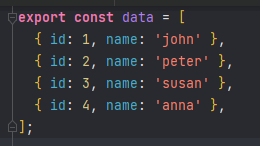
* Hooks starts with use.
* Components must be uppercase
* Hook must be inside the component body, or function
* We can’t call hook conditionally (for example, inside if block).
* **UseState:**

It returns an array. first value of array is the initial value and second one is the handler function.

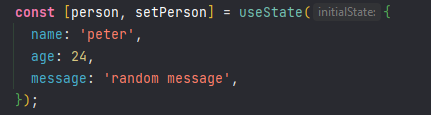


To change the value of text, we need to use setText function. It will trigger a re-render.

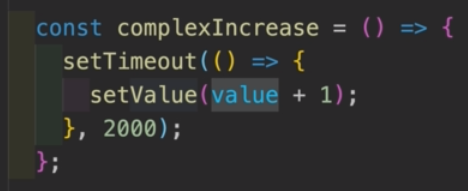
Array:

Object:

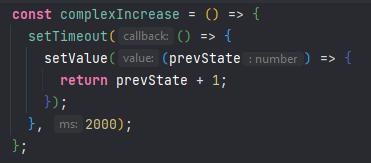


Async Example:

In this case, there is a delay between execution. So if we keep clicking the button that have this handler, It will take value = 0 for 2 sec. So, the value won’t be updated.

In this case, we can pass a function instead of a value inside the **setValue** function. The advantage here, that it gets the previous state value as parameter. Thus the issue can be solved.



* **UseEffect:**

It is used for side effect (means any work outside of the component). For example, changing document title, signing up for subscription, fetching data, setting up an event listener etc.

1. **Conditionally running logic:**

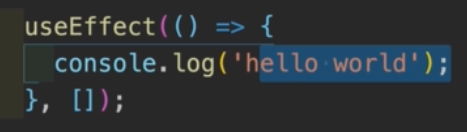
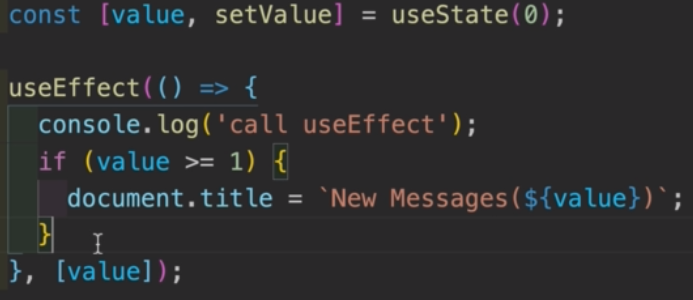
By default, it run once after every re-render. In many cases we might want to call useEffect conditionally. But in general, we cannot call hooks conditionally in react.

However, just like useState, we can pass a function inside the useEffect hook. Inside that function we can use condition as much as we want.

1. **Second Parameter/Dependency List:**

Apart from the callback function, we can pass an array as second argument to useEffect. It is a list of dependencies. If we leave this array blank, it will only run on the initial render. If we put a value in there, useEffect will only run when the value updates. We can have multiple useEffect.

1. **Cleanup function:**

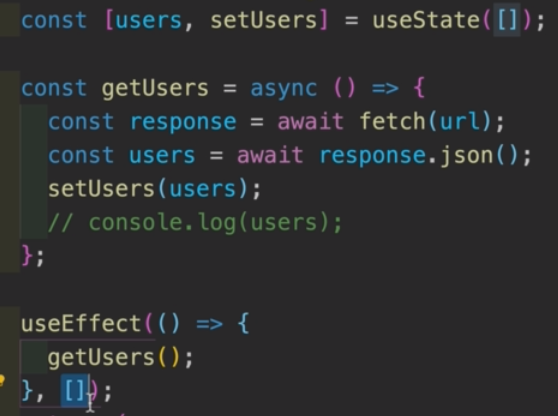
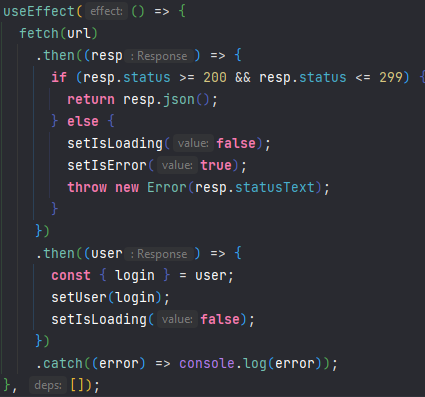
Event listeners can cause memory leaks if not cleaned up properly. In this case, we are adding an event listener everytime window is resized. But the problem is we are doing this in setSize function, which is causing a re-render. In this case a new event listener is added and it will keep growing.

To solve this, we can return a function from the first argument of useEffect. This will be invoked once we exit. Here we can do the cleanup work.

1. **Fetch Data:**

**Important**: we can not use async await to the callback function of the useEffect (first parameter). We can do this inside the callback function, or setup the function separately and call it inside.

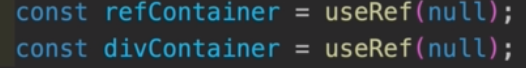
 

Here we are calling the async function inside the useEffect callback function. It is important to setup the dependency list as an empty array in this case. Otherwise it will cause an infinite loop of re-render (useEffect runs on every re-render and it calls getUser() which in turn calls setUser which causes a re-render)

Second example is for conditional rendering while fetching a data. Here we have logic to throw error in case server returns error response.

* **UseRef:**

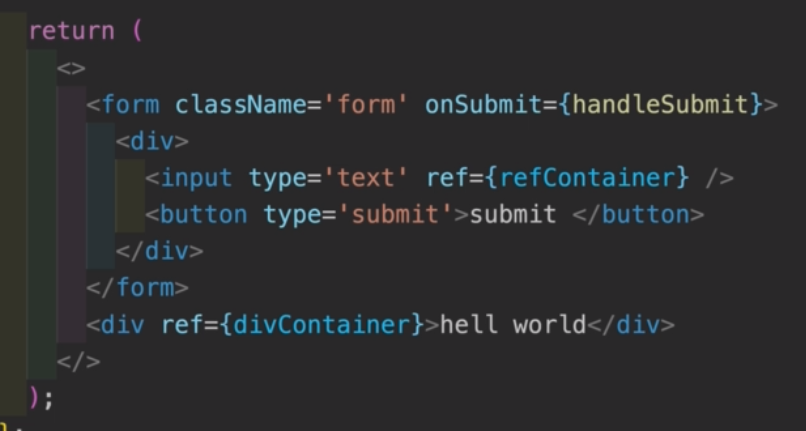
This hook is used for uncontrolled element. Most popular use of this hook is to target a dom element to have uncontrolled input (similar to javascript)



This is how to initialize a useRef. It is an object with a key of current.



In this case, the refcontainer contains reference of the input field. So we can access or manipulate its values like refContainer.current.value.



Some use cases:

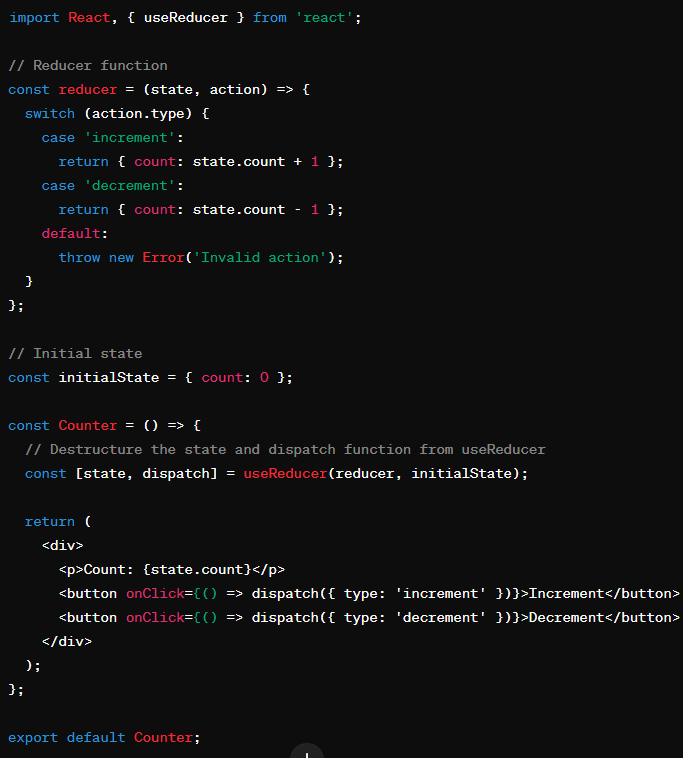
1. When you need to trigger imperative interactions with the DOM, such as focusing an input, scrolling to a specific position, or triggering animations, useRef can be used to achieve this without relying on component state.
2. You can use useRef to create a reference to a DOM element and access or manipulate it directly.
3. useRef can also be used to store values that persist across re-renders without triggering re-renders themselves. This is useful for storing mutable values that you want to maintain between renders, such as counters, flags, or previous values.

* **UseReducer:**

This is an alternative state management like useState for complex applications. The useReducer hook takes two arguments: a reducer function and an initial state. The reducer function receives the current state and an action, and returns the new state based on the action type and payload. It follows the same pattern as the reducer functions in Redux.

useReducer is especially useful when:

1. You have complex state logic that involves multiple sub-values.
2. You need to pass down the state and updater function through multiple levels of components without relying on props drilling.
3. You have state transitions that depend on the previous state.
4. You want to centralize state logic and make it easier to test.



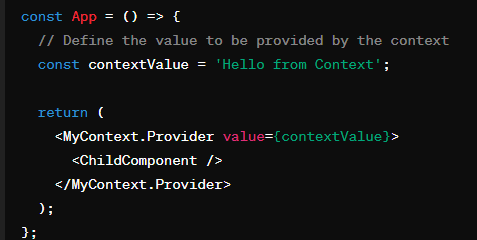
* **Context API:**

**Prop Drilling**: it is basically passing data from parent component to child component on multiple level. Often it so happens that one of those child components in some level doesn’t need that prop.

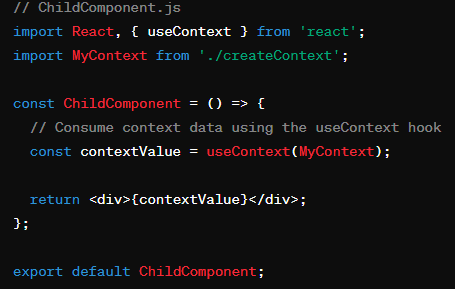
The Context API is a feature in React that provides a way to share data between components without having to pass props manually at every level of the component tree. It is designed to solve prop drilling issue.



Once we have created the context, we will have access to provider and consumer. Basic workflow is after we create the context, we will wrap the parent Component with Provider component. Provider component will have a prop named value, that is where we will pass our data



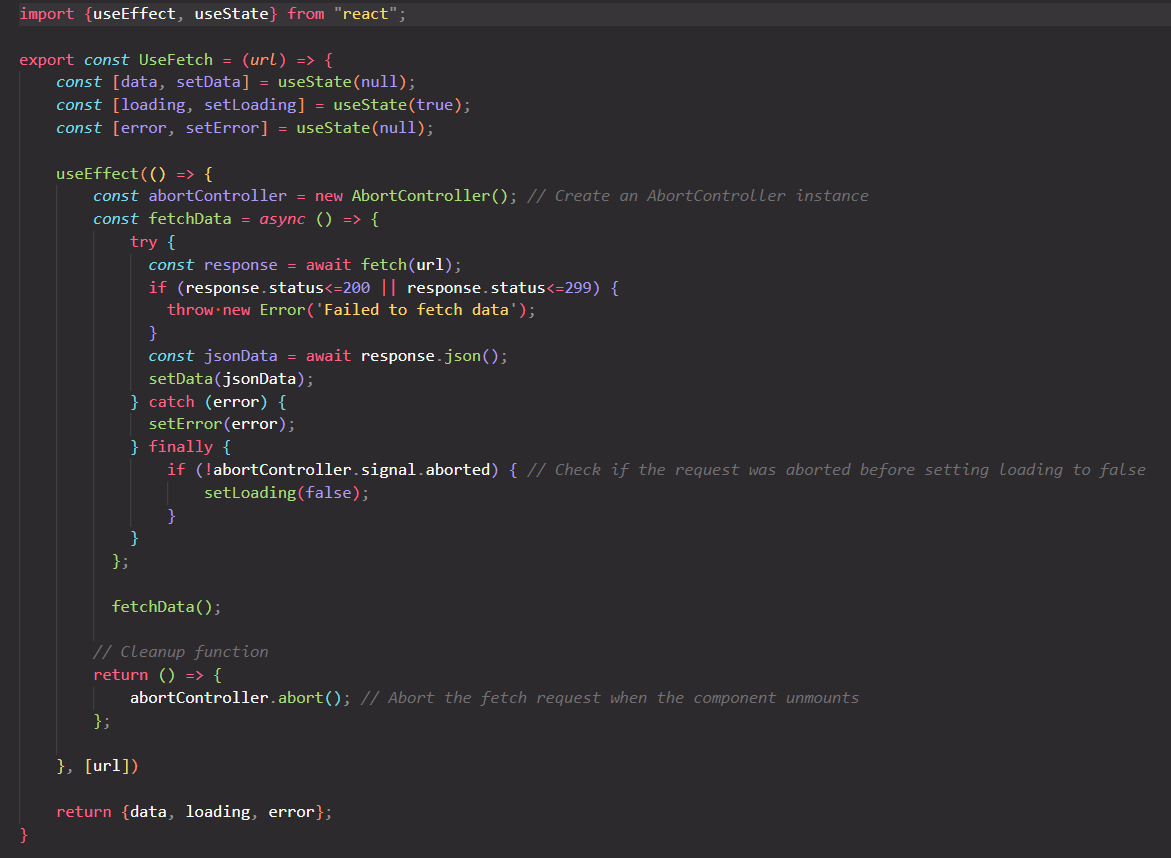
. It will be later accessable in all child element using useContext function



We can pass multiple context as well wrapping the component with multiple conext. We can pass redux functions/store in the value as well.

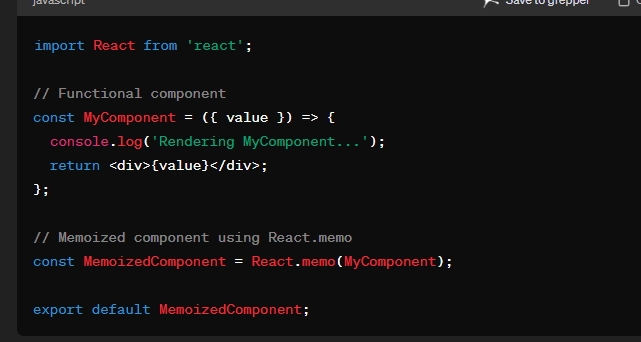
* **useFetch:**

This is a custom hook, not provided by react. It is used as an example of making our own custom hook.

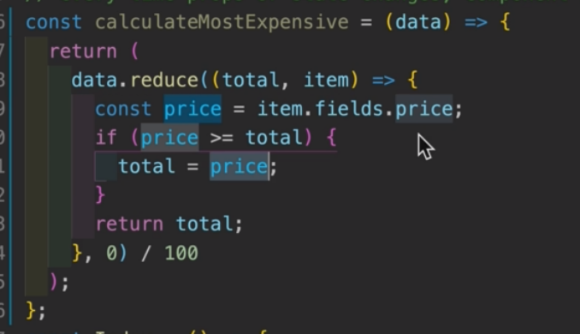


* **React.memo:**

React.memo is a higher-order component (HOC) provided by React that is used for memoizing functional components. In this example, if we wrap the whole component using memo, it will only re render when the value is changed. It is helpful in cases like where there is a list of 50 product cards that we don’t want reload, but another part like a header/title reloaded only.



* **useMemo:**



The React useMemo Hook returns a memoized value. Consider this CPU intensive function,

* **useCallback:**

A typical scenario can be a list of products that is in a page, each product card has a add to cart option. Now typically when we will click the button, the state will change and whole page will be re-rendered. The React useCallback Hook returns a memoized callback function.

The useCallback and useMemo Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function.

