

useEffect

With React hooks, we are able to manage state in normally stateless (functional) components. By using something like this:

```
const [pieceOfState, setPieceOfState] = useState('initial value here');
```

We can have information attached to our component and change it accordingly.

UseEffect is another hook that we can use in functional components. We use `useEffect` in order to manage "side Effects" in our React project. If you use `useEffect` in a functional component, this method will execute directly after the component is rendered, and every time the component updates.

A common way `useEffect` is used is when making a call to an API. For example, let's say we are creating an app that will display a list of people in the database. We want to make that API call when the component first renders.

```
const Example = (props) => {
  const [people, setPeople] = useState([]);

  useEffect(() => {
    fetch('https://swapi.dev/api/people/')
      .then(response => response.json())
      .then(response => setPeople(response.results))
  }, []);

  return (
    <div>
      {people.length > 0 && people.map((person, index)=>{
        return (<div key={index}>{person.name}</div>)
      })}
    </div>
  );
}
```

```
export default Example;
```

`Fetch` is simply a way to make a request in Javascript. When we make a request to the Star Wars API, we will get a response, and part of that response is a list of the people in the Star Wars API. We will then use `setPeople` in order to keep track of the list of people.

In our `return`, we will only map through the people's names if there is at least one person. Otherwise, it will just output nothing.









Second Argument

`useEffect` takes an optional second argument: an array that contains different variables. This second argument is very powerful, because we can be more specific and tell `useEffect` exactly when we want it to run. `useEffect` will always run on first render. It will also run when a variable in the second argument array changes. So, let's look at the following example:

```
useEffect(()=>{
  alert("When will this run?");
}, []);
```

When will this run? Well, we know it will run on first render (right when we load the component). However, it will also change whenever a variable in the second argument changes. If we have an empty array in the second argument, it will never change. Therefore, it will only run on first render.



	Pokemon API	
	Axios	
	Axios Pokemon API	
	useEffect	
	Chapter Survey	
React Routing		▼

Let's look at another example. Let's say we wanted it to run when something in state changed.

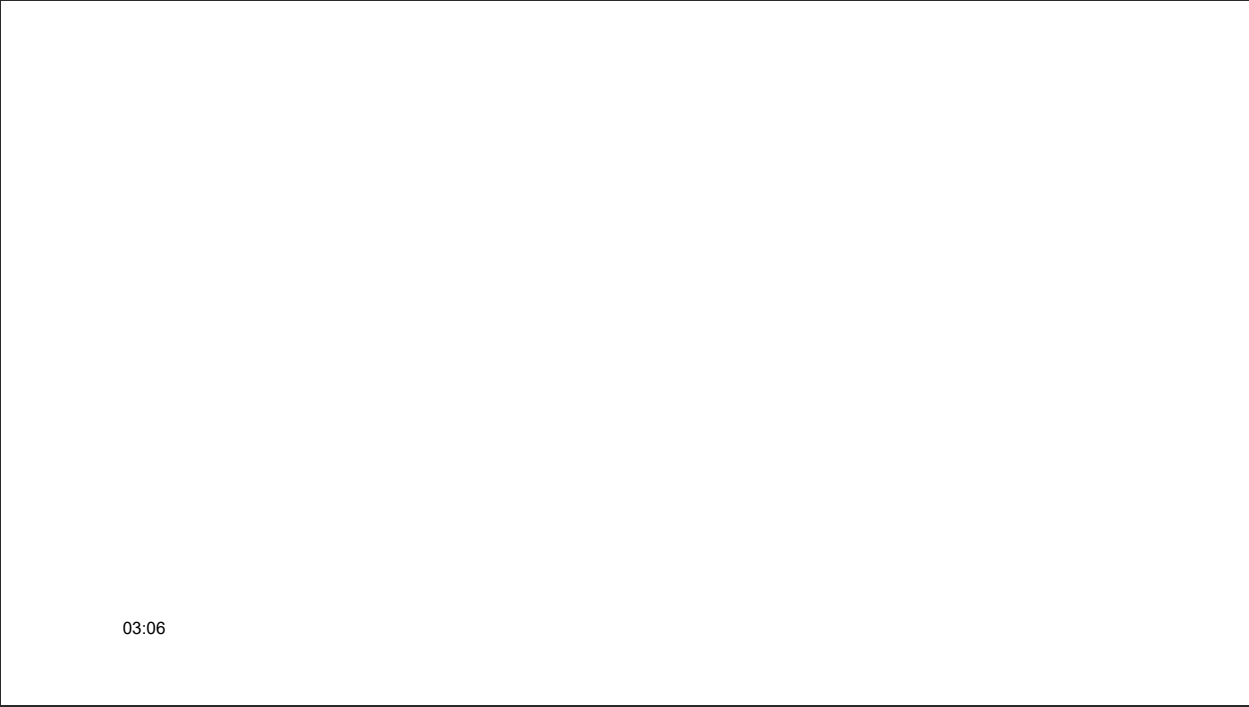
For example, we want to run the method when we submit a form. We could write the following:

```
useEffect(()=>{
  alert("When will this run?");
}, [state.isSubmitted]);
```

In this case, if we submit the form, `state.isSubmitted` will change in our `onSubmitHandler` . So, `state.isSubmitted` will change and `useEffect` will be triggered.

In conclusion, `useEffect` is a hook that will be called after every render (including the first one and every update thereafter). It is important to remember that we can use `useEffect` for making an API call when a component first loads.

NOTE: The Star Wars API has moved, so if you want to follow along with this demo, the new link is: <https://swapi.dev/>.



Cleaning Up On Unmounting

Sometimes, you might have a situation in which you've initiated something inside a `useEffect` call that is ongoing, such as a `setInterval` call or a socket connection. When the component is unmounted, as when the user changes to a different route, it is important to clean up so that your application doesn't develop a memory leak. To clean up, you can optionally return a function from your `useEffect` callback. For example, consider the following code:

```
// TimeDisplay.js
import React, { useEffect, useState } from 'react';

export default () => {
  const [time, setTime] = useState(new Date().toLocaleString());

  useEffect(() => {
    const int = setInterval(
      () => setTime(new Date().toLocaleString()),
      1000
    );

    return function clearInt() {
      clearInterval(int);
    }
  }, []);

  return (
    <div>Current Time: {time}</div>
  );
}
```

Let's go through what's happening in the body of our `useEffect` callback. We create an `int` variable and assign it to the result of invoking the `setInterval` function. We pass `setInterval` two arguments: (1) an anonymous function which updates the time with the current time, and (2) the interval, which is 1000 milliseconds, or 1 second. We then return a function called `clearInt`

(could also be an anonymous or even an arrow function) which clears the interval. If we didn't do this, our application would continue to run the setInterval callback at 1 second intervals until the user either did a hard refresh or left the app entirely.

Further Reading:

[clearInterval\(\)](#) | MDN

[Previous](#)

[Next](#)

[Privacy Policy](#)

