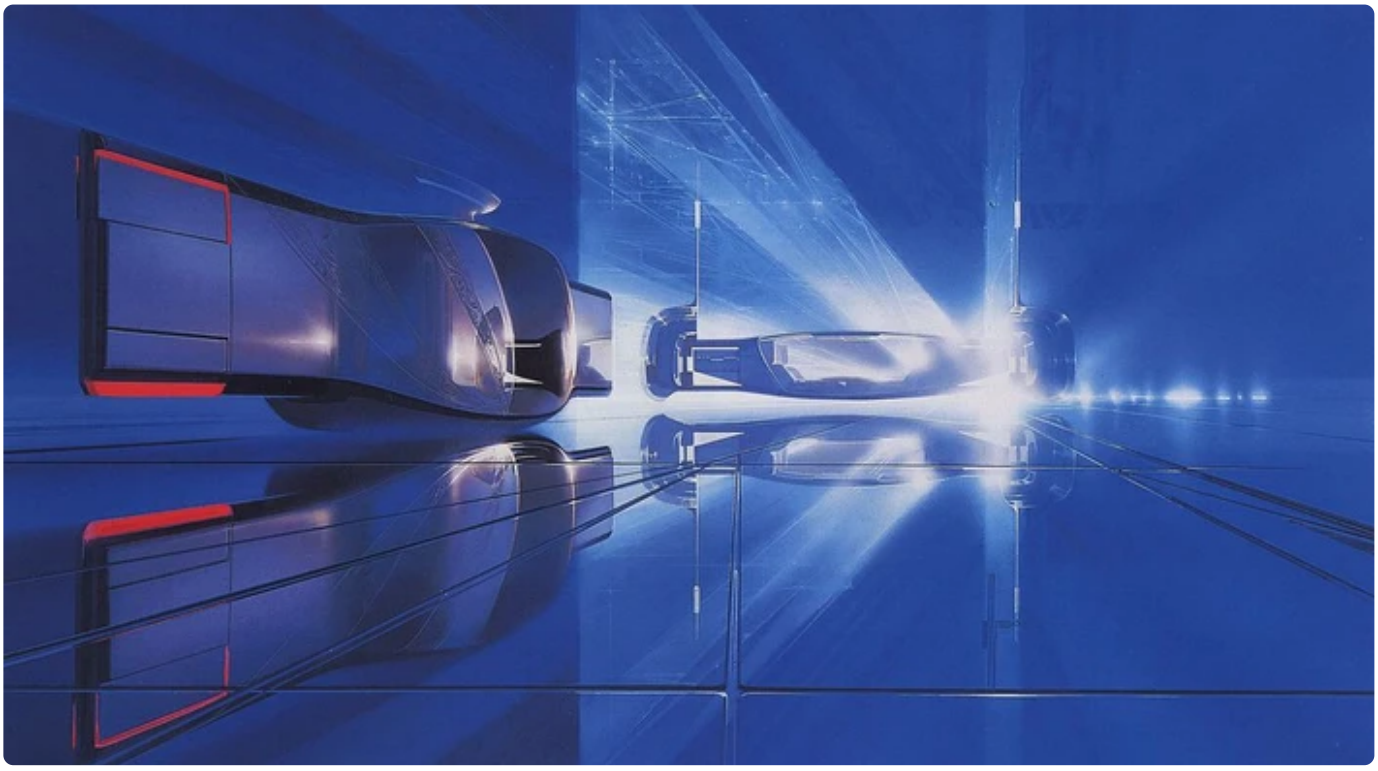


A Simple Explanation of React.useEffect()

Updated May 28, 2022

react hook useeffect

55 Comments

I am impressed by the expressiveness of React hooks. You can do so much by writing so little.

But the brevity of hooks has a price — they're relatively difficult to get started. Especially `useEffect()` — the hook that manages side-effects in functional React components.

In this post, you'll learn how and when to use `useEffect()` hook.

Table of Contents

1. *useEffect()* is for side-effects

A functional React component uses props and/or state to calculate the output. If the functional component makes calculations that don't target the output value, then these calculations are named *side-effects*.

Examples of side-effects are fetch requests, manipulating DOM directly, using timer functions like `setTimeout()`, and more.

The component rendering and side-effect logic are *independent*. It would be a mistake to perform side-effects directly in the body of the component, which is primarily used to compute the output.

How often the component renders isn't something you can control — if React wants to render the component, you cannot stop it.

```
function Greet({ name }) {
  const message = `Hello, ${name}!`; // Calculates output

  // Bad!
  document.title = `Greetings to ${name}`; // Side-effect!

  return <div>{message}</div>;       // Calculates output
```

```
}
```

How to decouple rendering from the side-effect? Welcome `useEffect()` — the hook that runs side-effects independently of rendering.

```
import { useEffect } from 'react';

function Greet({ name }) {
  const message = `Hello, ${name}!`;   // Calculates output

  useEffect(() => {
    // Good!
    document.title = `Greetings to ${name}`; // Side-effect!
  }, [name]);

  return <div>{message}</div>;         // Calculates output
}
```

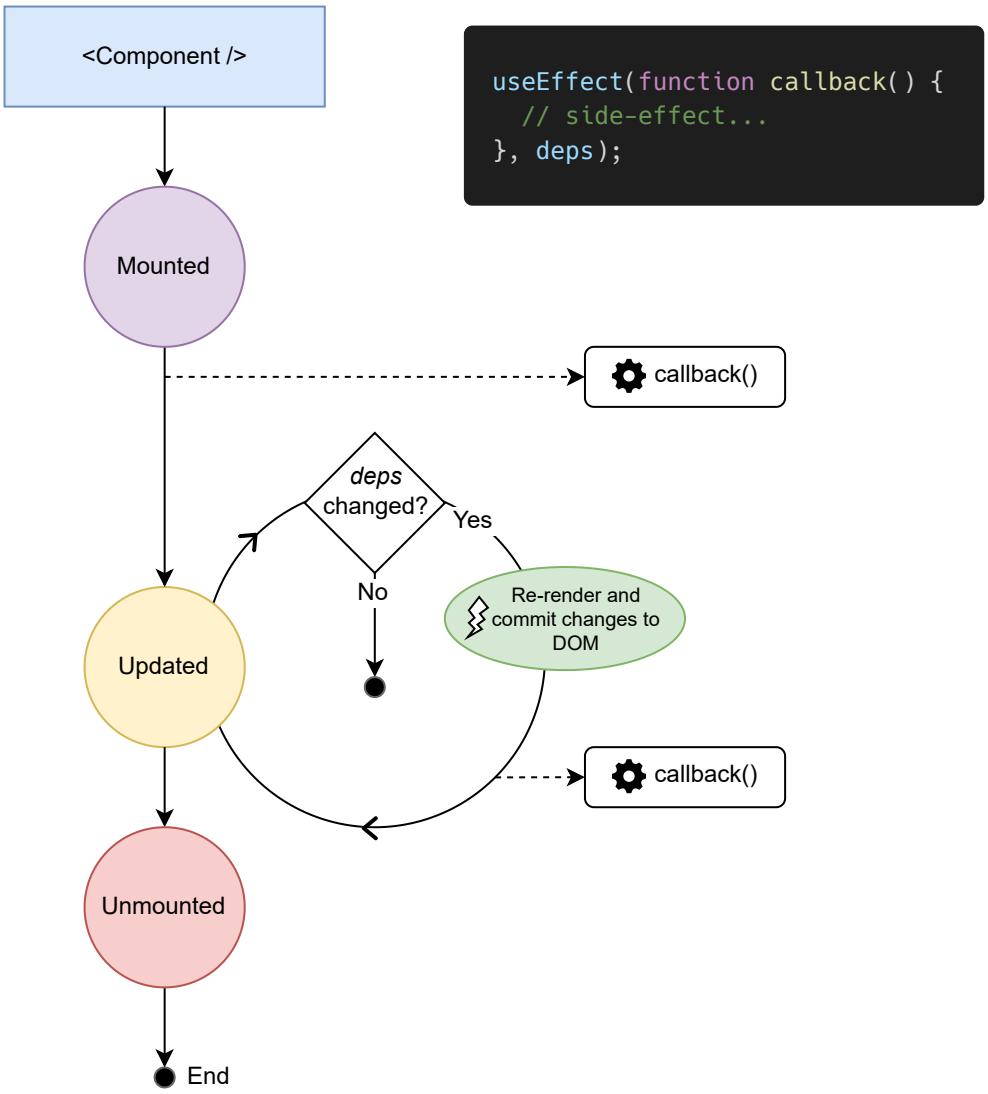
`useEffect()` hook accepts 2 arguments:

```
useEffect(callback[, dependencies]);
```

- `callback` is the function containing the side-effect logic. `callback` is executed right after changes were being pushed to DOM.
- `dependencies` is an optional array of dependencies. `useEffect()` executes `callback` only if the dependencies have changed between renderings.

Put your side-effect logic into the `callback` function, then use the `dependencies` argument to control when you want the side-effect to run. That's the sole purpose of `useEffect()`.

useEffect() Hook



For example, in the previous code snippet you saw the `useEffect()` in action:

```
useEffect(() => {
  document.title = `Greetings to ${name}`;
}, [name]);
```

The *document title update* is the side-effect because it doesn't directly calculate the component output. That's why document title update is placed in a callback and supplied to `useEffect()`.

Also, you don't want the document title update to execute every time `Greet` component renders. You just want it executed when `name` prop changes — that's the reason you supplied `name` as a dependency to `useEffect(callback, [name])`.

2. Dependencies argument

`dependencies` argument of `useEffect(callback, dependencies)` lets you control when the side-effect runs. When dependencies are:

A) Not provided: the side-effect runs after *every* rendering.

```
import { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // Runs after EVERY rendering
  });
}
```

B) An empty array `[]`: the side-effect runs *once* after the initial rendering.

```
import { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // Runs ONCE after initial rendering
  }, []);
}
```

C) Has props or state values `[prop1, prop2, ..., state1, state2]`: the side-effect runs *only when any depenendecy value changes*.

```
import { useEffect, useState } from 'react';

function MyComponent({ prop }) {
  const [state, setState] = useState('');
  useEffect(() => {
    // Runs ONCE after initial rendering
    // and after every rendering ONLY IF `prop` or `state` changes
  }, [prop, state]);
}
```

Let's detail into the cases B) and C) since they're used often.

3. Component lifecycle

3.1 Component did mount

Use an empty dependencies array to invoke a side-effect once after component mounting:

```
import { useEffect } from 'react';

function Greet({ name }) {
  const message = `Hello, ${name}!`;

  useEffect(() => {
    // Runs once, after mounting
    document.title = 'Greetings page';
  }, []);

  return <div>{message}</div>;
}
```

`useEffect(..., [])` was supplied with an empty array as the dependencies argument. When configured in such a way, the `useEffect()` executes the callback *just once*, after initial mounting.

Even if the component re-renders with different `name` property, the side-effect runs only once after the first render:

```
// First render
<Greet name="Eric" />   // Side-effect RUNS

// Second render, name prop changes
<Greet name="Stan" />   // Side-effect DOES NOT RUN

// Third render, name prop changes
<Greet name="Butters"/> // Side-effect DOES NOT RUN
```

Try the demo.

3.2 Component did update

Each time the side-effect uses props or state values, you must indicate these values as dependencies:

```
import { useEffect } from 'react';
```

```
function MyComponent({ prop }) {
  const [state, setState] = useState();

  useEffect(() => {
    // Side-effect uses `prop` and `state`
  }, [prop, state]);

  return <div>....</div>;
}
```

The `useEffect(callback, [prop, state])` invokes the `callback` after the changes are being committed to DOM and *if and only if* any value in the dependencies array `[prop, state]` has changed.

Using the dependencies argument of `useEffect()` you control when to invoke the side-effect, independently from the rendering cycles of the component. Again, *that's the essence of `useEffect()` hook.*

Let's improve the `Greet` component by using `name` prop in the document title:

```
import { useEffect } from 'react';

function Greet({ name }) {
  const message = `Hello, ${name}!`;

  useEffect(() => {
    document.title = `Greetings to ${name}`;
  }, [name]);

  return <div>{message}</div>;
}
```

`name` prop is mentioned in the dependencies argument of `useEffect(..., [name])`. `useEffect()` hook runs the side-effect after initial rendering, and on later renderings only if the `name` value changes.

```
// First render
<Greet name="Eric" />   // Side-effect RUNS

// Second render, name prop changes
<Greet name="Stan" />   // Side-effect RUNS

// Third render, name prop doesn't change
<Greet name="Stan" />   // Side-effect DOES NOT RUN

// Fourth render, name prop changes
```

```
<Greet name="Butters"/> // Side-effect RUNS
```

Try the demo.

5. Side-effect cleanup

Some side-effects need cleanup: close a socket, clear timers.

If the `callback` of `useEffect(callback, deps)` returns a function, then `useEffect()` considers this as an *effect cleanup*:

```
useEffect(() => {
  // Side-effect...

  return function cleanup() {
    // Side-effect cleanup...
  };
}, dependencies);
```

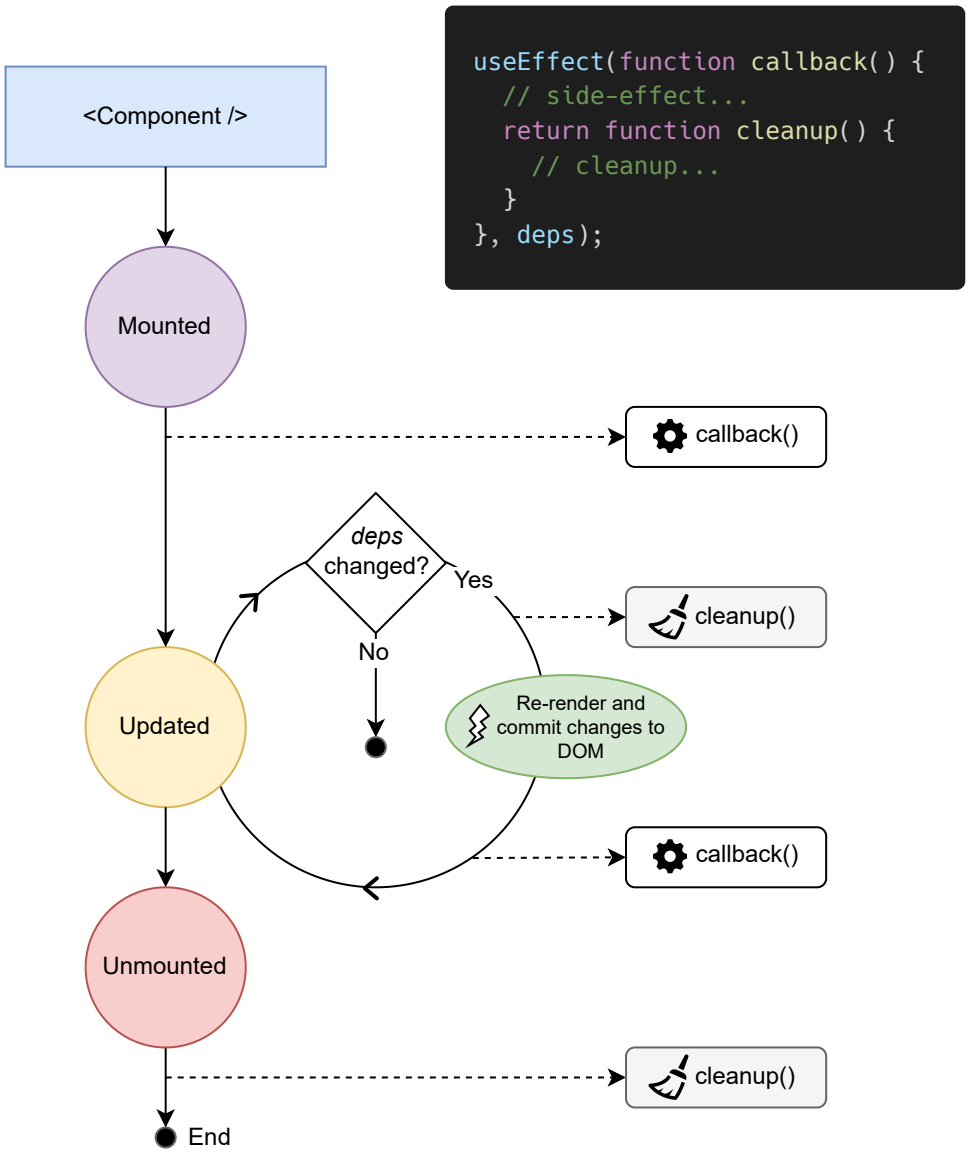
Cleanup works the following way:

A) After initial rendering, `useEffect()` invokes the callback having the side-effect. `cleanup` function is *not invoked*.

B) On later renderings, before invoking the next side-effect callback, `useEffect()` *invokes* the `cleanup` function from the previous side-effect execution (to clean up everything after the previous side-effect), then runs the current side-effect.

C) Finally, after unmounting the component, `useEffect()` *invokes* the cleanup function from the latest side-effect.

useEffect() Hook




```
useEffect(function callback() {
  // side-effect...
  return function cleanup() {
    // cleanup...
  }
}, deps);
```

Let's see an example when the side-effect cleanup is useful.

The following component `<RepeatMessage message="My Message" />` accepts a prop `message`. Then, every 2 seconds the `message` prop is logged to console:

```
import { useEffect } from 'react';

function RepeatMessage({ message }) {
  useEffect(() => {
    setInterval(() => {
      console.log(message);
    }, 2000);
  }, [message]);
```

```
  return <div>I'm logging to console "{message}"</div>;
}
```

Try the demo.

Open the demo and type some messages. The console logs every 2 seconds any message that's been ever typed into the input. However, you need to log only the latest message.

That's the case to clean up the side-effect: cancel the previous timer when starting a new one. Let's return a cleanup function that stops the previous timer before starting a new one:

```
import { useEffect } from 'react';

function RepeatMessage({ message }) {
  useEffect(() => {
    const id = setInterval(() => {
      console.log(message);
    }, 2000);
    return () => {
      clearInterval(id);
    };
  }, [message]);

  return <div>I'm logging to console "{message}"</div>;
}
```

Try the demo.

Open the demo and type some messages. You'll see that each 2 seconds only the latest message logs to console. Which means that all of the previous timers were cleanup.

6. *useEffect()* in practice

6.1 Fetching data

`useEffect()` can perform data fetching side-effect.

The following component `FetchEmployees` fetches the employees list over the network:

```
import { useEffect, useState } from 'react';

function FetchEmployees() {
  const [employees, setEmployees] = useState([]);

  useEffect(() => {
    async function fetchEmployees() {
      const response = await fetch('/employees');
      const fetchedEmployees = await response.json(response);
      setEmployees(fetchedEmployees);
    }

    fetchEmployees();
  }, []);

  return (
    <div>
      {employees.map(name => <div>{name}</div>)}
    </div>
  );
}
```

`useEffect()` starts a fetch request by calling `fetchEmployees()` async function after the initial mounting.

When the request completes, `setEmployees(fetchedEmployees)` updates the `employees` state with the just fetched employees list.

Note that the `callback` argument of `useEffect(callback)` cannot be an `async` function. But you can always define and then invoke an `async` function inside the callback itself:

```
function FetchEmployees() {
  const [employees, setEmployees] = useState([]);

  useEffect(() => {  // <--- CANNOT be an async function
    async function fetchEmployees() {
      // ...
    }
    fetchEmployees(); // <--- But CAN invoke async functions
  }, []);

  // ...
}
```

To run the fetch request depending on a prop or state value, simply indicate the required dependency in the dependencies argument: `useEffect(fetchSideEffect, [prop, stateValue])`.

7. Conclusion

`useEffect(callback, dependencies)` is the hook that manages the side-effects in functional components. `callback` argument is a function to put the side-effect logic. `dependencies` is a list of dependencies of your side-effect: being props or state values.

`useEffect(callback, dependencies)` invokes the `callback` after initial mounting, and on later renderings, if any value inside `dependencies` has changed.

The next step to mastering `useEffect()` is to understand and avoid the infinite loop pitfall.

Still have questions about `useEffect()` hook? Ask in the comments below!

Like the post? Please share!

Suggest Improvement

Enter your email

Join 5887 other subscribers.

About Dmitri Pavlutin

Tech writer and coach. My daily routine consists of (but not limited to) drinking coffee, coding, writing, coaching, overcoming boredom 😉.

Recommended reading:

Use React.memo() wisely

react memoization

Your Guide to React.useCallback()

react memoization