JULY 21, 2021 / #REACT

React Context for Beginners – The Complete Guide (2021)

Reed Barger

React context is an essential tool for every React developer to know. I lets you easily share state in your applications.

In this comprehensive guide, we will cover what React context is, how to use it, when and when not to use context, and lots more.

Even if you've never worked with React context before, you're in the right place. You will learn everything you need to know with simple, step-by-step examples.

Let's get started!

> Want the ultimate guide to learn React from front to back? Check out **The React Bootcamp**.

Table of Contents

What is React context?

React context allows us to pass down and use (consume) data in whatever component we need in our React app without using props.

In other words, React context allows us to share data (state) across our components more easily.

When should you use React context?

React context is great when you are passing data that can be used in any component in your application.

These types of data include:

- Theme data (like dark or light mode)
- User data (the currently authenticated user)
- Location-specific data (like user language or locale)

Why? Because context was not made as an entire state management system. It was made to make consuming data easier.

You can think of React context as the equivalent of global variables for our React components.

What problems does React context solve?

React context helps us avoid the problem of props drilling.

Props drilling is a term to describe when you pass props down multiple levels to a nested component, through components that don't need it.

Here is an example of props drilling. In this application, we have access to theme data that we want to pass as a prop to all of our app's components.

As you can see, however, the direct children of `App`, such as `Header`, also have to pass the theme data down using props.

```
export default function App({ theme }) {
  return (
    <>
      <Header theme={theme} />
      <Main theme={theme} />
      <Sidebar theme={theme} />
      <Footer theme={theme} />
    </>
  );
```

```
      <>
        <User theme={theme} />
        <Login theme={theme} />
        <Menu theme={theme} />
      </>
    );
  }
```

What is the issue with this example?

The issue is that we are drilling the `theme` prop through multiple components that don't immediately need it.

The `Header` component doesn't need `theme` other than to pass it down to its child component. In other words, it would be better for `User` , `Login` and `Menu` to consume the `theme` data directly.

This is the benefit of React context – we can bypass using props entirely and therefore avoid the issue of props drilling.

How do I use React context?

Context is an API that is built into React, starting from React version 16.

This means that we can create and use context directly by importing React in any React project.

There are four steps to using React context:

1. Create context using the `createContext` method.

3. Put any value you like on your context provider using the `value` prop.
4. Read that value within any component by using the context consumer.

Does all this sound confusing? It's simpler than you think.

Let's take a look at a very basic example. In our `App`, let's pass down our own name using Context and read it in a nested component: `User`.

```
import React from 'react';

export const UserContext = React.createContext();

export default function App() {
  return (
    <UserContext.Provider value="Reed">
      <User />
    </UserContext.Provider>
  )
}

function User() {
  return (
    <UserContext.Consumer>
      {value => <h1>{value}</h1>}
      {/* prints: Reed */}
    </UserContext.Consumer>
  )
}
```

Let's break down what we are doing, step-by-step:

we are doing here because your component will be in another file. Note that we can pass an initial value to our `value` prop when we call `React.createContext()`.

2. In our `App` component, we are using `UserContext`. Specifically `UserContext.Provider`. The created context is an object with two properties: `Provider` and `Consumer`, both of which are components. To pass our value down to every component in our App, we wrap our Provider component around it (in this case, `User`).
3. On `UserContext.Provider`, we put the value that we want to pass down our entire component tree. We set that equal to the `value` prop to do so. In this case, it is our name (here, Reed).
4. In `User`, or wherever we want to consume (or use) what was provided on our context, we use the consumer component: `UserContext.Consumer`. To use our passed down value, we use what is called the **render props pattern**. It is just a function that the consumer component gives us as a prop. And in the return of that function, we can return and use `value`.

What is the useContext hook?

Looking at the example above, the render props pattern for consuming context may look a bit strange to you.

Another way of consuming context became available in React 16.8 with the arrival of React hooks. We can now consume context with the **useContext hook**.

Here is the example above using the useContext hook:

```
import React from 'react';

export const UserContext = React.createContext();

export default function App() {
  return (
    <UserContext.Provider value="Reed">
      <User />
    </UserContext.Provider>
  )
}

function User() {
  const value = React.useContext(UserContext);

  return <h1>{value}</h1>;
}
```

The benefit of the useContext hook is that it makes our components more concise and allows us to create our own custom hooks.

You can either use the consumer component directly or the useContext hook, depending on which pattern you prefer.

You may not need context

The mistake many developers make is reaching for context when once they have to pass props down several levels to a component.

Here is an application with a nested `Avatar` component that requires

```
export default function App({ user }) {
  const { username, avatarSrc } = user;

  return (
    <main>
      <Navbar username={username} avatarSrc={avatarSrc} />
    </main>
  );
}

function Navbar({ username, avatarSrc }) {
  return (
    <nav>
      <Avatar username={username} avatarSrc={avatarSrc} />
    </nav>
  );
}

function Avatar({ username, avatarSrc }) {
  return <img src={avatarSrc} alt={username} />;
}
```

If possible, we want to avoid passing multiple props through components that don't need it.

What can we do?

Instead of immediately resorting to context because we are prop drilling, we should better compose our components.

Since only the top most component, `App`, needs to know about the `Avatar` component, we can create it directly within `App`.

This allows us to pass down a single prop, `avatar`, instead of two.

```
  const avatar = <img src={avatarSrc} alt={username} />;

  return (
    <main>
      <Navbar avatar={avatar} />
    </main>
  );
}

function Navbar({ avatar }) {
  return <nav>{avatar}</nav>;
}
```

In short: don't reach for context right away. See if you can better organize your components to avoid prop drilling.

Does React context replace Redux?

Yes and no.

For many React beginners, Redux is a way of more easily passing around data. This is because Redux comes with React context itself.

However, if you are not also *updating* state, but merely passing it down your component tree, you do not need a global state management library like Redux.

React context caveats

Why it is not possible to update the value that React context passes down?

any third-party library, it is generally not recommended for performance reasons.

The issue with this approach lies in the way that React context triggers a re-render.

If you are passing down an object on your React context provider and any property on it updates, what happens? *Any component which consumes that context will re-render.*

This may not be a performance issue in smaller apps with few state values that are not updated very often (such as theme data). But it is a problem if you will be performing many state updates in an application with a lot of components in your component tree.

Conclusion

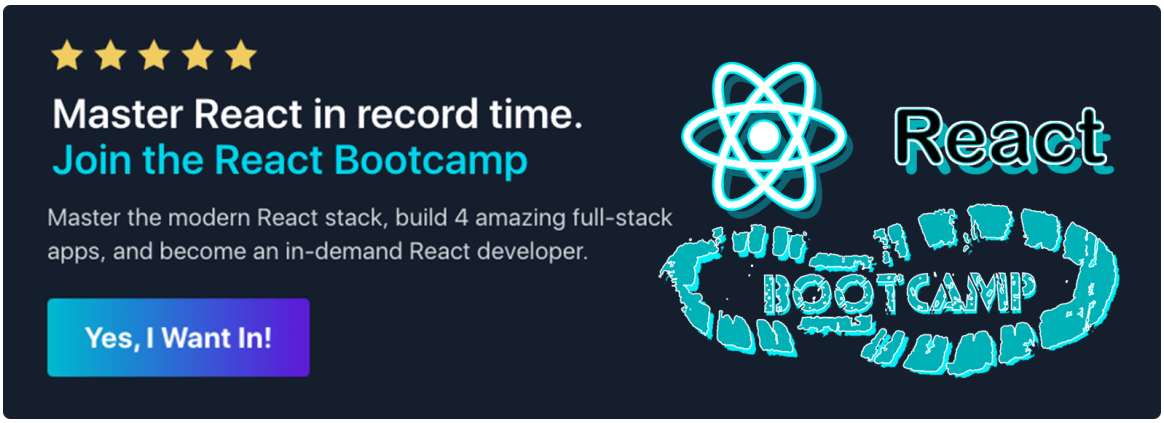
I hope this guide gave you a better understanding of how to use React context from front to back.

If you want an even more in-depth grasp of how to use React context to build amazing React projects, check out The React Bootcamp.

Want to become a React pro? Join The React Bootcamp

The React Bootcamp takes everything you should know about learning React and bundles it into one comprehensive package, including videos, cheatsheets, plus special bonuses.

future:



Click here to be notified when it opens

Reed Barger

React developer who loves to make incredible apps. Showing you how at ReactBootcamp.com

---

If this article was helpful, tweet it.

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Trending Guides

Big O Notation
SQL Outer Join
Python For Loop
What is JavaScript?
Learn How To Code
Chrome Bookmarks
Concatenate Excel
C# String to Int
Git Switch Branch
JavaScript Splice
Model View Controller
Git Checkout Remote Branch
Insert Checkbox in Word
Find and Replace in Word
C Programming Language
HTML Link
Bayes Rule
Python Map
HTML Italics
Python SQL
HTML Bold
GraphQL VS Rest
If Function Excel
HTML List
Wav File
Subnet Cheat Sheet
String to Char Array Java
JavaScript Append to Array
Add Page Numbers in Word
JavaScript Projects

Our Nonprofit