

React Context



[\(<https://github.com/learn-co-curriculum/react-hooks-use-context>\)](https://github.com/learn-co-curriculum/react-hooks-use-context)



[\(<https://github.com/learn-co-curriculum/react-hooks-use-context/issues/new>\)](https://github.com/learn-co-curriculum/react-hooks-use-context/issues/new)

Learning Goals

- Understand the use cases for React Context and what problems it solves
- Use `createContext` and the `useContext` hook to work with Context

Introduction

So far, we've learned that there is only one way to share data across multiple components: some parent component is responsible for passing down that data to any child components that need it via `props`. However, for large applications where many components need access to the same data, this approach can be a burden.

The React Context API, and the `useContext` hook, allows us to share "global" data between components without passing that data via props. Libraries like React Router and React Redux take advantage of Context under the hood, so let's see how we can use it in our applications as well!

The Prop Sharing Problem

In this lesson, we have the following components:

```
App
  └── Header
    └── ThemeButton
    └── DarkModeToggle
  └── Profile
    └── Interests
```

These components all need access to some shared state, which is currently kept in the App component. Here's a diagram of the state the components share:

```
App [theme]
  └─ Header [theme, user]
    |   └─ ThemeButton [theme]
    |   └─ DarkModeToggle [theme]
    └─ Profile [user]
      └─ Interests [theme]
```

As you can see, even in this small example, we have several components that need access to the same data.

In addition, because of the requirement that we must pass down data from parent to child components, we have a couple of components that take in some data via props, only to pass it along to a child component. For example, looking at the `Profile` component, we can see that it takes in a `theme` prop, even though it doesn't use it directly — it only needs to take this prop in so that it can pass it down to the `Interests` component:

```
// takes theme as a prop
function Profile({ user, theme }) {
  if (!user) return <h2>Please Login To View Profile</h2>;
  return (
    <div>
      <h2>{user.name}'s Profile</h2>
      {/* passes theme down to Interests */}
      <Interests interests={user.interests} theme={theme} />
    </div>
  );
}
```

This is known as **prop-drilling**, and it can become a burden for deeply-nested component hierarchies.

Let's see how to use React Context to solve this problem.

Creating Context

In order to create our context data, we need to create two things:

- The actual context object
- A context provider component

Let's start by creating the context for our `user` data. To organize our context code, make a new file called `/src/context/user.js`. Then, create our context:

```
// src/context/user.js
import React from 'react';

const UserContext = React.createContext();
```

After creating the context object, we need a special "provider" component that will give access to the context data to its child components. Here's how we can set up the context provider:

```
// src/context/user.js
import React from 'react';

// create the context
const UserContext = React.createContext();

// create a provider component
function UserProvider({ children }) {
  // the value prop of the provider will be our context data
  // this value will be available to child components of this provider
  return <UserContext.Provider value={null}>{children}</UserContext.Provider>;
}

export { UserContext, UserProvider };
```

With our context created, and our provider component all set up, let's see how we can use this context data from other components.

Curious about what children are? Review our [React Children Lesson ↗\(https://github.com/learn-co-curriculum/react-hooks-children\)](https://github.com/learn-co-curriculum/react-hooks-children) for a refresher!

Using Context

In order to give our components access to the context data, we must first use the provider component to wrap around any component that need access to the context. Based on our component hierarchy, the `Header` and `Profile` components both need access to the `user` data in our context:

```
App [theme]
└─ Header [theme, user]
  └─ ThemeButton [theme]
  └─ DarkModeToggle [theme]
└─ Profile [user]
  └─ Interests [theme]
```

So let's update the `App` component with the `UserProvider`:

```
import React, { useState } from 'react';
import Header from './Header';
import Profile from './Profile';
// import the provider
import { UserProvider } from '../context/user';

function App() {
  const [theme, setTheme] = useState('dark');
  return (
    <main className={theme}>
      {/* wrap components that need access to context data in the provider*/}
      <UserProvider>
```

```
<Header theme={theme} setTheme={setTheme} />
<Profile theme={theme} />
</UserProvider>
</main>
);
}

export default App;
```

You'll notice we also removed the `user` prop from these components, since we'll be accessing that data via context instead.

We've also included our `Header` and `Profile` components as children of our `UserProvider`.

Next, in order to access the context data from our components, we can use the `useContext` hook. This is another hook that's built into React, and it lets us access the `value` of our context provider in any child component. Here's how it looks when used in our `Profile` component:

```
// import the useContext hook
import React, { useContext } from 'react';
// import the UserContext we created
import { UserContext } from '../context/user';
import Interests from './Interests';

function Profile({ theme }) {
  // call useContext with our UserContext
  const user = useContext(UserContext);

  // now, we can use the user object just like we would if it was passed as a prop!
  console.log(user);
  if (!user) return <h2>Please Login To View Profile</h2>;
  return (
    <div>
      <h2>{user.name}'s Profile</h2>
      <Interests interests={user.interests} theme={theme} />
    </div>
  );
}

export default Profile;
```

```
</div>
);
}
```

You can test this out by updating the `value` prop in our `UserProvider` to something different, and see that the `Profile` component has access to the updated data:

```
function UserProvider({ children }) {
  const currentUser = {
    name: 'Duane',
    interests: ['Coding', 'Biking', "Words ending in 'ing'"],
  };
  return (
    <UserContext.Provider value={currentUser}>{children}</UserContext.Provider>
  );
}
```

Let's hook up the `Header` component to our context as well:

```
import React, { useContext } from 'react';
import ThemedButton from './ThemedButton';
import DarkModeToggle from './DarkModeToggle';
import defaultUser from '../data';
import { UserContext } from '../context/user';

function Header({ theme, setTheme }) {
  const user = useContext(UserContext);

  function handleLogin() {
    if (user) {
      // setUser(null);
    } else {
      // setUser(defaultUser);
    }
  }
}
```

```

    }
}

return (
  <header>
    <h1>React Context</h1>
    <nav>
      <ThemedButton onClick={handleLogin} theme={theme}>
        {user ? 'Logout' : 'Login'}
      </ThemedButton>
      <DarkModeToggle theme={theme} setTheme={setTheme} />
    </nav>
  </header>
);
}

```

One thing you'll notice is that our `Header` component also is meant to handle logging in/logging out a user. In the first version of our app, that functionality was available to use in the `App` component since we had a `user` variable as **state**:

```

function App() {
  const [theme, setTheme] = useState('dark');
  const [user, setUser] = useState(null);
  return (
    <main className={theme}>
      <Header theme={theme} setTheme={setTheme} user={user} setUser={setUser} />
      <Profile theme={theme} user={user} />
    </main>
  );
}

```

We can re-gain this functionality by setting up the **context** value to be stateful instead! (Remember to import `useState` !)

```
function UserProvider({ children }) {
  const [user, setUser] = useState(null);
  return (
    <UserContext.Provider value={{ user, setUser }}>
      {children}
    </UserContext.Provider>
  );
}
```

Since the `UserProvider` component is still just a React component, we can use any hooks we'd like within this component. You could also use the `useEffect` hook in the provider, if you'd like: for example, to have your provider component fetch some data from an API when it loads; or to read some saved data from `localStorage`.

In the code above, we're using `useState` to create a `user` state variable as well as a setter function. In the `Provider`, we're now using an **object** with `user` and `setUser` as the **value** for our context.

After this update, we can now use the `setUser` function in our `Header` component:

```
function Header({ theme, setTheme }) {
  const { user, setUser } = useContext(UserContext);
  function handleLogin() {
    if (user) {
      setUser(null);
    } else {
      setUser(defaultUser);
    }
  }
  // ...
}
```

We'll also need to update the `Profile` component since our context value has changed:

```
function Profile({ theme }) {  
  const { user } = useContext(UserContext);  
  // ...  
}
```

Exercise

Now that you've seen one approach to using React Context for our user data, try to implement React Context to handle the `theme` data for the app as well!

Completed code for this exercise is in the `solution` branch.

A Word of Caution

Once new developers encounter context, it's often tempting to reach for it as a solution to all your React state needs, since it helps save the pain of "prop drilling". However, React recommends using context sparingly:

Context is primarily used when some data needs to be accessible by many components at different nesting levels. Apply it sparingly because it makes component reuse more difficult.

If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context. — [Before You Use Context ↗](#) (<https://reactjs.org/docs/context.html#before-you-use-context>)

Keep this in mind when you're considering adding context to your application. Think about whether or not the data that's being held in context is truly *global*, and shared by many components.

This

[video ↗](#) (<https://youtu.be/3XaXKiXtNjw>)



(<https://youtu.be/3XaXKiXtNjw>)

by React Router creator Michael Jackson shows an alternative to using context for the sake of saving from props drilling, and demonstrates how to use *composition* instead.

Conclusion

React's Context system gives us a way to share global data across multiple components without needing to pass that data via props. Context should be used sparingly, but it is a helpful tool for simplifying our components and minimizing the need for prop drilling.

Resources

- [React Context ↗](https://reactjs.org/docs/context.html) (<https://reactjs.org/docs/context.html>)
- [useContext ↗](https://reactjs.org/docs/hooks-reference.html#usecontext) (<https://reactjs.org/docs/hooks-reference.html#usecontext>)
- [Using Composition Instead of Context ↗](https://youtu.be/3XaXKiXtNjw) (<https://youtu.be/3XaXKiXtNjw>)



(<https://youtu.be/3XaXKiXtNjw>)

- [Application State Management with React ↗](https://kentcdodds.com/blog/application-state-management-with-react) (<https://kentcdodds.com/blog/application-state-management-with-react>)