**Render Props Pattern**

A render prop is simply a function prop that is called in a render method. To truly understand what this means, let’s learn why such a pattern exists in the first place, then explore the various implementations.

React has made building interactive applications a lot more manageable. We’re able to break up our app into components, each with their own encapsulated logic and styling. It’s great — it means we’re able to work on lots of smaller problems instead of one really large one.

This method of building applications works really well because of abstractions. A component in an application is just an abstraction of logic, data, and, in some cases, styling.

const Button = () => (...)

The Button component above is an abstraction of the logic and styles required to make a button show up on a page. Now, every time we want a button, we simply use this abstraction — no need to reinvent the wheel.

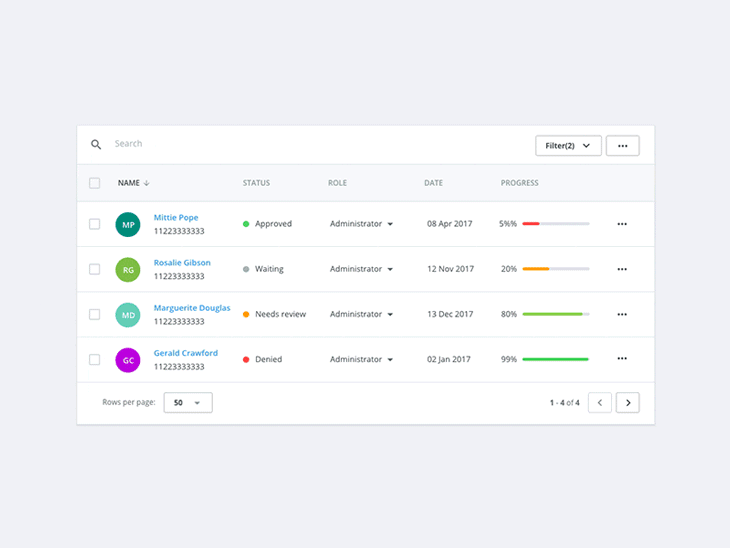
When we build apps this way, a new challenge arises: sometimes we want to share logic or data between components. We also want to do this in a way that is clean and composable. Just as we’re able to abstract logic and data into components, we also want to be able to compose these components to form larger building blocks.

Four paragraphs in and we still haven’t gotten to [render props](https://reactjs.org/docs/render-props.html) yet.

***Put simply, a render prop is a function prop that is called in a render method***. But to truly understand what this means, we need to know why such a pattern exists in the first place.

**Why do we need render props?**

Let’s take a legitimate problem we might face in the real world. Say we wanted to have a UI element that we could dismiss with an animation.

Delete from table by [Alex Lauderdale](https://dribbble.com/shots/4823330-Interaction-foundation-Delete-from-table).

What we want here is a component that can render anything, but we want to keep this dismissal animation. We could simply use the children prop and pass whatever we wanted to render as a child.

const Dismiss = ({ children }) => {

return (

<Fragment>

{children}

</Fragment>

)

}

That would work, but it’s limited.

* How would we fire the dismiss event from within the component being rendered? If we decide to fire it from the Dismiss component, that means we’ll have to hardcode some HTML.
* What if we wanted to set up different actions to fire on the dismiss event?
* Is there any way to pass data as props to the children being rendered?

In situations like this, we need something a bit more flexible than the children prop method. What we really want is to encapsulate the dismissal functionality so that we can share it between components.

With render props, we could render anything we wanted, just like the children prop method. But we will also be able to pass props to whatever is being rendered.

***This is the main purpose of render props — being able to communicate with any rendered component without having to couple implementation details***. This is possible because the render function now comes in as a prop, and you can pass data to it as arguments when you call it.

This will become clearer when we take a look at a practical use case for render props (based on [WAVE’s modal component](https://wave.free-now.com/components/Modal)). We’ll look at a possible solution to the problem we raised initially using the children prop method. Then, we’ll look at a more flexible solution with render props.

##### ***Now that you know the purpose of props, see how devs use them for modern reusable components.***

## Implementing render props

const Dismiss = ({ children }) => {

const dismiss = () => {

...code to implement dismissal animations etc

}

return (

<div>

{children}

<button onClick={dismiss}>dismiss</button>

</div>

)

}

const DismissableContent = () => {

return (

<Dismiss>

...code

</Dismiss>

)

}

Above, we have a Dismiss component that renders whatever is passed as the children prop but also contains code to implement the dismiss animation. Below it is how we’d use such a component. Notice that with this implementation:

* You can only fire the dismiss event from within the Dismiss component
* You can’t pass data to the children being rendered
* You have to account for the extra div in your layouts

You can see some of the limitations in sharing functionality this way. Now let’s look at the same solution, but with render props:

const Dismiss = (props) => {

const dismiss = () => {

...code to implement dismissal animations etc

}

return props.render(dismiss)

}

const DismissableContent = () => {

return (

<Dismiss render={

dismiss => <Content dismiss={dismiss} />

} />

)

}

Apart from the code being smaller, there are some more important advantages to this method.

We have moved the content of the render function from being hardcoded in the Dismiss component to accepting it from the props. By simply calling props.render(), the component is able to render whatever is returned from the render prop. This comes with some implications:

* Since the render prop is a function, we can pass data to it as arguments. This means that Dismiss can communicate with the component being rendered and vice versa
* You don’t have to account for any extra divs from Dismiss because it can be pure and not return its own HTML (most times, though, you’d return some accompanying HTML and styling)

We can now see how this solves the issues we had using the children prop method. This is much cleaner and more flexible by every metric.

## Implementing other props

Another advantage of render props is that the prop doesn’t have to be called render. You could call it whatever you want as long as the prop is invoked in the render method. This leads to interesting implementations, like this:

const Dismiss = ({ children }) => {

const dismiss = () => {

...code to implement dismissal animations etc

}

return children(dismiss)

}

const DismissableContent = () => {

return (

<Dismiss>

{(dismiss) => (

<Content dismiss={dismiss} />

)}

</Dismiss>

)

}