Module 8

Dealing with State

8. Dealing with State

Up until this point, the components we've created have been stateless. They have properties (aka props) that are passed in from their parent, but nothing (usually) changes about them once the components come alive. Your properties are considered immutable once they have been set. For many interactive scenarios, you don't want that. You want to be able to change aspects of your components as a result of some user interaction (or some data getting returned from a server or a billion other things!)

What we need is another way to store data on a component that goes beyond properties. We need a way to store data that can be changed. What we need is something known as **state**! In this chapter, you learn all about it and how you can use it to create stateful components.

Using State

If you know how to work with properties, you totally know how to work with states... sort of. There are some differences, but they are too subtle to bore you with right now. Instead, let's just jump right in and see states in action by using them in a small example.

What we are going to is create a simple lightning counter example as shown in Figure 8-1.

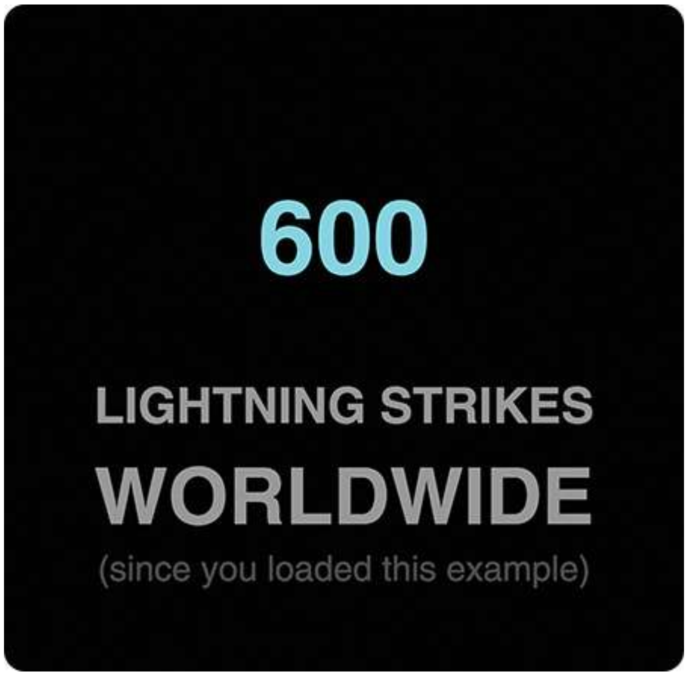

Figure 8-1 The app you will be building.

What this example does is nothing crazy. Lightning strikes the earth's surface about 100 times a second (http://environment.nationalgeographic.com/environment/natural-disasters/lightning-profile/). We have a counter that simply increments a number you see by that same amount. Let's create it.

Our Starting Point

The primary focus of this example is to see how we can work with state. There is no point in us spending a bunch of time creating the example from scratch and retracing paths that we've walked many times already. That's not the best use of anybody's time.
Instead of starting from scratch, modify an existing HTML document or create a new one with the following contents:

```
<!DOCTYPE html>
<html>

<head>
  <title>More State!</title>
  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
</head>

<body>
  <div id="container"></div>
  <script type="text/babel">
    var LightningCounter = React.createClass({
      render: function() {
        return (
          <h1>Hello!</h1>
        );
      }
    });

    var LightningCounterDisplay = React.createClass({
        render: function() {

          var divStyle = {
            width: 250,
            textAlign: "center",
            backgroundColor: "black",
            padding: 40,
            fontFamily: "sans-serif",
            color: "#999",
            borderRadius: 10
          };

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

    ReactDOM.render(
      <LightningCounterDisplay/>,
      document.querySelector("#container")
    );
  </script>
</body>

</html>
```

At this point, take a few moments to look at what our existing code does. First, we have a component called LightningCounterDisplay:

```
var LightningCounterDisplay = React.createClass({
    render: function() {

      var divStyle = {
        width: 250,
        textAlign: "center",
        backgroundColor: "black",
        padding: 40,
        fontFamily: "sans-serif",
        color: "#999",
        borderRadius: 10
      };

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

The bulk of this component is the divStyle object that contains the styling information responsible for the cool rounded background. The render function returns a div element that wraps the LightningCounter component.
The LightningCounter component is where all the action is going to be taking place:

```
var LightningCounter = React.createClass({
  render: function() {
    return (
      <h1>Hello!</h1>
    );
  }
});
```

This component, as it is right now, has nothing interesting going for it. It just returns the word Hello! That's OK—we'll fix this component up later.
The last thing to look at is our ReactDOM.render method:

```
ReactDOM.render(
  <LightningCounterDisplay/>,
  document.querySelector("#container")
);
```

It just pushes the LightningCounterDisplay component into our container div element in our DOM. That’s pretty much it. The end result is that you see the combination of markup from our ReactDOM.render method and the LightningCounterDisplay and LightningCounter components.

Getting Our Counter On

Now that we have an idea of what we are starting with, it's time to make plans for our next steps. The way our counter works is pretty simple. We are going to be using a setInterval function that calls some code every 1000 milliseconds (aka 1 second). That "some code" is going to increment a value by 100 each time it is called. Seems pretty straightforward, right?

To make this all work, we are going to be relying on three APIs that our React Component exposes:

- **getInitialState**—This method runs just before your component gets mounted, and it allows you to modify a component's state object.
- **componentDidMount**—This method gets called just after our component gets rendered (or mounted as React calls it).
- **setState**—This method allows you to update the value of the state object.

We'll see these APIs in use shortly, but I wanted to give you a preview of them so that you can spot them easily in a lineup!

Setting the Initial State Value

We need a variable to act as our counter, and let's call this variable strikes. There are a bunch of ways to create this variable. The most obvious one is the following:

```
var strikes = 0
```

We don't want to do that, though. For our example, the strikes variable is part of our component's state, and its value is what we display on screen. What we are going to do is use the getInitialState method that we briefly saw a few moments ago and take care of initializing our variable inside it. You'll see in a few moments what result that has on our component's state. Inside your LightningCounter component, add the following highlighted lines:

```
var LightningCounter = React.createClass({
  getInitialState: function() {
    return {
      strikes: 0
    };
  },
  render: function() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
});
```

The getInitialState method automatically runs waaaay before your component gets rendered, and what we are doing is telling React to return an object containing our strikes property (initialized to 0). You may be wondering to whom or what we are returning this object to? All of that is magic that happens under the covers. The object that gets returned is set as the initial value for our component's state object.

If we inspect the value of our state object after this code has run, it would look something like the following:

```
var state = {
  strikes: 0
}
```

Before we wrap this section up, let's visualize our strikes property. In our render method, make the following highlighted change:

```
var LightningCounter = React.createClass({
  getInitialState: function() {
    return {
      strikes: 0
    };
  },
  render: function() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
});
```

What we've done is replaced our default Hello! text with an expression that displays the value stored by the this.state.strikes property. If you preview your example in the browser, you will see a value of 0 displayed. That's a start!

Starting Our Timer and Setting State

Next up is getting our timer going and incrementing our strikes property. Like we mentioned earlier, we will be using the setInterval function to increase the strikes property by 100 every second. We are going to do all of this immediately after our component has been rendered using the **built-in componentDidMount** method.
The code for kicking off our timer looks as follows:

```
var LightningCounter = React.createClass({
  getInitialState: function() {
    return {
      strikes: 0
    };
  },
  componentDidMount: function() {
    setInterval(this.timerTick, 1000);
  },
  render: function() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
});
```

Go ahead and add these highlighted lines to our example. Inside our componentDidMount method that gets called once, our component gets rendered, we have our setInterval method that calls a timerTick function every second (or 1000 milliseconds).

We haven't defined our timerTick function, so let's fix that by adding the following highlighted lines to our code:

```
var LightningCounter = React.createClass({
  getInitialState: function() {
    return {
      strikes: 0
    };
  },
  timerTick: function() {
    this.setState({
      strikes: this.state.strikes + 100
    });
  },
  componentDidMount: function() {
    setInterval(this.timerTick, 1000);
  },
  render: function() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
});
```

What our timerTick function does is pretty simple. It just calls setState. The setState method comes in various flavors, but for what we are doing here, it just takes an object as its argument. This object contains all the properties you want to merge into the state object. In our case, we are specifying the strikes property and setting its value to be 100 more than what it is currently.

> How does timerTick maintain context?
> In regular JavaScript, the timerTick function won't maintain context. You have to do extra work to support that. The reason it works in the React world is because of something known as **autobinding**. Now, aren't you glad you know that?

Rendering the State Change

If you preview your app now, you'll see our strikes value start to increment by 100 every second (see Figure 8-2).

Figure 8-2 The strikes value increments by 100 every second.

Let's ignore for a moment what happens with our code. That is pretty straightforward. The interesting thing is how everything we've done ends up updating what you see on the screen. That updating has to do with this React behavior: Whenever you call setState and update something in the state object, your component's render method gets automatically called. This kicks off a cascade of render calls for any component whose output is also affected. The end result of all this is that what you see in your screen in the latest representation of your app's UI state. **Keeping your data and UI in sync is one of the hardest problems with UI development, so it's nice that React takes care of this for us**. It makes all of this pain of learning to use React totally worth it!

Optional: The Full Code

What we have right now is just a counter that increments by 100 every second. Nothing about it screams Lightning Counter, but it does cover everything about states that I wanted you to learn right now. If you want to optionally flesh out your example to look like my version that you saw at the beginning, below is the full code for what goes inside our script tag:

```
var LightningCounter = React.createClass({
  getInitialState: function() {
    return {
      strikes: 0
    };
  },
  timerTick: function() {
    this.setState({
      strikes: this.state.strikes + 100
    });
  },
  componentDidMount: function() {
    setInterval(this.timerTick, 1000);
  },
  render: function() {
    var counterStyle = {
      color: "#66FFFF",
      fontSize: 50
    };

    var count = this.state.strikes.toLocaleString();

    return (
      <h1 style={counterStyle}>{count}</h1>
    );
  }
});

var LightningCounterDisplay = React.createClass({
    render: function() {
      var commonStyle = {
        margin: 0,
        padding: 0
      }
      var divStyle = {
        width: 250,
        textAlign: "center",
        backgroundColor: "#020202",
        padding: 40,
        fontFamily: "sans-serif",
        color: "#999999",
        borderRadius: 10
      };

      var textStyles = {
        emphasis: {
          fontSize: 38,
          ...commonStyle
        },
        smallEmphasis: {
          ...commonStyle
        },
        small: {
          fontSize: 17,
          opacity: 0.5,
          ...commonStyle
        }
      }

      return(
        <div style={divStyle}>
          <LightningCounter/>
          <h2 style={textStyles.smallEmphasis}>LIGHTNING STRIKES</h2>
          <h2 style={textStyles.emphasis}>WORLDWIDE</h2>
          <p style={textStyles.small}>(since you loaded this example)</p>
        </div>
      );
    }
});

ReactDOM.render(
  <LightningCounterDisplay/>,
  document.querySelector("#container")
);
```

If you make your code look like everything you see above and run the example again, you will see our lightning counter example in all its cyan-colored glory. While you are at it, take a moment to look through the code to ensure you don’t see too many surprises.

Conclusion

We just scratched the surface on what we can do to create stateful components. While using a timer to update something in our state object is cool, the real action happens when we start combining user interaction with state. So far, we’ve shied away from the large amount of mouse, touch, keyboard, and other related things that your components will come into contact with. In an upcoming chapter, we are going to fix that. Along the way, you’ll see us taking what we’ve seen about states to a whole new level! If that doesn’t excite you, then I don’t know what will