## **HOOKS**

### **Introduction**

As mentioned in the previous lesson, the lifecycle methods such as componentDidMount, componentDidUpdate and also the render method can only be called in class components. This was a reason why functional components were for long referred to as “dumb” or “stateless” components. They were usually just used to return some JSX because of their inability to have a state or lifecycle methods. Since the introduction of Hooks, this has changed completely.

Hooks allow functional components to also have a lifecycle as well as a state.

Other than the basic hooks you will encounter in this section, there are many other hooks such as useRef or useContext, just to name two. React also provides the ability to write your own custom hooks. This changes the game between functional vs. class components. Functional components are no longer considered “dumb” components. Many programmers have increasingly begun to prefer functional components over class components. For more information have a look [here](https://dev.to/danielleye/react-class-component-vs-function-component-with-hooks-13dg)

Now we will discuss the most basic hooks. Create a create-react-app and use the App.js file for the following examples. This won’t actually create a lot of functionality, but when coding along you will remember everything better thanks to muscle memory.

### **useState**

So finally, as already mentioned in an earlier section, we will be coming back here to discuss how to use state in functional components. The useState hook makes it possible to declare a state in functional components. Here is an example on how to use it:

import React, { useState } from "react"; const App = () => { const [count, setCount] = useState(0); const incrementCount = () => { setCount(count + 1); }; return ( <div> <div>{count}</div> <button onClick={incrementCount}>Increment</button> </div> ); }; export default App;

Go try this out in the browser.

First, we have to import the useState hook from react.

Then we can declare a state like this: const [count, setCount] = useState(0). This is the syntax to declare a state. The name “count” and “setCount” is totally up to us, we could also call it “something” and “somethingElse” but this wouldn’t be very good, because then “somethingElse” would be called to set “something” in the state. It doesn’t read as easily. The convention is to always call the second value like the first, just with a “set” in front. The useState(0) call initializes our state with the value we set in brackets (0).

Afterwards we are declaring a function, which right now just sets a new count. In this case we are incrementing the current count with one. This function will be called anytime we click the button. In the div above the button we are just displaying the current state. One thing to mention and keep in mind here is that setting the state is an asynchronous task. Additionally, setting the state calls a re-render. That’s the reason we see the new count when clicking the button. If it wouldn’t trigger a re-render then we would still see the old count. Our component will update after setting the state. As mentioned in our last lecture, we might want to use this opportunity of updating to do some tasks. Unfortunately, we don’t have any lifecycle methods in functional components, right?

### **useEffect**

Well, we don’t have any lifecycle methods such as componentDidMount, componentDidUpdate or componentDidUnmount, but we do have something better. We have useEffect, which can actually do everything the above mentioned lifecycle methods can do. Let’s have a closer look.

import React, { useState, useEffect } from "react"; const App = () => { const [color, setColor] = useState("black"); useEffect(() => { const changeColorOnClick = () => { if (color === "black") { setColor("red"); } else { setColor("black"); } }; document.addEventListener("click", changeColorOnClick); return () => { document.removeEventListener("click", changeColorOnClick); }; }, [color]); return ( <div> <div id="myDiv" style={{ color: "white", width: "100px", height: "100px", position: "absolute", left: "50%", top: "50%", backgroundColor: color, }} > This div can change color. Click on me! </div> </div> ); }; export default App;

Try it out to get an idea of what is happening.

Once you’ve done that, let’s go through it in all detail.

You should already be familiar with the useState logic implemented. Basically, we are just saving a color in state and on click we are changing the color saved in state. This color is applied to our div element. Don’t worry about all the inline styles of the div, the only important thing within this styles object is where we set the background to the color that is currently saved in state.

Now the only thing we haven’t looked at is the useEffect hook. As mentioned, this hook can do a lot.

The syntax is as follows:

useEffect(() => {}, [])

In the curly brackets you can write the code that will be executed. The dependency array at the end is optional, however, you will include it more often than not. A dependency is any state, prop, context that is used within the useEffect callback. You can also include state or props that are not. useEffect will trigger based on changes in the dependencies listed. ESLint will warn you if it expects a dependency, but one is not added, however, this is a warning and they are not required.

You have three different options for the dependency array:

1. Leave it empty. If you leave it empty the useEffect hook would look something like this:

useEffect(() => { // Do something }, []);

This option is equal to a componentDidMount lifecycle method, meaning the hook runs one time when the component mounts (is inserted in the DOM tree)

1. Add a dependency to the array. Like we did it in our example code.

useEffect(() => { // Do something }, [color]);

This way, the useEffect hook will re-run anytime the dependency (color) changes. This is similar to a componentDidUpdate method, with the only difference that it only runs when a certain condition has changed.

1. Leave out the dependency array.

useEffect(() => { // Do something });

You can also completely leave out the dependency array. This way, the useEffect hook runs anytime the component is updated, AND right after the initial render. This is the difference compared to the componentDidUpdate lifecycle method, because it also runs after the initial render. This way it would be equal to a componentDidMount and componentDidUpdate method combined.

The return statement we are adding to our useEffect is actually equal to a componentWillUnmount method.

return () => { document.removeEventListener("click", changeColorOnClick); };

If you write a return statement like the above in a useEffect, it will do the same as a componentWillUnmount method. As you can see, there is a lot to the useEffect hook. You can also create your own custom hooks if desired. However, with the above mentioned hooks useState and useEffect you will be fine in most of your smaller projects.

### **Assignment**

1. The React documentation provides a whole section on hooks. Read through all steps [here](https://reactjs.org/docs/hooks-intro.html). You can also always refer back to this page if you get stuck. The main idea is to understand the concepts, the details come with time.
2. Now go back to your CV Project and rewrite it using functional components and hooks.

### **Additional Resources**

* [Here](https://www.ohansemmanuel.com/react-hooks-documentation-easy-to-read/) is another article about hooks, which provides a simpler version of the official React hooks documentation.
* [Watch this video](https://youtu.be/-MlNBTSg_Ww) for a look into useState(), useEffect() and custom hooks in great detail.