Using Context API in React (Hooks and Classes)

React Context API is a way to essentially create global variables that can be passed around in a React app. This is the alternative to "prop drilling", or passing props from grandparent to parent to child, and so on. Context is often touted as a simpler, lighter solution to using Redux for state management. I haven't used Redux myself yet, but every time I use React's Context API, I have to look it up because it doesn't seem obvious to me.

I'm going to leave some brief, concise steps to getting started with Context here.

Prerequisite

- Read **Getting Started with React** or **Build a React App with Hooks** if you don't know React or React Hooks yet.

Create Context

Imagine I have some information I want to be available anywhere or everywhere throughout a React app. A theme might be implemented using Context - for example, on this site I have Context serving three themes: dark mode, light mode, and MS-DOS mode (on the **404 page**).In this simple example, I'll use a logged in user.

I'll create Context, and call it `UserContext`. This will also give me `UserContext.Provider` and `UserContext.Consumer`. What these two components do is straightforward:

- **Provider -** The component that provides the value
- **Consumer -** A component that is consuming the value

So I'll create it with `React.createContext()` in a new file called `UserContext.js`.

src/UserContext.js

```
import React from 'react'

const UserContext = React.createContext()

export const UserProvider = UserContext.Provider
export const UserConsumer = UserContext.Consumer

export default UserContext
```

I'm passing in an empty object value here to represent that I might be filling in this data later with an API call. You can pre-populate this with whatever data you want, in case you're not retrieving the data through an API.

```
React.createContext(true)
```

Providing Context

The provider always needs to exist as a wrapper around the parent element, no matter how you choose to consume the values. I'll wrap the entire `App` component in the `Provider`. I'm just creating some value (`user`) and passing it down as the `Provider` value prop.

```
import React from 'react'
import HomePage from './HomePage'
import { UserProvider } from './UserContext'

function App() {
  const user = { name: 'Tania', loggedIn: true }

  return (
    <UserProvider value={user}>
      <HomePage />
    </UserProvider>
  )
}
```

Now any child, grandchild, great-grandchild, and so on will have access to `user` as a prop. Unfortunately, retrieving that value is slightly more involved than simply getting it like you might with `this.props` or `this.state`.

Consuming Context

The way you provide Context is the same for class and functional components, but consuming it is a little different for both.

Class component

The most common way to access Context from a class component is via the static `contextType`. If you need the value from Context outside of `render`, or in a lifecycle method, you'll use it this way.

src/HomePage.js (class example)

```
import React, { Component } from 'react'
import UserContext from './UserContext'

class HomePage extends Component {
  static contextType = UserContext

  componentDidMount() {
    const user = this.context

    console.log(user) // { name: 'Tania', loggedIn: true }
  }

  render() {
    return <div>{user.name}</div>
  }
}
```

The traditional way to retrieve Context values was by wrapping the child component in the `Consumer`. From there, you would be able to access the value prop as `props`. You may still see this, but it's more of a legacy way of accessing Context.

src/HomePage.js (class example)

```
import React, { Component } from 'react'
import { UserConsumer } from './UserContext'

class HomePage extends Component {
  render() {
    return (
```

```
        }}
      </UserConsumer>
    )
  }
}
```

Functional component and Hooks

For functional components, you'll use `useContext`, such as in the example below. This is the equivalent of `static contextType`.

src/HomePage.js

```
import React, { useContext } from 'react'
import UserContext from './UserContext'

export const HomePage = () => {
  const user = useContext(UserContext)

  return <div>{user.name}</div>
}
```

Updating Context

Updating context is not much different than updating regular state. We can create a wrapper class that contains the state of Context and the means to update it.

src/UserContext.js

```
import React, { Component } from 'react'

const UserContext = React.createContext()

class UserProvider extends Component {
  // Context state
  state = {
    user: {},
  }

  // Method to update state
  setUser = (user) => {
    this.setState((prevState) => ({ user }))
  }

  render() {
    const { children } = this.props
    const { user } = this.state
    const { setUser } = this

    return (
      <UserContext.Provider
        value={{
          user,
          setUser,
        }}
      >
        {children}
      </UserContext.Provider>
    )
```

```
export default UserContext

export { UserProvider }
```

Now you can update and view the user from the Context method.

```
import React, { Component } from 'react'
import UserContext from './UserContext'

class HomePage extends Component {
  static contextType = UserContext

  render() {
    const { user, setUser } = this.context

    return (
      <div>
        <button
          onClick={() => {
            const newUser = { name: 'Joe', loggedIn: true }

            setUser(newUser)
          }}
        >
          Update User
        </button>
        <p>{`Current User: ${user.name}`}</p>
      </div>
    )
  }
}
```

In my opinion, the biggest downfall of Context API with classes is that you cannot use multiple static `contextTypes` in one component. This leads to the necessity of having one really big Context for all global state in an application, so it's not sufficient for a large application. The method of creating a wrapper for Context is also difficult to test.

Conclusion

To summarize:

- Use `const ___Context = React.createContext()` to create context.
- Pull `___Context.Provider` and `___Context.Consumer` out of `___Context`
- Wrap `Provider` around your parent component.
- A class can consume with `static contextType = ___Context`
- A functional component can consume with `const x = useContext(___Context)`

Hope this helps!

Published

April 25, 2019

Tags

Author

Hey there, I'm Tania—a software engineer and open-source creator. This website is a compendium of things I've learned while writing code for fun and profit.

I believe there are pockets of the internet that can still be beautiful. That's why my site has:

- No ads
- No social media
- No tracking or analytics
- No sponsored posts
- No affiliate links
- No paywall
- No third-party scripts
- No bullshit

Support

If you like what I do and would like to support me, you can do so below!

Buy me a coffee

Become a Patron

Stay in touch

Every now and then I'll send out an email when I've created something new. No spam, unsubscribe whenever. Or follow on RSS.

Subscribe to the email list

RSS Feed

Up next

Using React Router for a Single Page Application

Getting Started with Vue: An Overview and Walkthrough Tutorial