LEARNING REACT

A Hands-On Guide to Building Web Applications Using React and Redux

SECOND EDITION

KIRUPA CHINNATHAMBI

Learning React

Second Edition

```
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
  }
}

class Label extends React.Component {
  render() {
    return (
      <Display {...this.props} />
    );
  }
}

class Shirt extends React.Component {
  render() {
    return (
      <div>
        <Label {...this.props} />
      </div>
    );
  }
}
```

If you run this code, the end result is unchanged from what we had earlier. The biggest difference is that we are no longer passing in expanded forms of each property as part of calling each component. This solves all the problems we originally set out to solve.

By using the spread operator, if you ever decide to add properties, rename properties, remove properties, or do any other sort of property-related shenanigans, you don't have to make a billion different changes. You make one change at the spot you define your property. You make another change at the spot you consume the property. That's it. All the intermediate components that merely transfer the properties remain untouched because the `{...this.props}` expression contains no details of what goes on inside it.

> **Is this the best way to transfer properties?**
>
> Using the spread operator to transfer properties is convenient, and it's a marked improvement over explicitly defining each property at each component as we were originally doing. The thing is, even the spread operator approach isn't a perfect solution. If all you want to do is transfer a property to a particular component, having each intermediate component play a role in passing it on is unnecessary. Worse, it has the potential to be a performance bottleneck. Any change to a property that you are passing along will trigger a component update on each component along the property's path. That's not a good thing! Later, we look at ways to solve this transferring properties problem in a much better, without any side effects.

Conclusion

As created by the ES6/ES2015 committee, the spread operator is designed to work only on arrays and arraylike creatures (a.k.a. something that has a `Symbol.iterator` property). The fact that it works on object literals such as our `props` object is a result of React extending the standard. No browser currently supports using the spread object on object literals. Our example works because of Babel. Besides turning all our JSX into something our browser understands, Babel turns cutting-edge and experimental features into something that's friendly across browsers. That's why we're able to get away with using the spread operator on an object literal, and that's why we're able to elegantly solve the problem of transferring properties across multiple layers of components.

Now, does any of this matter? Is it really critical that you know about the nuances of the spread operator and how it works in certain situations and doesn't work in others? For the most part, no. The important part to realize is that you can use the spread operator to transfer `props` from one component to another. The other important part to realize is that we will look at some other ways in the future to make transferring properties equally simple, without running into any performance issues.

> **Note: If you run into any issues, ask!**
>
> If you have any questions or your code isn't running like you expect, don't hesitate to ask! Post on the forums at https://forum.kirupa.com and get help from some of the friendliest and most knowledgeable people the Internet has ever brought together!

This page intentionally left blank

7

Meet JSX…Again!

As you've probably noticed by now, we've been using a lot of JSX. But we really haven't taken a good look at what JSX actually is. How does it work? Why don't we just call it HTML? What quirks does it have up its sleeve? In this chapter, we answer all those questions and more! We do some serious backtracking (and some forwardtracking) to see what we need to know about JSX in order to be dangerous.

What Happens with JSX?

One of the biggest things we've glossed over is trying to figure out what happens with our JSX after we've written it. How does it end up as the HTML that you see in the browser? Take a look at the following example, where we define a component called `Card`:

```
class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      boxShadow: "0px 0px 5px #666"
    };

    return (
      <div style={cardStyle}>
        <Square color={this.props.color} />
        <Label color={this.props.color} />
      </div>
    );
  }
}
```

We can quickly spot the JSX here. It's the following four lines:

```
<div style={cardStyle}>
  <Square color={this.props.color} />
  <Label color={this.props.color} />
</div>
```

Keep in mind that browsers have no idea what to do with JSX. They probably think you're crazy if you even try to describe JSX to them. That's why we've been relying on things like Babel to turn that JSX into something the browsers understand: JavaScript.

This means that the JSX we write is for human (and well-trained cat) eyes only. When this JSX reaches our browser, it ends up getting turned into pure JavaScript:

```
return React.createElement(
  "div",
  { style: cardStyle },
  React.createElement(Square, { color: this.props.color }),
  React.createElement(Label, { color: this.props.color })
);
```

All of those neatly nested HTML-like elements, their attributes, and their children get turned into a series of `createElement` calls with default initialization values. Here's what our entire Card component looks like when it gets turned into JavaScript:

```
class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      boxShadow: "0px 0px 5px #666"
    };

    return React.createElement(
      "div",
      { style: cardStyle },
      React.createElement(Square, { color: this.props.color }),
      React.createElement(Label, { color: this.props.color })
    );
  }
}
```

Notice that there's no trace of JSX anywhere! All these changes between what we wrote and what our browser sees are part of the transpiling step we talked about in Chapter 1, "Introducing React." That transpilation happens entirely behind the scenes, thanks to Babel,

which we've been using to perform this JSX-to-JS transformation entirely in the browser. We'll eventually look at using Babel as part of a more involved build environment in which we generate a transformed JS file, but you'll see more on that when we get there in the future.

So there you have it, an answer to what exactly happens to all our JSX: It gets turned into *sweet* JavaScript.

JSX Quirks to Remember

As we've been working with JSX, you've probably noticed that we've run into some arbitrary rules and exceptions on what we can and can't do. In this section, let's look at those quirks…and some brand new ones!

Evaluating Expressions

JSX is treated like JavaScript. As you've seen a few times already, this means that you aren't limited to dealing with static content like the following:

```
class Stuff extends React.Component {
  render() {
    return (
      <h1>Boring static content!</h1>
    );
  }
};
```

The values you return can be dynamically generated. All you have to do is wrap your expression in curly braces:

```
class Stuff extends React.Component {
  render() {
    return (
      <h1>Boring {Math.random() * 100} content!</h1>
    );
  }
}
```

Notice that we're throwing in a `Math.random()` call to generate a random number. It gets evaluated along with the static text alongside it, but because of the curly braces, what you see looks something like the following: *Boring 28.6388820148227 content!*

These curly braces allow your app to first evaluate the expression and then return the result of the evaluation. Without them, you would see your expression returned as text: *Boring Math.random() * 100 content!*

That isn't what you would probably want.