React's useContext hook makes it easy to pass data throughout your app without manually passing props down the tree.

It makes up part of React's Context API (the other parts being the Provider and Consumer components, which we'll see examples of below). Context can make a nice simple alternative to Redux when your data is simple or your app is small.

In this post we'll look at how to use `useContext`.

useContext vs. Consumer: First, the hard way

With the `Consumer` component, the typical way to use the Context API looks like this:

```
import React from "react";
import ReactDOM from "react-dom";

// Create a Context
const NumberContext = React.createContext();
// It returns an object with 2 values:
// { Provider, Consumer }

function App() {
  // Use the Provider to make a value available to all
  // children and grandchildren
  return (
    <NumberContext.Provider value={42}>
      <div>
        <Display />
      </div>
    </NumberContext.Provider>
  );
```

```
}

function Display() {
  // Use the Consumer to grab the value from context
  // Notice this component didn't get any props!
  return (
    <NumberContext.Consumer>
      {value => <div>The answer is {value}.</div>}
    </NumberContext.Consumer>
  );
}

ReactDOM.render(<App />, document.querySelector("#root"));
```

First, we create a new context, which we store in `NumberContext`. This is an object with 2 properties: `Provider` and `Consumer`. They're a matched pair, and they're born knowing how to communicate with each other (but not with other contexts).

Then, we render the `NumberContext.Provider` with some contents, and pass a `value` prop to it. It will make this value available to all of its descendants, and their descendants. The whole subtree will be able to use the Consumer (or useContext) to read out the value.

Finally, we read the value with the Consumer inside the `Display` component.

Destructuring Context is Optional

It's pretty common to destructure the context at creation time, like this:

```
const { Provider, Consumer } = React.createContext();
```

We're not doing that here, instead accessing the properties explicitly as `NumberContext.Provider` and `NumberContext.Consumer`, but I wanted to show it to you in case you see it around.

Consumer Adds Extra Nesting

Look at how we read the `value` inside the `Display` component: we have to wrap our content in a `NumberContext.Consumer` and use the render props pattern – passing a function as a child – to retrieve the value and display it.

This works fine, and "render props" can be a great pattern for passing dynamic data around, but it does introduce some extra nesting, cognitive overhead (especially if you're not used to it), and it just looks a bit weird.

useContext lets you "use" context without a Consumer

Let's rewrite the `Display` component with the `useContext` hook:

```
// import useContext (or we could write React.useContext)
import React, { useContext } from 'react';

// ...

function Display() {
  const value = useContext(NumberContext);
  return <div>The answer is {value}.</div>;
}
```

Call `useContext`, pass in the context object you got from `React.createContext`, and out pops the value. That's all there is to it! Way easier to read, right?

The only thing to watch out for is that you have to pass the *whole* context object to `useContext` – not just the Consumer! (this is why I didn't destructure the Context object right away) React will warn you if you forget, but try to rememeber, eh?

New to React? There's a lot to learn, but hooks aren't the place to start!

Nested Consumers vs. useContext

You might have a case where your component needs to receive data from *multiple* parent contexts, leading to code like this:

```
function HeaderBar() {
  return (
    <CurrentUser.Consumer>
      {user =>
        <Notifications.Consumer>
          {notifications =>
            <header>
              Welcome back, {user.name}!
              You have {notifications.length} notifications.
            </header>
          }
        </Notifications.Consumer>
      }
    </CurrentUser.Consumer>
  );
}
```

This is an awful lot of nesting just to receive 2 values. Here's what it can look like with `useContext`:

```
function HeaderBar() {
  const user = useContext(CurrentUser);
  const notifications = useContext(Notifications);
  return (
    <header>
      Welcome back, {user.name}!
      You have {notifications.length} notifications.
    </header>
  );
}
```