



BONUS: React Children

[New Attempt](#)

Due No Due Date **Points** 1 **Submitting** a website url

[FORK](#)

<https://github.com/learn-co-curriculum/react-hooks-children/fork>  <https://github.com/learn-co-curriculum/react-hooks-children>  <https://github.com/learn-co-curriculum/react-hooks-children/issues/new>

Learning Goals

- Use React Children to compose multiple components together
- Access the `children` prop to return a child component

Why do we need children?

In HTML, encapsulating several bits of UI can be as easy as wrapping them in a single element. For example:

```
<div class="container">
  <h1>Hello, I'm in a container!</h1>
  <p>I'm a description!</p>
</div>
```

In the above code, the `h1` and `p` tags are direct children of the `div`, meaning that they are rendered within `div`. In other words, they are **part** of the `div`.

In React, you might create a reusable version of this HTML by doing the following:

```
function Header(props) {
  return (
    <div class="container">
      <h1>{props.header}</h1>
      <p>{props.description}</p>
    </div>
  );
}
```

The `header` and `description` props help us make the Header component reusable, as seen here:

```
ReactDOM.render(
  <div>
    <Header header="Hello, I'm in a container!", description="I'm a description!"
    <Header header="I'm another container", description="Whoa that's weird!" />
    <Header header="A third container!", description="Cray cray" />
  </div>
)
```

```

</div>,
document.getElementById('root')
)

```

This is great when we want UI that has the same structure with different text/attributes. We can even use conditional rendering to help us choose when to render parts of the UI. Not bad! But consider the following HTML:

```

<div class="container">
  <h1>Hello, I'm in a container!</h1>
  <p>I'm a description!</p>
</div>
<div class="container">
  <strong>Image description</strong>
  <div class="image-wrapper">
    
  </div>
</div>
<div class="container">
  <h4>People</h4>
  <ul>
    <li>Evans "Wangtron" Wang</li>
    <li>Andrew "Chrome Boi" Cohn </li>
    <li>Tashawn "Thursdays" Williams</li>
    <li>Alex "Friggin'" Griffith</li>
  </ul>
</div>

```

In this example, we have 3 `div` s each with the same class name, but entirely different children and internal structure. `props` won't help us here: each `div` has such radically different content. From what we know of React, we would be forced to write 3 different components, each with the same wrapping `div` but entirely different contents. Wouldn't it be nice if you could write one component that can keep much of the same *external* structure but render different components *internally*? Enter the `children` prop.

How do we make children?

So far you've seen components rendered like this using the **self-closing tag** syntax:

```

function Example(props) {
  return <div>{props.exampleProp}</div>;
}

<Example exampleProp="example value" />;

```

However, React also allows you to use your components with an **opening and closing tag**, like most HTML elements:

```
<Example exampleProp="example value">
  <h1>Example header!</h1>
  <p>Some example text</p>
</Example>
```

If you were to use the above definition of the `Example` component, you would observe no difference at all: the `h1` and `p` would not be rendered. However, if you inspect the props in `Example`, you'll notice a new prop has been added: `children`.

A closer look at this prop reveals that it contains an array, and that each element of the array is a component! In this case, you'll see an `h1` and a `p` tag, in that order. Rendering these children is the same as rendering any array of components:

```
function Example(props) {
  return (
    <div>
      {props.exampleProp}
      /* using the children prop to render any elements inside the opening and cl
      {props.children}
    </div>
  );
}
```

And voila! You have a component that is able to render its children! Any valid JSX elements, including your own components and nested JSX elements, can be used as children.

An example of working children

To run our example, run `npm install && npm start`.

You'll notice that our `App` component renders two `Container` components, each with distinct children. Take a look at the code for `Container`: spend some time figuring out what each prop does. (Note: we're using default values for our destructured props in case values are not passed in by the parent component.) You'll notice there are 5 props: `direction`, `header`, `textPosition`, `contentPosition`, and `children`. We've already discussed `children`, but try to figure out what the others do!

Conclusion

Using React Children greatly expands your ability to make reusable components. While you won't see as many examples of this style of writing components when you're first getting started with React, this pattern is an incredibly useful one to master. In addition to helping with component reusability, it can also help with passing props down multiple levels of the component hierarchy.

Resources

- [Composition vs Inheritance](https://reactjs.org/docs/composition-vs-inheritance.html) [_\(https://reactjs.org/docs/composition-vs-inheritance.html\)](https://reactjs.org/docs/composition-vs-inheritance.html)