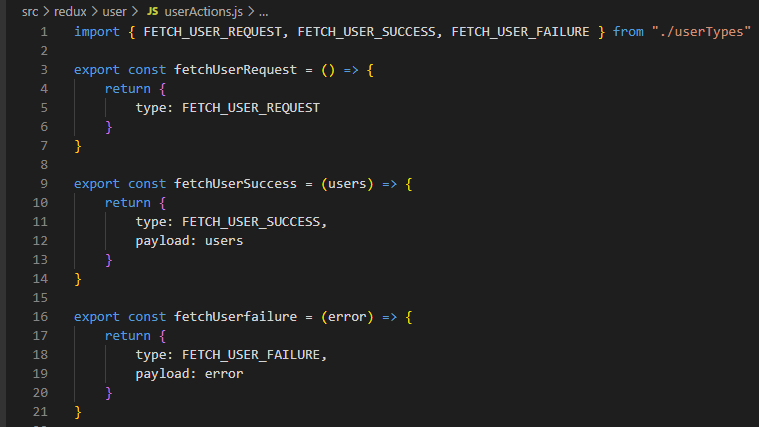


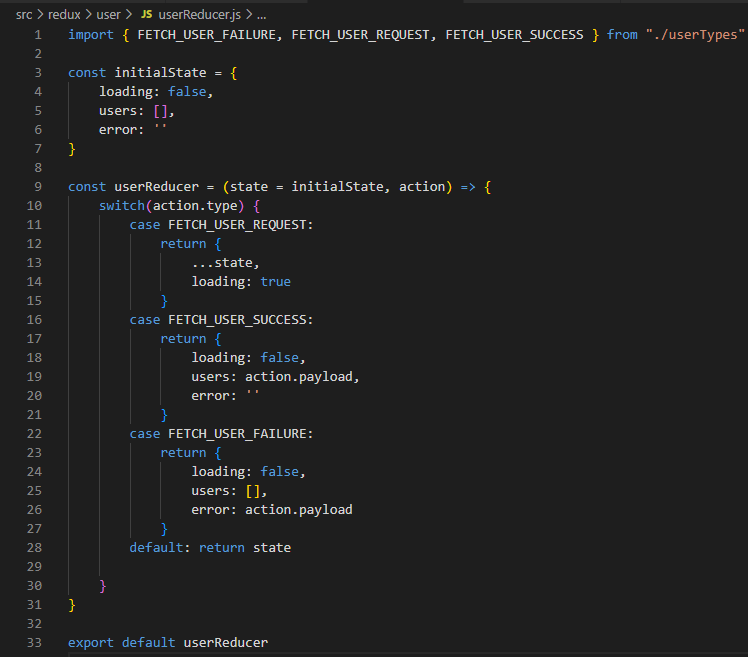


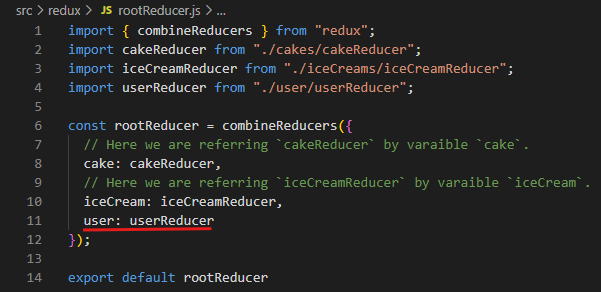
**NOTE:** the 2nd and 3rd actions are dependent on 1st action.











Now, we need to build an Async action to make an API call to dispatch an appropriate actions that we have defined already and render the list of users in the browser.

Explanation:

import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

Here,

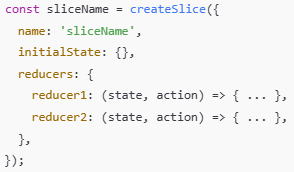
createSlice and createAsyncThunk — these are **core tools** for **managing state and async logic** in modern Redux apps.

**createSlice:**

Creates a **slice** of your Redux state — including:

* Initial state
* Reducers (for sync logic)
* Automatically generated action creators and action types

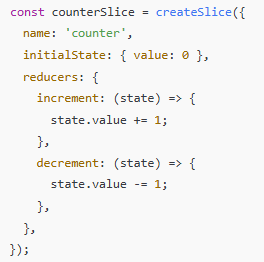
Syntax:



✅ What it gives you:

* A **reducer** function for the Redux store
* Auto-generated **action creators**
* Clean, concise, and organized structure

Example:



You get:

 counterSlice.reducer → use in store

 counterSlice.actions.increment() → dispatch in UI

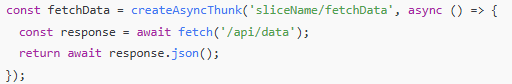
**createAsyncThunk:**

It helps you handle **async API calls** (like fetching data) using Redux.

It generates:

* Pending / Fulfilled / Rejected **action types**
* Simplified logic for API handling

**Syntax:**

****

**✅ Behind the scenes:**

It dispatches:

* fetchData.pending when request starts
* fetchData.fulfilled if successful
* fetchData.rejected if error occurs

You handle these cases inside your extraReducers in createSlice.

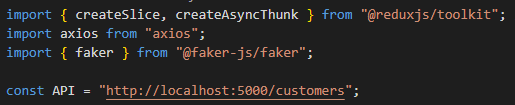
Brish Tasks on **Yoloh** Insurance project

**Task-1:**

**Build an Admin dashboard where Admin could perform CURD operation.**

**Code Implementation:**

* **Build customer module in nav bar.**
* **When Admin click on customer module it should render data from DB.**
* **Also Admin able to perform CURD operations on rendered customer data.**



**createAsyncThunk()**: to handle API calls (like GET, POST).

**Axios**(): for making HTTP requests.

**Faker** : to generate mock (fake) data like names, emails, etc.

**const API** is the **base endpoint** of your mock json-server. You’ll use this in all API requests.

**createSlice():**

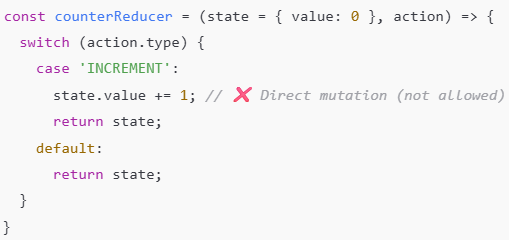
* It is a **function from Redux Toolkit** that automatically creates:
* action creators
* action types
* reducer function
* It Helps you write Redux logic in one place for a **"slice" of state** (e.g., users, products, claims).
* Reducers inside createSlice() use **Immer** under the hood, so you can mutate state directly like:



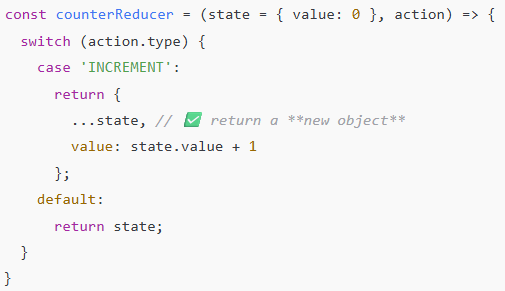
Even though normal Redux reducers need to return a **new object** every time.

Traditional Redux reducers (pure Redux) **vs** Redux Toolkit createSlice() reducers (with Immer)

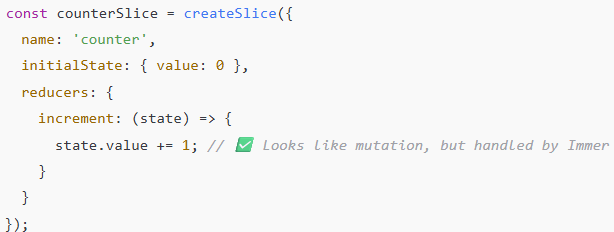
In Pure Redux (Traditional Reducer), you **must not mutate the state** — instead, return a **new copy**:



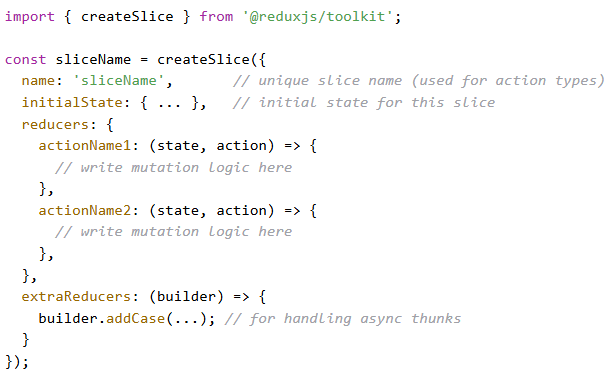
You **MUST** do this:



In Redux Toolkit using createSlice(, you **can** write code that looks like it’s mutating the state:



**Syntax of createSlice():**



**NOTE:**

reducers – Sync actions (increment, delete, etc)

extraReducers – Handle async actions (API calls)

Example:



Here what **createSlice()** gives is,

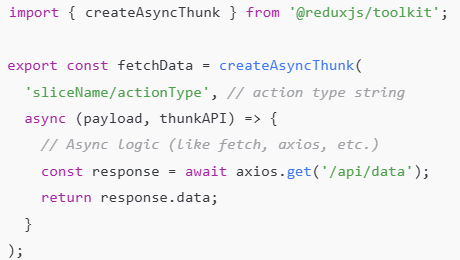
Let’s say your name is "counter" and action is "increment"

1. It automatically creates an action type:  
   "counter/increment"
2. It gives you an action creator:  
   increment() → { type: 'counter/increment' }
3. It gives you a reducer function that handles all your defined cases.

**createAsyncThunk()**: is a helper function **provided by Redux Toolkit** to **handle asynchronous logic** like API calls (GET, POST, DELETE, etc.) inside your Redux flow.

* It helps avoid boilerplate.
* Automatically tracks async status (pending, fulfilled, rejected)
* Easily integrates into extraReducers in createSlice
* Without createAsyncThunk:
  + You’d manually write action types (e.g., FETCH\_CUSTOMERS\_REQUEST, FETCH\_CUSTOMERS\_SUCCESS)
  + You’d manually dispatch and handle async states.
  + More boilerplate, harder to maintain.
* With createAsyncThunk:
  + Just write a simple async function.
  + Redux Toolkit automatically creates:
    - a pending request action (for loading)
    - a fulfilled action (for success)
    - a rejected action (for error)

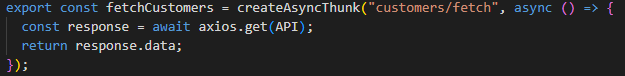
**Syntax:**

****

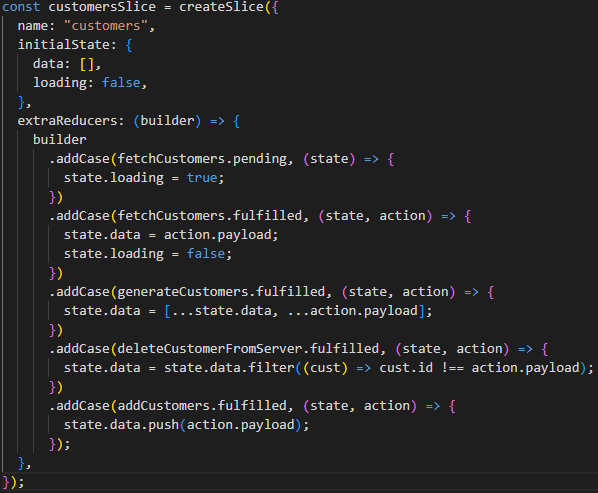
**Here,**

**“sliceName/action.type” – Used to generate action types.**

**async (payload, thunkAPI) – The async function that runs your API logic.**



* Fetches all customer data from your backend (JSON Server).
* Returns the data to be used in the Redux store.



**Here,**

* builder is an **object provided by Redux Toolkit**'s createSlice() internally to help you **add reducers for external actions** — like actions created by createAsyncThunk.
* Normally, reducers are defined inside **reducers: {}** in createSlice, which only handles **local sync actions** (like **increment**, **decrement**).
* But async actions (like API fetches) created using createAsyncThunk() generate **pending, fulfilled, and rejected actions**. These need to be handled inside extraReducers.
* So Redux Toolkit gives us the builder object to **build up a case-by-case reducer for each async state**.
* builder allows chaining the **addCase()**

✅ Syntax:

****

****