useContext

`useContext` is a React Hook that lets you read and subscribe to context from your component.

```
const value = useContext(SomeContext)
```

Reference

`useContext(SomeContext)`

Call `useContext` at the top level of your component to read and subscribe to context.

```
import { useContext } from 'react';

function MyComponent() {
  const theme = useContext(ThemeContext);
  // ...
```

See more examples below.

Parameters

- `SomeContext`: The context that you've previously created with `createContext`. The context itself does not hold the information, it only represents the kind of information you can provide or read from components.

Returns

`useContext` returns the context value for the calling component. It is determined as the `value` passed to the closest `SomeContext.Provider` above the calling component in the tree. If there is no such provider, then the returned value will be the `defaultValue` you have passed to `createContext` for that context. The returned value is always up-to-date. React automatically re-renders components that read some context if it changes.

Caveats

- `useContext()` call in a component is not affected by providers returned from the *same* component. The corresponding `<Context.Provider>` **needs to be *above*** the component doing the `useContext()` call.
- React **automatically re-renders** all the children that use a particular context starting from the provider that receives a different `value`. The previous and the next values are compared with the `Object.is` comparison. Skipping re-

renders with `memo` does not prevent the children receiving fresh context values.

- If your build system produces duplicates modules in the output (which can happen with symlinks), this can break context. Passing something via context only works if `SomeContext` that you use to provide context and `SomeContext` that you use to read it are ***exactly* the same object**, as determined by a `===` comparison.

Usage

Passing data deeply into the tree

Call `useContext` at the top level of your component to read and subscribe to context.

```
import { useContext } from 'react';

function Button() {
  const theme = useContext(ThemeContext);
  // ...
```

`useContext` returns the context value for the context you passed. To determine the context value, React searches the component tree and finds **the closest context provider above** for that particular context.

To pass context to a `Button`, wrap it or one of its parent components into the corresponding context provider:

```
function MyPage() {
  return (
    <ThemeContext.Provider value="dark">
      <Form />
```

```
    </ThemeContext.Provider>
  );
}

function Form() {
  // ... renders buttons inside ...
}
```

It doesn't matter how many layers of components there are between the provider and the `Button`. When a `Button` *anywhere* inside of `Form` calls `useContext(ThemeContext)`, it will receive `"dark"` as the value.

Pitfall

`useContext()` always looks for the closest provider *above* the component that calls it. It searches upwards and **does not** consider providers in the component from which you're calling `useContext()`.

App.js Download Reset

```
import { createContext, useContext } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
  return (
    <ThemeContext.Provider value="dark">
      <Form />
    </ThemeContext.Provider>
  )
}
```

Show more

Updating data passed via context

Often, you'll want the context to change over time. To update context, combine it with state. Declare a state variable in the parent component, and pass the current state down as the context value to the provider.

```
function MyPage() {
  const [theme, setTheme] = useState('dark');
  return (
    <ThemeContext.Provider value={theme}>
      <Form />
      <Button onClick={() => {
        setTheme('light');
      }}>
        Switch to light theme
      </Button>
    </ThemeContext.Provider>
  );
}
```

Now any `Button` inside of the provider will receive the current `theme` value. If you call `setTheme` to update the `theme` value that you pass to the provider, all `Button` components will re-render with the new `'light'` value.

Examples of updating context

1. Updating a value via context
2. Updating an object via context
3.

Example 1 of 5:

Updating a value via context

In this example, the `MyApp` component holds a state variable which is then passed to the `ThemeContext` provider. Checking the “Dark mode” checkbox updates the state. Changing the provided value re-renders all the components using that context.

App.js

Download Reset

```
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext.Provider value={theme}>
      <Form />
      <label>
        <input
          type="checkbox"
```

Show more

Note that `value="dark"` passes the `"dark"` string, but `value={theme}` passes the value of the JavaScript `theme` variable with JSX curly braces. Curly braces also let you pass context values that aren't strings.

Next Example

Specifying a fallback default value

If React can't find any providers of that particular context in the parent tree, the context value returned by `useContext()` will be equal to the default value that you specified when you created that context:

```
const ThemeContext = createContext(null);
```

The default value **never changes**. If you want to update context, use it with state as described above.

Often, instead of `null`, there is some more meaningful value you can use as a default, for example:

```
const ThemeContext = createContext('light');
```

This way, if you accidentally render some component without a corresponding provider, it won't break. This also helps your components work well in a test environment without setting up a lot of providers in the tests.

In the example below, the "Toggle theme" button is always light because it's **outside any theme context provider** and the default context theme value is `'light'`. Try editing the default theme to be `'dark'`.

App.js Download Reset

```
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext('light');

export default function MyApp() {
  const [theme, setTheme] = useState('light');
  return (
    <>
      <ThemeContext.Provider value={theme}>
        <Form />
      </ThemeContext.Provider>
      <Button onClick={() => {
```

Show more

Overriding context for a part of the tree

You can override the context for a part of the tree by wrapping that part in a provider with a different value.

```
<ThemeContext.Provider value="dark">
  ...
  <ThemeContext.Provider value="light">
    <Footer />
  </ThemeContext.Provider>
  ...
</ThemeContext.Provider>
```

You can nest and override providers as many times as you need.

Examples of overriding context

1. Overriding a theme
2. Automatically nested headings

Example 1 of 2:

Overriding a theme

Here, the button *inside* the `Footer` receives a different context value (`"light"`) than the buttons outside (`"dark"`).

App.js

Download Reset

```
import { createContext, useContext } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
  return (
    <ThemeContext.Provider value="dark">
      <Form />
    </ThemeContext.Provider>
  )
}
```

Show more

Next Example

Optimizing re-renders when passing objects and functions

You can pass any values via context, including objects and functions.

```
function MyApp() {
  const [currentUser, setCurrentUser] = useState(null);

  function login(response) {
    storeCredentials(response.credentials);
    setCurrentUser(response.user);
  }

  return (
    <AuthContext.Provider value={{ currentUser, login }}>
      <Page />
    </AuthContext.Provider>
  );
}
```

Here, the context value is a JavaScript object with two properties, one of which is a function. Whenever `MyApp` re-renders (for example, on a route update), this will be a *different* object pointing at a *different* function, so React will also have to re-render all components deep in the tree that call `useContext(AuthContext)`.

In smaller apps, this is not a problem. However, there is no need to re-render them if the underlying data, like `currentUser`, has not changed. To help React take advantage of that fact, you may wrap the `login` function with `useCallback` and wrap the object creation into `useMemo`. This is a performance optimization:

```
import { useCallback, useMemo } from 'react';

function MyApp() {
  const [currentUser, setCurrentUser] = useState(null);

  const login = useCallback((response) => {
    storeCredentials(response.credentials);
    setCurrentUser(response.user);
  }, []);

  const contextValue = useMemo(() => ({
    currentUser,
    login
  }), [currentUser, login]);

  return (
    <AuthContext.Provider value={contextValue}>
      <Page />
    </AuthContext.Provider>
  );
}
```

As a result of this change, even if `MyApp` needs to re-render, the components calling `useContext(AuthContext)` won't need to re-render unless `currentUser` has changed.

Read more about `useMemo` and `useCallback`.

Troubleshooting

My component doesn't see the value from my provider

There are a few common ways that this can happen:

1. You're rendering `<SomeContext.Provider>` in the same component (or below) as where you're calling `useContext()`. Move `<SomeContext.Provider>` *above and outside* the component calling `useContext()`.
2. You may have forgotten to wrap your component with `<SomeContext.Provider>`, or you might have put it in a different part of the tree than you thought. Check whether the hierarchy is right using React DevTools.
3. You might be running into some build issue with your tooling that causes `SomeContext` as seen from the providing component and `SomeContext` as seen by the reading component to be two different objects. This can happen if you use symlinks, for example. You can verify this by assigning them to globals like `window.SomeContext1` and `window.SomeContext2` and then checking whether `window.SomeContext1 === window.SomeContext2` in the console. If they're not the same, fix that issue on the build tool level.

I am always getting `undefined` from my context although the default value is different

You might have a provider without a `value` in the tree:

```
// 🚩 Doesn't work: no value prop
<ThemeContext.Provider>
   <Button />
</ThemeContext.Provider>
```

If you forget to specify `value`, it's like passing `value={undefined}`.

You may have also mistakingly used a different prop name by mistake:

```
// 🚩 Doesn't work: prop should be called "value"
<ThemeContext.Provider theme={theme}>
   <Button />
```

```
</ThemeContext.Provider>
```

In both of these cases you should see a warning from React in the console. To fix them, call the prop `value`:

```
// ✅ Passing the value prop
<ThemeContext.Provider value={theme}>
   <Button />
</ThemeContext.Provider>
```

Note that the default value from your `createContext(defaultValue)` call is only used **if there is no matching provider above at all.** If there is a `<SomeContext.Provider value={undefined}>` component somewhere in the parent tree, the component calling `useContext(SomeContext)` *will* receive `undefined` as the context value.

PREVIOUS
useCallback

NEXT
useDebugValue

Meta Open Source

Learn React

Quick Start

Installation

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

API Reference

React APIs

React DOM APIs

Community

Code of Conduct

Meet the Team

Docs Contributors

Acknowledgements

More

Blog

React Native

Privacy

Terms