Render Props

The term "render prop" refers to a technique for sharing code between React components using a prop whose value is a function.

A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)}/>
```

Libraries that use render props include React Router, Downshift and Formik.

In this document, we'll discuss why render props are useful, and how to write your own.

Use Render Props for Cross-Cutting Concerns

Components are the primary unit of code reuse in React, but it's not always obvious how to share the state or behavior that one component encapsulates to other components that need that same state.

For example, the following component tracks the mouse position in a web app:

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        <h1>Move the mouse around!</h1>
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}
```

As the cursor moves around the screen, the component displays its (x, y) coordinates in a `<p>`.

Now the question is: How can we reuse this behavior in another component? In other words, if another component needs to know about the cursor position, can we encapsulate that behavior so that we can easily share it with that component?

Since components are the basic unit of code reuse in React, let’s try refactoring the code a bit to use a `<Mouse>` component that encapsulates the behavior we need to reuse elsewhere.

```
// The <Mouse> component encapsulates the behavior we need...
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>

        {/* ...but how do we render something other than a <p>? */}
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}
```

```
}

class MouseTracker extends React.Component {
  render() {
    return (
      <>
        <h1>Move the mouse around!</h1>
        <Mouse />
      </>
    );
  }
}
```

Now the `<Mouse>` component encapsulates all behavior associated with listening for `mousemove` events and storing the (x, y) position of the cursor, but it's not yet truly reusable.

For example, let's say we have a `<Cat>` component that renders the image of a cat chasing the mouse around the screen. We might use a `<Cat mouse={{ x, y }}>` prop to tell the component the coordinates of the mouse so it knows where to position the image on the screen.

As a first pass, you might try rendering the `<Cat>` *inside <Mouse>'s render method*, like this:

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      <img src="/cat.jpg" style={{ position: 'absolute', left: mouse.x, top: mouse.y
}} />
    );
  }
}

class MouseWithCat extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }
  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>

        {/*
          We could just swap out the <p> for a <Cat> here ... but then
          we would need to create a separate <MouseWithSomethingElse>
          component every time we need to use it, so <MouseWithCat>
          isn't really reusable yet.
        */}
        <Cat mouse={this.state} />
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <MouseWithCat />
      </div>
    );
  }
}
```

This approach will work for our specific use case, but we haven't achieved the objective of truly encapsulating the behavior in a reusable way. Now, every time we want the mouse position for a different use case, we have to create a new component (i.e. essentially another `<MouseWithCat>`) that renders something specifically for that use case.

Here's where the render prop comes in: Instead of hard-coding a `<Cat>` inside a `<Mouse>` component, and effectively changing its rendered output, we can provide `<Mouse>` with a function prop that it uses to dynamically determine what to render–a render prop.

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      <img src="/cat.jpg" style={{ position: 'absolute', left: mouse.x, top: mouse.y
}} />
    );
  }
}

class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>

        {/*
          Instead of providing a static representation of what <Mouse> renders,
          use the `render` prop to dynamically determine what to render.
        */}
        {this.props.render(this.state)}
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}/>
      </div>
    );
  }
}
```

Now, instead of effectively cloning the `<Mouse>` component and hard-coding something else in its `render` method to solve for a specific use case, we provide a `render` prop that `<Mouse>` can use to dynamically determine what it renders.

More concretely, **a render prop is a function prop that a component uses to know what to render.**

This technique makes the behavior that we need to share extremely portable. To get that behavior, render a `<Mouse>` with a `render` prop that tells it what to render with the current (x, y) of the cursor.

One interesting thing to note about render props is that you can implement most higher-order components (HOC) using a regular component with a render prop. For example, if you would prefer to have a `withMouse` HOC instead of a `<Mouse>` component, you could easily create one using a regular `<Mouse>` with a render prop:

```
// If you really want a HOC for some reason, you can easily
// create one using a regular component with a render prop!
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}/>
      );
    }
  }
}
```

So using a render prop makes it possible to use either pattern.

Using Props Other Than render

It's important to remember that just because the pattern is called "render props" you don't *have to use a prop named render to use this pattern*. In fact, *any* prop that is a function that a component uses to know what to render is technically a "render prop".

Although the examples above use `render`, we could just as easily use the `children` prop!

```
<Mouse children={mouse => (
  <p>The mouse position is {mouse.x}, {mouse.y}</p>
)}/>
```

And remember, the `children` prop doesn't actually need to be named in the list of "attributes" in your JSX element. Instead, you can put it directly *inside* the element!

```
<Mouse>
  {mouse => (
    <p>The mouse position is {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

You'll see this technique used in the react-motion API.

Since this technique is a little unusual, you'll probably want to explicitly state that `children` should be a function in your `propTypes` when designing an API like this.

```
Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

Caveats

Be careful when using Render Props with React.PureComponent

Using a render prop can negate the advantage that comes from using `React.PureComponent` if you create the function inside a `render` method. This is because the shallow prop comparison will always return `false` for new props, and each `render` in this case will generate a new value for the render prop.

For example, continuing with our `<Mouse>` component from above, if `Mouse` were to extend `React.PureComponent` instead of `React.Component`, our example would look like this:

```
class Mouse extends React.PureComponent {
  // Same implementation as above...
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
```

```
          {/*
            This is bad! The value of the `render` prop will
            be different on each render.
          */}
          <Mouse render={mouse => (
            <Cat mouse={mouse} />
          )}/>
        </div>
      );
    }
  }
```

In this example, each time `<MouseTracker>` renders, it generates a new function as the value of the `<Mouse render>` prop, thus negating the effect of `<Mouse>` extending `React.PureComponent` in the first place!

To get around this problem, you can sometimes define the prop as an instance method, like so:

```
class MouseTracker extends React.Component {
  // Defined as an instance method, `this.renderTheCat` always
  // refers to *same* function when we use it in render
  renderTheCat(mouse) {
    return <Cat mouse={mouse} />;
  }

  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={this.renderTheCat} />
      </div>
    );
  }
}
```

In cases where you cannot define the prop statically (e.g. because you need to close over the component's props and/or state) `<Mouse>` should extend `React.Component` instead.

Is this page useful? Edit this page