



[LEARN REACT](#) > [ESCAPE HATCHES](#) >

Reusing Logic with Custom Hooks

React comes with several 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. You might not find these Hooks in React, but you can create your own Hooks for your application's needs.

You will learn

- What custom Hooks are, and how to write your own
- How to reuse logic between components
- How to name and structure your custom Hooks
- When and why to extract custom Hooks

Custom Hooks: Sharing logic between components

Imagine you're developing an app that heavily relies on the network (as most apps do). You want to warn the user if their network connection has accidentally gone off while they were using your app. How would you go about it? It seems like you'll need two things in your component:

1. A piece of state that tracks whether the network is online.
2. An Effect that subscribes to the global `online` and `offline` events, and updates that state

updates that state.

This will keep your component **synchronized** with the network status. You might start with something like this:

App.js

[Download](#)[Reset](#)

```
1  import { useState, useEffect } from 'react';
2
3  export default function StatusBar() {
4    const [isOnline, setIsOnline] = useState(true);
5    useEffect(() => {
6      function handleOnline() {
7        setIsOnline(true);
8      }
9      function handleOffline() {
10       setIsOnline(false);
11     }
12     window.addEventListener('online', handleOnline);
13     window.addEventListener('offline', handleOffline);
14     return () => {
15       window.removeEventListener('online', handleOnline);
16       window.removeEventListener('offline', handleOffline);
17     };
18   }, []);
19
20   return <h1>{isOnline ? '✅ Online' : '❌ Disconnected'}</h1>;
21 }
22
```

[Show less](#)

Try turning your network on and off, and notice how this `StatusBar` updates in response to your actions.

Now imagine you *also* want to use the same logic in a different component. You want to implement a Save button that will become disabled and show “Reconnecting...” instead of “Save” while the network is off.

To start, you can copy and paste the `isOnline` state and the Effect into `SaveButton`:

App.js

Download

Reset

```
1  import { useState, useEffect } from 'react';
2
3  export default function SaveButton() {
4    const [isOnline, setIsOnline] = useState(true);
5    useEffect(() => {
6      function handleOnline() {
7        setIsOnline(true);
8      }
9      function handleOffline() {
10       setIsOnline(false);
11     }
12     window.addEventListener('online', handleOnline);
13     window.addEventListener('offline', handleOffline);
14     return () => {
15       window.removeEventListener('online', handleOnline);
16       window.removeEventListener('offline', handleOffline);
17     };
18   }
19 }
```

```
18   }, []));
19
20   function handleSaveClick() {
21     console.log('✅ Progress saved');
22   }
23
24   return (
25     <button disabled={!isOnline} onClick={handleSaveClick}>
26       {isOnline ? 'Save progress' : 'Reconnecting...'}
27     </button>
28   );
29 }
30
```

Show less

Verify that, if you turn off the network, the button will change its appearance.

These two components work fine, but the duplication in logic between them is unfortunate. It seems like even though they have different *visual appearance*, you want to reuse the logic between them.

Extracting your own custom Hook from a component

Imagine for a moment that, similar to `useState` and `useEffect`, there was a built-in `useOnlineStatus` Hook. Then both of these components could be simplified and you could remove the duplication between them:

```
function StatusBar() {  
  const isOnline = useOnlineStatus();  
  return <h1>{isOnline ? '✅ Online' : '❌ Disconnected'}</h1>;  
}  
  
function SaveButton() {  
  const isOnline = useOnlineStatus();  
  
  function handleSaveClick() {  
    console.log('✅ Progress saved');  
  }  
  
  return (  
    <button disabled={!isOnline} onClick={handleSaveClick}>  
      {isOnline ? 'Save progress' : 'Reconnecting...'}  
    </button>  
  );  
}
```

Although there is no such built-in Hook, you can write it yourself. Declare a function called `useOnlineStatus` and move all the duplicated code into it from the components you wrote earlier:

```
function useOnlineStatus() {  
  const [isOnline, setIsOnline] = useState(true);  
  useEffect(() => {  
    function handleOnline() {
```

```
    setIsOnline(true);
  }
  function handleOffline() {
    setIsOnline(false);
  }
  window.addEventListener('online', handleOnline);
  window.addEventListener('offline', handleOffline);
  return () => {
    window.removeEventListener('online', handleOnline);
    window.removeEventListener('offline', handleOffline);
  };
}, []);
return isOnline;
}
```

At the end of the function, return `isOnline`. This lets your components read that value:

App.js useOnlineStatus.js

Reset

```
1  import { useOnlineStatus } from './useOnlineStatus.js';
2
3  function StatusBar() {
4    const isOnline = useOnlineStatus();
5    return <h1>{isOnline ? '✅ Online' : '❌ Disconnected'}</h1>;
6  }
7
8  function SaveButton() {
9    const isOnline = useOnlineStatus();
10
11    function handleSaveClick() {
12      console.log('✅ Progress saved');
13    }
14
15    return (
16      <button disabled={!isOnline} onClick={handleSaveClick}>
17        {isOnline ? 'Save progress' : 'Reconnecting...'}
18      </button>
```

```
19   );  
20 }  
21  
22 export default function App() {  
23   return (  
24     <>  
25       <SaveButton />  
26       <StatusBar />  
27     </>  
28   );  
29 }  
30
```

Show less

Verify that switching the network on and off updates both components.

Now your components don't have as much repetitive logic. **More importantly, the code inside them describes *what they want to do* (use the online status!) rather than *how to do it* (by subscribing to the browser events).**

When you extract logic into custom Hooks, you can hide the gnarly details of how you deal with some external system or a browser API. The code of your components expresses your intent, not the implementation.

Hook names always start with `use`

React applications are built from components. Components are built from Hooks, whether built-in or custom. You'll likely often use custom Hooks created by others, but occasionally you might write one yourself!

You must follow these naming conventions:

1. **React component names must start with a capital letter**, like `StatusBar` and `SaveButton`. React components also need to return something that React knows how to display, like a piece of JSX.
2. **Hook names must start with `use` followed by a capital letter**, like `useState` (built-in) or `useOnlineStatus` (custom, like earlier on the page). Hooks may return arbitrary values.

This convention guarantees that you can always look at a component and know where its state, Effects, and other React features might “hide”. For example, if you see a `getColor()` function call inside your component, you can be sure that it can't possibly contain React state inside because its name doesn't start with `use`. However, a function call like `useOnlineStatus()` will most likely contain calls to other Hooks inside!

Note

If your linter is [configured for React](#), it will enforce this naming convention. Scroll up to the sandbox above and rename `useOnlineStatus` to `getOnlineStatus`. Notice that the linter won't allow you to call `useState` or `useEffect` inside of it anymore. Only Hooks and components can call other Hooks!

DEEP DIVE

Should all functions called during rendering start with the use prefix?

[Hide Details](#)

No. Functions that don't *call* Hooks don't need to *be* Hooks.

If your function doesn't call any Hooks, avoid the `use` prefix. Instead, write it as a regular function *without* the `use` prefix. For example, `useSorted` below doesn't call Hooks, so call it `getSorted` instead:

```
// 🚫 Avoid: A Hook that doesn't use Hooks
function useSorted(items) {
  return items.slice().sort();
}

// ✅ Good: A regular function that doesn't use Hooks
function getSorted(items) {
  return items.slice().sort();
}
```

This ensures that your code can call this regular function anywhere, including conditions:

```
function list({ items, shouldSort }) {
```

```
function useItems(items, shouldSort) {  
  let displayedItems = items;  
  if (shouldSort) {  
    // ✅ It's ok to call getSorted() conditionally because it's not  
    displayedItems = getSorted(items);  
  }  
  // ...  
}
```

You should give `use` prefix to a function (and thus make it a Hook) if it uses at least one Hook inside of it:

```
// ✅ Good: A Hook that uses other Hooks  
function useAuth() {  
  return useContext(Auth);  
}
```

Technically, this isn't enforced by React. In principle, you could make a Hook that doesn't call other Hooks. This is often confusing and limiting so it's best to avoid that pattern. However, there may be rare cases where it is helpful. For example, maybe your function doesn't use any Hooks right now, but you plan to add some Hook calls to it in the future. Then it makes sense to name it with the `use` prefix:

```
// ✅ Good: A Hook that will likely use some other Hooks later  
function useAuth() {  
  // TODO: Replace with this line when authentication is implemented  
  // return useContext(Auth);  
  return TEST_USER;  
}
```

Then components won't be able to call it conditionally. This will become important when you actually add Hook calls inside. If you

don't plan to use Hooks inside it (now or later), don't make it a Hook.

Custom Hooks let you share stateful logic, not state itself

In the earlier example, when you turned the network on and off, both components updated together. However, it's wrong to think that a single `isOnline` state variable is shared between them. Look at this code:

```
function StatusBar() {  
  const isOnline = useOnlineStatus();  
  // ...  
}  
  
function SaveButton() {  
  const isOnline = useOnlineStatus();  
  // ...  
}
```

It works the same way as before you extracted the duplication:

```
function StatusBar() {  
  const [isOnline, setIsOnline] = useState(true);  
  useEffect(() => {  
    // ...  
  }, []);  
  // ...  
}  
  
function SaveButton() {  
  const [isOnline, setIsOnline] = useState(true);
```

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

These are two completely independent state variables and Effects! They happened to have the same value at the same time because you synchronized them with the same external value (whether the network is on).

To better illustrate this, we'll need a different example. Consider this `Form` component:

App.js

[Download](#)[Reset](#)

```
1  import { useState } from 'react';  
2  
3  export default function Form() {  
4    const [firstName, setFirstName] = useState('Mary');  
5    const [lastName, setLastName] = useState('Poppins');  
6  
7    function handleFirstNameChange(e) {  
8      setFirstName(e.target.value);  
9    }  
10  
11    function handleLastNameChange(e) {  
12      setLastName(e.target.value);  
13    }  
14  
15    return (  
16      <>  
17        <label>  
18          First name:  
19          <input value={firstName} onChange={handleFirstNameChange} />  
20        </label>  
21        <label>  
22          Last name:  
23          <input value={lastName} onChange={handleLastNameChange} />  
24        </label>  
25      </>  
26    );  
27  }  
28}
```

```
23       <input value={lastName} onChange={handleLastNameChange} />
24     </label>
25     <p><b>Good morning, {firstName} {lastName}.</b></p>
26   </>
27 );
28 }
29
```

Show less

There's some repetitive logic for each form field:

1. There's a piece of state (`firstName` and `lastName`).
2. There's a change handler (`handleFirstNameChange` and `handleLastNameChange`).
3. There's a piece of JSX that specifies the `value` and `onChange` attributes for that input.

You can extract the repetitive logic into this `useFormInput` custom Hook:

[App.js](#) [useFormInput.js](#)

Reset

```
1  import { useState } from 'react';
2
3  export function useFormInput(initialValue) {
4    const [value, setValue] = useState(initialValue);
5
6    function handleChange(e) {
7      setValue(e.target.value);
8    }
9
10   const inputProps = {
11     value: value,
12     onChange: handleChange
13   };
14
15   return inputProps;
16 }
17
```

Show less

Notice that it only declares *one* state variable called `value`.

However the `Form` component calls `useFormInput` *two times*.

However, the `Form` component can `useFormInput` two times.

```
function Form() {  
  const firstNameProps = useFormInput('Mary');  
  const lastNameProps = useFormInput('Poppins');  
  // ...  
}
```

This is why it works like declaring two separate state variables!

Custom Hooks let you share *stateful logic* but not *state itself*. Each call to a Hook is completely independent from every other call to the same Hook.

This is why the two sandboxes above are completely equivalent. If you'd like, scroll back up and compare them. The behavior before and after extracting a custom Hook is identical.

When you need to share the state itself between multiple components, [lift it up and pass it down](#) instead.

Passing reactive values between Hooks

The code inside your custom Hooks will re-run during every re-render of your component. This is why, like components, custom Hooks [need to be pure](#). Think of custom Hooks' code as part of your component's body!

Because custom Hooks re-render together with your component, they always receive the latest props and state. To see what this means, consider this chat room example. Change the server URL or the chat room:

[App.js](#) [ChatRoom.js](#) [chat.js](#) [notifications.js](#)

Reset

```
1 import { useState, useEffect } from 'react';  
2 import { createConnection } from './chat.js';  
3 import { showNotification } from './notifications.js';  
4  
5 // ...
```



```
5 export default function ChatRoom({ roomId }) {
6   const [serverUrl, setServerUrl] = useState('https://localhost:1234');
7
8   useEffect(() => {
9     const options = {
10       serverUrl: serverUrl,
11       roomId: roomId
12     };
13     const connection = createConnection(options);
14     connection.on('message', (msg) => {
15       showNotification('New message: ' + msg);
16     });
17     connection.connect();
18     return () => connection.disconnect();
19   }, [roomId, serverUrl]);
20
21   return (
22     <>
23       <label>
24         Server URL:
25         <input value={serverUrl} onChange={e => setServerUrl(e.target.value)} />
26       </label>
27       <h1>Welcome to the {roomId} room!</h1>
28     </>
29   );
30 }
31
```

[Show less](#)

▼ Console (3)

- ✓ Connecting to "general" room at https://localhost:1234...
- ✗ Disconnected from "general" room at https://localhost:1234
- ✓ Connecting to "general" room at https://localhost:1234...

When you change `serverUrl` or `roomId`, the Effect “reacts” to your changes and re-synchronizes. You can tell by the console messages that the chat re-connects every time that you change your Effect’s dependencies.

Now move the Effect’s code into a custom Hook:

```
export function useChatRoom({ serverUrl, roomId }) {  
  useEffect(() => {  
    const options = {  
      serverUrl: serverUrl,  
      roomId: roomId  
    };  
    const connection = createConnection(options);  
    connection.connect();  
    connection.on('message', (msg) => {  
      showNotification('New message: ' + msg);  
    });  
    return () => connection.disconnect();  
  }, [roomId, serverUrl]);  
}
```

This lets your `ChatRoom` component call your custom Hook without worrying about how it works inside:

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

```
    serverUrl: serverUrl
  });

  return (
    <>
      <label>
        Server URL:
        <input value={serverUrl} onChange={e => setServerUrl(e.target.valu
      </label>
      <h1>Welcome to the {roomId} room!</h1>
    </>
  );
}
```

This looks much simpler! (But it does the same thing.)

Notice that the logic *still responds* to prop and state changes. Try editing the server URL or the selected room:

App.js ChatRoom.js useChatRoom.js chat.js notificatic

Reset

```
1  import { useState } from 'react';
2  import { useChatRoom } from './useChatRoom.js';
3
4  export default function ChatRoom({ roomId }) {
5    const [serverUrl, setServerUrl] = useState('https://localhost:1234');
6
7    useChatRoom({
8      roomId: roomId,
9      serverUrl: serverUrl
10   });
11
12   return (
13     <>
14       <label>
15         Server URL:
16         <input value={serverUrl} onChange={e => setServerUrl(e.target.v
17       </label>
```

```
18     <h1>Welcome to the {roomId} room!</h1>
19   </>
20   );
21 }
22
```

Show less

▼ Console (3)

- ✓ Connecting to "general" room at https://localhost:1234...
- ✗ Disconnected from "general" room at https://localhost:1234
- ✓ Connecting to "general" room at https://localhost:1234...

Notice how you're taking the return value of one Hook:

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

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

and pass it as an input to another Hook:

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

Every time your `ChatRoom` component re-renders, it passes the latest `roomId` and `serverUrl` to your Hook. This is why your Effect re-connects to the chat whenever their values are different after a re-render. (If you ever worked with audio or video processing software, chaining Hooks like this might remind you of chaining visual or audio effects. It's as if the output of `useState` “feeds into” the input of the `useChatRoom`.)

Passing event handlers to custom Hooks

Under Construction

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

As you start using `useChatRoom` in more components, you might want to let components customize its behavior. For example, currently, the logic for what to do when a message arrives is hardcoded inside the Hook:

```
export function useChatRoom({ serverUrl, roomId }) {
```

```
useEffect(() => {
  const options = {
    serverUrl: serverUrl,
    roomId: roomId
  };
  const connection = createConnection(options);
  connection.connect();
  connection.on('message', (msg) => {
    showNotification('New message: ' + msg);
  });
  return () => connection.disconnect();
}, [roomId, serverUrl]);
}
```

Let's say you want to move this logic back to your component:

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

  useChatRoom({
    roomId: roomId,
    serverUrl: serverUrl,
    onReceiveMessage(msg) {
      showNotification('New message: ' + msg);
    }
  });
  // ...
}
```

To make this work, change your custom Hook to take `onReceiveMessage` as one of its named options:

```
export function useChatRoom({ serverUrl, roomId, onReceiveMessage }) {
  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
```

```
    roomId: roomId
  };
  const connection = createConnection(options);
  connection.connect();
  connection.on('message', (msg) => {
    onReceiveMessage(msg);
  });
  return () => connection.disconnect();
}, [roomId, serverUrl, onReceiveMessage]); // ✅ All dependencies declared
}
```

This will work, but there's one more improvement you can do when your custom Hook accepts event handlers.

Adding a dependency on `onReceiveMessage` is not ideal because it will cause the chat to re-connect every time the component re-renders. [Wrap this event handler into an Effect Event to remove it from the dependencies:](#)

```
import { useEffect, useEffectEvent } from 'react';
// ...

export function useChatRoom({ serverUrl, roomId, onReceiveMessage }) {
  const onMessage = useEffectEvent(onReceiveMessage);

  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
      roomId: roomId
    };
    const connection = createConnection(options);
    connection.connect();
    connection.on('message', (msg) => {
      onMessage(msg);
    });
    return () => connection.disconnect();
  }, [roomId, serverUrl]); // ✅ All dependencies declared
}
```

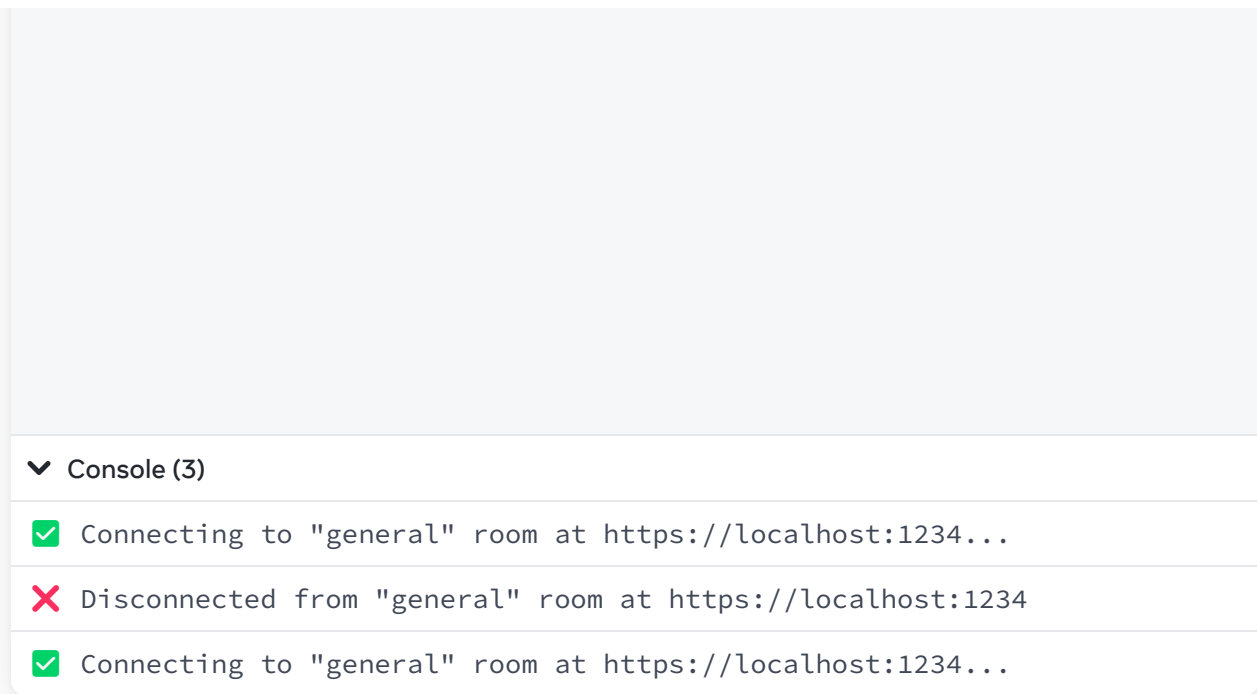
Now the chat won't re-connect every time that the `ChatRoom` component re-renders. Here is a fully working demo of passing an event handler to a custom Hook that you can play with:

[App.js](#) [ChatRoom.js](#) [useChatRoom.js](#) [chat.js](#) [notificatic](#)

Reset

```
1  import { useState } from 'react';
2  import { useChatRoom } from './useChatRoom.js';
3  import { showNotification } from './notifications.js';
4
5  export default function ChatRoom({ roomId }) {
6    const [serverUrl, setServerUrl] = useState('https://localhost:1234');
7
8    useChatRoom({
9      roomId: roomId,
10     serverUrl: serverUrl,
11     onReceiveMessage(msg) {
12       showNotification('New message: ' + msg);
13     }
14   });
15
16   return (
17     <>
18       <label>
19         Server URL:
20         <input value={serverUrl} onChange={e => setServerUrl(e.target.v
21       </label>
22       <h1>Welcome to the {roomId} room!</h1>
23     </>
24   );
25 }
26
```

Show less



Notice how you no longer need to know *how* `useChatRoom` works in order to use it. You could add it to any other component, pass any other options, and it would work the same way. That’s the power of custom Hooks.

When to use custom Hooks

You don’t need to extract a custom Hook for every little duplicated bit of code. Some duplication is fine. For example, extracting a `useFormInput` Hook to wrap a single `useState` call like earlier is probably unnecessary.

However, whenever you write an Effect, consider whether it would be clearer to also wrap it in a custom Hook. [You shouldn’t need Effects very often](#), so if you’re writing one, it means that you need to “step outside React” to synchronize with some external system or to do something that React doesn’t have a built-in API for. Wrapping it into a custom Hook lets you precisely communicate your intent and how the data flows through it.

For example, consider a `ShippingForm` component that displays two dropdowns: one shows the list of cities, and another shows the list of areas in the selected city. You might start with some code that looks like this:

```
function ShippingForm({ country }) {  
  const [cities, setCities] = useState(null);  
  // This Effect fetches cities for a country  
  useEffect(() => {  
    let ignore = false;  
    fetch(`/api/cities?country=${country}`)  
      .then(response => response.json())  
      .then(json => {  
        if (!ignore) {  
          setCities(json);  
        }  
      });  
    return () => {  
      ignore = true;  
    };  
  }, [country]);
```

```
  const [city, setCity] = useState(null);  
  const [areas, setAreas] = useState(null);  
  // This Effect fetches areas for the selected city  
  useEffect(() => {  
    if (city) {  
      let ignore = false;  
      fetch(`/api/areas?city=${city}`)  
        .then(response => response.json())  
        .then(json => {  
          if (!ignore) {  
            setAreas(json);  
          }  
        });  
      return () => {  
        ignore = true;  
      };  
    }  
  }, [city]);
```

```
  // ...
```

Although this code is quite repetitive, [it's correct to keep these Effects separate from each other](#). They synchronize two different things, so you shouldn't merge them into one Effect. Instead, you can simplify the `ShippingForm` component above by extracting the common logic between them into your own `useData` Hook:

```
function useData(url) {  
  const [data, setData] = useState(null);  
  useEffect(() => {  
    if (url) {  
      let ignore = false;  
      fetch(url)  
        .then(response => response.json())  
        .then(json => {  
          if (!ignore) {  
            setData(json);  
          }  
        });  
      return () => {  
        ignore = true;  
      };  
    }  
  }, [url]);  
  return data;  
}
```

Now you can replace both Effects in the `ShippingForm` components with calls to `useData`:

```
function ShippingForm({ country }) {  
  const cities = useData(`/api/cities?country=${country}`);  
  const [city, setCity] = useState(null);  
  const areas = useData(city ? `/api/areas?city=${city}` : null);  
  // ...  
}
```

Extracting a custom Hook makes the data flow explicit. You feed the `url` in and you get the `data` out. By “hiding” your Effect inside `useData`, you also prevent someone working on the `ShippingForm` component from adding **unnecessary dependencies** to it. With time, most of your app’s Effects will be in custom Hooks.

DEEP DIVE

Keep your custom Hooks focused on concrete high-level use cases

Hide Details




Start by choosing your custom Hook’s name. If you struggle to pick a clear name, it might mean that your Effect is too coupled to the rest of your component’s logic, and is not yet ready to be extracted.

Ideally, your custom Hook’s name should be clear enough that even a person who doesn’t write code often could have a good guess about what your custom Hook does, what it takes, and what it returns:




- ✓ `useData(url)`
- ✓ `useImpressionLog(eventName, extraData)`
- ✓ `useChatRoom(options)`

When you synchronize with an external system, your custom Hook name may be more technical and use jargon specific to that system. It’s good as long as it would be clear to a person familiar with that system:


system.

-  `useMediaQuery(query)`
-  `useSocket(url)`
-  `useIntersectionObserver(ref, options)`

Keep custom Hooks focused on concrete high-level use cases. Avoid creating and using custom “lifecycle” Hooks that act as alternatives and convenience wrappers for the `useEffect` API itself:

-  `useMount(fn)`
-  `useEffectOnce(fn)`
-  `useUpdateEffect(fn)`

For example, this `useMount` Hook tries to ensure some code only runs “on mount”:

```
function ChatRoom({ roomId }) {  
  const [serverUrl, setServerUrl] = useState('https://localhost:12  
  
  //  Avoid: using custom "lifecycle" Hooks  
  useMount(() => {  
    const connection = createConnection({ roomId, serverUrl });  
    connection.connect();  
  
    post('/analytics/event', { eventName: 'visit_chat' });  
  });  
  // ...  
}  
  
//  Avoid: creating custom "lifecycle" Hooks  
function useMount(fn) {  
  useEffect(() => {  
    fn();  
  }, []); //  React Hook useEffect has a missing dependency: 'fn'  
}
```

Custom “lifecycle” Hooks like `useMount` don’t fit well into the React paradigm. For example, this code example has a mistake (it doesn’t “react” to `roomId` or `serverUrl` changes), but the linter won’t warn you about it because the linter only checks direct `useEffect` calls. It won’t know about your Hook.

If you’re writing an Effect, start by using the React API directly:

```
function ChatRoom({ roomId }) {  
  const [serverUrl, setServerUrl] = useState('https://localhost:12  
  
  // ✅ Good: two raw Effects separated by purpose  
  
  useEffect(() => {  
    const connection = createConnection({ serverUrl, roomId });  
    connection.connect();  
    return () => connection.disconnect();  
  }, [serverUrl, roomId]);  
  
  useEffect(() => {  
    post('/analytics/event', { eventName: 'visit_chat', roomId });  
  }, [roomId]);  
  
  // ...  
}
```

Then, you can (but don’t have to) extract custom Hooks for different high-level use cases:

```
function ChatRoom({ roomId }) {  
  const [serverUrl, setServerUrl] = useState('https://localhost:12  
  
  // ✅ Great: custom Hooks named after their purpose  
  useChatRoom({ serverUrl, roomId });  
  useImpressionLog('visit chat', { roomId });
```

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

A good custom Hook makes the calling code more declarative by constraining what it does. For example, `useChatRoom(options)` can only connect to the chat room, while `useImpressionLog(eventName, extraData)` can only send an impression log to the analytics. If your custom Hook API doesn't constrain the use cases and is very abstract, in the long run it's likely to introduce more problems than it solves.

Custom Hooks help you migrate to better patterns

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. With time, the React team's goal is to reduce the number of the Effects in your app to the minimum by providing more specific solutions to more specific problems. Wrapping your Effects in custom Hooks makes it easier to upgrade your code when these solutions become available.

Let's return to this example:

App.js useOnlineStatus.js

Reset

```
1 import { useState, useEffect } from 'react';  
2  
3 export function useOnlineStatus() {  
4   const [isOnline, setIsOnline] = useState(true);  
5   useEffect(() => {  
6     function handleOnline() {  
7       setIsOnline(true);
```

```
8      }
9      function handleOffline() {
10         setIsOnline(false);
11     }
12     window.addEventListener('online', handleOnline);
13     window.addEventListener('offline', handleOffline);
14     return () => {
15         window.removeEventListener('online', handleOnline);
16         window.removeEventListener('offline', handleOffline);
17     };
18 }, []);
19 return isOnline;
20 }
21
```

Show less

In the above example, `useOnlineStatus` is implemented with a pair of `useState` and `useEffect`. However, this isn't the best possible solution. There is a number of edge cases it doesn't consider. For example, it assumes that when the component mounts, `isOnline` is already `true`, but this may be wrong if the network already went offline. You can use the browser

`navigator.onLine` API to check for that, but using it directly would not work on the server for generating the initial HTML. In short, this code could be improved.

Luckily, React 18 includes a dedicated API called `useSyncExternalStore` which takes care of all of these problems for you. Here is how your `useOnlineStatus` Hook, rewritten to take advantage of this new API:

App.js useOnlineStatus.js

Reset

```
1  import { useSyncExternalStore } from 'react';
2
3  function subscribe(callback) {
4    window.addEventListener('online', callback);
5    window.addEventListener('offline', callback);
6    return () => {
7      window.removeEventListener('online', callback);
8      window.removeEventListener('offline', callback);
9    };
10 }
11
12 export function useOnlineStatus() {
13   return useSyncExternalStore(
14     subscribe,
15     () => navigator.onLine, // How to get the value on the client
16     () => true // How to get the value on the server
17   );
18 }
19
20
```

Show less

Notice how **you didn't need to change any of the components** to make this migration:

```
function StatusBar() {  
  const isOnline = useOnlineStatus();  
  // ...  
}  
  
function SaveButton() {  
  const isOnline = useOnlineStatus();  
  // ...  
}
```

This is another reason for why wrapping Effects in custom Hooks is often beneficial:

1. You make the data flow to and from your Effects very explicit.
2. You let your components focus on the intent rather than on the exact implementation of your Effects.
3. When React adds new features, you can remove those Effects without changing any of your components.

Similar to a [design system](#), you might find it helpful to start extracting common idioms from your app's components into custom Hooks. This will

keep your components' code focused on the intent, and let you avoid writing raw Effects very often. Many excellent custom Hooks are maintained by the React community.

DEEP DIVE

Will React provide any built-in solution for data fetching?

Hide Details

We're still working out the details, but we expect that in the future, you'll write data fetching like this:

```
import { use } from 'react'; // Not available yet!

function ShippingForm({ country }) {
  const cities = use(fetch(`/api/cities?country=${country}`));
  const [city, setCity] = useState(null);
  const areas = city ? use(fetch(`/api/areas?city=${city}`)) : null
  // ...
}
```

If you use custom Hooks like `useData` above in your app, it will require fewer changes to migrate to the eventually recommended approach than if you write raw Effects in every component manually. However, the old approach will still work fine, so if you feel happy writing raw Effects, you can continue to do that.

There is more than one way to do it

Let's say you want to implement a fade-in animation *from scratch* using the browser `requestAnimationFrame` API. You might start with an Effect that sets up an animation loop. During each frame of the animation, you could change the opacity of the DOM node you [hold in a ref](#) until it reaches `1`. Your code might start like this:

App.js

Download

Reset

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

8   const node = ref.current;
9
10  let startTime = performance.now();
11  let frameId = null;
12
13  function onFrame(now) {
14    const timePassed = now - startTime;
15    const progress = Math.min(timePassed / duration, 1);
16    onProgress(progress);
17    if (progress < 1) {
18      // We still have more frames to paint
19      frameId = requestAnimationFrame(onFrame);
20    }
21  }
22
23  function onProgress(progress) {
24    node.style.opacity = progress;
25  }
26
27  function start() {
```

```
28     onProgress(0);
29     startTime = performance.now();
30     frameId = requestAnimationFrame(onFrame);
31   }
32
33   function stop() {
34     cancelAnimationFrame(frameId);
35     startTime = null;
36     frameId = null;
37   }
38
39   start();
40   return () => stop();
41 }, []);
42
43 return (
44   <h1 className="welcome" ref={ref}>
45     Welcome
46   </h1>
47 );
48 }
49
50 export default function App() {
51   const [show, setShow] = useState(false);
52   return (
53     <>
54       <button onClick={() => setShow(!show)}>
55         {show ? 'Remove' : 'Show'}
56       </button>
57       <hr />
58       {show && <Welcome />}
59     </>
60   );
61 }
62
```

[Show less](#)

To make the component more readable, you might extract the logic into a `useFadeIn` custom Hook:

`App.js` `useFadeIn.js`

Reset

```
1  import { useState, useEffect, useRef } from 'react';
2  import { useFadeIn } from './useFadeIn.js';
3
4  function Welcome() {
5    const ref = useRef(null);
6
7    useFadeIn(ref, 1000);
8
9    return (
10     <h1 className="welcome" ref={ref}>
11       Welcome
12     </h1>
13   );
14 }
15
16 export default function App() {
17   const [show, setShow] = useState(false);
18   return (
19     <>
```

```
--  
20     <button onClick={() => setShow(!show)}>  
21       {show ? 'Remove' : 'Show'}  
22     </button>  
23     <hr />  
24     {show && <Welcome />}  
25   </>  
26 );  
27 }  
28
```

Show less

You could keep the `useFadeIn` code as is, but you could also refactor it more. For example, you could extract the logic for setting up the animation loop out of `useFadeIn` into a custom `useAnimationLoop` Hook:

App.js useFadeIn.js

Reset

```
1  import { useState, useEffect } from 'react';  
2  import { experimental_useEffectEvent as useEffectEvent } from 'react';  
3
```

```
4 export function useFadeIn(ref, duration) {  
5   const [isRunning, setIsRunning] = useState(true);  
6  
7   useAnimationLoop(isRunning, (timePassed) => {  
8     const progress = Math.min(timePassed / duration, 1);  
9     ref.current.style.opacity = progress;  
10    if (progress === 1) {  
11      setIsRunning(false);  
12    }  
  })  
}
```

▼ Show more

However, you didn't *have to* do that. As with regular functions, ultimately you decide where to draw the boundaries between different parts of your code. You could also take a very different approach. Instead of keeping the logic in the Effect, you could move most of the imperative logic inside a JavaScript [class](#):

App.js useFadeIn.js animation.js

Reset

```
1 import { useState, useEffect } from 'react';  
2 import { FadeInAnimation } from './animation.js';
```



```
3
4 export function useFadeIn(ref, duration) {
5   useEffect(() => {
6     const animation = new FadeInAnimation(ref.current);
7     animation.start(duration);
8     return () => {
9       animation.stop();
10    };
11  }, [ref, duration]);
12 }
```

Effects let you connect React to external systems. The more coordination between Effects is needed (for example, to chain multiple animations), the more it makes sense to extract that logic out of Effects and Hooks *completely* like in the sandbox above. Then, the code you extracted *becomes* the “external system”. This lets your Effects stay simple because they only need to send messages to the system you’ve moved outside React.

The examples above assume that the fade-in logic needs to be written in JavaScript. However, this particular fade-in animation is both simpler and much more efficient to implement with a plain [CSS Animation](#):

App.js welcome.css

Reset

```
1  .welcome {
2    color: white;
3    padding: 50px;
4    text-align: center;
5    font-size: 50px;
6    background-image: radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(
7
8    animation: fadeIn 1000ms;
9  }
10
11 @keyframes fadeIn {
12   0% { opacity: 0; }
```

Sometimes, you don't even need a Hook!

Recap

- Custom Hooks let you share logic between components.
- Custom Hooks must be named starting with `use` followed by a capital letter.

- Custom Hooks only share stateful logic, not state itself.
- You can pass reactive values from one Hook to another, and they stay up-to-date.
- All Hooks re-run every time your component re-renders.
- The code of your custom Hooks should be pure, like your component's code.
- Wrap event handlers received by custom Hooks into Effect Events.
- Don't create custom Hooks like `useMount`. Keep their purpose specific.
- It's up to you how and where to choose the boundaries of your code.

Try out some challenges

1. Extract a `useCounter` Hook 2. Make the counter delay configurable

Challenge 1 of 5:

Extract a `useCounter` Hook

This component uses a state variable and an Effect to display a number that increments every second. Extract this logic into a custom Hook called `useCounter`. Your goal is to make the `Counter` component implementation look exactly like this:

```
export default function Counter() {  
  const count = useCounter();  
  return <h1>Seconds passed: {count}</h1>;  
}
```

You'll need to write your custom Hook in `useCounter.js` and import it into the `Counter.js` file.

App.js useCounter.js

Reset

```
1 import { useState, useEffect } from 'react';
2
3 export default function Counter() {
4   const [count, setCount] = useState(0);
5   useEffect(() => {
6     const id = setInterval(() => {
7       setCount(c => c + 1);
8     }, 1000);
9     return () => clearInterval(id);
10  }, []);
11  return <h1>Seconds passed: {count}</h1>;
12 }
```

Show solution

Next Challenge

PREVIOUS

[Removing Effect Dependencies](#)

How do you like these docs?

Take our survey!

 Meta Open Source

©2023

Learn React

Quick Start

Installation

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

Community

Code of Conduct

Meet the Team

Docs Contributors

Acknowledgements

API Reference

React APIs

React DOM APIs

More

Blog

React Native

Privacy

Terms