LEARN REACT > ESCAPE HATCHES >

Synchronizing with Effects

Some components need to synchronize with external systems. For example, you might want to control a non-React component based on the React state, set up a server connection, or send an analytics log when a component appears on the screen. *Effects* let you run some code after rendering so that you can synchronize your component with some system outside of React.

You will learn

- What Effects are
- How Effects are different from events
- How to declare an Effect in your component
- How to skip re-running an Effect unnecessarily
- Why Effects run twice in development and how to fix them

What are Effects and how are they different from events?

Before getting to Effects, you need to be familiar with two types of logic inside React components:

- **Rendering code** (introduced in Describing the UI) lives at the top level of your component. This is where you take the props and state, transform them, and return the JSX you want to see on the screen. Rendering code must be pure. Like a math formula, it should only *calculate* the result, but not do anything else.

- **Event handlers** (introduced in Adding Interactivity) are nested functions inside your components that *do* things rather than just calculate them. An event handler might update an input field, submit an HTTP POST request to buy a product, or navigate the user to another screen. Event handlers contain "side effects" (they change the program's state) caused by a specific user action (for example, a button click or typing).

Sometimes this isn't enough. Consider a `ChatRoom` component that must connect to the chat server whenever it's visible on the screen. Connecting to a server is not a pure calculation (it's a side effect) so it can't happen during rendering. However, there is no single particular event like a click that causes `ChatRoom` to be displayed.

***Effects* let you specify side effects that are caused by rendering itself, rather than by a particular event.** Sending a message in the chat is an *event* because it is directly caused by the user clicking a specific button. However, setting up a server connection is an *Effect* because it should happen no matter which interaction caused the component to appear. Effects run at the end of a commit after the screen updates. This is a good time to synchronize the React components with some external system (like network or a third-party library).

Note

Here and later in this text, capitalized "Effect" refers to the React-specific definition above, i.e. a side effect caused by rendering. To refer to the broader programming concept, we'll say "side effect".

You might not need an Effect

Don't rush to add Effects to your components. Keep in mind that Effects are typically used to “step out” of your React code and synchronize with some *external* system. This includes browser APIs, third-party widgets, network, and so on. If your Effect only adjusts some state based on other state, you might not need an Effect.

How to write an Effect

To write an Effect, follow these three steps:

1. **Declare an Effect.** By default, your Effect will run after every render.
2. **Specify the Effect dependencies.** Most Effects should only re-run *when needed* rather than after every render. For example, a fade-in animation should only trigger when a component appears. Connecting and disconnecting to a chat room should only happen when the component appears and disappears, or when the chat room changes. You will learn how to control this by specifying *dependencies.*
3. **Add cleanup if needed.** Some Effects need to specify how to stop, undo, or clean up whatever they were doing. For example, “connect” needs “disconnect”, “subscribe” needs “unsubscribe”, and “fetch” needs either “cancel” or “ignore”. You will learn how to do this by returning a *cleanup function*.

Let’s look at each of these steps in detail.

Step 1: Declare an Effect

To declare an Effect in your component, import the `useEffect` Hook from React:

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

Then, call it at the top level of your component and put some code inside your Effect.

```
function MyComponent() {
  useEffect(() => {
    // Code here will run after *every* render
  });
  return <div />;
}
```

Every time your component renders, React will update the screen *and then* run the code inside `useEffect`. In other words, **`useEffect` "delays" a piece of code from running until that render is reflected on the screen.**

Let's see how you can use an Effect to synchronize with an external system. Consider a `<VideoPlayer>` React component. It would be nice to control whether it's playing or paused by passing an `isPlaying` prop to it:

```
<VideoPlayer isPlaying={isPlaying} />;
```

Your custom `VideoPlayer` component renders the built-in browser `<video>` tag:

```
function VideoPlayer({ src, isPlaying }) {
  // TODO: do something with isPlaying
  return <video src={src} />;
}
```

However, the browser `<video>` tag does not have an `isPlaying` prop. The only way to control it is to manually call the `play()` and `pause()` methods on the DOM element. **You need to synchronize the value of `isPlaying` prop, which tells whether the video *should* currently be playing, with calls like** `play()` and `pause()`.

We'll need to first get a ref to the `<video>` DOM node.

You might be tempted to try to call `play()` or `pause()` during rendering, but that isn't correct:

App.js Download Reset

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

function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  if (isPlaying) {
    ref.current.play();  // Calling these while rendering isn't allowed
  } else {
    ref.current.pause(); // Also, this crashes.
  }

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  return (
    <>
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-vic
      />
    </>
  );
}

```

Show less

Runtime Error

```
App.js: ref.current is null (9:4)

   6 |   if (isPlaying) {
   7 |     ref.current.play();  // Calling these while rendering
isn't allowed.
   8 |   } else {
>  9 |     ref.current.pause(); // Also, this crashes.
         ^
  10 |   }
  11 |
  12 |   return <video ref={ref} src={src} loop playsInline />;
```

The reason this code isn't correct is that it tries to do something with the DOM node during rendering. In React, rendering should be a pure calculation of JSX and should not contain side effects like modifying the DOM.

Moreover, when `VideoPlayer` is called for the first time, its DOM does not exist yet! There isn't a DOM node yet to call `play()` or `pause()` on, because React doesn't know what DOM to create until you return the JSX.

The solution here is to **wrap the side effect with `useEffect` to move it out of the rendering calculation:**

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

function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      ref.current.play();
    } else {
      ref.current.pause();
    }
  });

  return <video ref={ref} src={src} loop playsInline />;
}
```

By wrapping the DOM update in an Effect, you let React update the screen first. Then your Effect runs.

When your `VideoPlayer` component renders (either the first time or if it re-renders), a few things will happen. First, React will update the screen, ensuring the `<video>` tag is in the DOM with the right props. Then React will run your Effect. Finally, your Effect will call `play()` or `pause()` depending on the value of `isPlaying`.

Press Play/Pause multiple times and see how the video player stays synchronized to the `isPlaying` value:

App.js Download Reset

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

function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      ref.current.play();
    } else {
      ref.current.pause();
    }
  });

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  return (
    <>
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-vic
      />
    </>
  );
}

```

Show less

In this example, the "external system" you synchronized to React state was the browser media API. You can use a similar approach to wrap legacy non-

React code (like jQuery plugins) into declarative React components.

Note that controlling a video player is much more complex in practice. Calling `play()` may fail, the user might play or pause using the built-in browser controls, and so on. This example is very simplified and incomplete.

Pitfall

By default, Effects run after *every* render. This is why code like this **will produce an infinite loop:**

```
const [count, setCount] = useState(0);
useEffect(() => {
  setCount(count + 1);
});
```

Effects run as a *result* of rendering. Setting state *triggers* rendering. Setting state immediately in an Effect is like plugging a power outlet into itself. The Effect runs, it sets the state, which causes a re-render, which causes the Effect to run, it sets the state again, this causes another re-render, and so on.

Effects should usually synchronize your components with an *external* system. If there's no external system and you only want to adjust some state based on other state, you might not need an Effect.

Step 2: Specify the Effect dependencies

By default, Effects run after *every* render. Often, this is **not what you want:**

- Sometimes, it’s slow. Synchronizing with an external system is not always instant, so you might want to skip doing it unless it’s necessary. For example, you don’t want to reconnect to the chat server on every keystroke.
- Sometimes, it’s wrong. For example, you don’t want to trigger a component fade-in animation on every keystroke. The animation should only play once when the component appears for the first time.

To demonstrate the issue, here is the previous example with a few `console.log` calls and a text input that updates the parent component’s state. Notice how typing causes the Effect to re-run:

App.js Download Reset

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

function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      console.log('Calling video.play()');
      ref.current.play();
    } else {
      console.log('Calling video.pause()');
      ref.current.pause();
    }
  });

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  const [text, setText] = useState('');
  return (
    <>
24        <input value={text} onChange={e => setText(e.target.value)} />
25        <button onClick={() => setIsPlaying(!isPlaying)}>
26          {isPlaying ? 'Pause' : 'Play'}
27        </button>
28        <VideoPlayer
29          isPlaying={isPlaying}
30          src="https://interactive-examples.mdn.mozilla.net/media/cc0-vic
31        />
32      </>
33    );
34  }
35
```

Show less

Console (2)

```
Calling video.pause()
Calling video.pause()
```

You can tell React to **skip unnecessarily re-running the Effect** by specifying an array of *dependencies* as the second argument to the `useEffect` call. Start by adding an empty `[]` array to the above example on line 14:

```
useEffect(() => {
  // ...
  }, []);
```

You should see an error saying `React Hook useEffect has a missing dependency: 'isPlaying'`:

App.js Download Reset

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

function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      console.log('Calling video.play()');
      ref.current.play();
    } else {
      console.log('Calling video.pause()');
      ref.current.pause();
    }
  }, []); // This causes an error

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  const [text, setText] = useState('');
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-vic
      />
    </>
  );
}

```

Show less

Lint Error

```
14:6 - React Hook useEffect has a missing dependency: 'isPlaying'.
Either include it or remove the dependency array.
```

The problem is that the code inside of your Effect *depends on* the `isPlaying` prop to decide what to do, but this dependency was not explicitly declared. To fix this issue, add `isPlaying` to the dependency array:

```
useEffect(() => {
  if (isPlaying) { // It's used here...
    // ...
  } else {
    // ...
  }
}, [isPlaying]); // ...so it must be declared here!
```

Now all dependencies are declared, so there is no error. Specifying

Now all dependencies are declared, so there is no error. Specifying `[isPlaying]` as the dependency array tells React that it should skip re-running your Effect if `isPlaying` is the same as it was during the previous render. With this change, typing into the input doesn't cause the Effect to re-run, but pressing Play/Pause does:

App.js Download Reset

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

function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      console.log('Calling video.play()');
      ref.current.play();
    } else {
      console.log('Calling video.pause()');
      ref.current.pause();
    }
  }, [isPlaying]);

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  const [text, setText] = useState('');
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-vic
      />
    </>
  );
}

```

Show less

Console (2)

```
Calling video.pause()
Calling video.pause()
```

The dependency array can contain multiple dependencies. React will only skip re-running the Effect if *all* of the dependencies you specify have exactly the same values as they had during the previous render. React compares the dependency values using the `Object.is` comparison. See the `useEffect` reference for details.

Notice that you can't "choose" your dependencies. You will get a lint error if the dependencies you specified don't match what React expects based on the code inside your Effect. This helps catch many bugs in your code. If you don't want some code to re-run, *edit the Effect code itself* to not "need" that dependency.

Pitfall

The behaviors without the dependency array and with an *empty* `[]` dependency array are different:

```
useEffect(() => {
  // This runs after every render
});

useEffect(() => {
  // This runs only on mount (when the component appears)
}, []);

useEffect(() => {
  // This runs on mount *and also* if either a or b have changed s
}, [a, b]);
```

We'll take a close look at what "mount" means in the next step.

DEEP DIVE

Why was the ref omitted from the dependency array?

Hide Details

This Effect uses *both* `ref` and `isPlaying`, but only `isPlaying` is declared as a dependency:

```
function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);
  useEffect(() => {
    if (isPlaying) {
      ref.current.play();
    } else {
      ref.current.pause();
    }
  }, [isPlaying]);
```

This is because the `ref` object has a *stable identity:* React guarantees you'll always get the same object from the same `useRef` call on every render. It never changes, so it will never by itself cause the Effect to re-run. Therefore, it does not matter whether you include it or not. Including it is fine too:

```
function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);
  useEffect(() => {
    if (isPlaying) {
      ref.current.play();
    } else {
      ref.current.pause();
    }
  }, [isPlaying, ref]);
```

The `set` functions returned by `useState` also have stable identity, so you will often see them omitted from the dependencies too. If the linter lets you omit a dependency without errors, it is safe to do.

Omitting always-stable dependencies only works when the linter can "see" that the object is stable. For example, if `ref` was passed from a parent component, you would have to specify it in the dependency array. However, this is good because you can't know whether the parent component always passes the same ref, or passes one of several refs conditionally. So your Effect *would* depend on which ref is passed.

Step 3: Add cleanup if needed

Consider a different example. You're writing a `ChatRoom` component that needs to connect to the chat server when it appears. You are given a `createConnection()` API that returns an object with `connect()` and `disconnect()` methods. How do you keep the component connected while it is displayed to the user?

Start by writing the Effect logic:

```
useEffect(() => {
  const connection = createConnection();
  connection.connect();
});
```

It would be slow to connect to the chat after every re-render, so you add the dependency array:

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

The code inside the Effect does not use any props or state, so your dependency array is `[]` (empty). This tells React to only run this code when

the component "mounts", i.e. appears on the screen for the first time.

Let's try running this code:

App.js chat.js Reset

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

export default function ChatRoom() {
  useEffect(() => {
    const connection = createConnection();
    connection.connect();
  }, []);
  return <h1>Welcome to the chat!</h1>;
}

```

Console (2)

```
✅ Connecting...
✅ Connecting...
```

This Effect only runs on mount, so you might expect `"✅ Connecting..."` to be printed once in the console. **However, if you check the console, `"✅ Connecting..."` gets printed twice. Why does it happen?**

Imagine the `ChatRoom` component is a part of a larger app with many different screens. The user starts their journey on the `ChatRoom` page. The component mounts and calls `connection.connect()`. Then imagine the user navigates to another screen—for example, to the Settings page. The `ChatRoom` component unmounts. Finally, the user clicks Back and `ChatRoom` mounts again. This would set up a second connection—but the first connection was never destroyed! As the user navigates across the app, the connections would keep piling up.

Bugs like this are easy to miss without extensive manual testing. To help you spot them quickly, in development React remounts every component once immediately after its initial mount.

Seeing the `"✅ Connecting..."` log twice helps you notice the real issue: your code doesn't close the connection when the component unmounts.

To fix the issue, return a *cleanup function* from your Effect:

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

React will call your cleanup function each time before the Effect runs again, and one final time when the component unmounts (gets removed). Let's see what happens when the cleanup function is implemented:

App.js chat.js Reset

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

export default function ChatRoom() {
  useEffect(() => {
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, []);
  return <h1>Welcome to the chat!</h1>;
}

```

Console (3)

```
✅ Connecting...
❌ Disconnected.
✅ Connecting...
```

Now you get three console logs in development:

1. "✅ Connecting..."
2. "❌ Disconnected."
3. "✅ Connecting..."

This is the correct behavior in development. By remounting your component, React verifies that navigating away and back would not break your code. Disconnecting and then connecting again is exactly what should happen! When you implement the cleanup well, there should be no user-visible difference between running the Effect once vs running it, cleaning it up, and running it again. There's an extra connect/disconnect call pair because React is probing your code for bugs in development. This is normal —don't try to make it go away!

In production, you would only see `"✅ Connecting..."` printed once. Remounting components only happens in development to help you find Effects that need cleanup. You can turn off Strict Mode to opt out of the development behavior, but we recommend keeping it on. This lets you find many bugs like the one above.

How to handle the Effect firing twice in development?

React intentionally remounts your components in development to find bugs like in the last example. **The right question isn't "how to run an Effect once", but "how to fix my Effect so that it works after remounting".**

Usually, the answer is to implement the cleanup function. The cleanup function should stop or undo whatever the Effect was doing. The rule of thumb is that the user shouldn't be able to distinguish between the Effect running once (as in production) and a *setup* → *cleanup* → *setup* sequence (as you'd see in development).

Most of the Effects you'll write will fit into one of the common patterns below.

Controlling non-React widgets

Sometimes you need to add UI widgets that aren't written to React. For example, let's say you're adding a map component to your page. It has a `setZoomLevel()` method, and you'd like to keep the zoom level in sync with a `zoomLevel` state variable in your React code. Your Effect would look like similar to this:

similar to this:

```
useEffect(() => {
  const map = mapRef.current;
  map.setZoomLevel(zoomLevel);
}, [zoomLevel]);
```

Note that there is no cleanup needed in this case. In development, React will call the Effect twice, but this is not a problem because calling `setZoomLevel` twice with the same value does not do anything. It may be slightly slower, but this doesn't matter because it won't remount needlessly in production.

Some APIs may not allow you to call them twice in a row. For example, the `showModal` method of the built-in `<dialog>` element throws if you call it twice. Implement the cleanup function and make it close the dialog:

```
useEffect(() => {
  const dialog = dialogRef.current;
  dialog.showModal();
  return () => dialog.close();
}, []);
```

In development, your Effect will call `showModal()`, then immediately `close()`, and then `showModal()` again. This has the same user-visible behavior as calling `showModal()` once, as you would see in production.

Subscribing to events

If your Effect subscribes to something, the cleanup function should unsubscribe:

```
useEffect(() => {
  function handleScroll(e) {
    console.log(window.scrollX, window.scrollY);
  }
  window.addEventListener('scroll', handleScroll);
  return () => window.removeEventListener('scroll', handleScroll);
}, []);
```

In development, your Effect will call `addEventListener()`, then immediately `removeEventListener()`, and then `addEventListener()` again with the same handler. So there would be only one active subscription at a time. This has the same user-visible behavior as calling `addEventListener()` once, as in production.

Triggering animations

If your Effect animates something in, the cleanup function should reset the animation to the initial values:

```
useEffect(() => {
  const node = ref.current;
  node.style.opacity = 1; // Trigger the animation
  return () => {
    node.style.opacity = 0; // Reset to the initial value
  };
}, []);
```

In development, opacity will be set to `1`, then to `0`, and then to `1` again. This should have the same user-visible behavior as setting it to `1` directly, which is what would happen in production. If you use a third-party animation library with support for tweening, your cleanup function should reset the timeline to its initial state.

Fetching data

If your Effect fetches something, the cleanup function should either abort the fetch or ignore its result:

```
useEffect(() => {
  let ignore = false;

  async function startFetching() {
    const json = await fetchTodos(userId);
    if (!ignore) {
      setTodos(json);
    }
  }

  startFetching();

  return () => {
    ignore = true;
  };
}, [userId]);
```

You can't "undo" a network request that already happened, but your cleanup function should ensure that the fetch that's *not relevant anymore* does not keep affecting your application. If the `userId` changes from `'Alice'` to `'Bob'`, cleanup ensures that the `'Alice'` response is ignored if even it arrives after `'Bob'`.

In development, you will see two fetches in the Network tab. There is nothing wrong with that. With the approach above, the first Effect will immediately get cleaned up so its copy of the `ignore` variable will be set to `true`. So even though there is an extra request, it won't affect the state thanks to the `if (!ignore)` check.

In production, there will only be one request. If the second request in development is bothering you, the best approach is to use a solution that deduplicates requests and caches their responses between components:

```
function TodoList() {
  const todos = useSomeDataLibrary(`/api/user/${userId}/todos`);
  // ...
```

This will not only improve the development experience, but also make your application feel faster. For example, the user pressing the Back button won't have to wait for some data to load again because it will be cached. You can either build such a cache yourself or use one of the many alternatives to manual fetching in Effects.

DEEP DIVE

What are good alternatives to data fetching in Effects?

Hide Details

Writing `fetch` calls inside Effects is a popular way to fetch data, especially in fully client-side apps. This is, however, a very manual approach and it has significant downsides:

- **Effects don't run on the server.** This means that the initial server-rendered HTML will only include a loading state with no data. The client computer will have to download all JavaScript and render your app only to discover that now it needs to load the data. This is not very efficient.
- **Fetching directly in Effects makes it easy to create "network**

waterfalls". You render the parent component, it fetches some data, renders the child components, and then they start fetching their data. If the network is not very fast, this is significantly slower than fetching all data in parallel.
- **Fetching directly in Effects usually means you don't preload or cache data.** For example, if the component unmounts and then mounts again, it would have to fetch the data again.
- **It's not very ergonomic.** There's quite a bit of boilerplate code involved when writing `fetch` calls in a way that doesn't suffer from bugs like race conditions.

This list of downsides is not specific to React. It applies to fetching data on mount with any library. Like with routing, data fetching is not trivial to do well, so we recommend the following approaches:

- **If you use a framework, use its built-in data fetching mechanism.** Modern React frameworks have integrated data fetching mechanisms that are efficient and don't suffer from the above pitfalls.
- **Otherwise, consider using or building a client-side cache.** Popular open source solutions include React Query, useSWR, and React Router 6.4+. You can build your own solution too, in which case you would use Effects under the hood, but add logic for deduplicating requests, caching responses, and avoiding network waterfalls (by preloading data or hoisting data requirements to routes).

You can continue fetching data directly in Effects if neither of these approaches suit you.

Sending analytics

Consider this code that sends an analytics event on the page visit:

```
useEffect(() => {
  logVisit(url); // Sends a POST request
}, [url]);
```

In development, `logVisit` will be called twice for every URL, so you might be tempted to try to fix that. **We recommend keeping this code as is.** Like with earlier examples, there is no *user-visible* behavior difference between running it once and running it twice. From a practical point of view, `logVisit` should not do anything in development because you don't want the logs from the development machines to skew the production metrics. Your component remounts every time you save its file, so it logs extra visits in development anyway.

In production, there will be no duplicate visit logs.

To debug the analytics events you're sending, you can deploy your app to a staging environment (which runs in production mode) or temporarily opt out of Strict Mode and its development-only remounting checks. You may also send analytics from the route change event handlers instead of Effects. For more precise analytics, intersection observers can help track which components are in the viewport and how long they remain visible.

Not an Effect: Initializing the application

Some logic should only run once when the application starts. You can put it outside your components:

```
if (typeof window !== 'undefined') { // Check if we're running in the brow
  checkAuthToken();
  loadDataFromLocalStorage();
}
```

```
function App() {
  // ...
}
```

This guarantees that such logic only runs once after the browser loads the page.

Not an Effect: Buying a product

Sometimes, even if you write a cleanup function, there's no way to prevent user-visible consequences of running the Effect twice. For example, maybe your Effect sends a POST request like buying a product:

```
useEffect(() => {
  // 🔴 Wrong: This Effect fires twice in development, exposing a problem
  fetch('/api/buy', { method: 'POST' });
}, []);
```

You wouldn't want to buy the product twice. However, this is also why you shouldn't put this logic in an Effect. What if the user goes to another page and then presses Back? Your Effect would run again. You don't want to buy the product when the user *visits* a page; you want to buy it when the user *clicks* the Buy button.

Buying is not caused by rendering; it's caused by a specific interaction. It should run only when the user presses the button. **Delete the Effect and move your `/api/buy` request into the Buy button event handler:**

```
  function handleClick() {
    // ✅ Buying is an event because it is caused by a particular interact
    fetch('/api/buy', { method: 'POST' });
```

```
}
```

This illustrates that if remounting breaks the logic of your application, this usually uncovers existing bugs. From the user's perspective, visiting a page shouldn't be different from visiting it, clicking a link, and pressing Back. React verifies that your components abide by this principle by remounting them once in development.

Putting it all together

This playground can help you "get a feel" for how Effects work in practice.

This example uses `setTimeout` to schedule a console log with the input text to appear three seconds after the Effect runs. The cleanup function cancels the pending timeout. Start by pressing "Mount the component":

App.js Download Reset

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

function Playground() {
  const [text, setText] = useState('a');

  useEffect(() => {
    function onTimeout() {
      console.log('⏰ ' + text);
    }

    console.log('🔵 Schedule "' + text + '" log');
    const timeoutId = setTimeout(onTimeout, 3000);

    return () => {
      console.log('🟡 Cancel "' + text + '" log');
      clearTimeout(timeoutId);
    };
  }, [text]);
```

```jsx

  return (
    <>
      <label>
        What to log:{' '}
        <input
          value={text}
          onChange={e => setText(e.target.value)}
        />
      </label>
      <h1>{text}</h1>
    </>
  );
}

export default function App() {
  const [show, setShow] = useState(false);
  return (
    <>
      <button onClick={() => setShow(!show)}>
        {show ? 'Unmount' : 'Mount'} the component
      </button>
      {show && <hr />}
      {show && <Playground />}
    </>
  );
}

```

Show less

You will see three logs at first: `Schedule "a" log`, `Cancel "a" log`, and `Schedule "a" log` again. Three second later there will also be a log saying `a`. As you learned earlier, the extra schedule/cancel pair is because React remounts the component once in development to verify that you've implemented cleanup well.

Now edit the input to say `abc`. If you do it fast enough, you'll see `Schedule "ab" log` immediately followed by `Cancel "ab" log` and `Schedule "abc" log`. **React always cleans up the previous render's Effect before the next render's Effect.** This is why even if you type into the input fast, there is at most one timeout scheduled at a time. Edit the input a few times and watch the console to get a feel for how Effects get cleaned up.

Type something into the input and then immediately press "Unmount the component". Notice how unmounting cleans up the last render's Effect. Here, it clears the last timeout before it has a chance to fire.

Finally, edit the component above and comment out the cleanup function so that the timeouts don't get cancelled. Try typing `abcde` fast. What do you expect to happen in three seconds? Will `console.log(text)` inside the timeout print the *latest* `text` and produce five `abcde` logs? Give it a try to check your intuition!

Three seconds later, you should see a sequence of logs (`a`, `ab`, `abc`, `abcd`, and `abcde`) rather than five `abcde` logs. **Each Effect "captures" the `text` value from its corresponding render.** It doesn't matter that the `text` state changed: an Effect from the render with `text = 'ab'` will always see `'ab'`.

In other words, Effects from each render are isolated from each other. If you're curious how this works, you can read about closures.

DEEP DIVE

Each render has its own Effects

Hide Details

You can think of `useEffect` as "attaching" a piece of behavior to the render output. Consider this Effect:

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

  return <h1>Welcome to {roomId}!</h1>;
}
```

Let's see what exactly happens as the user navigates around the app.

Initial render

The user visits `<ChatRoom roomId="general" />`. Let's mentally substitute `roomId` with `'general'`:

```
  // JSX for the first render (roomId = "general")
```

```
return <h1>Welcome to general!</h1>;
```

The Effect is *also* a part of the rendering output. The first render's Effect becomes:

```
// Effect for the first render (roomId = "general")
() => {
  const connection = createConnection('general');
  connection.connect();
  return () => connection.disconnect();
},
// Dependencies for the first render (roomId = "general")
['general']
```

React runs this Effect, which connects to the `'general'` chat room.

Re-render with same dependencies

Let's say `<ChatRoom roomId="general" />` re-renders. The JSX output is the same:

```
// JSX for the second render (roomId = "general")
return <h1>Welcome to general!</h1>;
```

React sees that the rendering output has not changed, so it doesn't update the DOM.

The Effect from the second render looks like this:

```
// Effect for the second render (roomId = "general")
() => {
  const connection = createConnection('general');
```

```
    connection.connect();
    return () => connection.disconnect();
  },
  // Dependencies for the second render (roomId = "general")
  ['general']
```

React compares `['general']` from the second render with `['general']` from the first render. **Because all dependencies are the same, React *ignores* the Effect from the second render.** It never gets called.

Re-render with different dependencies

Then, the user visits `<ChatRoom roomId="travel" />`. This time, the component returns different JSX:

```
  // JSX for the third render (roomId = "travel")
  return <h1>Welcome to travel!</h1>;
```

React updates the DOM to change `"Welcome to general"` into `"Welcome to travel"`.

The Effect from the third render looks like this:

```
  // Effect for the third render (roomId = "travel")
  () => {
    const connection = createConnection('travel');
    connection.connect();
    return () => connection.disconnect();
  },
  // Dependencies for the third render (roomId = "travel")
  ['travel']
```

React compares `['travel']` from the third render with `['general']` from the second render. One dependency is different: `Object.is('travel', 'general')` is `false`. The Effect can't be skipped.

Before React can apply the Effect from the third render, it needs to clean up the last Effect that *did* run. The second render's Effect was skipped, so React needs to clean up the first render's Effect. If you scroll up to the first render, you'll see that its cleanup calls `disconnect()` on the connection that was created with `createConnection('general')`. This disconnects the app from the `'general'` chat room.

After that, React runs the third render's Effect. It connects to the `'travel'` chat room.

Unmount

Finally, let's say the user navigates away, and the `ChatRoom` component unmounts. React runs the last Effect's cleanup function. The last Effect was from the third render. The third render's cleanup destroys the `createConnection('travel')` connection. So the app disconnects from the `'travel'` room.

Development-only behaviors

When Strict Mode is on, React remounts every component once after mount (state and DOM are preserved). This helps you find Effects that need cleanup and exposes bugs like race conditions early. Additionally, React will remount the Effects whenever you save a file in development. Both of these behaviors are development-only.

Recap

- Unlike events, Effects are caused by rendering itself rather than a particular interaction.
- Effects let you synchronize a component with some external system (third-party API, network, etc).
- By default, Effects run after every render (including the initial one).
- React will skip the Effect if all of its dependencies have the same values as during the last render.
- You can't "choose" your dependencies. They are determined by the code inside the Effect.
- Empty dependency array (`[]`) corresponds to the component "mounting", i.e. being added to the screen.
- In Strict Mode, React mounts components twice (in development only!) to stress-test your Effects.
- If your Effect breaks because of remounting, you need to implement a cleanup function.
- React will call your cleanup function before the Effect runs next time, and during the unmount.

Try out some challenges

1. Focus a field on mount 2. Focus a field conditionally 3. Fix an i

Challenge 1 of 4:

Focus a field on mount

In this example, the form renders a `<MyInput />` component.

Use the input's `focus()` method to make `MyInput` automatically focus when it appears on the screen. There is already a commented out implementation, but it doesn't quite work. Figure out why it doesn't work, and fix it. (If you're familiar with the `autoFocus` attribute, pretend that it does not exist: we are reimplementing the same functionality from scratch.)

MyInput.js Reset

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

export default function MyInput({ value, onChange }) {
  const ref = useRef(null);

  // TODO: This doesn't quite work. Fix it.
  // ref.current.focus()

  return (
    <input
      ref={ref}
      value={value}
      onChange={onChange}
    />
  );
}

```

Show less

To verify that your solution works, press “Show form” and verify that the input receives focus (becomes highlighted and the cursor is placed inside). Press “Hide form” and “Show form” again. Verify the input is highlighted again.

`MyInput` should only focus *on mount* rather than after every render. To verify that the behavior is right, press “Show form” and then repeatedly press the “Make it uppercase” checkbox. Clicking the checkbox should *not* focus the input above it.

Show solution Next Challenge

PREVIOUS

Manipulating the DOM with Refs

NEXT

You Might Not Need an Effect

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