How to useEffect in React

NOVEMBER 24, 2020 BY ROBIN WIERUCH - EDIT THIS POST

In this tutorial you will learn everything about React's useEffect Hook. Let's say we have these two components whereas the parent component manages state with React's useState Hook and its child component consumes the state and modifies the state with a callback event handler:

```
import * as React from 'react';

const App = () => {
  const [toggle, setToggle] = React.useState(true);

  const handleToggle = () => {
    setToggle(!toggle);
  };

  return <Toggler toggle={toggle} onToggle={handleToggle} />;
};
```

```
const Toggler = ({ toggle, onToggle }) => {
  return (
    <div>
      <button type="button" onClick={onToggle}>
        Toggle
      </button>

      {toggle && <div>Hello React</div>}
    </div>
  );
};

export default App;
```

Based on the stateful boolean flag coming from the parent component, the child component renders "Hello React" conditionally. Now let's dive into React's useEffect Hook. Essentially useEffect runs a side-effect function whenever you want to run it. It can run only when the component mounts, when the component renders, or only when the component re-renders, and so on. We will go through various useEffect examples to demonstrate its usage.

REACT USEEFFECT HOOK: ALWAYS

Let's see the first example of React's useEffect Hook where we pass in the side-effect function as an argument:

```
const Toggler = ({ toggle, onToggle }) => {
  React.useEffect(() => {
    console.log('I run on every render: mount + update.');
  });

  return (
    <div>
      <button type="button" onClick={onToggle}>
        Toggle
      </button>

      {toggle && <div>Hello React</div>}
    </div>
  );
};
```

This is the most straightforward usage of useEffect where we only pass one argument -- a function. This function will render on every render -- meaning **it runs on the first render of the**

component (also called on mount or mounting of the component) and **on every re-render of the component** (also called on update or updating of the component).

REACT USEEFFECT HOOK: MOUNT

If you want to run React's useEffect Hook **only on the first render** of a component (also called **only on mount**), then you can pass in a second argument to useEffect:

```
const Toggler = ({ toggle, onToggle }) => {
  React.useEffect(() => {
    console.log('I run only on the first render: mount.');
  }, []);

  return (
    <div>
      <button type="button" onClick={onToggle}>
        Toggle
      </button>

      {toggle && <div>Hello React</div>}
    </div>
  );
};
```

The second argument -- here an empty array -- is called **dependency array**. If the dependency array is empty, the side-effect function used in React's useEffect Hook has no dependencies, meaning it runs only the first time a component renders.

REACT USEEFFECT HOOK: UPDATE

Previously you have learned about React's useEffect Hook's dependency array. This array can be used to run the side-effect function of useEffect only if a certain variable changes:

```
const Toggler = ({ toggle, onToggle }) => {
  React.useEffect(() => {
    console.log('I run only if toggle changes (and on mount).');
  }, [toggle]);

  return (
    <div>
      <button type="button" onClick={onToggle}>
        Toggle
```

```
      </button>

      {toggle && <div>Hello React</div>}
    </div>
  );
};
```

Now the side-effect function for this React component **runs only when the variable in the dependency array changes**. However, note that the function runs also on the component's first render (mount). Anyhow, the dependency array can grow in size, because it's an array after all, so you can pass in more than one variable. Let's check this out with the following addition to our component:

```
const Toggler = ({ toggle, onToggle }) => {
  const [title, setTitle] = React.useState('Hello React');

  React.useEffect(() => {
    console.log('I still run only if toggle changes (and on mount).');
  }, [toggle]);

  const handleChange = (event) => {
    setTitle(event.target.value);
  };

  return (
    <div>
      <input type="text" value={title} onChange={handleChange} />

      <button type="button" onClick={onToggle}>
        Toggle
      </button>

      {toggle && <div>{title}</div>}
    </div>
  );
};
```

The side-effect function in React's useEffect Hook still only runs when the one variable in the dependency array changes. Even though the component updates whenever we type something into the input element, useEffect will not run on this update. Only if we provide the new variable in the dependency array, the side-effect function will run for both updates:

```
const Toggler = ({ toggle, onToggle }) => {
  const [title, setTitle] = React.useState('Hello React');

  React.useEffect(() => {
    console.log('I run if toggle or title change (and on mount).');
  }, [toggle, title]);
```

```
  const handleChange = (event) => {
    setTitle(event.target.value);
  };

  return (
    <div>
      <input type="text" value={title} onChange={handleChange} />

      <button type="button" onClick={onToggle}>
        Toggle
      </button>

      {toggle && <div>{title}</div>}
    </div>
  );
};
```

However, in this case you could leave out the second argument -- the dependency array -- of useEffect entirely, because only these two variables trigger an update of this compinent, so by not having a second argument the side-effect would run on every re-render anyway.

There are various use cases for having React's useEffect run on an updated variable. For example, after updating the state, one might want to have a callback function based on this state change.

REACT USEEFFECT HOOK: ONLY ON UPDATE

If you have been attentive the previous section, you know that React's useEffect Hook with an array of dependencies run for the first render of the component too. What if you would want to **run this effect only on the update**? We can achieve this by using React's useRef Hook for an instance variable:

```
const Toggler = ({ toggle, onToggle }) => {
  const didMount = React.useRef(false);

  React.useEffect(() => {
    if (didMount.current) {
      console.log('I run only if toggle changes.');
    } else {
      didMount.current = true;
    }
  }, [toggle]);

  return (
    <div>
      <button type="button" onClick={onToggle}>
```

```
        Toggle
      </button>

      {toggle && <div>Hello React</div>}
    </div>
  );
};
```

When the side-effect function runs for the first time on mount, it only flips the instance variable and doesn't run the implementation details (here `console.log`) of the side-effect. Only for the next time the side-effect runs (on the first re-render / update of the component), the real implementation logic runs. If you want to have a custom hook for this purpose, check out this guide: custom hook for React useEffect only on update.

REACT USEEFFECT HOOK: ONLY ONCE

As you have seen, you can run React's useEffect Hook's function only once by passing an empty dependency array. This runs the function only once, however, only on the component's first render. What if you would want to run the effect function for a different case -- for example, only once when a variable updates? Let's see:

```
const Toggler = ({ toggle, onToggle }) => {
  const calledOnce = React.useRef(false);

  React.useEffect(() => {
    if (calledOnce.current) {
      return;
    }

    if (toggle === false) {
      console.log('I run only once if toggle is false.');

      calledOnce.current = true;
    }
  }, [toggle]);

  return (
    <div>
      <button type="button" onClick={onToggle}>
        Toggle
      </button>

      {toggle && <div>Hello React</div>}
    </div>
  );
};
```

Same as before, we implement this with a instance variable from React's useRef Hook to track non stateful information. Once our condition is met, for example here that the boolean flag is set to false, we remember that we have called the effect's function and don't call it ever again. If you want to have a custom hook for this purpose, check out this guide: custom hook for React useEffect only on update.

REACT USEEFFECT HOOK: CLEANUP

Sometimes you need to cleanup your effect from React's useEffect Hook when a component re-renders. Fortunatly this is a built-in feature of useEffect by returning a cleanup function in useEffects's effect function. The following example shows you a timer implementation with React's useEffect Hook:

```
import * as React from 'react';

const App = () => {
  const [timer, setTimer] = React.useState(0);

  React.useEffect(() => {
    const interval = setInterval(() => setTimer(timer + 1), 1000);

    return () => clearInterval(interval);
  }, [timer]);

  return <div>{timer}</div>;
};

export default App;
```

When the component renders for the first time, it sets up an interval with React's useEffect Hook which ticks every 1 second. Once the interval ticks, the state of the timer gets incremented by one. The state change initiates a re-render of the component. Since the timer state has changed, without the cleanup function the useEffect function would run again and set up *another* interval. This wouldn't be the desired behavior, because we only need one interval after all. That's why the useEffect function clears the interval before the component updates and then the component sets up a new interval. Essentially the interval is only running for one second before it gets cleaned up in this example.

If you are interested in setting up a stopwatch example from scratch with React's useEffect Hook, check out this React Hooks tutorial.

REACT USEEFFECT HOOK: UNMOUNT

The useEffect hook's cleanup function runs on unmounting of a component too. This makes sense for intervals or any other memory consuming objects that should stop running after the component isn't there anymore. In the following useEffect example, we alternate the previous example to another version:

```
import * as React from 'react';

const App = () => {
  const [timer, setTimer] = React.useState(0);

  React.useEffect(() => {
    const interval = setInterval(
      () => setTimer((currentTimer) => currentTimer + 1),
      1000
    );

    return () => clearInterval(interval);
  }, []);

  return <div>{timer}</div>;
};

export default App;
```

Now we are using useState hook's ability to use a function instead of a value to update the state. This function has as parameter the current timer. Hence we don't need to provide the timer from the outside anymore and can run the effect only once on mount (empty dependency array). That's why the cleanup function here gets only called when the component unmounts (due to page transition or to conditional rendering).

. . .

If you want to dive deeper into React's useEffect Hook and its usages, check out these guides:

- React useEffect to Fetch Data
- React useEffect Best Practices

Show Comments

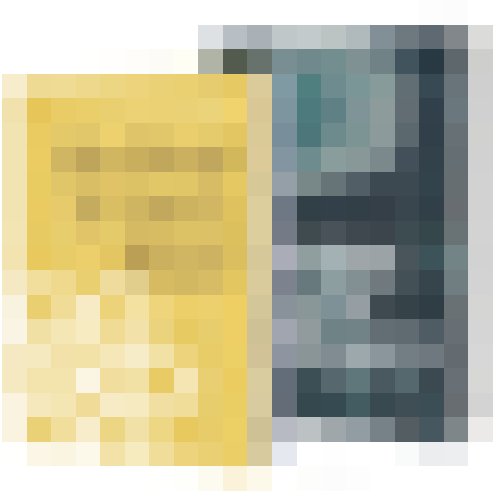
KEEP READING ABOUT REACT

REACT STATE WITHOUT CONSTRUCTOR

The article is a short tutorial on how to have state in React without a constructor in a class component and how to have state in React without a class component at all. It may be a great refresher on...

HOW TO USECALLBACK IN REACT

React's useCallback Hook can be used to optimize the rendering behavior of your React function components . We will go through an example component to illustrate the problem first, and then solve...



THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.



- ✔ Join 50.000+ Developers
- ✔ Learn Web Development with JavaScript
- ✔ Tips and Tricks
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

Your email address SUBSCRIBE

View our Privacy Policy.

PORTFOLIO **ABOUT**

Online Courses
Open Source
Tutorials
About me
What I use
How to work with me
How to support me

Contact Me Privacy & Terms