

# Custom Hooks



[\(https://github.com/learn-co-curriculum/react-hooks-custom-hooks\)](https://github.com/learn-co-curriculum/react-hooks-custom-hooks)



[\(https://github.com/learn-co-curriculum/react-hooks-custom-hooks/issues/new\)](https://github.com/learn-co-curriculum/react-hooks-custom-hooks/issues/new)

## Learning Goals

- Refactor existing React code into custom hooks

## Introduction

One of the most powerful features of React hooks is that they give us the ability to share logic and state between multiple components by writing our own **custom hooks**. You've already encountered some custom hooks: the `useParams` and `useHistory` hooks from React Router are hooks that let us access the `params` and `history` objects from React Router in any component we want. In this lesson, you'll learn how to create your own custom hooks by extracting hooks-related logic out of components and into a reusable hook function.

## Setup

This lesson has some starter code for a blog site using React Router. The data for the blog is saved in a `db.json` file, which we'll serve up using `json-server`. To get started, run `npm install`. Then, run `npm run server` to run our `json-server` backend in one terminal tab. Open another terminal tab and run `npm start` to run our React frontend.

We'll be focusing on two components: the `HomePage` and `ArticlePage` components. Make sure to run the app and familiarize yourself with the code before moving on.

## Extracting Custom Hooks

In both the `HomePage` and `ArticlePage` components, you'll notice that we are using the `useEffect` hook in order to set the document title. The [document title](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/title) is an important part of any website, because it:

- is displayed in the browser tab

- is also displayed in the browser history
- helps with accessibility and search engine optimization (SEO)

Since we have similar logic for updating the title in both of our components (and we might want this functionality in other components as our app grows), this logic is a good candidate for a custom hook! Let's start by creating a new file for our custom hook:

`/src/hooks/useDocumentTitle.js` . Let's take the `useEffect` code from the `HomePage` component and place it in that file, inside a function called `useDocumentTitle` :

```
// src/hooks/useDocumentTitle.js
import { useEffect } from "react";

function useDocumentTitle() {
  useEffect(() => {
    document.title = "Underreacted | Home";
  }, []);
}

export default useDocumentTitle;
```

There are a couple important things going on with this custom hook already, so let's review.

So far, any time we've wanted to use a React hook (like `useState` or `useEffect` ), we've only been able to do so inside of our React components (not inside of any other JavaScript functions).

**Custom hooks also allow us to call React hooks**, so long as we call our custom hook from a React component.

Another important convention to note here: the name of our custom hook starts with the word `use` . This is a signal to React (and ESLint) that our hook should follow the [Rules of Hooks](https://reactjs.org/docs/hooks-rules.html) ↗(<https://reactjs.org/docs/hooks-rules.html>) , and also a signal to other developers that this code is meant to be used as a React hook.

## Using our Custom Hook

Now that we've extracted this custom hook to its own file, we can import it and use it in our `HomePage` component:

```
import React, { useEffect, useState } from "react";
import About from "./About";
import ArticleList from "./ArticleList";
import useDocumentTitle from "../hooks/useDocumentTitle";

function HomePage() {
  // fetch data for posts

  // set the document title
  useDocumentTitle();

  // return ...
}

export default HomePage;
```

After that update, our component should work the same. But now our `HomePage` component doesn't have to worry about the logic for updating the document title: it just needs to call the `useDocumentTitle` hook, which will handle that work.

Updating our `ArticlePage` component won't quite work with our new custom hook just yet, since the title is dynamic in this component:

```
// src/components/ArticlePage.js
function ArticlePage() {
  //...

  const pageTitle = post ? `Underreacted | ${post.title}` : "Underreacted";
  useEffect(() => {
    // depends on the page title
    document.title = pageTitle;
  }, [pageTitle]);
```

```
//...  
}
```

To solve this, we can update our `useDocumentTitle` hook to accept an argument of the page title:

```
function useDocumentTitle(pageTitle) {  
  useEffect(() => {  
    document.title = pageTitle;  
  }, [pageTitle]);  
}
```

Now, both our components can use this custom hook by passing in a page title when calling the hook:

```
// src/components/ArticlePage.js  
function ArticlePage() {  
  //...  
  
  const pageTitle = post ? `Underreacted | ${post.title}` : "Underreacted";  
  useDocumentTitle(pageTitle);  
  
  //...  
}  
  
// src/components/HomePage.js  
function HomePage() {  
  //...  
  
  useDocumentTitle("Underreacted | Home");  
  
  //...  
}
```

# Returning Data from Custom Hooks

One other common piece of logic that is shared between our components is fetching data from our API. Both the `ArticlePage` and `HomePage` share some similarities with regards to data fetching:

- They both have a couple state variables related to data fetching ( `isLoading` , `posts` , `posts` )
- They both use the `useEffect` hook to fetch data as a side effect of rendering the component

By noticing these similarities, we can recognize what logic is coupled together and what we'd need to extract in order to build out our custom hook.

Let's start with the `HomePage` component once again. Here's all of the logic that is related to working with our API:

```
// src/components/HomePage.js
function HomePage() {
  // fetch data for posts
  const [isLoading, setIsLoaded] = useState(false);
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    setIsLoaded(false);
    fetch("http://localhost:4000/posts")
      .then((r) => r.json())
      .then((posts) => {
        setPosts(posts);
        setIsLoaded(true);
      });
  }, []);
}

// return ...
```

To start off with, let's take all this code out from our `HomePage` component and create a new custom hook called `useQuery`, for querying data from our API:

```
// src/hooks/useQuery.js
import { useState, useEffect } from "react";

function useQuery() {
  const [isLoaded, setIsLoaded] = useState(false);
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    setIsLoaded(false);
    fetch("http://localhost:4000/posts")
      .then((r) => r.json())
      .then((posts) => {
        setPosts(posts);
        setIsLoaded(true);
      });
  }, []);
}

export default useQuery;
```

Just like before, we created a new file for our custom hook, and imported the React hooks that our custom hook will use. We also gave our custom hook a name that starts with `use`.

Unlike our previous custom hook, however, we're going to need to get some data back out of this component. Specifically, when we're using this component, we'll need access to two things:

- the data returned by the fetch request ( `posts` )
- the `isLoading` state

But how can we get this data **out** of the custom hook? Well, since a custom hook is **just a function**, all we need to do is have our hook **return** whatever data we need!

```
// src/hooks/useQuery.js
import { useState, useEffect } from "react";

function useQuery() {
  const [isLoaded, setIsLoaded] = useState(false);
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    setIsLoaded(false);
    fetch("http://localhost:4000/posts")
      .then((r) => r.json())
      .then((posts) => {
        setPosts(posts);
        setIsLoaded(true);
      });
  }, []);

  // return an *object* with the data and isLoading state
  return {
    posts: posts,
    isLoading: isLoaded,
  };
}

export default useQuery;
```

Now, in order to use this custom hook, we can call it from our `HomePage` component, and **destructure** the return value to get the `posts` and `isLoading` state out:

```
import useQuery from "../hooks/useQuery";

function HomePage() {
  // fetch data for posts
  const { posts, isLoading } = useQuery();

  // ...
}
```

Our `HomePage` component is now significantly cleaner, because it no longer has to worry about all the logic related to handling the fetch request and setting state based on the response — all of that logic is now nicely encapsulated in our `useQuery` hook!

In order to get this hook to work with the `ArticlePage` component as well, we need to refactor it a bit and abstract away the logic that is specific to the `HomePage` component's needs.

```
// take in the url
function useQuery(url) {
  const [isLoading, setIsLoaded] = useState(false);
  // rename `posts` to a more generic `data`
  const [data, setData] = useState(null);

  useEffect(() => {
    setIsLoaded(false);
    fetch(url)
      .then((r) => r.json())
      .then((data) => {
        setData(data);
        setIsLoaded(true);
      });
  }, [url]);
  // the url is now a dependency
  // we want to use the side effect whenever the url changes
```

```
// return an *object* with the data and isLoading state
return { data, isLoading };
}
```

Now, to use our more generic version of this hook in the `HomePage` component, we just need to make a couple small changes:

```
function HomePage() {
  const { data: posts, isLoading } = useQuery("http://localhost:4000/posts");

  // ...
}
```

Since `useQuery` now accepts a URL for the fetch request, we must pass that URL in when we call the hook. It now also returns an object with a more generic name ( `data` ), so we can [re-name that variable ↗\(https://wesbos.com/destructuring-renaming\)](#) to `posts` when destructuring.

The `useQuery` hook should now also work with our `ArticlePage` component:

```
function ArticlePage() {
  const { id } = useParams();
  const { data: post, isLoading } = useQuery(
    `http://localhost:4000/posts/${id}`
  );

  // ...
}
```

With our custom hooks in place, the completed versions of the `HomePage` and `ArticlePage` components are now both significantly shorter. Also, adding new components to our application that need access to similar functionality is now significantly easier, since we don't have to rewrite that functionality from scratch in each new component.

**Note:** While our `useQuery` hook works nicely in this example, there are some optimizations we could make to improve it, such as:

- Handling errors with `.catch` and adding an error state
- Using one state variable instead of [multiple state variables](https://reactjs.org/docs/hooks-faq.html#should-i-use-one-or-many-state-variables) ↗ (<https://reactjs.org/docs/hooks-faq.html#should-i-use-one-or-many-state-variables>), so that it doesn't re-render more than necessary
- [Using the useReducer hook instead of useState](https://reactjs.org/docs/hooks-reference.html#usereducer) ↗ (<https://reactjs.org/docs/hooks-reference.html#usereducer>) to manage state transitions
- [Caching our fetched data](https://flaviocopes.com/javascript-memoization/) ↗ (<https://flaviocopes.com/javascript-memoization/>) to prevent unnecessary network requests
- [Cancel the fetch](https://davidwalsh.name/cancel-fetch) ↗ (<https://davidwalsh.name/cancel-fetch>) if the component un-mounts before the fetch is complete

You're encouraged to try adding a few of these optimizations to this hook yourself! There's also a version of the `useQuery` hook in the solution branch called `useQueryAdvanced` that handles some of these optimizations. However, there are also pre-built solutions out there, such as [React Query](https://react-query.tanstack.com/) ↗ (<https://react-query.tanstack.com/>), that handle this logic (and more) with a pre-built custom hook.

## Conclusion

Creating custom hooks allows us to share stateful logic across multiple components. The ability to use custom hooks lets us create more concise components that are focused more on the UI logic.

The React community has also embraced custom hooks in a big way — major libraries like [React Redux](https://react-redux.js.org/api/hooks) ↗ (<https://react-redux.js.org/api/hooks>) and [React Router](https://v5.reactrouter.com/web/api/Hooks) ↗ (<https://v5.reactrouter.com/web/api/Hooks>) use custom hooks to provide a lot of their functionality, and there are lots of [community-generated custom hooks](https://github.com/rehooks/awesome-react-hooks) ↗ (<https://github.com/rehooks/awesome-react-hooks>) out there to explore and add to your projects!

## Resources