NOTE: This book is currently placed at the end of React Native.

Voice:

This will be a brief introduction to using hooks in React, specifically the useState and useEffect hooks. You may have heard of the React Hooks API, which showed up as an experimental feature a couple years ago and by now is part of standard React usage.

Here at Nucamp, we will be converting our curriculum to using React hooks, as well as making some other updates along the way. But updating all the videos and lessons takes time, and we know many of you are eager to learn hooks. So we are releasing a few videos now specifically to help Nucamp students who have been through our pre-hooks curriculum get acquainted with hooks. Later, when we do update our entire curriculum, you'll be able to come back and get access to those updates as well. Remember, you have perpetual access to the Nucamp curriculum updates and the Nucamp Slack, so your journey with us doesn't ever have to end.

# A Brief Introduction to Hooks

**[TITLE] SLIDE 1: A Brief Introduction to Hooks**

**Screencast:** VS Code showing function component and class component example

class ExampleClassComponent extends Component {

render() {

return <h1>Example Class Component</h1>;

}

}

function ExampleFunctionComponent() {

return <h1>Example Function Component</h1>;

}

**Voice:**

In the React course, you learned about function components and class components. You learned how to write class components [screencast: spotlight class component] like this...

And you learned how to write function components [screencast: spotlight function component], like this...

A major motivation behind the creation of hooks was to get rid of the need for writing class components [in screencast: comment out class component], in favor of always using function components.

**SLIDE 2: What are hooks?**

* The React Hooks API provides us with built-in functions, called hooks
* Hooks let function components 'hook' into React state and access lifecycle features, which previously only class components could do
* Simpler and easier to write and manage than class components
* You can also write custom hooks
* Many third party libraries such as react-redux and react-router have already implemented custom hooks in their APIs

**Voice:**

The React Hooks API provides us with built-in functions that are called hooks. These let us use function components to 'hook' into React state and access lifecycle features, both of which previously only class components could do. And, as you will learn, they let us do it in a way that's simpler and easier to write and manage.

Along with the built-in hooks, you can even write your own custom hooks. Many third-party libraries are already using this feature to add custom hooks to their APIs. The React-Redux library for example now has Redux hooks that make dispatching actions and accessing the Redux store much easier. The React Router library has its own custom hooks too, and so on. Hooks have caught on like wildfire across the React ecosystem, and once you learn them, I think you're really going to like them.

**SLIDE 3: Class components**

Q: Is what I know about class components useless now?

A:

* No! There are countless pre-existing React components written as class components. There are no plans to remove support for class components.
* The React team recommends writing new components as function components with hooks instead of classes, but that there is no need to go back and rewrite pre-existing class components.
* You will continue to encounter class components in existing React codebases, and you will need to know what they are.

**Voice:**

You might be thinking of this question, what about all that stuff I learned about class components? Is that all useless now? Absolutely not. Remember, React was introduced in 2013, and there are many thousands of class components in React code out in the world. The React team has specifically said that there are no plans to remove support for class components. While they recommend writing new components as function components with hooks, they are not encouraging anyone to waste time going back and rewriting all those class components.

What that means is that even if you don't write class components yourself, it's more than likely that as a React developer, you will encounter and work with older code that uses class components, and you're going to need to know what they are.

**SLIDE 4: In the following lessons**

useState and useEffect

Two major rules of hooks

Converting class components

**Voice:**

In the following lessons, I will introduce you to the useState and useEffect hooks, two of the most important built-in React hooks. Let's go.

# Review: Component State

I want to start with a review of two concepts: React component state, and array destructuring syntax. It will make learning useState much easier.

If you feel like you already understand these concepts well enough and you want to go straight to the hooks part, you can skip ahead to around the \_ minute mark.

**SLIDE 1: Review: What is component state?**

* When using Redux, global state is kept in the Redux store, and any component can connect to it to access that state
* The state for useState is different from Redux state; it is (local) component state that React itself stores for a specific component
* With class components, this component state is created using the constructor method and an object named this.state

**Voice:**

What is state all about?

When you learned to use Redux, you learned how to keep global state in the Redux store, where any component can connect to the Redux store to access that state.

With the useState hook, we are not talking about Redux state. Instead, we are talking about component state, which is data that React itself stores for a specific component. This can also be called local state, because it's local to that component.

You learned in the React course that in class components, you can create local component state using the class constructor method and a this.state object that can hold multiple state properties.

Let's look at an example of a class component that uses component state.

Inside this class component's constructor, I've created a state value named color and initialized it with the string 'red'.

I've created a handleClick() method that uses React's this.setState method to change that color to blue, and I've bound that method in the constructor.

class App extends Component {

constructor(props) {

super(props);

this.state = {

color: 'red',

};

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

this.setState({ color: 'blue' });

}

render() {

return (

<div>

<h1 style={{ background: this.state.color }}>

React Component State Example

</h1>

<button onClick={this.handleClick}>Click Me</button>

</div>

);

}

}

Inside the render method, I'm returning this div that contains an h1 element and a button.

I've applied a style to set the background color of the h1 element, to whatever color is in this.state.color.

Then I've set up the button to call the handleClick method when clicked.

[screencast: bring in browser]

Here it is in the browser. This red background will turn blue when I click the button, like that.

Now we could take for granted that when I click this button, and the handleClick method sets this.state.color to blue, the browser will magically update with that color. But there is a reason that the browser updates when the color is changed, and that reason is this.

*All React components will automatically re-render if any of the state or prop data in it changes.*

That's how React is designed to work. When state or prop data changes, React re-renders any affected components *and* any direct child components, and updates the browser DOM as needed.

This is important to understand and to not take for granted. It doesn't happen on its own, React makes this happen.

This may make more sense if I show you what happens if I don't use state and setState.

Let's say I just have a normal class variable, this.color. Because I'm not using this.state and setState, React will not recognize this as part of the component state.

this.color = 'red';

In the handleClick method, I'll replace this.setState with this line to set this.color to blue. With state values, we must use setState instead of assigning a new value, but we can't use setState for non-state values.

Let me add a console.log here, to prove to you that the value of this.color changes:

console.log(this.color);

Then I'll use this.color here instead of this.state.color, for the background of the h1.

<h1 style={{ background: this.color }}>

I click the button, and the background color does not change this time, but if you look at the console, you can see that the handleClick method did run, because it logged the value of this.color as blue. This is because we've changed the value, but it's not a state or prop value that's been updated via setState, so React won't automatically re-render this component.

Now, we could use JavaScript DOM manipulation methods to set the background color directly, but this is not the way that React is designed to work. React uses a declarative approach where we tell React what we want to see, using JSX, state, and props, then React handles updating the DOM for us in a consistent, optimized way.

~~I mentioned earlier that React component state is different from Redux state. Redux state is generally used to store application data that's used by multiple components. For larger applications especially, it's very useful to have a central repository for that data, and a structured way to access and update that data. It's not the only way to manage application data, but it's a time-tested and popular solution.~~

~~React component state, on the other hand, is typically used for non-critical UI state for a single component. For example, you used it in the React nucampsite project to keep track of whether a modal was open or closed. [show that code]~~

~~One thing I haven't mentioned is that state is used for values that can change. If it never changes, we wouldn't need to keep any state for it, we could just say 'red' or 'blue'.~~

In summary, component state values, and the props that are created with them, are special values in React because any updates to them will cause components to automatically re-render.

Now let's review the array destructuring syntax.

# Review: Array Destructuring Syntax

**SLIDE: Review: Array Destructuring Syntax**

* Not React-specific; it is a part of vanilla JavaScript
* Related to the object destructuring syntax
* A shortcut to assign array items as values for variables

Array destructuring syntax is not a React specific syntax, it is a part of vanilla JavaScript.

It is related to the object destructuring syntax, which we have used more often during your bootcamp. It is used as a shortcut to assign array items as values for variables.

We will use it with the useState hook, so let's try to understand what it does and how it works.

For this, I'll use my browser's JavaScript console to demonstrate.

Let me initialize an array with three values. These values could be numbers, objects, even functions, anything that can be stored in an array. Yes, functions can be stored in arrays.

For this example, I'll use strings:

**SCREENCAST**: const myArr = ['apple', 'banana', 'cherry'];

Let's say I want to take each item in this array and assign it to its own variable.

I could do this.

**SCREENCAST:**

const first = myArr[0];

const second = myArr[1];

const third = myArr[2];

I'm using the array index, 0, 1, and 2, to explicitly assign each array item to a const variable.

Now let's check what each of these variables contain. In the JavaScript console, I can do that just by entering each variable name.

**SCREENCAST**:

first

second

third

So that works...

But we can do better. Let's try that again. [screencast: refresh]

And let me show you how array destructuring syntax can be used as a shortcut to accomplish exactly what we just did.

I'm going to refresh the console, which clears all variables so I'll create myArr again.

const myArr = ['apple', 'banana', 'cherry'];

Then I'll use a variable declaration keyword, I'm using const but you could use let as well, and for the variable name I'm going to put all three names inside square brackets, separated by commas. Then I'll say equals myArr.

const [first, second, third] = myArr;

If this worked, then the variable first should contain the first array item, the variable second should contain the second array item, and so on.

And you can see that worked.

Now, these names I chose to use, first, second, third, aren't anything special. I could use any valid variable name and grab those array item values again, like this:

const [jack, john, jim] = myArr;

And the value of jack should be apple, the value of john should be banana, and the value of jim should be cherry. Why? It's the order that matters. Whatever variable name is first on the left side, gets the value of the first item in the array, and so on.

Let's say I don't care about the third item in myArr and I just want to assign the first two items to a value. Can I do that? Yes, like this: [refresh] I can just leave out the third variable name.

const myArr = ['apple', 'banana', 'cherry'];

const [first, second] = myArr;

first

second

What if I don't want the first item, and only the second and third? In that case I can skip the first item with a comma like this:

[refresh]

const myArr = ['apple', 'banana', 'cherry'];

const [ , second, third] = myArr;

So to summarize, the array destructuring syntax is a shortcut for assigning items from arrays as values for variables, by declaring those variable names inside square brackets on the left side of the assignment operator, in the same order as the items in the array on the right side.

# Introduction to useState

**SCREENCAST: File Explorer - NucampFolder**

**VOICE:**

For this demo, I'll start by scaffolding out a React app using npx create-react-app. If you want to follow along, you can do this as well. Open your Git Bash or Terminal application to the folder where you want to create this project. I'll do this in my NucampFolder, and I'll enter npx create-react-app react-hooks-demo.

At the end of the installation, and I will fast forward here, if I type ls, I can see there's a new folder named react-hooks-demo.

I can open this folder in VS Code from the command by typing code space react-hooks-demo. Or you can choose to open this folder via the File menu in VS Code. If you are following along, however you do it, make sure you now have the react-hooks-demo folder open at the top level in VS Code. [screencast: spotlight VS Code Explorer]

I'm going to use App.js for this demo because it's already set up as the root component, so let me open that from the src folder. I'll go ahead and delete the code we don't need.

Let's return an h1 element from this component, containing the text React Hooks Demo

<h1>React Hooks Demo</h1>

I'll start the app with yarn start, and here it is in the browser. So far, this is a very basic React function component without hooks.

Now let's implement the useState hook. First, we'll import the useState hook from react.

import { useState } from 'react';

There are two main rules that need to be followed by all hooks, whether built-in or custom hooks. The first rule is this: Hooks can only be used inside React functions.

By React function, I mean either a React function component, or a React custom hook function. They won't work anywhere else. So you cannot use hooks inside React class components.

We'll get back to the second rule of hooks later. Let's get back to the useState hook.

Let's say that I want to use component state to update this h1 element's background color, as you saw me do earlier with the class component in the module where I reviewed the concept of component state.

Let's see how we can do this with the useState hook.

The useState hook is a function, so I call it with an argument list like this. And I'll pass it one argument, which will be used as the default value for the state variable I want to set, and I'll use the string 'red'.

Now, I happen to know, from reading the docs, that the return value from this call to the useState hook function is going to be an array with two items in it. The first item in that array will be the current value for this state that we are creating, which we initialized to 'red'. The second item will be a function that we can use to update that value, which I'll call the updater function.

Knowing this, I can use the array destructuring syntax to assign those values to a variable and function name. And I'm going to choose to call them color, and setColor. These names are arbitrary and decided by me, they could be anything, but the convention is to name the updater function the same as the state variable name, prefixed by set and camel cased.

const [color, setColor] = useState('red');

This one line of code initializes a local component state value as 'red', and creates a variable name we can use to access that value, which I named color, and a function we can call to update that value, which I named setColor.

Let's try using this color variable to set the background color for the h1 element.

<h1 style={{ background: color }}>React Hooks Example</h1>

In the browser, you see that that worked.

Now what if I want to change the background color?

Let's set up a button element with an onClick event handler, let's call it handleClick.

<button onClick={handleClick}>Click Me</button>

Since I can only return one top-level element from a React component, I'll wrap these two in a div.

Now let's set up that handleClick function, which will be called whenever this button is clicked, and it will use the updater function we got from the useState hook to update the color state value to blue.

function handleClick() {

setColor('blue');

};

If all went well, when I click this button, the red color should turn to blue. There we are.

const [color, setColor] = useState('red');

function handleClick() => {

//setColor('blue');

color === 'red' ? setColor('blue') : setColor('red');

}

return (

<div>

<h1 style={{ background: color }}>React Hooks Example</h1>

<button onClick={handleClick}>Click Me</button>

</div>

);

Let's say we want the color to toggle between blue and red when we click the button. We can use a ternary operator to write the logic for that like this. We'll check if the value of the 'color' variable is triple equal to the string 'red'. If it is, then we'll use setColor to set the value to 'blue'. If it's *not* red, then we'll use setColor to set the value to 'red'.

In the browser now, every time I click on the button, it should toggle between red and blue. Just like that. And if I refresh this page, since color is initialized with the value of red, it should go back to being red.

What if we want more than one state value? We can use the useState hook as many times as we want in a component. So let's say we want to set some text to appear here when the button is clicked. Let's give it an initial value of an empty string. And let's give this value a name of msg, and its updater function a name of setMsg.

Now we can use this msg variable here:

<h1>{msg}</h1>

Let's use the same handleClick function to update it. When the button is clicked, the text will be updated to say "Super Secret Message".

setMsg("Super Secret Message")

Now if I click this button, both the color and msg state gets updated, the component re-renders, and we see the color change and the super secret message.

---

Now that we have two useState hooks being used here, we can talk about the second rule of hooks.

There are only two, and you already learned about the first one: that hooks can only be used in react functions.

The second rule is this. Hooks must always be called at the top level of the component. Don't call hooks inside any loops, conditions, or functions inside this component.

This is because React depends on the order in which hooks are called to always be the same, every time the component is rendered, in order to access them correctly.

But if you have a hook inside an if condition, for example ... then the first time the component renders, this hook might be rendered first, and this one second. Then the next time, this hook might not be called because the if condition changed to false, so this hook is the one that gets called first, and that will just result in chaos, so let's just try to avoid that by always remembering this rule of hooks. Only use hooks at the top level of the component.

You have now learned the basic use of the useState hook, and you've learned the two rules of hooks. In the next video, we'll take a look at how to convert stateful class components to stateful function components using hooks.

----

# Converting Stateful Class Components to Functions

To help you better understand how component state is set and changed in class components vs function components, I'll demonstrate a conversion from one to the other.

First, we change this from a function to a class that extends the Component class from React.

I'll import Component from react and stop importing useState.

Then we need to set up a constructor, with super(props), then this.state =

Now in the state object we add color and msg as properties, and set their initial values

Then we have to bind the handleClick method...

Remove the calls to useState...

this.handleClick = this.handleClick.bind(this);

And we have to change this function to a method, and use setState instead of setColor.

Then we need to wrap the return in a render.

And we have to use this.state.color instead of just color

And this.state.msg instead of msg.

And this.handleClick instead of handleClick...

Let's check that in the browser. It works exactly the same.

Now let's change it back for practice.

Change the class to a function

Remove the constructor

set up the useState hook for color, and setColor...

set up the useState hook for msg, and setMsg

Import useState once again

Change the method back to a function, remove this.state, and use setColor and setMsg.

Remove the render, and remove this.state

Remove this. from handleClick

And let's check that it works the same in the browser.

Oops, looks like I initialized msg to the string msg instead of an empty string, let me fix that...

There we go. Now it works the same as before.

With this, I will end your initiation to the useState hook and we will continue to the useEffect hook, which will let us access lifecycle features of components. There is more to learn about the useState hook, and we will cover it in more depth in the full curriculum updates. This has been an introduction that I hope will get you feeling more confident about trying out useState in your own projects.

# Review: Side Effects and Lifecycle Methods

The useEffect hook is used to add side effects in your function components. It is similar, though not identical, to the use of several lifecycle methods by class components for the same purpose.

The useEffect hook is used to add side effects in function components

Similar, though not identical, to the use of several lifecycle methods by class class components

As with useState, there are a couple of concepts here to review before we get to useEffect: What are side effects, and what are lifecycle methods?

First, what are side effects?

The concept of a side effect is a general concept in computer programming, not only React. Side effects are operations that cause any changes outside of its scope beyond the main effect. Consider the main effect of a React component: That would be a return value, typically written with JSX, that is used to render the component.

At times, we want a component to do more than just return this value. We might want it to, for example, perform data fetching by making a request to a server. Or we might want to set up a subscription, like in React Native, when we subscribed to network status changes, or make other API calls, or use timer functions like setTimer. These are typical examples of side effects. Consider what happens when your component sends a request to a server. That server receives the request, it may log the request, it sends back a response, your application then needs to deal with that response, and so on. All these things are happening separately from your component's main effect, and that's why we call them side effects.

Class components use what are called lifecycle methods to run side effects. Which leads us to the next topic for review: Lifecycle methods

When we run a React application, each component has what we call a lifecycle. There are three major stages of this lifecycle:

First, the component is rendered and inserted into the DOM tree. React calls this process of first inserting the component into the DOM "mounting".

Second, the component may receive updates and be re-rendered.

Lastly, when the component is no longer needed, it is removed from the DOM tree. React calls this removal process unmounting.

Lifecycle methods give class components a way to run code whenever a component enters one of these stages. componentDidMount runs after a component is mounted, componentDidUpdate runs after a component is re-rendered, and componentWillUnmount runs right before a component is unmounted.

ComponentWillUnmount is typically used for cleaning up side effects, such as canceling any timers or subscriptions.

The useEffect hook can be conceptualized as a replacement for all three of the major lifecycle methods, componentDidMount, componentDidUpdate, and ComponentWillUnmount. It's not an exact duplicate.

It's not an exact duplicate. There are ways in which it doesn't behave the same as those methods. But the basic idea is the same - it's a way to run extra code when a component is in a certain stage of its lifecycle.

Next, in the following lesson, we'll go over the basic usage of the useEffect hook in function components.

# Introduction to the useEffect hook

Let's bring back our demo code from the introduction to useState. We'll continue using this code to learn about useEffect.

class App extends Component {

constructor(props) {

super(props);

this.state = {

color: 'red',

};

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

this.setState({ color: 'blue' });

}

render() {

return (

<div>

<h1 style={{ background: this.state.color }}>

React Component State Example

</h1>

<button onClick={this.handleClick}>Click Me</button>

</div>

);

}

}

~~Let's add these lifecycle methods, componentDidMount, and I'm going to just set up each of them to log to the console so we can see the method has run.~~

~~componentDidMount() {~~

~~console.log("mount");~~

~~}~~

~~componentDidUpdate() {~~

~~console.log('update');~~

~~}~~

~~componentWillUnmount() {~~

~~console.log('unmount');~~

~~}~~

~~Now when this component is first mounted, you can see that the componentDidMount method ran.~~

~~Remember, every time that we update the state by clicking this button, this component will update, so if I click this button, the componentDidUpdate method runs.~~

~~And now if I navigate away from this page, there's the componentWillUnmount method.~~

~~Let me bring up the function component version of this example.~~

Like useState, we can import the useEffect hook from react.

Now we'll call useEffect from inside this function component.

The useEffect hook has one required argument that we need to pass in, and that argument is a function. Typically we'll use an arrow function for this, because it's a more concise form. We can call this function our "effect".

Inside this effect function, we'll put the code that we want to run. For this, let's do a console.log of the msg.

console.log(msg);

And let me actually go back to this useState hook and give the msg state an initial value, so it's easier to see what's going on.

When this page first loads, the App component will render for the first time and get mounted to the DOM. Then the effect function runs, and you can see the msg state value is logged to the console. What will happen if I click this button, which causes the component to re-render? Note that the browser console will usually just increment a number here for repeated identical logs to the console, instead of logging the same message over and over again.

From this, we can see that the effect function runs after each re-render.

This is its default behavior, similar to the ComponentDidUpdate lifecycle method. One difference is that ComponentDidUpdate doesn't run on the initial render, only on re-renders. But useEffect runs on the initial render and every re-render after that.

There's a nice feature to useEffect that goes beyond what lifecycle methods offered, and it's that we can tell useEffect to only run if certain values have been updated since the previous render. Doing this can improve performance.

How we do this is by adding an optional second argument after the function, called a dependency array, or deps for short.

Inside this array, you should put the names of all component state and props values that are used inside the effect. THe useEffect hook will then only run if one of these values changes. For example, if I put msg inside this array, then it will only run if the value of msg changes.

To give you an idea of how this works, let's separate out some code. We'll change the handleClick method to changeColor. And we'll make a new handler of changeMsg and that will handle updating the message. Here, let's change this button to use changeColor for its onClick event handler. Then we can make a second button, and let's give it an onclick handler of changeMsg.

I'm going to save and refresh, and we can see that when the component first rendered, the effect function ran and we see a console log of Initial Msg.

Now if I click the second button, that will change the value of msg, so the effect function runs again. You can see that it did run again, because it console logged the updated message.

Now if I click the first button, that changes the color. Our useEffect hook doesn't care about updates to any value except msg, so the effect function does NOT run this time.

If we were to go back and add color to the dependency array, then the effect function will run anytime the color changes as well.

So once again, if you remove the dependency array, then this effect function will run every time the component renders. If you add a dependency array, then the effect function will only run if the values in that array have changed since the previous render.

Notice that even though msg is in the dependency array, it only changes the first time we click the changeMsg button. After that, the msg is always the same, so it doesn't fire off the effect.

By the way, as with useState, we can have more than one useEffect hook in a component. Let me add another one to demonstrate, and I'll also show you how we can mimic the behavior of the ComponentDIdMount lifecycle method, where code is run once only when the component is first mounted, instead of for every render. We'll log to the console here, then we'll give this an empty dependency array.

Remember, whenever you pass useEffect a dependency array, it will then only run the effect function if values inside that array change. If the array is empty, then the effect function will never run after the first render, when the component mounts. So this is how we can make an effect function run when the component mounts, and ignore updates.

useEffect(() => {

console.log('mount');

}, []);

You can see in the browser that I'm causing the component to re-render by clicking the buttons, but the effect that console.logs the text "mount" does not run again after the first time.

The last concept I want to go over with you is the cleanup function.

Let me first delete these useEffects, not because it's required, but just to make it easier to read the console.

For this part of the demo

, let's make a child component so that you can see what happens when it gets unmounted.

I'll create a function component named ExampleChild.

return (

<h2>I am the example child</h2>

);

I'll render the child component here, like this <ExampleChild />

And here you can see the child component text in the browser.

Let's say in the ExampleChild component, we want to make an API call to subscribe to some updates. I'll console log it here and we can pretend that a subscription is being made.

Notice that inside this effect function, we're not returning any value.

However, we can return a function, which React refers to as a cleanup function because it's typically used to perform cleanup routines such as canceling any subscriptions to avoid memory leaks.

When the component re-renders, if there is a cleanup function to run, it will by default first run that function *then* run the effect function. It will also run the cleanup function when it unmounts.

So I can use this cleanup function to unsubscribe. Again, I'll just console.log that that's what we're doing.

Take a look at the console. You can see that when the component first mounted, the effect ran and console logged "Subscribing." Now if I click this button to re-render the component, the cleanup function runs to unsubscribe, and the effect function runs again to re-subscribe. This ensures that we don't have a bunch of duplicate subscriptions when we only want one.

Now, you might have a case where you only want the cleanup function to run once, when the component is unmounted, instead of at every re-render.

We can do that by once again passing this useEffect hook an empty dependency array.

Let me set up a scenario where we can force this child component to unmount, so you can see how that works. We'll go to the App component and create a situation where the child component first gets mounted, then in a later render, is excluded from the return. Then React will automatically unmount the child component.

One way is to introduce a conditional in the render.

One way to do that is by using curly braces to indicate JavaScript in JSX, then we can write JavaScript to check for the condition, is the msg set to 'Initial Message'? Then we'll use the Logical And operator, which means that the right operand, the <ExampleChild> component, will only be rendered if it's true that the message is that initial value. If the message has changed, then because the logical && operator returns false if the left operand is false, the ExampleChild component will not be rendered, and it will get unmounted.

Let's look at that in the browser. When the app first loads, the component state of msg *is* the string Initial Message, so the ExampleChild component *did* render and mount, so you can see that the subscribe effect function ran.

Now I can re-render the component by changing its color, but due to the empty dependency array, the effect does not run again, nor does the cleanup.

Let's unmount this ExampleChild component. I can do that by clicking the second button, and you can see that causes the ExampleChild component to be unmounted from the DOM, due to the code I added earlier. When that happens, you can see in the console that the cleanup function also ran.

So that's how you can also use the empty dependency array to cause the cleanup function to only run when the component unmounts, instead of every single time it re-renders.

~~The useEffect hook can be used more than once. Maybe I want one behavior to run only at component mount:~~

~~useEffect(() => console.log("mount"), []);~~

~~And another to run every time the component renders:~~

~~useEffect(()=> console.log('render');~~

~~If I click ... you can see the first effect runs only the first time the component is rendered, and the second effect runs every time.~~

With that, we will end our introduction to useEffect and hooks.

There is of course much more to hooks. If you want to dive deeper, some concepts to explore are the function form of useState, storing objects in useState, how useEffect captures state and props values for each render, other built-in React hooks such as useContext and useRef, and writing custom hooks.

I've included some links in the Additional Resources section for you to explore more on your own, and/or, you can check back for when we have completed integrating hooks fully into the React and React Native curriculum. Happy learning!

mention function version of setState

useState function version

<https://www.taniarascia.com/redux-react-guide/>

state component always object, hooks anything

state object, must create new object, mutating existing object won't trigger re-render

also won't merge, need to use spread to merge if you need that

https://blog.logrocket.com/a-guide-to-usestate-in-react-ecb9952e406c/\`

from acemarke re: useEffect and redux action creators: (reactiflux):

the issue is that the React hooks lint rule assumes that:

- all the logic in the hook needs to be re-run any time any of the values or functions you used changed

- anything that's not one of the state setter functions from useState/useReducer could change

so here, the bound action creators from mapDispatch should be stable, unless you're writing mapDispatch as a function like (dispatch, ownProps) => {}

but the lint rule doesn't know that

similarly, even with useDispatch, it doesn't know that dispatch will be stable unless you happen to switch a different store into the <Provider>, so it yells at you if you don't have [dispatch] as the deps array

so, your options are either stick all of those in the deps array, or // eslint-disable-next-line