**useEffect cleanup function**

As the name implies, useEffect cleanup is a function in the useEffect Hook that allows us to tidy up our code before our component unmounts. When our code runs and reruns for every render, useEffect also cleans itself up using the cleanup function.

The useEffect Hook is designed to allow the return of a function within it, which serves as a cleanup function. The cleanup function prevents memory leaks — a situation where your application tries to update a state memory location that no longer exists — and removes unnecessary and unwanted behaviors.

Note that you don’t update the state inside the return function either:

useEffect(() => {

effect

return () => {

cleanup

}

}, [input])

## **Why is the useEffect cleanup function useful?**

As previously stated, the useEffect cleanup function helps developers clean effects that prevent unwanted behaviors, thereby optimizing application performance.

However, it is important to note that the useEffect cleanup function does not only run when our component wants to unmount — it also runs right before the execution of the next scheduled effect.

In fact, after our effect executes, the next scheduled effect is usually based on the dependency array:

// The `dependency` in the code below is an array

useEffect(callback, dependency)

Therefore, when our effect is dependent on our prop or whenever we set up something that persists, we have a reason to call the cleanup function.

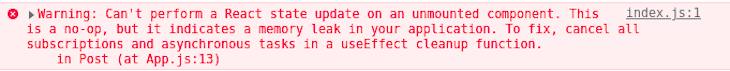
Let’s look at this scenario: imagine we request the server to fetch a particular user’s information using the user’s id. Before the request is completed, we change our mind and try to make another request to get a different user’s information.

At this point, both fetch requests would continue to run even after the component unmounts or the dependencies change. This can lead to unexpected behavior or errors, such as displaying outdated information or attempting to update components that are no longer mounted.

So, it is necessary for us to abort the fetch using the cleanup function. That way, we prevent these memory leak-related issues in our application.

## **When should we use the useEffect cleanup?**

Let’s say we have a React component that fetches and renders data. If our component unmounts before our promise resolves, useEffect will try to update the state (on an unmounted component) and send an error that looks like this:



To fix this error, we use the cleanup function. According to [React’s official documentation](https://legacy.reactjs.org/docs/hooks-effect.html" \l ":~:text=When%20exactly%20does,issues%20later%20below.), “React performs the cleanup when the component unmounts. However, effects run for every render and not just once. This is why React also cleans up effects from the previous render before running the effects next time.”

As a side note before we continue: useEffects can be made to run once by simply passing an empty array to the dependency list. When you provide an empty array as the dependency list for useEffect, it indicates that the effect does not depend on any values from the component’s state or props. As a result, the effect will only run once, after the initial render, and it won’t run again for subsequent renders unless the component is unmounted and remounted:

useEffect(() => {

// Effect implementation

}, []); // Empty dependency array indicates the effect should only run once

Now that we understand how to make useEffect run once, let’s get back to our cleanup function conversation.

The cleanup function is commonly used to cancel all active subscriptions and async requests. Now, let’s write some code and see how we can accomplish these cancellations.

**fetch data from API request:**

1. open your terminal and install axios

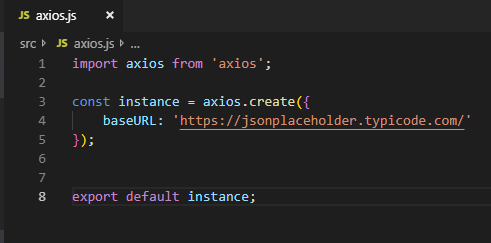
npm install axios –save

2. Now we will create an instance for baseURL. This instance is created so that we don’t have to manually change the base URl of the API in each component. So to create the instance we need to go our app **src**folder and create a file name **axios.js**

**3.**Now open your project inside your favorite IDE and paste this inside your newly created **axios.js** file

import axios from 'axios';const instance = axios.create({  
 baseURL: '[https://jsonplaceholder.typicode.com](https://jsonplaceholder.typicode.com/)/['](http://backend.wespotyou.com/')  
});export default instance;

Note: change [**https://jsonplaceholder.typicode.com**](https://jsonplaceholder.typicode.com/)**/**with your own API base URL.



4. Now we will create our **components** folder to put all our components in one place. Now in your **src** folder create a new folder name **components**and inside that create another folder name **users.**

5. Now create a file **users.js**inside your **users** folder and create the react class component by pasting this

import React, { Component } from 'react'export default class users extends Component {  
 render() {  
 return (  
 <div>  
   
 </div>  
 )  
 }  
}

6. Now import your **users.js**file in your **app.js** file inside your **src** folder.

|  |
| --- |
| import React from 'react'; |
| import logo from './logo.svg'; |
| import './App.css'; |
| import Users from './components/users/users' |
|  |
| function App() { |
| return ( |
| <div className="App"> |
| <Users></Users> |
| </div> |
| ); |
| } |
|  |
| export default App; |

7. Now import axios from your newly created **axios.js** file from your **src** folder to the component.

import axios from '../../axios'

8. Now it’s time to add a class **constructor** and assigns the initial **state**Users : []

constructor(props) {  
 super(props);  
 this.state = {  
 Users: []  
 };  
 }

9. Now we will create a function to call our users data from an API using axios to our component. Create a function name **getUsersData**after your class constructor and paste this inside your function

axios  
 .get(`/users`, {})  
 .then(res => {  
 const data = res.data  
 console.log(data)})  
 .catch((error) => {  
 console.log(error)  
 })

Note: you need to call **getUsersData function**inside your componentDidMount lifecycle method by insertingthis.getUsersData()

10. Your **users.js** will look something like this.

|  |
| --- |
| import React, {Component} from 'react' |
| import axios from '../../axios' |
|  |
| export default class users extends Component { |
| constructor(props) { |
| super(props); |
| this.state = { |
| Users: [] |
| }; |
| } |
| getUsersData() { |
| axios |
| .get(`/users`, {}) |
| .then(res => { |
| const data = res.data |
| console.log(data) |
|  |
| }) |
| .catch((error) => { |
| console.log(error) |
| }) |
|  |
| } |
| componentDidMount(){ |
| this.getUsersData() |
| } |
| render() { |
|  |
| return ( |
| <div></div> |
| ) |
| } |
| } |

11. Now run your app. You will see an array of objects inside your console.

12. Hola 😎 we have successfully called our data from an API to our react app using axios.

13. Now it’s time to print the data in our app. For this, we will use the javascript [**map**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)and [**arrow**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)function to loop through our array of objects.

14. Just after your console.log(data) paste this

const users = data.map(u =>  
 <div>  
 <p>{u.id}</p>  
 <p>{u.name}</p>  
 <p>{u.email}</p>  
 <p>{u.website}</p>  
 <p>{u.company.name}</p>  
 </div>  
 )this.setState({users})

15. Now call your users data using **this.state.users** inside return to print in your app

{this.state.users}

Refresh your browser and tada 😱 all your data is printed.

**React Memo:**

Using memo will cause React to skip rendering a component if its props have not changed.

This can improve performance.

This section uses React Hooks. See the [React Hooks](https://www.w3schools.com/react/react_hooks.asp) section for more information on Hooks.

## **Problem**

In this example, the Todos component re-renders even when the todos have not changed.

index.js:

import { useState } from "react";

import ReactDOM from "react-dom/client";

import Todos from "./Todos";

const App = () => {

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

const [todos, setTodos] = useState(["todo 1", "todo 2"]);

const increment = () => {

setCount((c) => c + 1);

};

return (

<>

<Todos todos={todos} />

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

</div>

</>

);

};

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

Todos.js:

const Todos = ({ todos }) => {

console.log("child render");

return (

<>

<h2>My Todos</h2>

{todos.map((todo, index) => {

return <p key={index}>{todo}</p>;

})}

</>

);

};

export default Todos;

When you click the increment button, the Todos component re-renders.

If this component was complex, it could cause performance issues.

## **Solution**

To fix this, we can use memo.

Use memoto keep the Todos component from needlessly re-rendering.

Wrap the Todos component export in memo:

### **Example:**

index.js:

import { useState } from "react";

import ReactDOM from "react-dom/client";

import Todos from "./Todos";

const App = () => {

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

const [todos, setTodos] = useState(["todo 1", "todo 2"]);

const increment = () => {

setCount((c) => c + 1);

};

return (

<>

<Todos todos={todos} />

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

</div>

</>

);

};

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

Todos.js:

import { memo } from "react";

const Todos = ({ todos }) => {

console.log("child render");

return (

<>

<h2>My Todos</h2>

{todos.map((todo, index) => {

return <p key={index}>{todo}</p>;

})}

</>

);

};

export default memo(Todos);

Now the Todos component only re-renders when the todos that are passed to it through props are updated.

# useCallback Hook

The React useCallback Hook returns a memoized callback function.

Think of memoization as caching a value so that it does not need to be recalculated.

This allows us to isolate resource intensive functions so that they will not automatically run on every render.

The useCallback Hook only runs when one of its dependencies update.

This can improve performance.

The useCallback and useMemo Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function. You can learn more about useMemo in the useMemo [chapter](https://www.w3schools.com/react/react_usememo.asp).

### **Problem**

One reason to use useCallback is to prevent a component from re-rendering unless its props have changed.

In this example, you might think that the Todos component will not re-render unless the todos change:

This is a similar example to the one in the [React.memo](https://www.w3schools.com/react/react_memo.asp) section.

index.js

import { useState } from "react";

import ReactDOM from "react-dom/client";

import Todos from "./Todos";

const App = () => {

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

const [todos, setTodos] = useState([]);

const increment = () => {

setCount((c) => c + 1);

};

const addTodo = () => {

setTodos((t) => [...t, "New Todo"]);

};

return (

<>

<Todos todos={todos} addTodo={addTodo} />

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

</div>

</>

);

};

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

Todos.js

import { memo } from "react";

const Todos = ({ todos, addTodo }) => {

console.log("child render");

return (

<>

<h2>My Todos</h2>

{todos.map((todo, index) => {

return <p key={index}>{todo}</p>;

})}

<button onClick={addTodo}>Add Todo</button>

</>

);

};

export default memo(Todos);

Try running this and click the count increment button.

You will notice that the Todos component re-renders even when the todos do not change.

Why does this not work? We are using memo, so the Todos component should not re-render since neither the todos state nor the addTodo function are changing when the count is incremented.

This is because of something called "referential equality".

Every time a component re-renders, its functions get recreated. Because of this, the addTodo function has actually changed.

## **Solution**

To fix this, we can use the useCallback hook to prevent the function from being recreated unless necessary.

Use the useCallback Hook to prevent the Todos component from re-rendering needlessly:

### **Example:**

index.js

import { useState, useCallback } from "react";

import ReactDOM from "react-dom/client";

import Todos from "./Todos";

const App = () => {

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

const [todos, setTodos] = useState([]);

const increment = () => {

setCount((c) => c + 1);

};

const addTodo = useCallback(() => {

setTodos((t) => [...t, "New Todo"]);

}, [todos]);

return (

<>

<Todos todos={todos} addTodo={addTodo} />

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

</div>

</>

);

};

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

Todos.js

import { memo } from "react";

const Todos = ({ todos, addTodo }) => {

console.log("child render");

return (

<>

<h2>My Todos</h2>

{todos.map((todo, index) => {

return <p key={index}>{todo}</p>;

})}

<button onClick={addTodo}>Add Todo</button>

</>

);

};

export default memo(Todos);

Now the Todos component will only re-render when the todos prop changes.