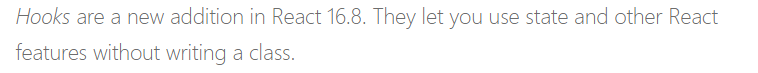
# **React Hooks**

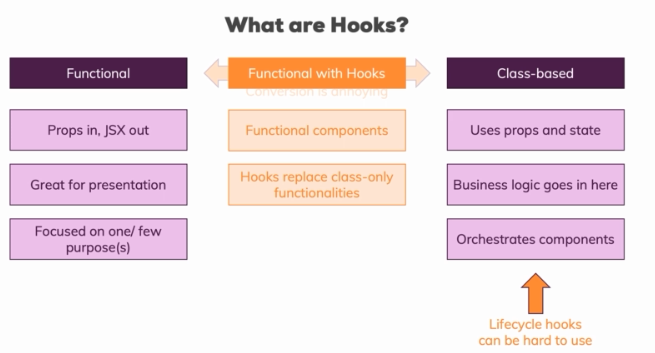




React hooks are a very new feature added to React with version 16.8 and this feature allows you to use functional components only. You can still use class-based components.

React hooks as you will learn in this module give you a new way of writing your React components, you can use functional components only and with the help of React hooks, you can manage state and so on in these components.

## **What are Hooks**



**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.**

class-based components are the components which actually hold the majority of our business logic, so with business logic, I mean things like we make an HTTP request and we need to handle the response and to change the internal state of the app or maybe even without HTTP. A user fills out the form and we want to show this somewhere on the screen, we need state for this, we need class-based components for this and therefore we also typically use class based components to orchestrate our other components and pass our state down as props to functional components for example.

one problem we have with this separation, with all the benefits it adds but one problem we have is that converting from one component form to the other is annoying. It's not really difficult but it is annoying. If you ever found yourself in a situation where you needed to convert a functional component into a class based one, it's a lot of typing and a lot of typing of always the same things, so it's annoying.

**A bigger problem in quotation marks is that lifecycle hooks can be hard to use right**

**I should say. Obviously it's not hard to add componentDidMount and execute some code in there but knowing which lifecycle hook to use, when and how to use it correctly, that can be challenging especially in more complex applications and anyways, wouldn't it be nice if we had one way of creating components and that super component could then handle both state and side effects like HTTP requests and also render the user interface?**

**Well this is exactly what hooks are all about.**

**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.**

## **Enabling react hooks**

**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.**

**So make sure to update your project to React 16.8.0 then delete node modules and in the terminal, navigate into your project, run npm install and this will now install React in that version, into your project and make sure that you can use hooks and all the syntax I'm about to teach you in this module.**

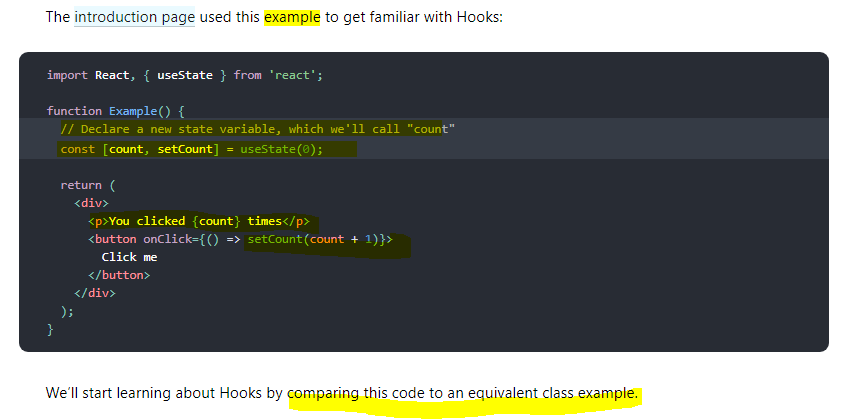
## **useState() hook**

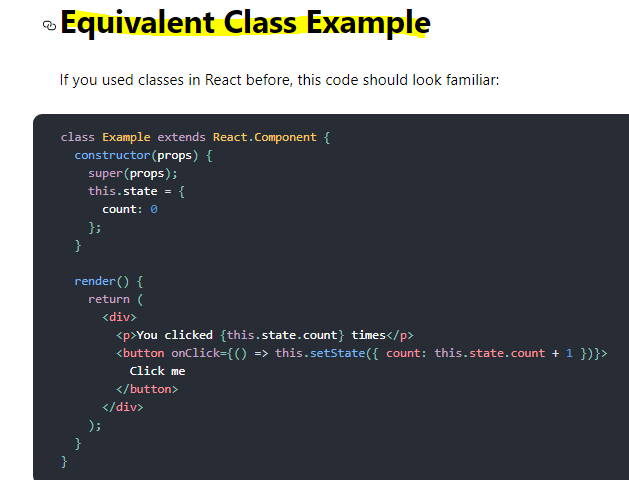
**we import it from the React package and it's called useState.**

**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.**

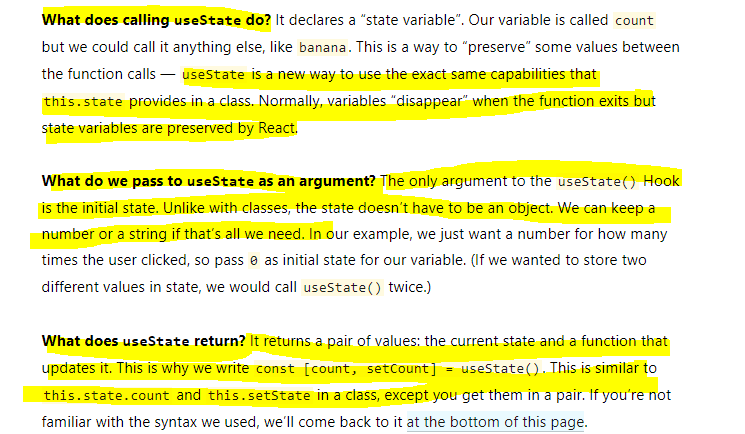
Now this alone doesn't do much because there is no magic connection between calling this function and this input here but **useState returns something**. **It returns us an array, input state here is an array and this array will have exactly two elements.**

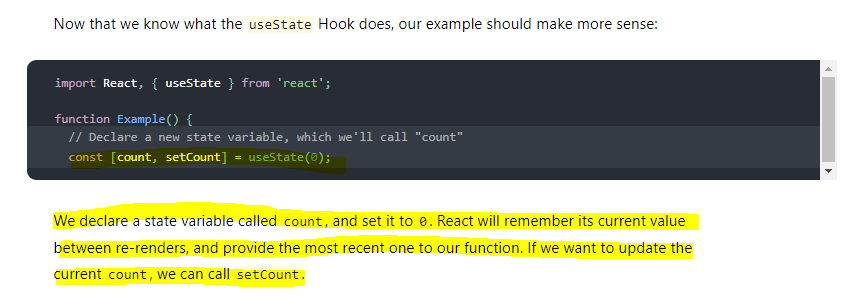
**Now which two elements are that? The first element is the current state. The second element in this input state object here or array to be precise is then a function which we can use to manipulate that state and this function is also given to us by React.**

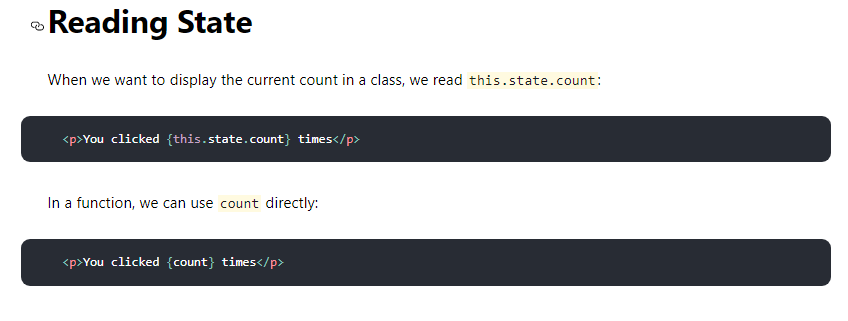


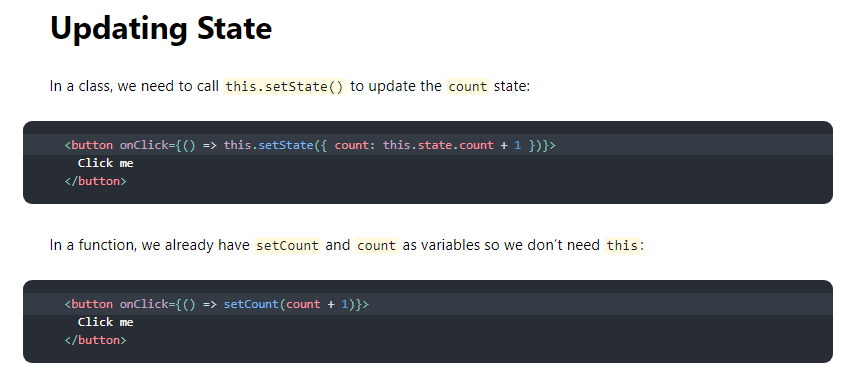




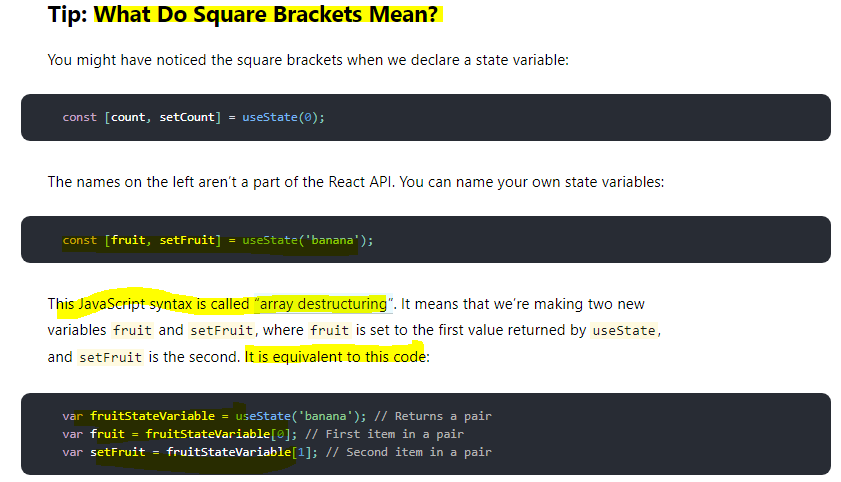
**Very Important** 



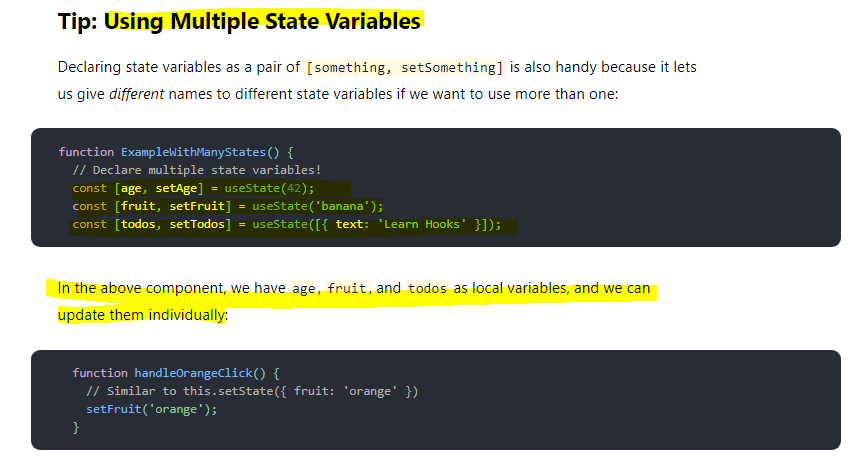


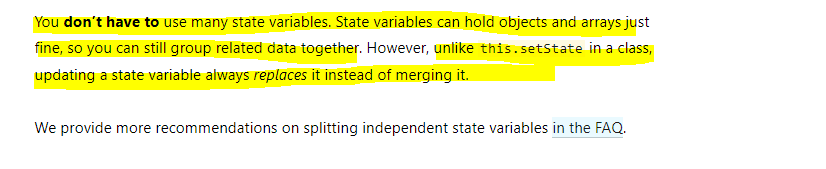


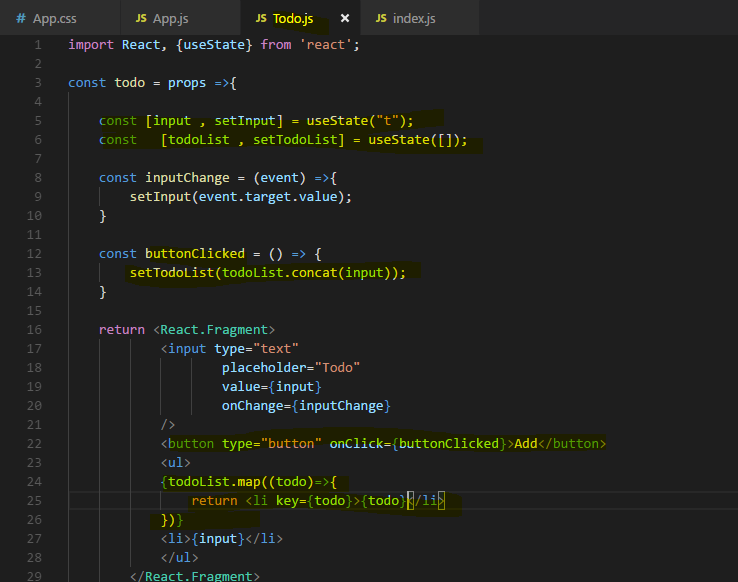
### **Array Destructuring**



### **Using multiple state Variables**





**Example -** 

important, since we set a new value, we overwrite the old one.

So if I want to change my value depending on the previous value, then of course I have to take my previous value into account here and I can safely do this. With set state, you might remember that we had two ways of calling this, right?

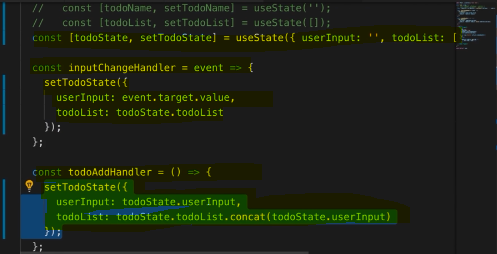
You could call set state like this with an object, with something like count and then count number or we could use the functional version where you got the previous state as an argument, then you returned this mergible state object and there you could then safely access the previous state if you needed that for your state update.

**Now this was required for class-based components, for functional components with hooks, we don't need to do that, instead here when I depend on my old state, my old todoList, I can use it right away.**

**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.

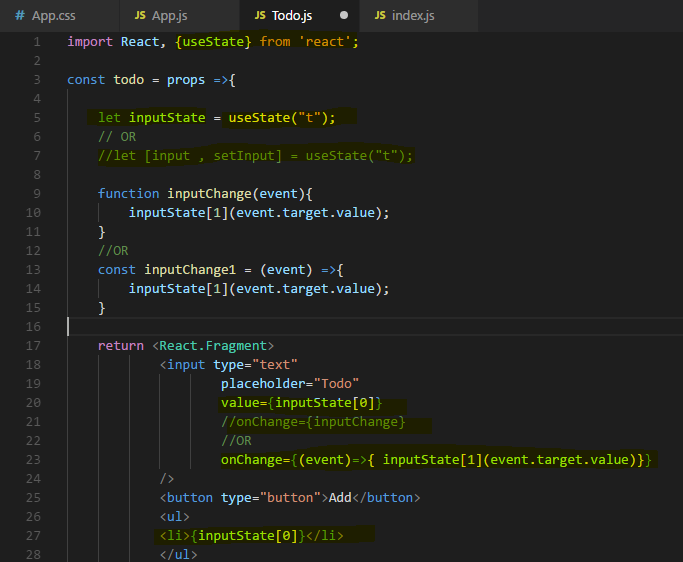
**why having separate states might be beneficial. Because one important takeaway is that unlike set state which took an object and merged it with the existing state, the hook created update function here will not merge whatever you pass in with the old state, it will simply replace the old state with 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.**

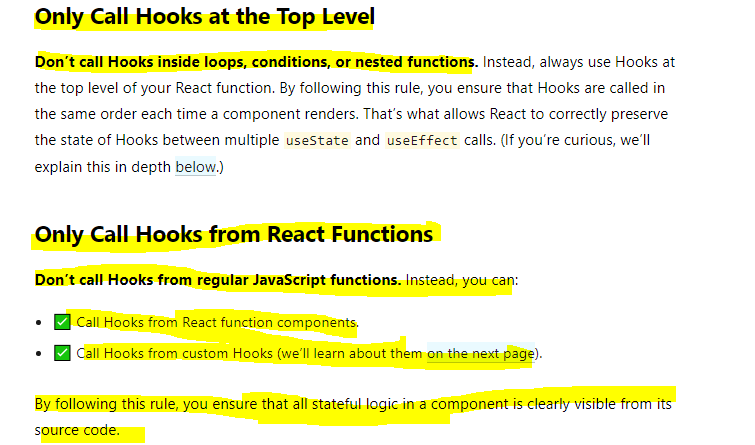
**Real code example**



**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**

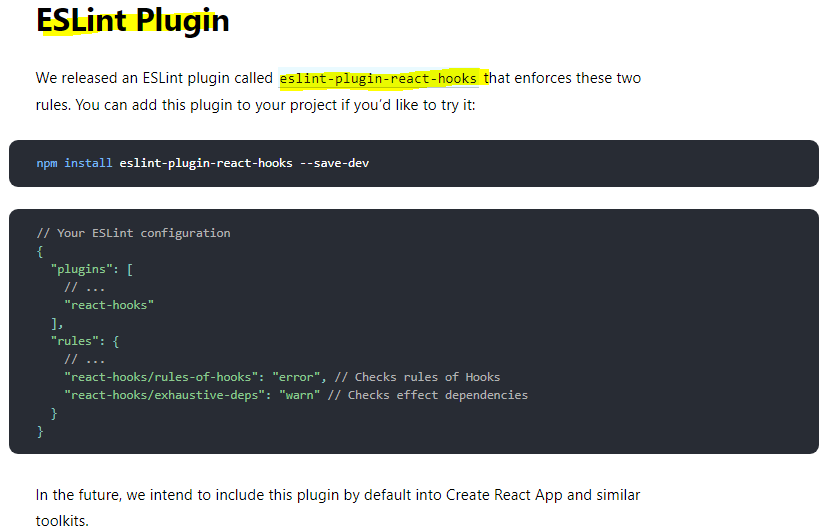
Hooks are JavaScript functions, but you need to follow two rules when using them. We provide a [linter plugin](https://www.npmjs.com/package/eslint-plugin-react-hooks) to enforce these rules automatically:

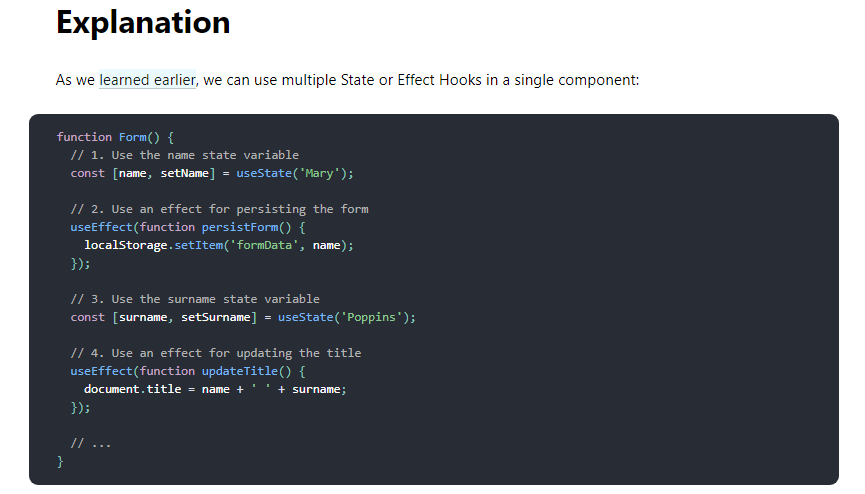


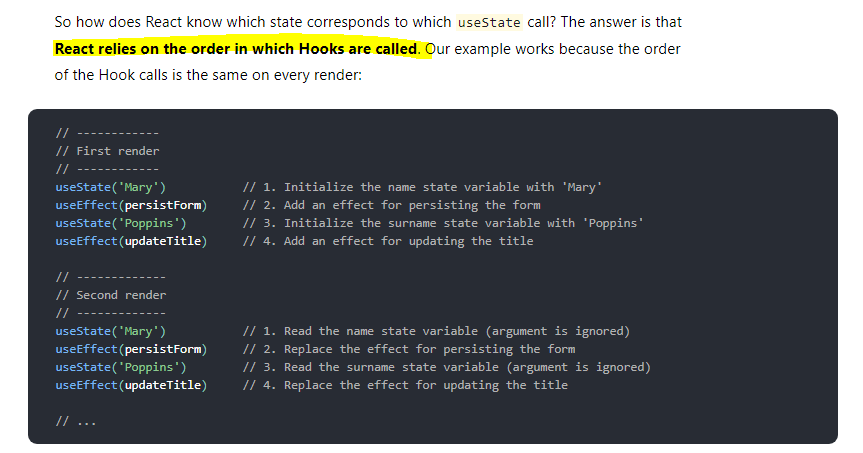
**You must only use useState and all the other hooks at the top level of your component function and that already contains a lot of important things. Important is that it has to be a component function, so such a function which takes props and returns JSX, a function which React can use to render a component. It has to be such a function, you can't use another function and call a hook in there. Well you can call it of course, it's a function in a Javascript function but it will not work correctly, for example state changes will not be picked up by React.**

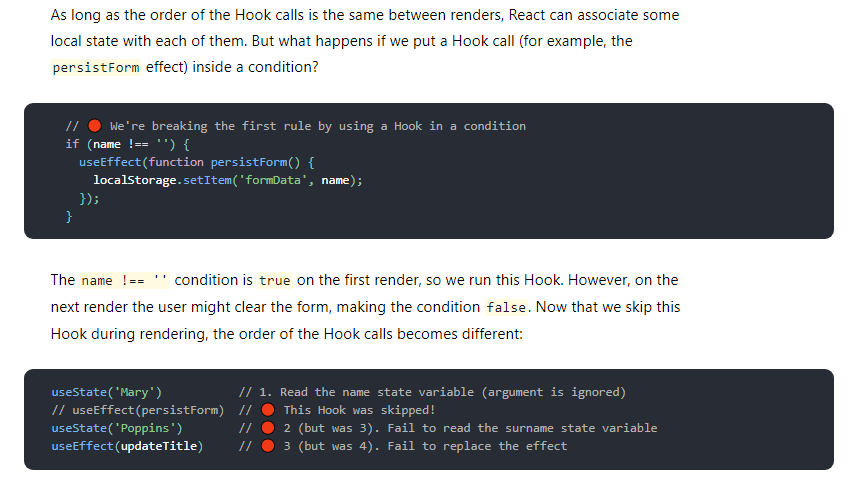
**The next important takeaway is that the top level here has to be guaranteed, you must only call useState directly in your top level of the function body. You must not call useState in here for example, in a function of the function, so in this nested function here, you can't or you must not call it. You must also not call it in an if statement. So if you had a if condition, in there also not ok. The same goes for for loops or any other form of nesting, only call useState at the root level of this function here.**

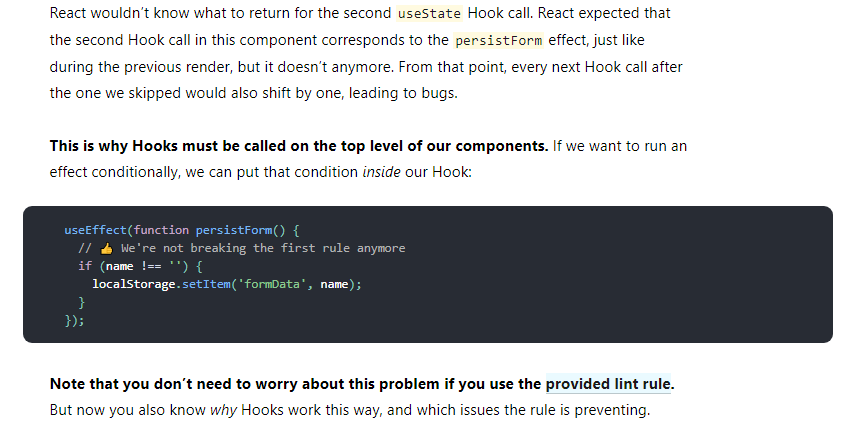
Now with these **rules in mind of which the important ones are that you have to have a React component function and that you have to use hooks at the top level.**











## **useEffect() hook**

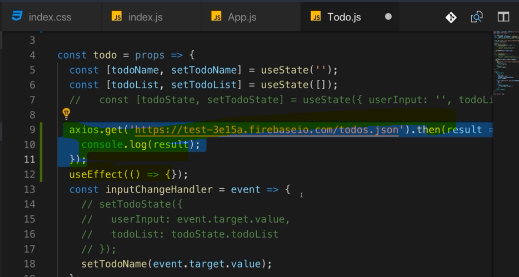
**The Effect Hook lets you perform side effects in function components.**

**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.**

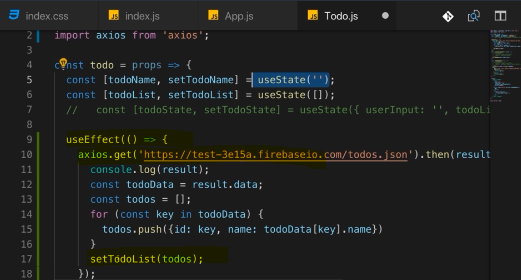
**So this function here will execute after this component renders** and now correctly you might say if I want to execute code when this function renders, I could add it right here, right here, I could add my code there.



We could do that and this code would work but it is a very bad idea to do that.

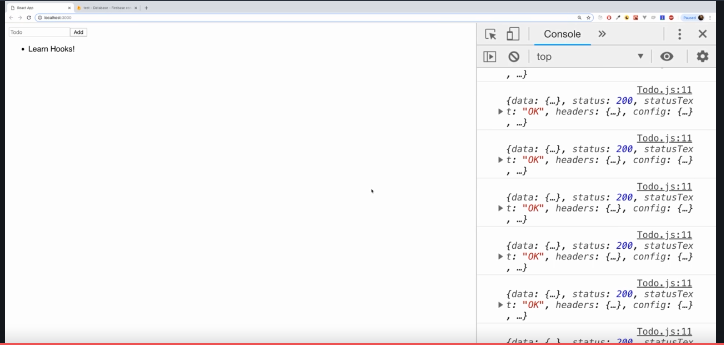
**The problem with calling it here would be that we call it inside of the render cycle and that is bad for performance reasons. React does a lot of behind the scenes magic stuff when it renders our component and updates the DOM and when we do operations like this one which then may change the state once they have a response, when we do them inside of that render cycle, we can end up with a bad performance and unexpected results in the UI because maybe we get this response, the result of this request, we get this when React completed its render cycle and therefore changes might end up not being reflected or we just hit the performance and perform too many updates.**

**As a general takeaway, you should not cause any side effects, so things like make HTTP requests or manipulate the DOM on your own or send analytics data to some server, don't do these things here in the render function, do them in a callback function passed to useEffect instead.**



So this is OK because this again, **this useEffect hook here hooks into React's internals and makes sure that this code executes at the right time which is after this render cycle finished, so that this can run in a high performant way and that the UI is always updated correctly and you don't end up with some strange state changes behind the scenes outside of what React expected or anything like that.**

**And just to make this really clear, calling setTodoList in here is absolutely fine, it would not be OK to call useState in here or to call useEffect in here because of the rules of hooks I mentioned earlier, you must only call hooks at the top level of your function, not in a nested function as this one is and this applies to all hooks, useState, useEffect and any other hook. So setTodoList is OK because this is just a function returned by the hook, it's not the hook itself, not the hook function itself. So we can safely call setTodoList in here, then I pass my todos to that list or I replace the oldest with my new todoList to be precise.**



And now if we save that, this page reloads and it actually loads the data from the back-end but what you'll also notice is that we enter an **infinite loop here**,

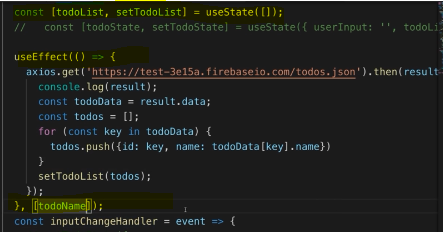
we get more and more responses coming in. So that is a problem which we certainly have to take care about.

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?

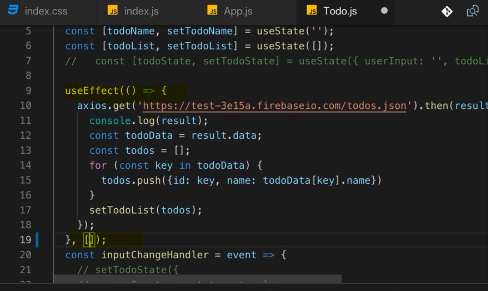
**Well useEffect takes actually two arguments.It does not just take this first argument which is the function it executes, it takes a second argument instead which is an array of values we want to have a look at before it executes this and only if the values we have a look at changed, only in this case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



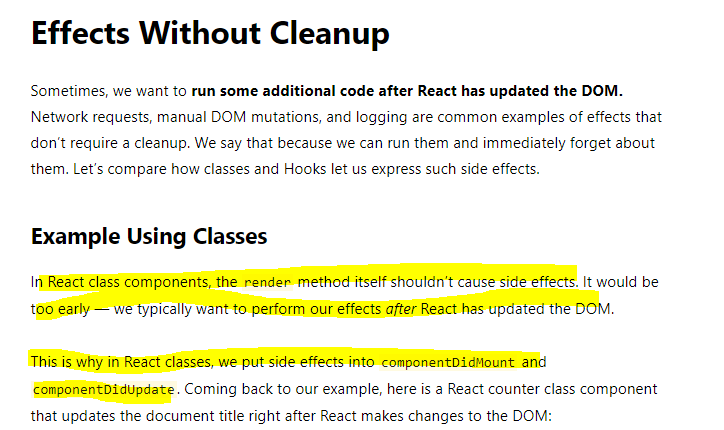
**So if you want to replicate componentDidMount, then pass an empty array, if you want to replicate componentDidMount in combination with componentDidUpdate with an if check included in it, then you should add state variable or other variable because this is now just like your own componentDidUpdate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

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

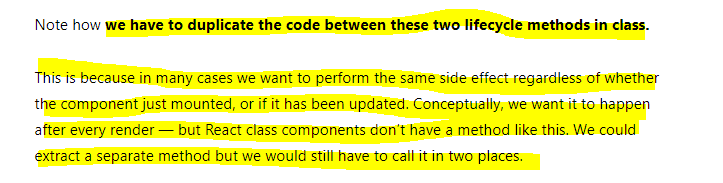
**anything you would have run in componentDidMount or componentDidUpdate before should go in there and with the second argument, you can control when this function should run. If you have no second argument, it'll run for every render cycle. If you have an empty array, it will run for mounting of the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

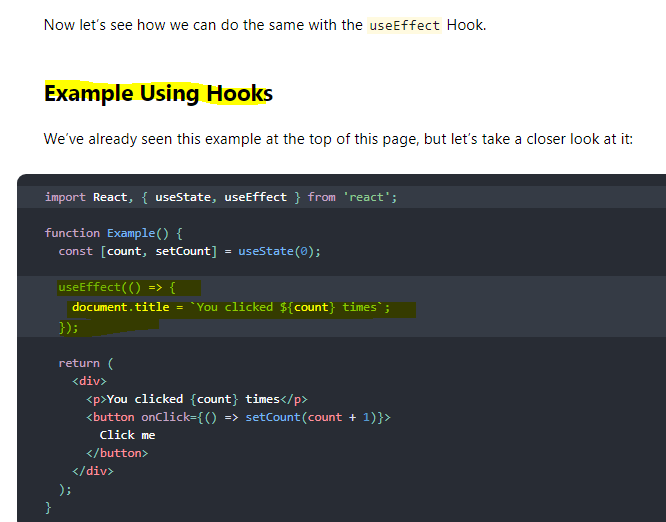
### **React website documentation**

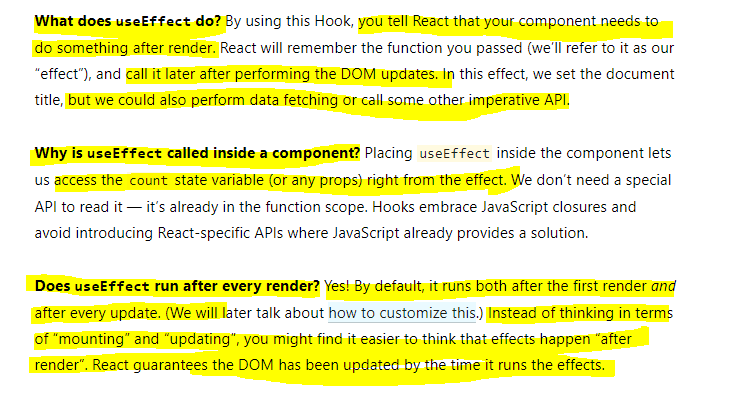
There are two common kinds of side effects in React components: those that don’t require cleanup, and those that do.

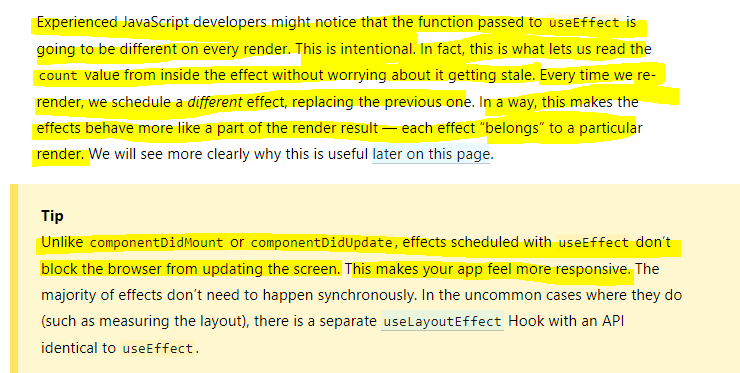








**Very Important** 

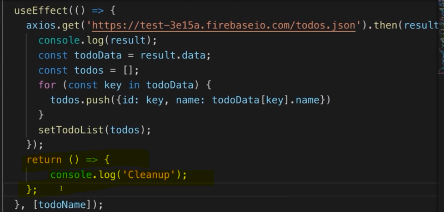


### **Effect Cleanup**

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.**



Now obviously for a normal HTTP request which we fire like this and where we don't need todo any clean up even if we re-render early, this doesn't make too much sense

but consider an event listener and let me show you an example for that actually.

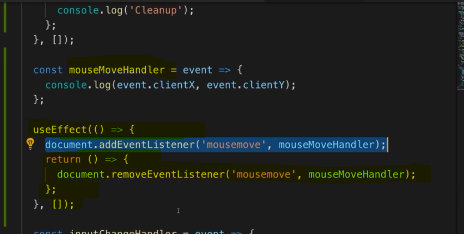
Consider we have another effect, just as we can have multiple useState calls,

**we can have multiple effects and in this effect here, let's say we have an event listener we want to add to our document. We want to listen to mousemove events so when the user moves the mouse because maybe we're collecting that data and we send it to our analytics server so that we can track where the user moved the mouse on our page.**

**you'll see tons of stuff being printed on the right, these are the coordinates of my mouse.**

**Now I might want to clean this up and not apply a new effect whenever this page reloads because if I type, you see, watch the numbers next to my coordinates, more and more calls, more and more console log statements and that simply happens because I add more and more event listeners which is awful for performance and absolutely has to be avoided.**

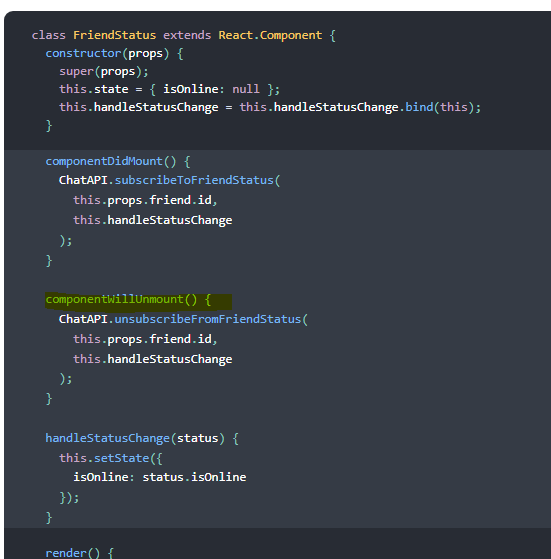
**So we certainly want to clean up our old listener before we attach a new one.**

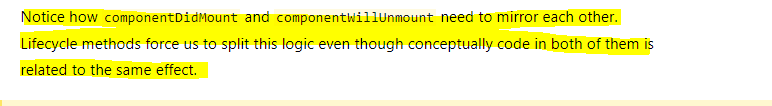


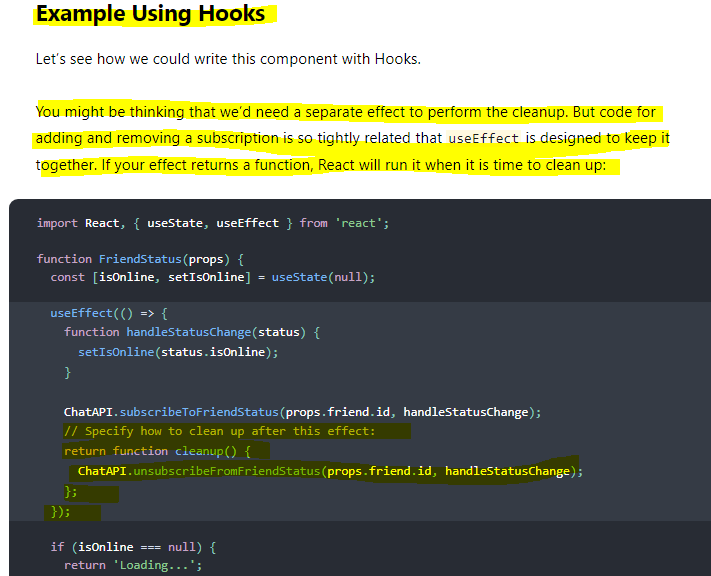
**this will be executed by React because React automatically executes the function we return in useEffect and it executes it as a cleanup function or as a cleanup work to do before our other code here takes effect or has an effect again.**

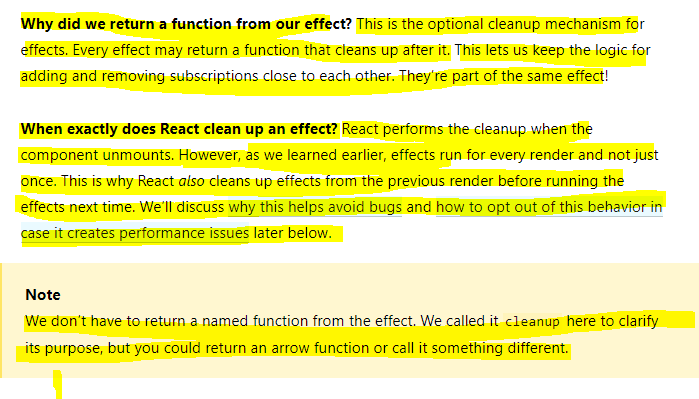
**maybe you only want to clean up when the component gets destroyed and you can do that by adding this empty array as a second argument which basically again means this should be executed on componentDidMount and then as an extra feature, the cleanup work will only be done on componentDidUnmount basically,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

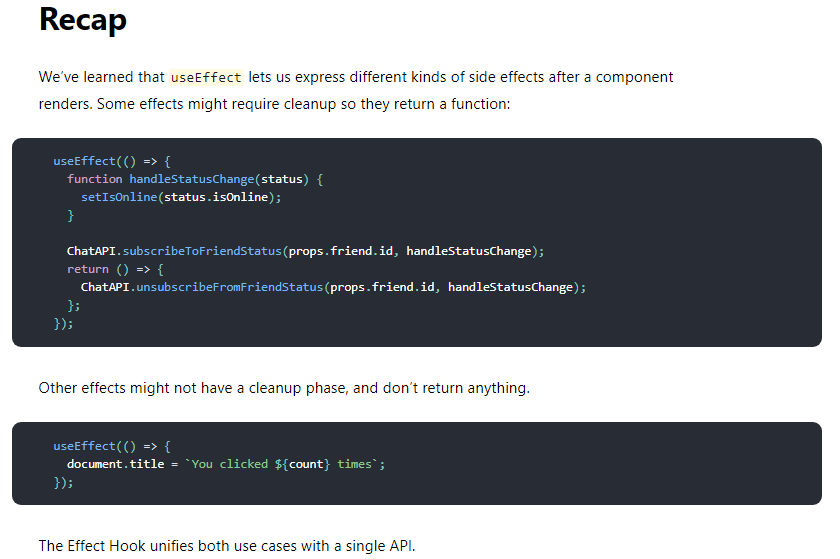
#### **React Website Document**











## Tips for Using Effects

