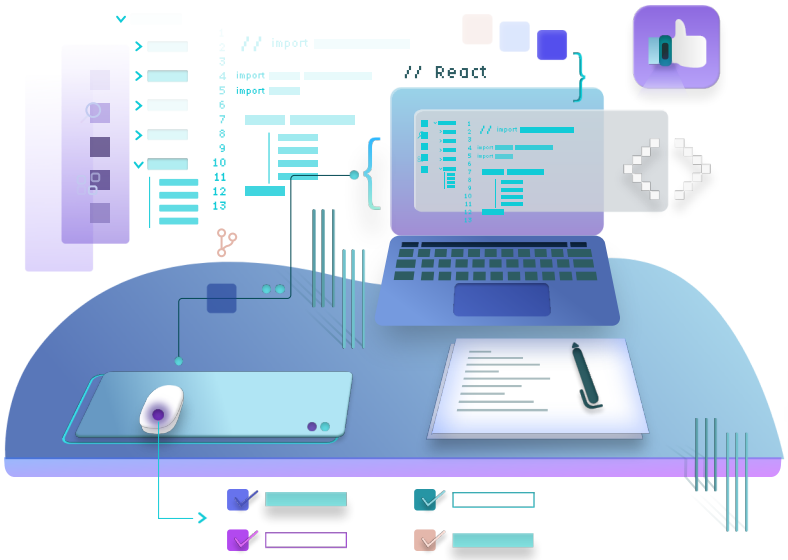
Courses



useEffect Hook

Manage with your component's lifecycle with the useEffect hook

React is known for its powerful state management process that doesn't require any page reload. One of the tools used for managing the state is the useEffect hook, that deals with the

component's lifecycle. Read on to learn how to use the useEffect hook.

CodeSandbox link

You can find the full code for this tutorial at **https://codesandbox.io/s/useeffect-yuwht**.

Understanding the component lifecycle

Using the useEffect hook means understanding the lifecycle of the component. This lifecycle consists of three main parts: mounting, updating, and unmounting. If you're familiar with the old wait to deal with the component lifecycle, these three parts correspond to **ComponentDidMount**, **ComponentDidUpdate**, **ComponentWillUnmount**.

To understand better a component's lifecycle, we can look at the lifecycle of a flower. When the flower's seed is planted, it is **mounted**. Then, as it grows, it is **updating** and, unfortunately, it will die (or **unmount**) someday. The component's lifecycle is exactly the same: when the user lands on the page, the component is mounted, then as the states update, the component is also updated, and, finally, the component is unmounted when the user leaves the page.

Anatomy of the useEffect hook

The component lifecycle is represented in three different parts of the useEffect hook, as shown in the code snippet below.

```
ect(() => {
Mounting
```

```
    return () => {
      // Cleanup function
    }
  }, [//Updating])
```

The first part is the mounting part. That's when the component is initially rendered when the user lands on the page. The return function is the cleanup function, or when the user leaves the page and the component will unmount. The array is the last part, and it is where you put the states that will update throughout the component's lifecycle. You can discard the array if your component won't update during its lifecycle.

When to use the useEffect hook?

The useEffect hook is useful when you wish to run some functions during the component's lifecycle. For example, if you want to update the UI when a state changes, the useEffect hook is the way to go. You can also define a state on first load (when componentDidMount), and also clean the state when the component is unmounting (componentWillUnmount).

Let's look at an example below. We want to update the page title every time the user clicks on a button on the page, using the useEffect and useState hooks.

Create your component

Start by creating your component. We'll reuse the button component we created in the **useState hook section** and incorporate the useEffect hook as well.

```
import React, { useState } from "react"

Button = () => {
st [count, setCount] = useState(0)
```

```
    return <button onClick={() => setCount(count + 1)}>You clicked {count} times</
}

export default Button
```

Import useEffect

In order to use **useEffect**, you'll need to import it at the top of the file. Don't forget to keep React and useState as we still need them.

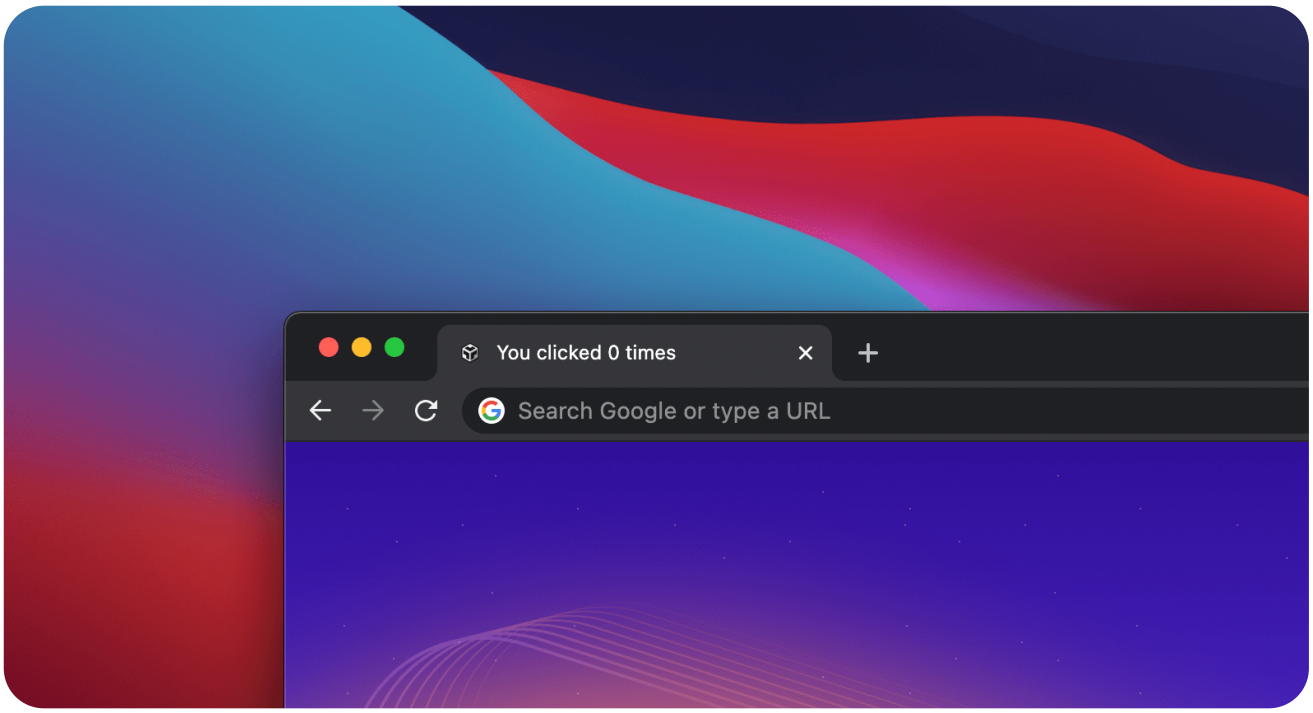
```
import React, { useState, useEffect } from "react"
```

Define useEffect

Next, create a new **useEffect** just above the return statement in your component. We want to update the document title every time the user clicks on the button, so let's set it in the mounting part of the useEffect hook.

```
useEffect(() => {
    document.title = `You clicked ${count} times.` // This is the mounting part
}, [])
```

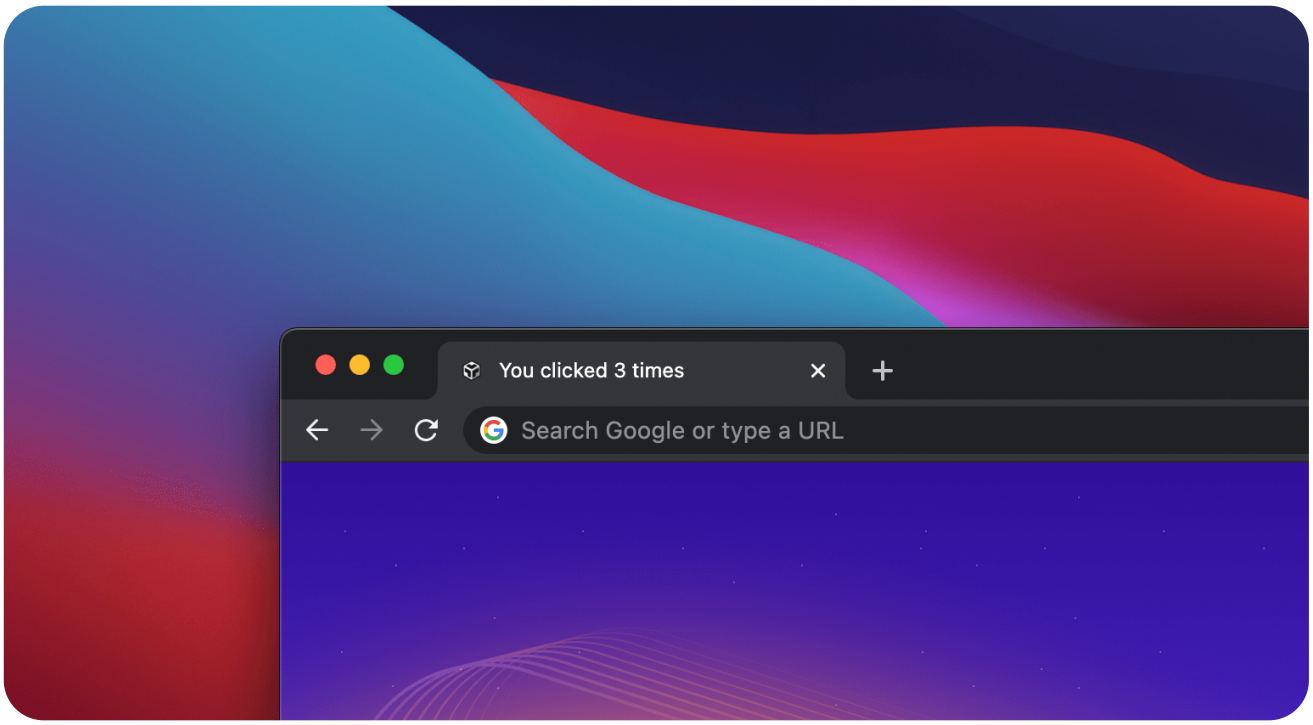
You'll see the title of the tab change to **You clicked 0 times**, since the value of **count** is initially set to 0.



However, when you click on the button, the text on the button updates but not the title of the tab. Therefore, we need to add the count state in the update array of the useEffect hook. Every time the **count** is updated, the useEffect will be re-run, updating the document title in our case.

```
useEffect(() => {
    document.title = `You clicked ${count} times.`
}, [count]) // Add the count state in the array here
```

Now, every time you click on the button, the title of the tab will update to reflect the number of times the user clicked on the button.



Final code with styling

You can style your components by using the **styled-components** package in your project. Learn more about it in **the Styling in React section** of this handbook.

The final code incorporating useEffect as well as useState and styled-components will look like this:

```
import React, { useState, useEffect } from "react";
import styled from "styled-components";

const MyButton = () => {
  const [count, setCount] = useState(0);

  Effect(() => {
  ocument.title = `You clicked ${count} times`;
  [count]);
```

```
  return (
    <StyledButton onClick={() => setCount(count + 1)}>
      You clicked {count} times
    </StyledButton>
  );
};

export default MyButton;

const StyledButton = styled.button`
  background: linear-gradient(91.4deg, #2fb8ff 0%, #9eecd9 100%);
  padding: 12px 0;
  width: 200px;
  border: none;
  border-radius: 30px;
  color: white;
  font-weight: bold;
  font-family: Segoe UI, sans-serif;
`;
```

BACK TO
useState Hook

READ NEXT
useRef Hook

15 **useInput Hook** 6:04
Create a hook to get the value and the onChange event of input fields

16 **Intro to Firebase** 6:59
Learn about Firebase and set it up in your React project

17 **Firebase Auth** 11:59
Set up authentication in your React application using Firebase Auth

18 **Firestore** 10:51
Enable Firestore as your database in your React application

19 **Firebase Storage** 6:40
Enable Firebase Storage in your application to store photos or...

20 **Gatsby and React** 6:44
Create a static content-oriented website using React on Gatsby

21 **NextJS and React** 5:24
Create your first NextJS React application

22 **React TypeScript Par...** 8:19
Learn how to create a React TypeScript application using the...

23 **React TypeScript Par...** 7:35
Learn the basics of TypeScript and how to use TypeScript in a React...

24 **useScrollPositio n Hook** 4:26
Create a custom hook to listen to the current window position of th...

25 **useOnScreen hook** 8:08
Create a custom hook to listen to when an element is visible on...

26 **useContext Hook** 8:32
Manage global states throughout the entire application

27 **Fragments** 2:43
Group multiple children together with React Fragments

28 **Lazy Loading** 4:05
Lazy Load heavy components to improve performance

29 **React Suspense** 3:13
Wait for data with React Suspense and React.lazy

30 **Environment Variables** 4:43
Make environment variables secret with a .env file

31 **Reach Router** 5:31
Create a multiple-pages React application with Reach Router

32 **URL Params** 4:04
Create unique URL with URL Params

SEO and Metadata 6:47
Optimize a React application for search engines with React Helmet

34 **Favicon** 3:03
Add an icon to a React website

35 **Dynamic Favicon** 2:14

Change the favicon's fill color depending on the user's system...

36 **PropTypes** 3:54

Implement props type-checking with PropTypes

37 **Custom PropTypes** 3:58

Create a custom PropType using a validator function

38 **useMemo** 4:05

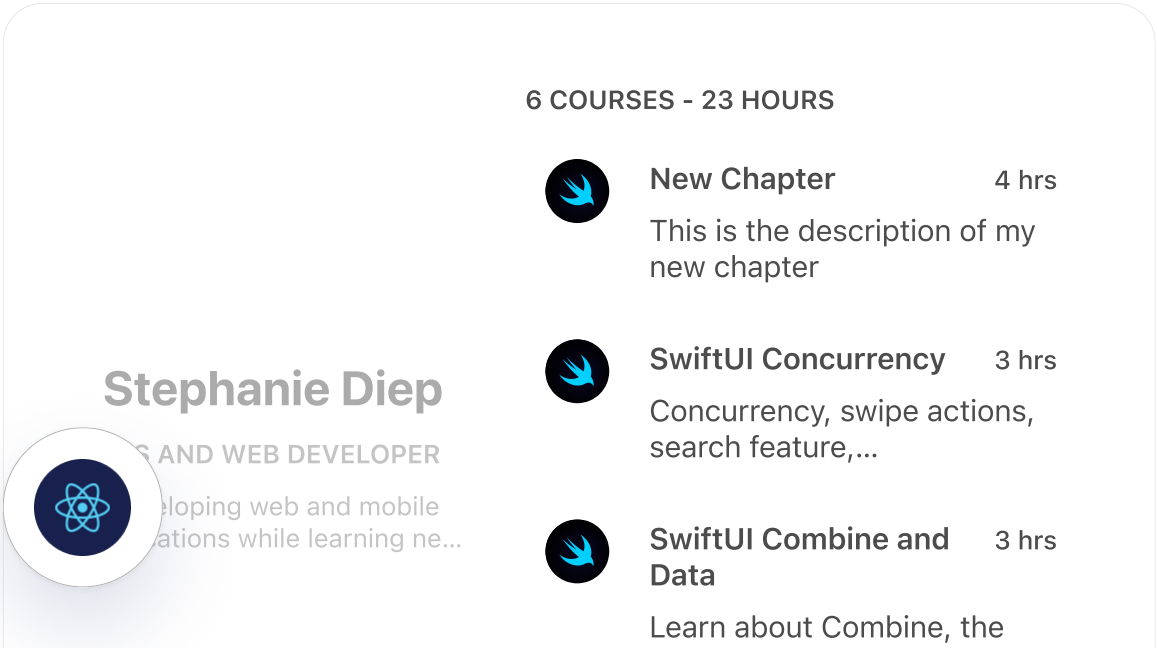
Prevent unnecessary re-renders when the component stays the...

39 **Serverless Email Sending** 10:02

Use EmailJS to send an email without backend

Meet the instructor

We all try to be consistent with our way of teaching step-by-step, providing source files and prioritizing design in our courses.



MVVM architecture, data,...

SwiftUI Advanced 3 hrs

Home
Downloads
Courses
Search
Tutorials
Account
Pricing
Licenses
Updates

Site made with React, Gatsby,
Netlify and Contentful. Learn **how**.