useEffect

`useEffect` is a React Hook that lets you synchronize a component with an external system.

```
useEffect(setup, dependencies?)
```

- My cleanup logic runs even though my component didn't unmount
- My Effect does something visual, and I see a flicker before it runs

---

Reference

`useEffect(setup, dependencies?)`

Call `useEffect` at the top level of your component to declare an Effect:

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

See more examples below.

Parameters

- `setup` : The function with your Effect's logic. Your setup function may also optionally return a *cleanup* function. When your component is added to the DOM, React will run your setup function. After every re-render with changed dependencies, React will first run the cleanup function (if you provided it) with the old values, and then run your setup function with the new values.

After your component is removed from the DOM, React will run your cleanup function.

- **optional** `dependencies`: The list of all reactive values referenced inside of the `setup` code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like `[dep1, dep2, dep3]`. React will compare each dependency with its previous value using the `Object.is` comparison. If you omit this argument, your Effect will re-run after every re-render of the component. See the difference between passing an array of dependencies, an empty array, and no dependencies at all.

Returns

`useEffect` returns `undefined`.

Caveats

- `useEffect` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.
- If you're **not trying to synchronize with some external system,** you probably don't need an Effect.
- When Strict Mode is on, React will **run one extra development-only setup+cleanup cycle** before the first real setup. This is a stress-test that ensures that your cleanup logic "mirrors" your setup logic and that it stops or undoes whatever the setup is doing. If this causes a problem, implement the cleanup function.
- If some of your dependencies are objects or functions defined inside the component, there is a risk that they will **cause the Effect to re-run more often than needed.** To fix this, remove unnecessary object and function dependencies. You can also extract state updates and non-reactive logic outside of your Effect.

- If your Effect wasn't caused by an interaction (like a click), React will generally let the browser **paint the updated screen first before running your Effect.** If your Effect is doing something visual (for example, positioning a tooltip), and the delay is noticeable (for example, it flickers), replace `useEffect` with `useLayoutEffect`.
- Even if your Effect was caused by an interaction (like a click), **the browser may repaint the screen before processing the state updates inside your Effect.** Usually, that's what you want. However, if you must block the browser from repainting the screen, you need to replace `useEffect` with `useLayoutEffect`.
- Effects **only run on the client.** They don't run during server rendering.

Usage

Connecting to an external system

Some components need to stay connected to the network, some browser API, or a third-party library, while they are displayed on the page. These systems aren't controlled by React, so they are called *external.*

To connect your component to some external system, call `useEffect` at the top level of your component:

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
```

```
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

You need to pass two arguments to `useEffect`:

1. A *setup function* with setup code that connects to that system.
   - It should return a *cleanup function* with cleanup code that disconnects from that system.
2. A list of dependencies including every value from your component used inside of those functions.

React calls your setup and cleanup functions whenever it's necessary, which may happen multiple times:

1. Your setup code runs when your component is added to the page *(mounts)*.
2. After every re-render of your component where the dependencies have changed:
   - First, your cleanup code runs with the old props and state.
   - Then, your setup code runs with the new props and state.
3. Your cleanup code runs one final time after your component is removed from the page *(unmounts)*.

Let's illustrate this sequence for the example above.

When the `ChatRoom` component above gets added to the page, it will connect to the chat room with the initial `serverUrl` and `roomId`. If either `serverUrl` or `roomId` change as a result of a re-render (say, if the user picks a different chat room in a dropdown), your Effect will *disconnect from the previous room, and connect to the next one.* When the `ChatRoom` component is removed from the page, your Effect will disconnect one last time.

To help you find bugs, in development React runs `setup` and `cleanup` one extra time before the `setup`. This is a stress-test that verifies your Effect's logic is implemented correctly. If this causes visible issues, your cleanup function is missing some logic. The cleanup function should stop or undo whatever the setup function was doing. The rule of thumb is that the user shouldn't be able to distinguish between the setup being called once (as in production) and a *setup* → *cleanup* → *setup* sequence (as in development). See common solutions.

Try to write every Effect as an independent process and think about a single setup/cleanup cycle at a time. It shouldn't matter whether your component is mounting, updating, or unmounting. When your cleanup logic correctly "mirrors" the setup logic, your Effect is resilient to running setup and cleanup as often as needed.

Note

An Effect lets you keep your component synchronized with some external system (like a chat service). Here, *external system* means any piece of code that's not controlled by React, such as:

- A timer managed with `setInterval()` and `clearInterval()`.
- An event subscription using `window.addEventListener()` and `window.removeEventListener()`.
- A third-party animation library with an API like `animation.start()` and `animation.reset()`.

If you're not connecting to any external system, you probably don't need an Effect.

Examples of connecting to an external system

1. Connecting to a chat server 2. Listening to a global browser event

Example 1 of 5:

Connecting to a chat server

In this example, the `ChatRoom` component uses an Effect to stay connected to an external system defined in `chat.js`. Press “Open chat” to make the `ChatRoom` component appear. This sandbox runs in development mode, so there is an extra connect-and-disconnect cycle, as explained here. Try changing the `roomId` and `serverUrl` using the dropdown and the input, and see how the Effect re-connects to the chat. Press “Close chat” to see the Effect disconnect one last time.

App.js chat.js Reset

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
```

Show more

Next Example

Wrapping Effects in custom Hooks

Effects are an “escape hatch”: you use them when you need to “step outside React” and when there is no better built-in solution for your use case. If you find yourself often needing to manually write Effects, it’s usually a sign that you need to extract some custom Hooks for common behaviors your components rely on.

For example, this `useChatRoom` custom Hook “hides” the logic of your Effect behind a more declarative API:

```
function useChatRoom({ serverUrl, roomId }) {
  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
      roomId: roomId
    };
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, serverUrl]);
```

```
}
```

Then you can use it from any component like this:

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useChatRoom({
    roomId: roomId,
    serverUrl: serverUrl
  });
  // ...
```

There are also many excellent custom Hooks for every purpose available in the React ecosystem.

Learn more about wrapping Effects in custom Hooks.

Examples of wrapping Effects in custom Hooks

1. Custom `useChatRoom` Hook
2. Custom `useWindowListener` Hook

Example 1 of 3:

Custom `useChatRoom` Hook

This example is identical to one of the earlier examples, but the logic is extracted to a custom Hook.

App.js useChatRoom.js chat.js Reset

```
import { useState } from 'react';
import { useChatRoom } from './useChatRoom.js';
```

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:

  useChatRoom({
    roomId: roomId,
    serverUrl: serverUrl
  });
```

Show more

Next Example

Controlling a non-React widget

Sometimes, you want to keep an external system synchronized to some prop or state of your component.

For example, if you have a third-party map widget or a video player component written without React, you can use an Effect to call methods on it that make its state match the current state of your React component. This Effect creates an instance of a `MapWidget` class defined in `map-widget.js`. When you change the `zoomLevel` prop of the `Map` component, the Effect calls the `setZoom()` on the class instance to keep it synchronized:

App.js Map.js map-widget.js Reset

```
import { useRef, useEffect } from 'react';
import { MapWidget } from './map-widget.js';

export default function Map({ zoomLevel }) {
  const containerRef = useRef(null);
  const mapRef = useRef(null);

  useEffect(() => {
    if (mapRef.current === null) {
      mapRef.current = new MapWidget(containerRef.current);
    }
```

Show more

In this example, a cleanup function is not needed because the `MapWidget` class manages only the DOM node that was passed to it. After the `Map` React component is removed from the tree, both the DOM node and the `MapWidget` class instance will be automatically garbage-collected by the browser JavaScript engine.

Fetching data with Effects

You can use an Effect to fetch data for your component. Note that if you use a framework, using your framework's data fetching mechanism will be a lot more efficient than writing Effects manually.

If you want to fetch data from an Effect manually, your code might look like this:

```
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
  const [person, setPerson] = useState('Alice');
  const [bio, setBio] = useState(null);

  useEffect(() => {
    let ignore = false;
    setBio(null);
    fetchBio(person).then(result => {
      if (!ignore) {
        setBio(result);
      }
    });
    return () => {
      ignore = true;
    };
  }, [person]);
```

```
// ...
```

Note the `ignore` variable which is initialized to `false`, and is set to `true` during cleanup. This ensures your code doesn't suffer from "race conditions": network responses may arrive in a different order than you sent them.

App.js

Reset

```
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
  const [person, setPerson] = useState('Alice');
  const [bio, setBio] = useState(null);
  useEffect(() => {
    let ignore = false;
    setBio(null);
    fetchBio(person).then(result => {
      if (!ignore) {
        setBio(result);
```

Show more

You can also rewrite using the `async / await` syntax, but you still need to provide a cleanup function:

App.js

Reset

```
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
  const [person, setPerson] = useState('Alice');
  const [bio, setBio] = useState(null);
  useEffect(() => {
    async function startFetching() {
      setBio(null);
      const result = await fetchBio(person);
      if (!ignore) {
        setBio(result);
```

Show more

Writing data fetching directly in Effects gets repetitive and makes it difficult to add optimizations like caching and server rendering later. It's easier to use a

custom Hook—either your own or maintained by the community.

DEEP DIVE

What are good alternatives to data fetching in Effects?

Show Details

Specifying reactive dependencies

Notice that you can't "choose" the dependencies of your Effect. Every reactive value used by your Effect's code must be declared as a dependency. Your Effect's dependency list is determined by the surrounding code:

```
function ChatRoom({ roomId }) { // This is a reactive value
  const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // T

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId); // This Effect
    connection.connect();
    return () => connection.disconnect();
  }, [serverUrl, roomId]); // ✅ So you must specify them as dependencies
  // ...
}
```

If either `serverUrl` or `roomId` change, your Effect will reconnect to the chat using the new values.

Reactive values include props and all variables and functions declared directly inside of your component. Since `roomId` and `serverUrl` are reactive values, you can't remove them from the dependencies. If you try to omit them and your linter is correctly configured for React, the linter will flag this as a mistake you need to fix:

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []); // 🔴 React Hook useEffect has missing dependencies: 'roomId' and
  // ...
}
```

To remove a dependency, you need to "prove" to the linter that it *doesn't need* to be a dependency. For example, you can move `serverUrl` out of your component to prove that it's not reactive and won't change on re-renders:

```
const serverUrl = 'https://localhost:1234'; // Not a reactive value anymore

function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]); // ✅ All dependencies declared
  // ...
}
```

Now that `serverUrl` is not a reactive value (and can't change on a re-render), it doesn't need to be a dependency. **If your Effect's code doesn't use any reactive values, its dependency list should be empty (`[]`):**

```
const serverUrl = 'https://localhost:1234'; // Not a reactive value anymore
const roomId = 'music'; // Not a reactive value anymore

function ChatRoom() {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []); // ✅ All dependencies declared
  // ...
}
```

An Effect with empty dependencies doesn't re-run when any of your component's props or state change.

Pitfall

If you have an existing codebase, you might have some Effects that suppress the linter like this:

```
useEffect(() => {
  // ...
  // 🔴 Avoid suppressing the linter like this:
  // eslint-ignore-next-line react-hooks/exhaustive-deps
}, []);
```

When dependencies don't match the code, there is a high risk of introducing bugs. By suppressing the linter, you "lie" to React about the values your Effect depends on. Instead, prove they're unnecessary.

Examples of passing reactive dependencies

1. Passing a dependency array
2. Passing an empty dependency array

Example 1 of 3:

Passing a dependency array

If you specify the dependencies, your Effect runs **after the initial render *and* after re-renders with changed dependencies.**

```
useEffect(() => {
  // ...
}, [a, b]); // Runs again if a or b are different
```

In the below example, `serverUrl` and `roomId` are reactive values, so they both must be specified as dependencies. As a result, selecting a different room in the dropdown or editing the server URL input causes the chat to re-connect. However, since `message` isn't used in the Effect (and so it isn't a dependency), editing the message doesn't re-connect to the chat.

App.js chat.js Reset

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';
```

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:
  const [message, setMessage] = useState('');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
```

Show more

Next Example

Updating state based on previous state from an Effect

When you want to update state based on previous state from an Effect, you might run into a problem:

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const intervalId = setInterval(() => {
      setCount(count + 1); // You want to increment the counter every second.
    }, 1000)
    return () => clearInterval(intervalId);
  }, [count]); // 🚩 ... but specifying `count` as a dependency always reset
  // ...
}
```

Since `count` is a reactive value, it must be specified in the list of dependencies. However, that causes the Effect to cleanup and setup again every time the `count` changes. This is not ideal.

To fix this, pass the `c => c + 1` state updater to `setCount`:

App.js Download Reset

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

export default function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const intervalId = setInterval(() => {
      setCount(c => c + 1); // ✅ Pass a state updater
    }, 1000);
    return () => clearInterval(intervalId);
  }, []); // ✅ Now count is not a dependency
```

Now that you're passing `c => c + 1` instead of `count + 1`, your Effect no longer needs to depend on `count`. As a result of this fix, it won't need to cleanup and setup the interval again every time the `count` changes.

Removing unnecessary object dependencies

If your Effect depends on an object or a function created during rendering, it might run too often. For example, this Effect re-connects after every render because the `options` object is different for every render:

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const options = { // 🚩 This object is created from scratch on every re-r
    serverUrl: serverUrl,
    roomId: roomId
  };

  useEffect(() => {
    const connection = createConnection(options); // It's used inside the E
    connection.connect();
    return () => connection.disconnect();
```

```
}, [options]); // 🚩 As a result, these dependencies are always differen
// ...
```

Avoid using an object created during rendering as a dependency. Instead, create the object inside the Effect:

App.js chat.js Reset

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
```

Show more

Now that you create the `options` object inside the Effect, the Effect itself only depends on the `roomId` string.

With this fix, typing into the input doesn't reconnect the chat. Unlike an object which gets re-created, a string like `roomId` doesn't change unless you set it to another value. Read more about removing dependencies.

Removing unnecessary function dependencies

If your Effect depends on an object or a function created during rendering, it might run too often. For example, this Effect re-connects after every render because the `createOptions` function is different for every render:

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  function createOptions() { // 🚩 This function is created from scratch on e
    return {
      serverUrl: serverUrl,
      roomId: roomId
    };
  }

  useEffect(() => {
    const options = createOptions(); // It's used inside the Effect
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // 🚩 As a result, these dependencies are always diffe
  // ...
```

By itself, creating a function from scratch on every re-render is not a problem. You don't need to optimize that. However, if you use it as a dependency of your Effect, it will cause your Effect to re-run after every re-render.

Avoid using a function created during rendering as a dependency. Instead, declare it inside the Effect:

App.js chat.js Reset

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    function createOptions() {
      return {
```

Show more

Now that you define the `createOptions` function inside the Effect, the Effect itself only depends on the `roomId` string. With this fix, typing into the input doesn't reconnect the chat. Unlike a function which gets re-created, a string like `roomId` doesn't change unless you set it to another value. Read more about removing dependencies.

Reading the latest props and state from an Effect

Under Construction

This section describes an **experimental API that has not yet been released** in a stable version of React.

By default, when you read a reactive value from an Effect, you have to add it as a dependency. This ensures that your Effect "reacts" to every change of that value. For most dependencies, that's the behavior you want.

However, sometimes you'll want to read the *latest* props and state from an Effect without "reacting" to them. For example, imagine you want to log the number of the items in the shopping cart for every page visit:

```
function Page({ url, shoppingCart }) {
  useEffect(() => {
    logVisit(url, shoppingCart.length);
  }, [url, shoppingCart]); // ✅ All dependencies declared
  // ...
}
```

What if you want to log a new page visit after every `url` change, but *not* if only the `shoppingCart` changes? You can't exclude `shoppingCart` from dependencies without breaking the reactivity rules. However, you can express that you *don't want* a piece of code to "react" to changes even though it is called from inside an Effect. Declare an *Effect Event* with the `useEffectEvent` Hook, and move the code reading `shoppingCart` inside of it:

```
function Page({ url, shoppingCart }) {
  const onVisit = useEffectEvent(visitedUrl => {
    logVisit(visitedUrl, shoppingCart.length)
  });

  useEffect(() => {
    onVisit(url);
  }, [url]); // ✅ All dependencies declared
  // ...
}
```

Effect Events are not reactive and must always be omitted from dependencies of your Effect. This is what lets you put non-reactive code (where you can read the latest value of some props and state) inside of them. By reading `shoppingCart` inside of `onVisit`, you ensure that `shoppingCart` won't re-run your Effect.

Read more about how Effect Events let you separate reactive and non-reactive code.

Displaying different content on the server and the client

If your app uses server rendering (either directly or via a framework), your component will render in two different environments. On the server, it will

render to produce the initial HTML. On the client, React will run the rendering code again so that it can attach your event handlers to that HTML. This is why, for hydration to work, your initial render output must be identical on the client and the server.

In rare cases, you might need to display different content on the client. For example, if your app reads some data from `localStorage`, it can't possibly do that on the server. Here is how you could implement this:

```
function MyComponent() {
  const [didMount, setDidMount] = useState(false);

  useEffect(() => {
    setDidMount(true);
  }, []);

  if (didMount) {
    // ... return client-only JSX ...
  }  else {
    // ... return initial JSX ...
  }
}
```

While the app is loading, the user will see the initial render output. Then, when it's loaded and hydrated, your Effect will run and set `didMount` to `true`, triggering a re-render. This will switch to the client-only render output. Effects don't run on the server, so this is why `didMount` was `false` during the initial server render.

Use this pattern sparingly. Keep in mind that users with a slow connection will see the initial content for quite a bit of time—potentially, many seconds—so you don't want to make jarring changes to your component's appearance. In many cases, you can avoid the need for this by conditionally showing different things with CSS.

Troubleshooting

My Effect runs twice when the component mounts

When Strict Mode is on, in development, React runs setup and cleanup one extra time before the actual setup.

This is a stress-test that verifies your Effect's logic is implemented correctly. If this causes visible issues, your cleanup function is missing some logic. The cleanup function should stop or undo whatever the setup function was doing. The rule of thumb is that the user shouldn't be able to distinguish between the setup being called once (as in production) and a setup → cleanup → setup sequence (as in development).

Read more about how this helps find bugs and how to fix your logic.

My Effect runs after every re-render

First, check that you haven't forgotten to specify the dependency array:

```
useEffect(() => {
  // ...
}); // 🚩 No dependency array: re-runs after every render!
```

If you've specified the dependency array but your Effect still re-runs in a loop, it's because one of your dependencies is different on every re-render.

You can debug this problem by manually logging your dependencies to the console:

```
  useEffect(() => {
    // ..
```

```
  }, [serverUrl, roomId]);

  console.log([serverUrl, roomId]);
```

You can then right-click on the arrays from different re-renders in the console and select “Store as a global variable” for both of them. Assuming the first one got saved as `temp1` and the second one got saved as `temp2`, you can then use the browser console to check whether each dependency in both arrays is the same:

```
Object.is(temp1[0], temp2[0]); // Is the first dependency the same between th
Object.is(temp1[1], temp2[1]); // Is the second dependency the same between t
Object.is(temp1[2], temp2[2]); // ... and so on for every dependency ...
```

When you find the dependency that is different on every re-render, you can usually fix it in one of these ways:

- Updating state based on previous state from an Effect
- Removing unnecessary object dependencies
- Removing unnecessary function dependencies
- Reading the latest props and state from an Effect

As a last resort (if these methods didn’t help), wrap its creation with `useMemo` or `useCallback` (for functions).

My Effect keeps re-running in an infinite cycle

If your Effect runs in an infinite cycle, these two things must be true:

- Your Effect is updating some state.
- That state leads to a re-render, which causes the Effect’s dependencies to change.

Before you start fixing the problem, ask yourself whether your Effect is connecting to some external system (like DOM, network, a third-party widget, and so on). Why does your Effect need to set state? Does it synchronize with that external system? Or are you trying to manage your application's data flow with it?

If there is no external system, consider whether removing the Effect altogether would simplify your logic.

If you're genuinely synchronizing with some external system, think about why and under what conditions your Effect should update the state. Has something changed that affects your component's visual output? If you need to keep track of some data that isn't used by rendering, a ref (which doesn't trigger re-renders) might be more appropriate. Verify your Effect doesn't update the state (and trigger re-renders) more than needed.

Finally, if your Effect is updating the state at the right time, but there is still a loop, it's because that state update leads to one of the Effect's dependencies changing. Read how to debug dependency changes.

My cleanup logic runs even though my component didn't unmount

The cleanup function runs not only during unmount, but before every re-render with changed dependencies. Additionally, in development, React runs setup+cleanup one extra time immediately after component mounts.

If you have cleanup code without corresponding setup code, it's usually a code smell:

```
useEffect(() => {
  // 🔴 Avoid: Cleanup logic without corresponding setup logic
  return () => {
```

```
    doSomething();
  };
}, []);
```

Your cleanup logic should be “symmetrical” to the setup logic, and should stop or undo whatever setup did:

```
useEffect(() => {
  const connection = createConnection(serverUrl, roomId);
  connection.connect();
  return () => {
    connection.disconnect();
  };
}, [serverUrl, roomId]);
```

Learn how the Effect lifecycle is different from the component’s lifecycle.

My Effect does something visual, and I see a flicker before it runs

If your Effect must block the browser from painting the screen, replace `useEffect` with `useLayoutEffect`. Note that **this shouldn’t be needed for the vast majority of Effects.** You’ll only need this if it’s crucial to run your Effect before the browser paint: for example, to measure and position a tooltip before the user sees it.

PREVIOUS
useDeferredValue

NEXT

useId

How do you like these docs?

Take our survey!

Meta Open Source

Learn React

- Quick Start
- Installation
- Describing the UI
- Adding Interactivity
- Managing State
- Escape Hatches

API Reference

- React APIs
- React DOM APIs

Community

- Code of Conduct
- Meet the Team
- Docs Contributors
- Acknowledgements

More

- Blog
- React Native
- Privacy
- Terms