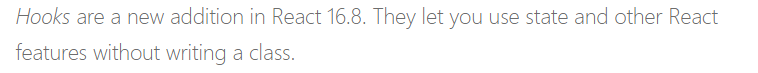
# **React Hooks**



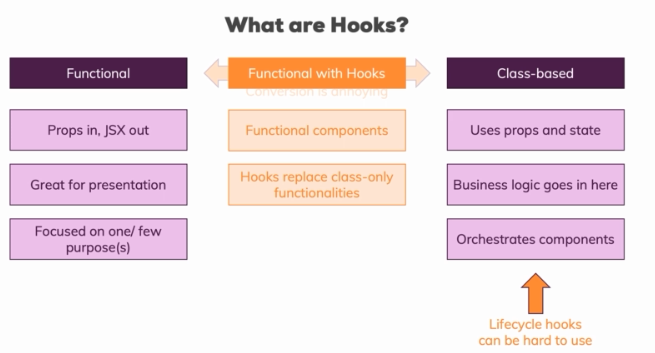
React hooks are a very new feature added to React with version 16.8 and this feature allows you to use functional components only. You can still use class-based components.



**Hooks don’t work inside classes. But you can use them instead of writing classes.**

**What is a Hook? A Hook is a special function that lets you “hook into” React features. For example, useState is a Hook that lets you add React state to function components**

**When would I use a Hook? If you write a function component and realize you need to add some state to it, previously you had to convert it to a class. Now you can use a Hook inside the existing function component.**



**hooks are a new way or offer us a new way of writing our components. Hooks give us a new way of creating functional components.we don't have a new way of creating a component,we will use the functional approach instead,but hooks are extra features, extra functions we can call in our functional component which give us access to certain capabilities we could only use in class-based components before.**

**to use React hooks, you need to use the right version. any version, 16.8 or 9, doesn't matter but any version higher than 16.8 or at least 16.8 will do because the hooks feature was included in that version.**

**Hooks don’t work inside classes. But you can use them instead of writing classes.**

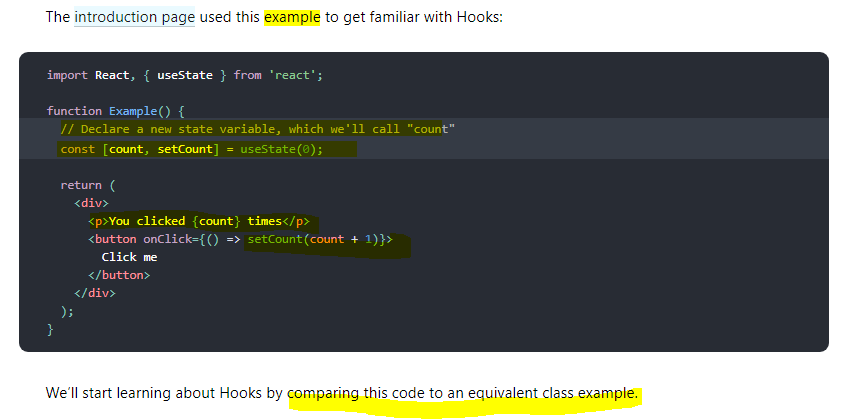
## **useState() hook**

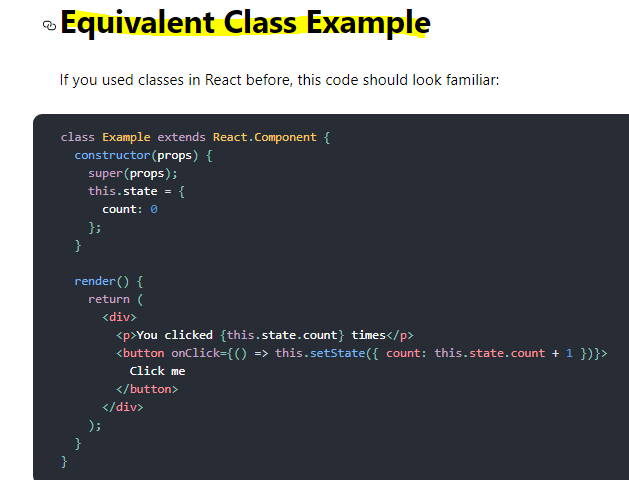
**we import it from the React package and it's called useState.**

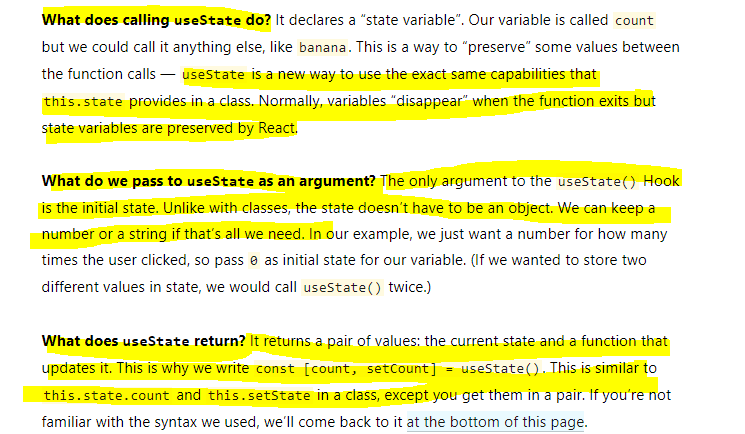
**This function takes an initial state. We can pass anything here by the way, an empty string, 0, null, an empty array, an empty object, whichever kind of state you want to manage.**

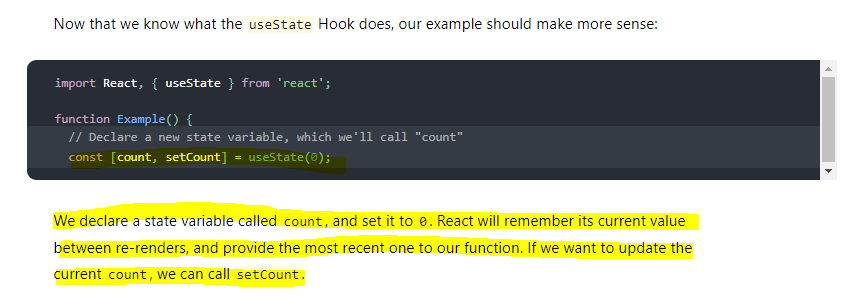
**useState returns something**. **It returns us an array, input state here is an array and this array will have exactly two elements.**

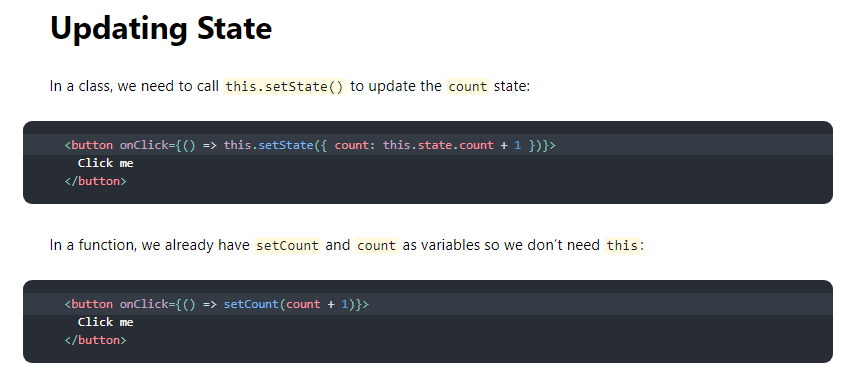
**Now which two elements are that? The first element is the current state. The second element in this input state object here or array to be precise is then a function which we can use to manipulate that state and this function is also given to us by React.**

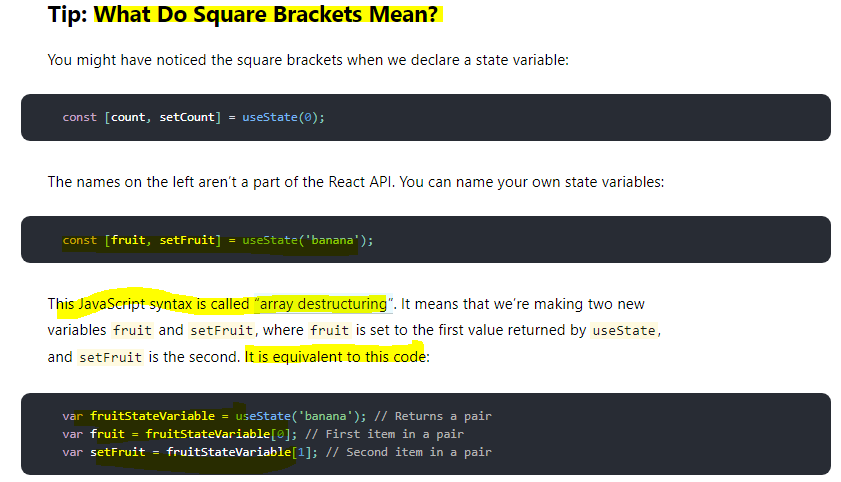




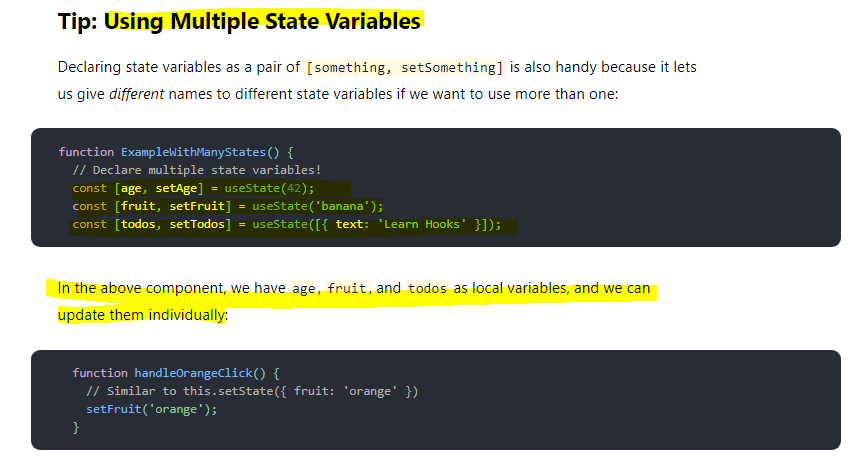


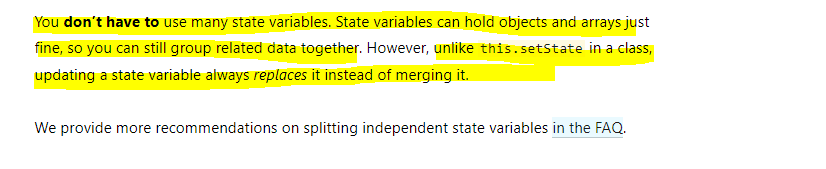


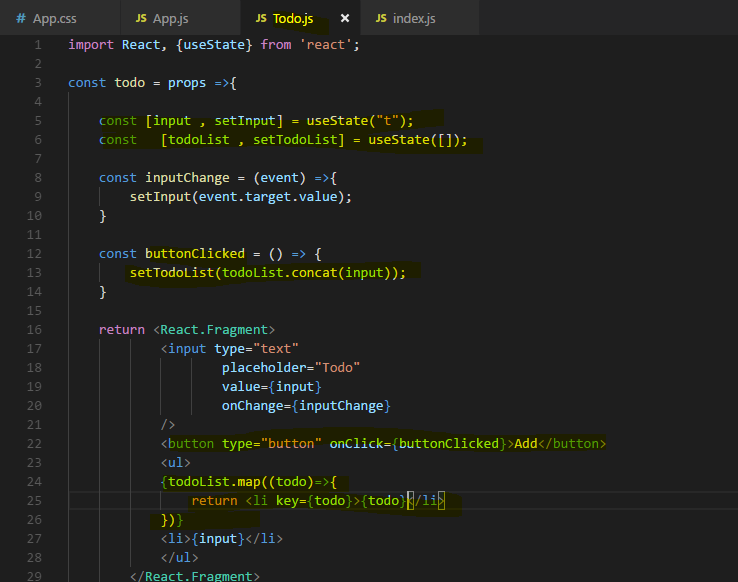




### **Using multiple state Variables**



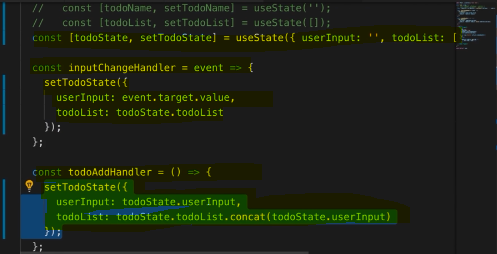


**Example -**

**Using one State Instead**

We now use two instances of the useState function, we call it two times and that is perfectly fine and actually the way I would recommend working with that, that you separate your states across multiple hooks where each hook manages one individual state. Like here, we got the user input and then we get the todoList and of course we use the user input for the todoList but when one of the states changes, it does so independently of the other, when the user enters something in todoInput, we don't need to change the list right away and when we add something to the list, we don't need to change the user input.

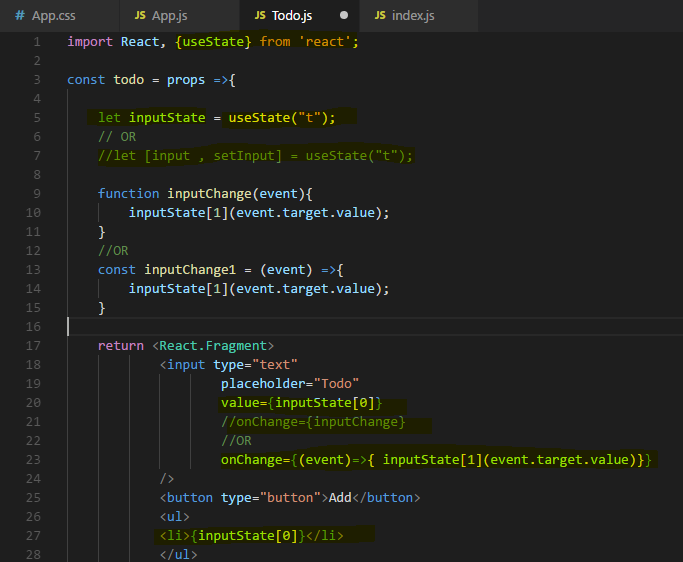
**why having separate states might be beneficial. Because one important takeaway is that unlike set state which took an object and merged it with the existing state, the hook created update function here will not merge whatever you pass in with the old state, it will simply replace the old state with the new one and that is super important to understand. And therefore any merging has to be done manually by you,** **exactly what we do here and that is again a reason why you might want to split it up because manually merging it all the time just leads to more code being written and is of course more error prone.**



**Very Important**

**unlike this.setState in a class, updating a state variable always replaces it instead of merging it.**

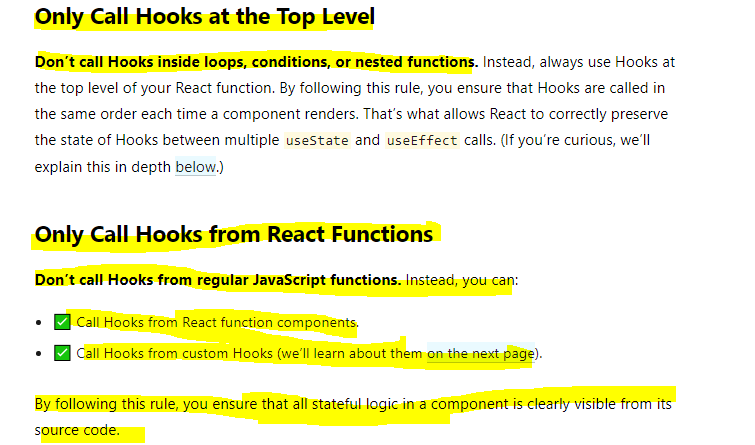
**Real code example**



**we're using this useState function which is coming from React to which we pass our initial state and which then seems to return an element or data which is actually an array with two values and exactly two values and the first value is always our latest state and the second value in input state is a function which we can execute to update this state here with a new value.**

## **Rules of Hooks**

Hooks are JavaScript functions, but you need to follow two rules when using them. We provide a [linter plugin](https://www.npmjs.com/package/eslint-plugin-react-hooks) to enforce these rules automatically:



**rules in mind of which the important ones are that you have to have a React component function and that you have to use hooks at the top level.**

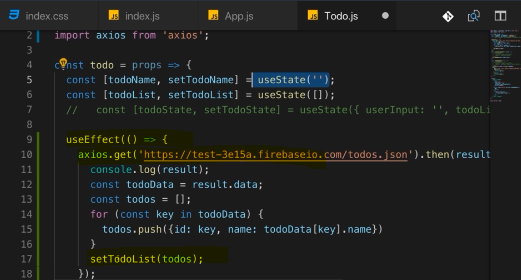
## **useEffect() hook**

**The Effect Hook lets you perform side effects in function components.**

**Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects. Whether or not you’re used to calling these operations “side effects” (or just “effects”), you’ve likely performed them in your components before.**

**If you’re familiar with React class lifecycle methods, you can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.**

**useEffect is also a function which you call and to this useEffect function here, you pass a function that should be executed.**

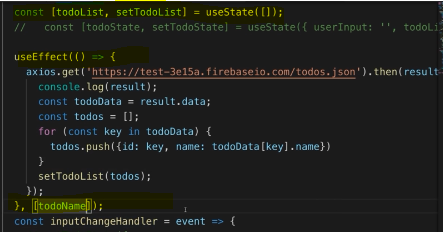


So I added the useEffect hook to cause side effects, it worked but well we caused quite a big side effect because we entered an infinite loop.

**The reason for that is that useEffect does not only run once, like for example componentDidMount did but it runs ofter every render cycle. So we make more and more requests here because I update my state in there, hence we re-render the UI, hence we make a new request, that is of course bad.**

Now how can we avoid doing that?

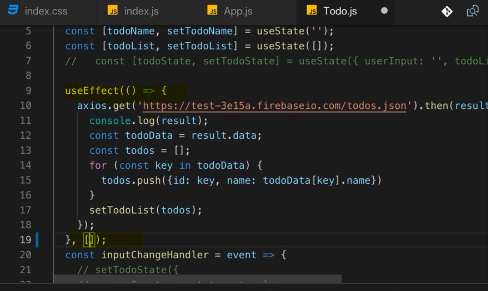
**Well useEffect takes actually two arguments.It does not just take this first argument which is the function it executes, it takes a second argument instead which is an array of values we want to have a look at before it executes this and only if the values we have a look at changed, only in this case this effect should run again.**



**The second element, the second argument you pass to useEffect is an array where you list all the values, all the variables you want to look for and only if a value passed here in this second argument array, only if one of the values you pass in here changed, this function here, this first argument to useEffect will run again.**

**So if you have multiple elements here, multiple things you're watching for, then any change in any item will be enough to run this again.**

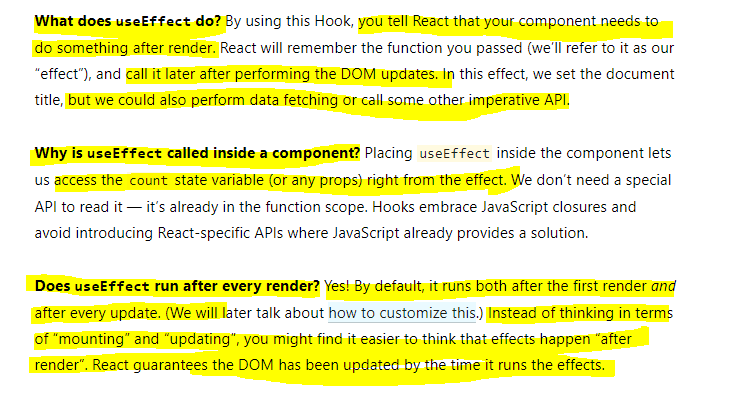
**if you only want to run an effect after only mounting (that is on ComponentDidMount only) well then you pass an empty array here because what you're saying here is this should only run when the items in here, when one of the items in here changed and if you pass no items, well then React hasn't anything to watch and therefore it will not detect any changes and therefore this first argument function will never run again.**

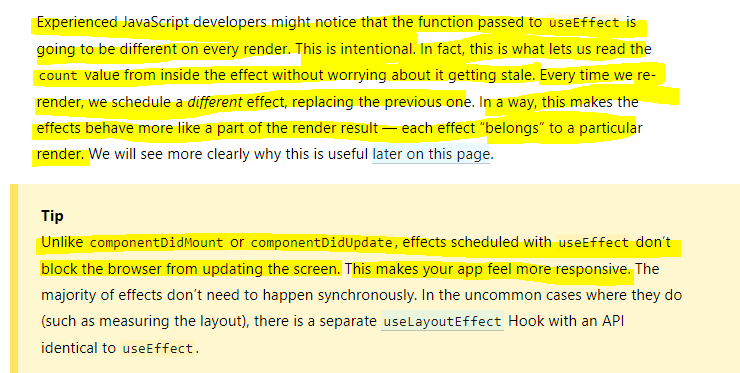


**So if you want to replicate componentDidMount, then pass an empty array, if you want to replicate componentDidMount in combination with componentDidUpdate with an if check included in it, then you should add state variable or other variable because this is now just like your own componentDidUpdate implementation where you also check for whether variable changed.**

**useEffect can be tricky to understand but it's actually straightforward,**

**anything you would have run in componentDidMount or componentDidUpdate before should go in there and with the second argument, you can control when this function should run. If you have no second argument, it'll run for every render cycle. If you have an empty array, it will run for mounting of the component and if you have an array with a value to watch, it will run whenever this value changes.**

**Very Important** 

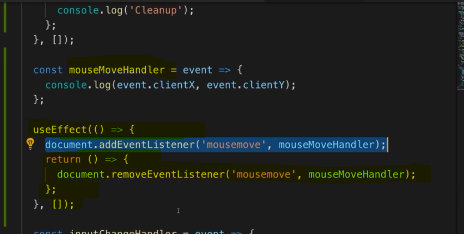


### **Effect Cleanup**

Now what about componentDidUnmount, what if you want to run something whenever a component gets removed or in general, what if you have side effects that require cleanup

work? Maybe not just on component removal but whenever this component re-renders? Well you can do cleanups too.

**You can return something in this function you pass as a first argument. If you add return statement here, then this return statement should also be a function, an anonymous function like this and you don't have to use arrow functions by the way, you can use the function keyword as well and this function will be executed by React on every render cycle too and React will actually execute this as a cleanup before it applies the effect of your main code again. So you can clean up after your last useEffect call basically.**



**this will be executed by React because React automatically executes the function we return in useEffect and it executes it as a cleanup function or as a cleanup work to do before our other code here takes effect or has an effect again.**

**maybe you only want to clean up when the component gets destroyed and you can do that by adding this empty array as a second argument which basically again means this should be executed on componentDidMount and then as an extra feature, the cleanup work will only be done on componentDidUnmount basically, so React will now only act when the component gets loaded and when it gets removed**

