9

Going from Data to UI in React

When you're building your apps, thinking in terms of props, state, components, JSX tags, `render` methods, and other React-isms might be the last thing on your mind. Most of the time, you're dealing with data in the form of JSON objects, arrays, and other data structures that have no knowledge (or interest) in React or anything visual. Bridging the gulf between your data and what you eventually see can be frustrating! Not to worry, though. This chapter helps reduce some of those frustrating moments by running through some common scenarios you'll encounter.

THE EXAMPLE

To help make sense of everything you're about to see, we need an example. It's nothing too complicated, so go ahead and create a new HTML document and throw the following stuff into it:

Click here to view code image

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>From Data to UI</title>
  <script src="https://unpkg.com/react@16
/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0
/babel.min.js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>
<body>
  <div id="container"></div>

  <script type="text/babel">
    class Circle extends React.Component {
      render() {
        var circleStyle = {
          padding: 10,
          margin: 20,
          display: "inline-block",
          backgroundColor: this.props.bgColor,
          borderRadius: "50%",
          width: 100,
          height: 100,
        };

        return (
          <div style={circleStyle}>
          </div>
        );
      }
    }

    ReactDOM.render(
      <div>
        <Circle bgColor="#F9C240" />
      </div>,
      document.querySelector("#container")
    );
  </script>
</body>

</html>
```

When you have your document set up, go ahead and preview what you have in your browser. If everything went well, you'll be greeted by a happy yellow circle (see Figure 9.1).

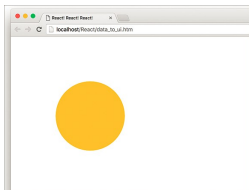

Figure 9.1 If everything went well, you'll get this yellow circle.

If you see what I see, great! Now, let's take a moment to understand what this example is doing. The bulk of what you see comes from the `Circle` component:

Click here to view code image

```
class Circle extends React.Component {
  render() {
    var circleStyle = {
      padding: 10,
      margin: 20,
      display: "inline-block",
      backgroundColor: this.props.bgColor,
      borderRadius: "50%",
      width: 100,
      height: 100,
    };

    return (
      <div style={circleStyle}>
      </div>
    );
  }
}
```

It's mostly made up of our `circleStyle` object that contains the inline style properties that turn our boring `div` into an awesome circle. All the style values are hard-coded except for the `backgroundColor` property, which takes its value from the `bgColor` prop that gets passed in.

Going beyond our component, we ultimately display our circle via our usual `ReactDOM.render` method:

Click here to view code image

```
ReactDOM.render(
  <div>
    <Circle bgColor="#F9C240"/>
  </div>,
  destination
);
```

We have a single instance of our Circle component declared, and we declare it with the `bgColor` prop set to the color we want our circle to appear. Now, having our Circle component be defined as is inside our `render` method is a bit limiting, especially if we're going to be dealing with data that could affect what our Circle component does. In the next couple sections, we'll look at the ways we have for solving that.

YOUR JSX CAN BE ANYWHERE, PART II

In Chapter 7, "Meet JSX...Again", you learned that JSX can actually live outside a `render` function and can be used as a value assigned to a variable or property. For example, we can fearlessly do something like this:

Click here to view code image

```
var theCircle = <Circle bgColor="#F9C240" />;

ReactDOM.render(
  <div>
    {theCircle}
  </div>,
  destination
);
```

The `theCircle` variable stores the JSX for instantiating our Circle component. Evaluating this variable inside our `ReactDOM.render` function results in a circle getting displayed. The end result is no different than what we had earlier, but freeing our Circle component instantiation from the shackles of the `render` method gives us more options to do crazy and cool things.

For example, you can go further and create a function that returns a Circle component:

Click here to view code image

```
function showCircle() {
  var colors = ["#393E41", "#E94F37", "#1C89BF", "#A1D363"];
  var ran = Math.floor(Math.random() * colors.length);

  // return a Circle with a randomly chosen color
  return <Circle bgColor={colors[ran]} />;
}
```

In this case, the `showCircle` function returns a Circle component (boring!) with the value for the `bgColor` prop set to a random color value (awesome sauce!). To have our example use the `showCircle` function, all you have to do is evaluate it inside `ReactDOM.render`:

Click here to view code image

```
ReactDOM.render(
  <div>
    {showCircle()}
  </div>,
  destination
);
```

As long as the expression you're evaluating returns JSX, you can put pretty much anything you want inside the { and } brackets. That flexibility is really nice because you can do *a lot* when your JavaScript lives outside the `render` function.

DEALING WITH ARRAYS

Now we get to some fun stuff! When you're displaying multiple components, you can't always manually specify them:

Click here to view code image

```
ReactDOM.render(
  <div>
    {showCircle()}
    {showCircle()}
    {showCircle()}
  </div>,
  destination
);
```

In many real-world scenarios, the number of components you display is related to the number of items in an array or arraylike (a.k.a. iterator) object you're working with. That brings up a few simple complications. For example, let's say that we have an array called `colors` that looks as follows:

Click here to view code image

```
var colors = ["#393E41", "#E94F37", "#1C89BF", "#A1D363",
              "#85FFC7", "#297373", "#FF8552", "#A40E4C"];
```

We want to create a Circle component for each item in this array (and set the `bgColor` prop to the value of each array item). We can do this by creating an array of Circle components:

Click here to view code image

```
var colors = ["#393E41", "#E94F37", "#1C89BF", "#A1D363",
              "#85FFC7", "#297373", "#FF8552", "#A40E4C"];

var renderData = [];

for (var i = 0; i < colors.length; i++) {
  renderData.push(<Circle bgColor={colors[i]} />);
}
```

In this snippet, we populate our `renderData` array with Circle components just as we originally set out to do. So far, so good. React makes displaying all of these components very simple. Take a look at the highlighted line for all you have to do:

Click here to view code image

```
ReactDOM.render(
  <div>
    {renderData}
  </div>,
  destination
);
```

In our `render` method, all we do is specify our `renderData` array as an expression that we need to evaluate. We don't need to take any other step to go from an array of components to something that looks like Figure 9.2 when you preview in your browser.

Figure 9.2 What you should see in your browser.

Okay, I lied. There's actually one more thing we need to do, and it's a subtle one. React makes UI updates really fast by having a good idea of what exactly is going on in your DOM. It does this in several ways, but one really noticeable way is by internally marking each element with some sort of an identifier.

When you create elements dynamically (such as what we're doing with our array of Circle components), these identifiers are not automatically set. We need to do some extra work. That extra work takes the form of a key prop whose value React uses to uniquely identify each particular component.

For our example, we can do something like this:

Click here to view code image

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];
  renderData.push(<Circle key={i + color} bgColor=
{color} />);
}
```

On each component, we specify our key prop and set its value to a combination of color and index position inside the colors array. This ensures

that each component we dynamically create ends up getting a unique identifier that React can then use to optimize any future UI updates.

> **Check Your Console, Yo!**
>
> React is really good at telling you when you might be doing something wrong. For example, if you dynamically create elements or components and don't specify a key prop on them, you'll be greeted with the following warning in your console:
>
> **Click here to view code image**
>
> ```
> Warning: Each child in an array or iterator should have a unique "key" prop.
> Check the top-level render call using .
> ```
>
> When you're working with React, it's a good idea to periodically check your console for any messages. Even if things seem to be working just fine, you never know what you might find.

CONCLUSION

All the tips and tricks you've seen in this article are made possible because of one thing: *JSX is JavaScript*. This is what allows you to have your JSX live wherever JavaScript thrives. To us, it looks like we're doing something absolutely bizarre when we specify something like this:

Click here to view code image

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];
  renderData.push(<Circle key={i + color} bgColor=
{color} />);
}
```

Even though we're pushing pieces of JSX to an array, just like magic, everything works in the end when `renderData` is evaluated inside our `render` method. I hate to sound like a broken record, but this is because what our browser ultimately sees looks like this:

Click here to view code image

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];

  renderData.push(React.createElement(Circle,
    {
      key: i + color,
      bgColor: color
    }));
}
```

When our JSX gets converted into pure JS, everything makes sense again. This is what allows us to get away with putting our JSX in all sorts of uncomfortable (yet photogenic!) situations and still get the end result we want. In the end, it's all just JavaScript.

> **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!