Escape Hatches

ADVANCED

Some of your components may need to control and synchronize with systems outside of React. For example, you might need to focus an input using the browser API, play and pause a video player implemented without React, or connect and listen to messages from a remote server. In this chapter, you'll learn the escape hatches that let you "step outside" React and connect to external systems. Most of your application logic and data flow should not rely on these features.

In this chapter

Referencing values with refs

When you want a component to "remember" some information, but you don't want that information to trigger new renders, you can use a *ref*:

```
const ref = useRef(0);
```

Like state, refs are retained by React between re-renders. However, setting state re-renders a component. Changing a ref does not! You can access the current value of that ref through the `ref.current` property.

App.js   Download Reload Clear Fork

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

export default function Counter() {
  let ref = useRef(0);

  function handleClick() {
    ref.current = ref.current + 1;
    alert('You clicked ' + ref.current + ' times!');
  }

  return (
    <button onClick={handleClick}>
```

Show more

A ref is like a secret pocket of your component that React doesn't track. For example, you can use refs to store timeout IDs, DOM elements, and other objects that don't impact the component's rendering output.

Ready to learn this topic?

Read **Referencing Values with Refs** to learn how to use refs to remember information.

Read More >

Manipulating the DOM with refs

React automatically updates the DOM to match your render output, so your components won't often need to manipulate it. However, sometimes you might need access to the DOM elements managed by React—for example, to focus a node, scroll to it, or measure its size and position. There is no built-in way to do those things in React, so you will need a ref to the DOM node. For example, clicking the button will focus the input using a ref:

App.js   Download Reload Clear Fork

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

export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputRef} />
```

Show more

Ready to learn this topic?

Read **Manipulating the DOM with Refs** to learn how to access DOM elements managed by React.

Read More

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. Unlike event handlers, which let you handle particular events, *Effects* let you run some code after rendering. Use them to synchronize your component with a system outside of React.

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

App.js Download Reload Clear Fork

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

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

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

Show more

Many Effects also “clean up” after themselves. For example, an Effect that sets up a connection to a chat server should return a *cleanup function* that tells React how to disconnect your component from that server:

App.js Reload Clear Fork

```
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>;
}
```

In development, React will immediately run and clean up your Effect one extra time. This is why you see `"✅ Connecting..."` printed twice. This ensures that you don't forget to implement the cleanup function.

Ready to learn this topic?

Read **Synchronizing with Effects** to learn how to synchronize components with external systems.

Read More ›

You Might Not Need An Effect

Effects are an escape hatch from the React paradigm. They let you “step outside” of React and synchronize your components with some external system. If there is no external system involved (for example, if you want to update a component’s state when some props or state change), you shouldn’t need an Effect. Removing unnecessary Effects will make your code easier to follow, faster to run, and less error-prone.

There are two common cases in which you don’t need Effects:

- **You don’t need Effects to transform data for rendering.**
- **You don’t need Effects to handle user events.**

For example, you don’t need an Effect to adjust some state based on other state:

```
function Form() {
  const [firstName, setFirstName] = useState('Taylor');
  const [lastName, setLastName] = useState('Swift');

  // 🔴 Avoid: redundant state and unnecessary Effect
  const [fullName, setFullName] = useState('');
  useEffect(() => {
    setFullName(firstName + ' ' + lastName);
  }, [firstName, lastName]);
  // ...
}
```

Instead, calculate as much as you can while rendering:

```
function Form() {
  const [firstName, setFirstName] = useState('Taylor');
  const [lastName, setLastName] = useState('Swift');
  // ✅ Good: calculated during rendering
  const fullName = firstName + ' ' + lastName;
  // ...
}
```

However, you *do* need Effects to synchronize with external systems.

Ready to learn this topic?

Read **You Might Not Need an Effect** to learn how to remove unnecessary Effects.

Read More >

Lifecycle of reactive effects

Effects have a different lifecycle from components. Components may mount, update, or unmount. An Effect can only do two things: to start synchronizing something, and later to stop synchronizing it. This cycle can happen multiple times if your Effect depends on props and state that change over time.

This Effect depends on the value of the `roomId` prop. Props are *reactive values,* which means they can change on a re-render. Notice that the Effect *re-synchronizes* (and re-connects to the server) if `roomId` changes:

App.js chat.js Reload Clear Fork

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

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

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

Show more

React provides a linter rule to check that you've specified your Effect's dependencies correctly. If you forget to specify `roomId` in the list of dependencies in the above example, the linter will find that bug automatically.

Ready to learn this topic?

Read **Lifecycle of Reactive Events** to learn how an Effect's lifecycle is different from a component's.

Read More

Separating events from Effects

Event handlers only re-run when you perform the same interaction again. Unlike event handlers, Effects re-synchronize if any of the values they read, like props or state, are different than during last render. Sometimes, you want a mix of both behaviors: an Effect that re-runs in response to some values but not others.

All code inside Effects is *reactive.* It will run again if some reactive value it reads has changed due to a re-render. For example, this Effect will re-connect to the chat if either `roomId` or `theme` have changed:

App.js chat.js notifications.js Reload Clear Fork

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

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

function ChatRoom({ roomId, theme }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.on('connected', () => {
      showNotification('Connected!', theme);
    });
```

Show more

This is not ideal. You want to re-connect to the chat only if the `roomId` has changed. Switching the `theme` shouldn't re-connect to the chat! Move the code reading `theme` out of your Effect into an *Effect Event*:

App.js chat.js Reload Clear Fork

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

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

function ChatRoom({ roomId, theme }) {
  const onConnected = useEffectEvent(() => {
    showNotification('Connected!', theme);
  });
```

Show more

Code inside Effect Events isn't reactive, so changing the `theme` no longer makes your Effect re-connect.

Ready to learn this topic?

Read Separating Events from Effects to learn how to prevent some values from re-triggering Effects.

Read More

Removing Effect dependencies

When you write an Effect, the linter will verify that you've included every reactive value (like props and state) that the Effect reads in the list of your Effect's dependencies. This ensures that your Effect remains synchronized with the latest props and state of your component. Unnecessary dependencies may cause your Effect to run too often, or even create an infinite loop. The way you remove them depends on the case.

For example, this Effect depends on the `options` object which gets re-created every time you edit the input:

App.js chat.js Reload Clear Fork

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

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

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

  const options = {
    serverUrl: serverUrl,
    roomId: roomId
  };
```

Show more

You don't want the chat to re-connect every time you start typing a message in that chat. To fix this problem, move creation of the `options` object inside the Effect so that the Effect only depends on the `roomId` string:

App.js chat.js Reload Clear Fork

```
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,
      roomId: roomId
```

Show more

Notice that you didn't start by editing the dependency list to remove the `options` dependency. That would be wrong. Instead, you changed the surrounding code so that the dependency became *unnecessary.* Think of the dependency list as a list of all the reactive values used by your Effect's code. You don't intentionally choose what to put on that list. The list describes your code. To change the dependency list, change the code.

Ready to learn this topic?

Read **Removing Effect Dependencies** to learn how to make your Effect re-run less often.

Read More ›

Reusing logic with custom Hooks

React comes with built-in Hooks like `useState`, `useContext`, and `useEffect`. Sometimes, you'll wish that there was a Hook for some more specific purpose: for example, to fetch data, to keep track of whether the user is online, or to connect to a chat room. To do this, you can create your own Hooks for your application's needs.

In this example, the `usePointerPosition` custom Hook tracks the cursor position, while `useDelayedValue` custom Hook returns a value that's "lagging behind" the value you passed by a certain number of milliseconds. Move the cursor over the sandbox preview area to see a moving trail of dots following the cursor:

App.js usePointerPosition.js useDelayedValue.js Reload Clear Fork

```
import { usePointerPosition } from './usePointerPosition.js';
import { useDelayedValue } from './useDelayedValue.js';

export default function Canvas() {
  const pos1 = usePointerPosition();
  const pos2 = useDelayedValue(pos1, 100);
  const pos3 = useDelayedValue(pos2, 200);
  const pos4 = useDelayedValue(pos3, 100);
  const pos5 = useDelayedValue(pos4, 50);
  return (
    <>
      <Dot position={pos1} opacity={1} />
```

Show more

You can create custom Hooks, compose them together, pass data between them, and reuse them between components. As your app grows, you will write fewer Effects by hand because you'll be able to reuse custom Hooks you already wrote. There are also many excellent custom Hooks maintained by the React community.

Ready to learn this topic?

Read **Reusing Logic with Custom Hooks** to learn how to share logic between components.

Read More >

What's next?

Head over to Referencing Values with Refs to start reading this chapter page by page!