Passing Data Deeply with Context

Usually, you will pass information from a parent component to a child component via props. But passing props can become verbose and inconvenient if you have to pass them through many components in the middle, or if many components in your app need the same information. *Context* lets the parent component make some information available to any component in the tree below it—no matter how deep—without passing it explicitly through props.

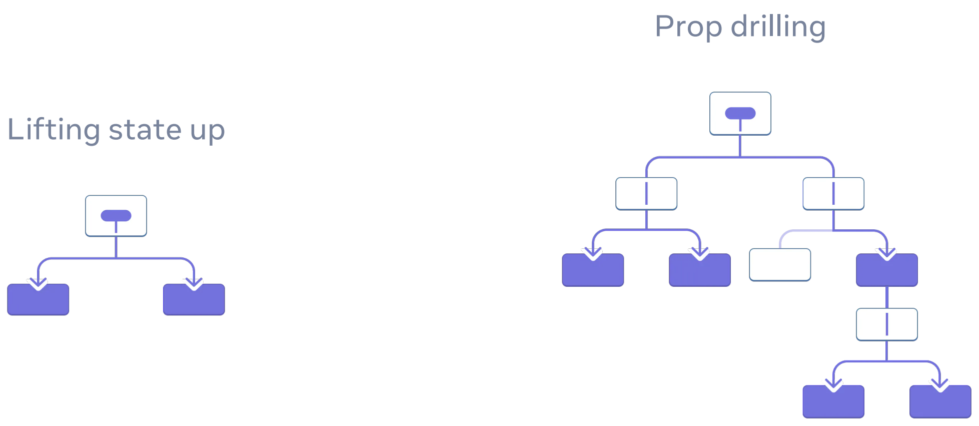
You will learn

- What "prop drilling" is
- How to replace repetitive prop passing with context
- Common use cases for context
- Common alternatives to context

The problem with passing props

Passing props is a great way to explicitly pipe data through your UI tree to the components that use it.

But passing props can become verbose and inconvenient when you need to pass some prop deeply through the tree, or if many components need the same prop. The nearest common ancestor could be far removed from the components that need data, and lifting state up that high can lead to a situation called “prop drilling”.



Wouldn’t it be great if there were a way to “teleport” data to the components in the tree that need it without passing props? With React’s context feature, there is!

Context: an alternative to passing props

Context lets a parent component provide data to the entire tree below it. There are many uses for context. Here is one example. Consider this `Heading` component that accepts a `level` for its size:

App.js Section.js Heading.js Reset

```
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
  return (
    <Section>
      <Heading level={1}>Title</Heading>
      <Heading level={2}>Heading</Heading>
      <Heading level={3}>Sub-heading</Heading>
      <Heading level={4}>Sub-sub-heading</Heading>
      <Heading level={5}>Sub-sub-sub-heading</Heading>
      <Heading level={6}>Sub-sub-sub-sub-heading</Heading>
```

Let’s say you want multiple headings within the same `Section` to always have the same size:

App.js Section.js Heading.js Reset

```
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
  return (
    <Section>
      <Heading level={1}>Title</Heading>
      <Section>
        <Heading level={2}>Heading</Heading>
        <Heading level={2}>Heading</Heading>
        <Heading level={2}>Heading</Heading>
        <Section>
          <Heading level={3}>Sub-heading</Heading>
          <Heading level={3}>Sub-heading</Heading>
          <Heading level={3}>Sub-heading</Heading>
          <Heading level={3}>Sub-heading</Heading>
          <Section>
            <Heading level={4}>Sub-sub-heading</Heading>
            <Heading level={4}>Sub-sub-heading</Heading>
            <Heading level={4}>Sub-sub-heading</Heading>
          </Section>
        </Section>
      </Section>
    </Section>
  );
}

```

Show less

Currently, you pass the `level` prop to each `<Heading>` separately:

```
<Section>
  <Heading level={3}>About</Heading>
  <Heading level={3}>Photos</Heading>
  <Heading level={3}>Videos</Heading>
```

```
</Section>
```

It would be nice if you could pass the `level` prop to the `<Section>` component instead and remove it from the `<Heading>`. This way you could enforce that all headings in the same section have the same size:

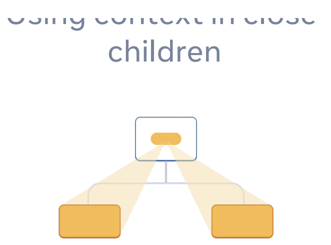
```
<Section level={3}>
  <Heading>About</Heading>
  <Heading>Photos</Heading>
  <Heading>Videos</Heading>
</Section>
```

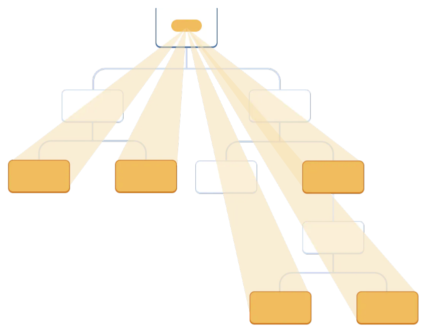
But how can the `<Heading>` component know the level of its closest `<Section>`? **That would require some way for a child to “ask” for data from somewhere above in the tree.**

You can’t do it with props alone. This is where context comes into play. You will do it in three steps:

1. **Create** a context. (You can call it `LevelContext`, since it’s for the heading level.)
2. **Use** that context from the component that needs the data. (`Heading` will use `LevelContext`.)
3. **Provide** that context from the component that specifies the data. (`Section` will provide `LevelContext`.)

Context lets a parent—even a distant one!—provide some data to the entire tree inside of it.





Step 1: Create the context

First, you need to create the context. You'll need to **export it from a file** so that your components can use it:

App.js Section.js Heading.js LevelContext.js Reset

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

export const LevelContext = createContext(1);

```

The only argument to `createContext` is the *default* value. Here, `1` refers to the biggest heading level, but you could pass any kind of value (even an object). You will see the significance of the default value in the next step.

Step 2: Use the context

Import the `useContext` Hook from React and your context:

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';
```

Currently, the `Heading` component reads `level` from props:

```
export default function Heading({ level, children }) {
  // ...
}
```

Instead, remove the `level` prop and read the value from the context you just imported, `LevelContext`:

```
export default function Heading({ children }) {
  const level = useContext(LevelContext);
  // ...
}
```

`useContext` is a Hook. Just like `useState` and `useReducer`, you can only call a Hook immediately inside a React component (not inside loops or conditions). **`useContext` tells React that the `Heading` component wants to read the `LevelContext`.**

Now that the `Heading` component doesn't have a `level` prop, you don't need to pass the level prop to `Heading` in your JSX like this anymore:

```
<Section>
  <Heading level={4}>Sub-sub-heading</Heading>
  <Heading level={4}>Sub-sub-heading</Heading>
  <Heading level={4}>Sub-sub-heading</Heading>
</Section>
```

Update the JSX so that it's the `Section` that receives it instead:

```
<Section level={4}>
  <Heading>Sub-sub-heading</Heading>
  <Heading>Sub-sub-heading</Heading>
  <Heading>Sub-sub-heading</Heading>
</Section>
```

As a reminder, this is the markup that you were trying to get working:

App.js Section.js Heading.js LevelContext.js Reset

```
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
  return (
    <Section level={1}>
    <Section level={1}>
      <Heading>Title</Heading>
      <Section level={2}>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Section level={3}>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Section level={4}>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
          </Section>
        </Section>
      </Section>
    </Section>
  );
}

```

Show less

Notice this example doesn't quite work, yet! All the headings have the same size because **even though you're *using* the context, you have not *provided* it yet.** React doesn't know where to get it!

If you don't provide the context, React will use the default value you've specified in the previous step. In this example, you specified `1` as the argument to `createContext`, so `useContext(LevelContext)` returns `1`, setting all those headings to `<h1>`. Let's fix this problem by having each `Section` provide its own context.

Step 3: Provide the context

The `Section` component currently renders its children:

```
export default function Section({ children }) {
  return (
    <section className="section">
      {children}
    </section>
  );
}
```

Wrap them with a context provider to provide the `LevelContext` to them:

```
import { LevelContext } from './LevelContext.js';

export default function Section({ level, children }) {
  return (
    <section className="section">
      <LevelContext.Provider value={level}>
        {children}
      </LevelContext.Provider>
    </section>
  );
}
```

This tells React: “if any component inside this `<Section>` asks for `LevelContext`, give them this `level`.” The component will use the value of the nearest `<LevelContext.Provider>` in the UI tree above it.

App.js Section.js Heading.js LevelContext.js Reset

```
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
  return (
    <Section level={1}>
      <Heading>Title</Heading>
      <Section level={2}>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Section level={3}>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Section level={4}>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
          </Section>
        </Section>
      </Section>
    </Section>
  );
}

```

Show less

It’s the same result as the original code, but you did not need to pass the `level` prop to each `Heading` component! Instead, it “figures out” its heading level by asking the closest `Section` above:

1. You pass a `level` prop to the `<Section>`.
2. `Section` wraps its children into `<LevelContext.Provider value={level}>`.
3. `Heading` asks the closest value of `LevelContext` above with `useContext(LevelContext)`.

Using and providing context from the same component

Currently, you still have to specify each section’s `level` manually:

```
export default function Page() {
  return (
    <Section level={1}>
```

```
      ...
      <Section level={2}>
        ...
        <Section level={3}>
          ...
```

Since context lets you read information from a component above, each `Section` could read the `level` from the `Section` above, and pass `level + 1` down automatically. Here is how you could do it:

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Section({ children }) {
  const level = useContext(LevelContext);
  return (
    <section className="section">
      <LevelContext.Provider value={level + 1}>
        {children}
      </LevelContext.Provider>
    </section>
  );
}
```

With this change, you don't need to pass the `level` prop *either* to the `<Section>` or to the `<Heading>`:

App.js Section.js Heading.js LevelContext.js Reset

```
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
  return (
    <Section>
```

```
      <Heading>Title</Heading>
      <Section>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Section>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Section>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
          </Section>
        </Section>
      </Section>
    </Section>
  );
}

```

Show less

Now both `Heading` and `Section` read the `LevelContext` to figure out how “deep” they are. And the `Section` wraps its children into the `LevelContext` to specify that anything inside of it is at a “deeper” level.

Note

This example uses heading levels because they show visually how nested components can override context. But context is useful for many other use cases too. You can pass down any information needed by the entire subtree: the current color theme, the currently logged in user, and so on.

Context passes through intermediate components

You can insert as many components as you like between the component that provides context and the one that uses it. This includes both built-in components like `<div>` and components you might build yourself.

In this example, the same `Post` component (with a dashed border) is rendered at two different nesting levels. Notice that the `<Heading>` inside of it gets its level automatically from the closest `<Section>`:

App.js Section.js Heading.js LevelContext.js Reset

```
import Heading from './Heading.js';
import Section from './Section.js';

export default function ProfilePage() {
  return (
    <Section>
    <Section>
      <Heading>My Profile</Heading>
      <Post
        title="Hello traveller!"
        body="Read about my adventures."
      />
      <AllPosts />
    </Section>
  );
}

function AllPosts() {
  return (
    <Section>
      <Heading>Posts</Heading>
      <RecentPosts />
    </Section>
  );
}

function RecentPosts() {
  return (
    <Section>
      <Heading>Recent Posts</Heading>
      <Post
        title="Flavors of Lisbon"
        body="...those pastéis de nata!"
      />
      <Post
        title="Buenos Aires in the rhythm of tango"
        body="I loved it!"
      />
    </Section>
  );
}

function Post({ title, body }) {
  return (
    <Section isFancy={true}>
      <Heading>
        {title}
```

```
          {title}
        </Heading>
        <p><i>{body}</i></p>
      </Section>
  );
}

```

Show less

You didn’t do anything special for this to work. A `Section` specifies the context for the tree inside it, so you can insert a `<Heading>` anywhere, and it will have the correct size. Try it in the sandbox above!

Context lets you write components that “adapt to their surroundings” and display themselves differently depending on *where* (or, in other words, *in which context*) they are being rendered.

How context works might remind you of CSS property inheritance. In CSS, you can specify `color: blue` for a `<div>`, and any DOM node inside of it, no matter how deep, will inherit that color unless some other DOM node in the middle overrides it with `color: green`. Similarly, in React, the only way to override some context coming from above is to wrap children into a context provider with a different value.

In CSS, different properties like `color` and `background-color` don't override each other. You can set all `<div>`'s `color` to red without impacting `background-color`. Similarly, **different React contexts don't override each other.** Each context that you make with `createContext()` is completely separate from other ones, and ties together components using and providing *that particular* context. One component may use or provide many different contexts without a problem.

Before you use context

Context is very tempting to use! However, this also means it's too easy to overuse it. **Just because you need to pass some props several levels deep doesn't mean you should put that information into context.**

Here's a few alternatives you should consider before using context:

1. **Start by passing props.** If your components are not trivial, it's not unusual to pass a dozen props down through a dozen components. It may feel like a slog, but it makes it very clear which components use which data! The person maintaining your code will be glad you've made the data flow explicit with props.
2. **Extract components and pass JSX as `children` to them.** If you pass some data through many layers of intermediate components that don't use that data (and only pass it further down), this often means that you forgot to extract some components along the way. For example, maybe you pass data props like `posts` to visual components that don't use them directly, like `<Layout posts={posts} />`. Instead, make `Layout` take `children` as a prop, and render `<Layout><Posts posts={posts} /></Layout>`. This reduces the number of layers between the component specifying the data and the one that needs it.

If neither of these approaches works well for you, consider context.

Use cases for context

- **Theming:** If your app lets the user change its appearance (e.g. dark mode), you can put a context provider at the top of your app, and use that context in components that need to adjust their visual look.
- **Current account:** Many components might need to know the currently logged in user. Putting it in context makes it convenient to read it anywhere in the tree. Some apps also let you operate multiple accounts at the same time (e.g. to leave a comment as a different user). In those cases, it can be convenient to wrap a part of the UI into a nested provider with a different current account value.
- **Routing:** Most routing solutions use context internally to hold the current route. This is how every link "knows" whether it's active or not. If you build your own router, you might want to do it too.
- **Managing state:** As your app grows, you might end up with a lot of state closer to the top of your app. Many distant components below may want to change it. It is common to use a reducer together with context to manage complex state and pass it down to distant components without too much hassle.

Context is not limited to static values. If you pass a different value on the next render, React will update all the components reading it below! This is why context is often used in combination with state.

In general, if some information is needed by distant components in different parts of the tree, it's a good indication that context will help you.

Recap

- Context lets a component provide some information to the entire tree below it.
- To pass context:

1. Create and export it with `export const MyContext = createContext(defaultValue)`.
2. Pass it to the `useContext(MyContext)` Hook to read it in any child component, no matter how deep.
3. Wrap children into `<MyContext.Provider value={...}>` to provide it from a parent.

- Context passes through any components in the middle.
- Context lets you write components that “adapt to their surroundings”.
- Before you use context, try passing props or passing JSX as `children`.

Try out some challenges

Challenge 1 of 1:

Replace prop drilling with context

In this example, toggling the checkbox changes the `imageSize` prop passed to each `<PlaceImage>`. The checkbox state is held in the top-level `App` component, but each `<PlaceImage>` needs to be aware of it.

Currently, `App` passes `imageSize` to `List`, which passes it to each `Place`, which passes it to the `PlaceImage`. Remove the `imageSize` prop, and instead pass it from the `App` component directly to `PlaceImage`.

You can declare context in `Context.js`.

App.js Context.js data.js utils.js Reset

```
import { useState } from 'react';
import { places } from './data.js';
import { getImageUrl } from './utils.js';
```

```

export default function App() {
  const [isLarge, setIsLarge] = useState(false);
  const imageSize = isLarge ? 150 : 100;
  return (
    <>
      <label>
        <input
          type="checkbox"
          checked={isLarge}
          onChange={e => {
            setIsLarge(e.target.checked);
          }}
        />
        Use large images
      </label>
      <hr />
      <List imageSize={imageSize} />
    </>
  )
}

function List({ imageSize }) {
  const listItems = places.map(place =>
    <li key={place.id}>
      <Place
        place={place}
        imageSize={imageSize}
      />
    </li>
  );
  return <ul>{listItems}</ul>;
}

function Place({ place, imageSize }) {
  return (
    <>
      <PlaceImage
        place={place}
        imageSize={imageSize}
      />
      <p>
        <b>{place.name}</b>
        {': ' + place.description}
      </p>
    </>
  );
}

function PlaceImage({ place, imageSize }) {
  return (
    <img
      src={getImageUrl(place)}
      alt={place.name}
      width={imageSize}
      height={imageSize}
    />
  );
}

```

Show less

Show solution

PREVIOUS

Extracting State Logic into a Reducer

NEXT

Scaling Up with Reducer and Context

How do you like these docs?

Take our survey!

Meta Open Source

©2023

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