Module 10

Working with Events

10. Working with Events

So far, most of our examples only did their work on page load. As you probably guessed, that isn't normal. In most apps, especially the kind of UI-heavy ones we will be building, there is going to be a ton of things the app does only as a reaction to something. That something could be triggered by a mouse click, a key press, window resize, or a whole bunch of other gestures and interactions. The glue that makes all of this possible is something known as events.

Now, you probably know all about events from your experience using them in the DOM world. (If you don't, then I suggest getting a quick refresher first: https://www.kirupa.com/html5/javascript_events.htm. The way React deals with events is a bit different, and these differences can surprise you in various ways if you aren't paying close attention. Don't worry. We start off with a few simple examples and then gradually look at increasingly more bizarre, complex, and (yes!) boring things.

Listening and Reacting to Events

The easiest way to learn about events in React is to actually use them, and that's exactly what we are going to do! To help with this, we have a simple example made up of a counter that increments each time you click on a button. Initially, our example will look like Figure 10-1.

Figure 10-1 Our example.

Each time you click on the plus button, the counter value will increase by 1. After clicking the plus button a bunch of times, it will look sorta like Figure 10-2.

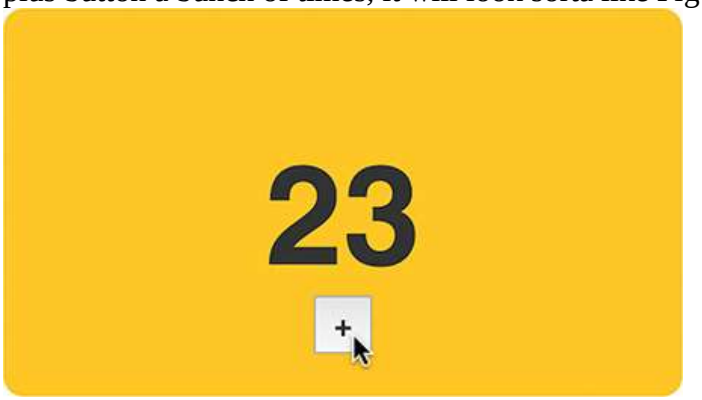

Figure 10-2 After clicking the plus button a bunch of times (23?).

Under the covers, the way this example works is pretty simple. Each time you click on the button, an event gets fired. We listen for this event and do all sorts of React-ey things to get the counter to update when this event gets overheard.

Starting Point

To save all of us some time, we aren't going to be creating everything in our example from scratch. By now, you probably have a good idea of how to work with components, styles, state, and so on. Instead, we are going to start off with a partially implemented example that contains everything except the event-related functionality that we are here to learn.

First, create a new HTML document and ensure your starting point looks as follows:

```
<!DOCTYPE html>
<html>

<head>
  <title>React! React! React!</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>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>

<body>
  <div id="container"></div>
  <script type="text/babel">

  </script>
</body>

</html>
```

Once your new HTML document looks like what you see above, it's time to add our partially implemented counter example. **Inside our script tag below the container div, add the following:**

```
var destination = document.querySelector("#container");

var Counter = React.createClass({
  render: function() {
      var textStyle = {
        fontSize: 72,
        fontFamily: "sans-serif",
        color: "#333",
```

```
      fontWeight: "bold"
    };

    return (
      <div style={textStyle}>
        {this.props.display}
      </div>
    );
  }
});

var CounterParent = React.createClass({
  getInitialState: function() {
    return {
      count: 0
    };
  },
  render: function() {
    var backgroundStyle = {
      padding: 50,
      backgroundColor: "#FFC53A",
      width: 250,
      height: 100,
      borderRadius: 10,
      textAlign: "center"
    };

    var buttonStyle = {
      fontSize: "1em",
      width: 30,
      height: 30,
      fontFamily: "sans-serif",
      color: "#333",
      fontWeight: "bold",
      lineHeight: "3px"
    };

    return (
      <div style={backgroundStyle}>
        <Counter display={this.state.count}/>
        <button style={buttonStyle}>+</button>
      </div>
    );
  }
});
```

```
ReactDOM.render(
  <div>
    <CounterParent/>
  </div>,
  destination
);
```

Once you have added all of this, preview everything in your browser to make sure things get displayed. You should see the beginning of our counter. Take a few moments to look at what all of this code does. There shouldn't be anything that looks strange. The only odd thing will be that clicking the plus button won't do anything. We'll fix that right up in the next section.

Making the Button Click Do Something

Each time we click on the plus button, we want the value of our counter to increase by one. What we need to do is going to look roughly like this:

1. Listen for the click event on the button and specify an event handler.
2. Implement the event handler where we increase the value of our this.state.count property that our counter relies on.

We'll just go straight down the list—starting with listening for the click event. In React, you listen to an event by specifying everything inline in your JSX itself. More specifically, you specify both the event you are listening for and the event handler that will get called, all inside your markup. To do this, find the return function inside our CounterParent component, and make the following highlighted change:

```
  .
  .
  .
return (
  <div style={backgroundStyle}>
    <Counter display={this.state.count}/>
    <button onClick={this.increase} style={buttonStyle}>+</button>
  </div>
);
```

What we've done is told React to call the increase function when the onClick event is overheard.

Next, let's go ahead and implement the increase function—aka our event handler. Inside our CounterParent component, add the following highlighted lines:

```
var CounterParent = React.createClass({
  getInitialState: function() {
    return {
      count: 0
    };
  },
  increase: function(e) {
    this.setState({
      count: this.state.count + 1
    });
  },
  render: function() {
      var backgroundStyle = {
        padding: 50,
        backgroundColor: "#FFC53A",
        width: 250,
        height: 100,
        borderRadius: 10,
        textAlign: "center"
      };

      var buttonStyle = {
        fontSize: "1em",
        width: 30,
        height: 30,
        fontFamily: "sans-serif",
        color: "#333",
        fontWeight: "bold",
        lineHeight: "3px"
      };

      return (
        <div style={backgroundStyle}>
          <Counter display={this.state.count}/>
          <button onClick={this.increase} style={buttonStyle}>+</button>
        </div>
      );
    }
});
```

All we are doing with these lines is making sure that each call to the increase function increments the value of our this.state.count property by 1. Because we are dealing with events, your increase function (as the designated event handler) will get access to the event argument.

We have set this event argument to be accessed by e, and you can see that by looking at our increase function's signature (aka what its declaration looks like). We'll talk about the various events and their properties in a little bit.

Now, go ahead and preview what you have in your browser. Once everything has loaded, click on the plus button to see all of our newly added code in action. Our counter value should increase with each click!

Event Properties

As you know, our events pass what is known as an event argument to our event handler. This event argument contains a bunch of properties that are specific to the type of event you are dealing with. In the regular DOM world, each event has its own type. For example, if you are dealing with a mouse event, your event and its event argument object will be of **type MouseEvent**. This MouseEvent object will allow you to access mouse-specific information, like which button was pressed or the screen position of the mouse click. Event arguments for a keyboard-related event are of **type KeyboardEvent**. Your KeyboardEvent object contains properties which (among many other things) allow you to figure out which key was actually pressed. I could go on forever for every other Event type, but you get the point. Each Event type contains its own set of properties that you can access via the event handler for that event!

Meet Synthetic Events

In React, when you specify an event in JSX like we did with onClick, you are not directly dealing with regular DOM events. **Instead, you are dealing with a React-specific event type known as a SyntheticEvent.** Your event handlers don't get native event arguments of type MouseEvent, KeyboardEvent, etc. They always get event arguments of type SyntheticEvent that wrap your browser's native event instead. What is the fallout of this in our code? Surprisingly not a whole lot.

Each SyntheticEvent contains the following properties:

Property Name	Type
bubbles	boolean
cancelable	boolean
currentTarget	DOMEventTarget
defaultPrevented	boolean
eventPhase	number
isTrusted	boolean
nativeEvent	DOMEvent
preventDefault()	void
isDefaultPrevented()	boolean
isPropagationStopped	void
target	DOMEventTarget
timeStamp	number
type	string

These properties should seem pretty straightforward—and generic! **The non-generic stuff depends on what type of native event our SyntheticEvent is wrapping.** This means that a SyntheticEvent that wraps a MouseEvent will have access to mouse-specific properties such as the following:

boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey

Similarly, a SyntheticEvent that wraps a KeyboardEvent will have access to these additional keyboard-related properties:

boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which

In the end, all of this means that you still get the same functionality in the SyntheticEvent world that you had in the vanilla DOM world.

Now, here is something I learned the hard way. Don't refer to traditional DOM event documentation when using Synthetic events and their properties. Because the SyntheticEvent wraps your native DOM event, events and their properties may not map one-to-one. Some DOM events don't even exist in React. To avoid running into any issues, if you want to know the name of a SyntheticEvent or any of its properties, refer to the **React Event System document**(https://facebook.github.io/react/docs/events.html) instead.

Doing Stuff With Event Properties

By now, you've probably seen more about the DOM and SyntheticEvent stuff than you'd probably like. To wash away the taste of all that text, let's write some code and put all of this newfound knowledge to good use. Right now, our counter example increments by one each time you click on the plus button. What we want to do is increment our counter by ten when the Shift key on the keyboard is pressed while clicking the plus button with our mouse.

The way we are going to do that is by using the shiftKey property that exists on the SyntheticEvent when using the mouse:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

The way this property works is simple. If the Shift key is pressed when this mouse event fires, then the shiftKey property value is true. Otherwise, the shiftKey property value is false. To increment our counter by 10 when the Shift key is pressed, go back to our increase function and make the following highlighted changes:

```
increase: function(e) {
  var currentCount = this.state.count;

  if (e.shiftKey) {
    currentCount += 10;
  } else {
    currentCount += 1;
  }

  this.setState({
    count: currentCount
  });
},
```

Once you've made the changes, preview our example in the browser. Each time you click on the plus button, your counter will increment by one just like it had always done. If you click on the plus button with your Shift key pressed, notice that our counter increments by 10 instead.

The reason that all of this works is because we change our incrementing behavior depending on whether the Shift key is pressed or not. That is primarily handled by the following lines:

```
if (e.shiftKey) {
  currentCount += 10;
} else {
  currentCount += 1;
}
```

If the shiftKey property on our SyntheticEvent event argument is true, we increment our counter by 10. If the shiftKey value is false, we just increment by 1.

More Eventing Shenanigans

We are not done yet! Up until this point, we've looked at how to work with events in React in a very simplistic way. In the real world, rarely will things be as direct as what we've seen. Your real apps will be more complex, and because React insists on doing things differently, we'll need to learn (or re-learn) some new event-related tricks and techniques to make our apps work. That's where this section comes in. We are going to look at some common situations you'll run into and how to deal with them.

You Can't Directly Listen to Events on Components

Let's say your component is nothing more than a button or another type of UI element that users will be interacting with. You **can't** get away with doing something like what we see in the following highlighted line **(NOT ALLOWED):**

```
var CounterParent = React.createClass({
  getInitialState: function() {
    return {
      count: 0
    };
  },
  increase: function() {
    this.setState({
      count: this.state.count + 1
    });
  },
  render: function() {
    return (
      <div>
        <Counter display={this.state.count}/>
        <PlusButton onClick={this.increase}/>
      </div>
    );
  }
});
```

On the surface, this line of JSX looks totally valid. When somebody clicks on our PlusButton component, the increase function will get called. In case you are curious, this is what our PlusButton component looks like:

```
var PlusButton = React.createClass({
  render: function() {
      return (
        <button>
          +
        </button>
      );
    }
});
```

Our PlusButton component doesn't do anything crazy. It only returns a single HTML element!

No matter how you slice and dice this, none of this matters. It doesn't matter how simple or obvious the HTML we are returning via a component looks like. **You simply can't listen for events on them directly.** The reason is because components are wrappers for DOM elements. What does it even mean to listen for an event on a component? Once your component gets unwrapped into DOM elements, does the outer HTML element act as the thing you are listening for the event on? Is it some other element? How do you distinguish between listening for an event and declaring a prop with a value?

There is no clear answer to any of those questions. It's too harsh to say that the solution is to simply not listen to events on components, either. **Fortunately, there is a workaround where we treat the event handler as a prop and pass it on to the component. Inside the component, we can then assign the event to a DOM element and set the event handler to the value of the prop we just passed in.** I realize that probably makes no sense, so let's walk through an example.
Take a look at the following highlighted line:

```
var CounterParent = React.createClass({
   .
   .
   .
  render: function() {
    return (
      <div>
        <Counter display={this.state.count}/>
        <PlusButton clickHandler={this.increase}/>
      </div>
    );
  }
});
```

In this example, we create a property called clickHandler whose value is the increase event handler. Inside our PlusButton component, we can then do something like this:

```
var PlusButton = React.createClass({
  render: function() {
     return (
       <button onClick={this.props.clickHandler}>
         +
       </button>
     );
   }
});
```

On our button element, we specify the onClick event and set its value to the clickHandler prop. At runtime, this prop gets evaluated as our increase function, and clicking the plus button ensures the increase function gets called. This solves our problem while still allowing our component to participate in all this eventing!

Listening to Regular DOM Events

If you thought the previous section was a doozy, wait till you see what we have here. **Not all DOM events have SyntheticEvent equivalents.** It may seem like you can just add the on prefix and capitalize the event you are listening for when specifying it inline in your JSX:

```
var Something = React.createClass({
  handleMyEvent: function(e) {
    // do something
  },
  render: function() {
     return (
       <div onMyWeirdEvent={this.handleMyEvent}>Hello!</div>
     );
   }
});
```

It doesn't work that way! For those events that aren't officially recognized by React, you have to use the traditional approach that uses **addEventListener** with a few extra hoops to jump through.
Take a look at the following section of code:

```
var Something = React.createClass({
  handleMyEvent: function(e) {
    // do something
  },
  componentDidMount: function() {
    window.addEventListener("someEvent", this.handleMyEvent);
  },
  componentWillUnmount: function() {
    window.removeEventListener("someEvent", this.handleMyEvent);
  },
  render: function() {
     return (
       <div>Hello!</div>
     );
   }
});
```

We have our Something component that listens for an event called someEvent. We start listening for this event under the **componentDidMount** method which is automatically called when our component gets rendered. The way we listen for our event is by using addEventListener and specifying both the event and the event handler to call:

```
var Something = React.createClass({
  handleMyEvent: function(e) {
    // do something
  },
  componentDidMount: function() {
    window.addEventListener("someEvent", this.handleMyEvent);
  },
  componentWillUnmount: function() {
    window.removeEventListener("someEvent", this.handleMyEvent);
  },
  render: function() {
      return (
        <div>Hello!</div>
      );
    }
});
```

That should be pretty straightforward. The only other thing you need to keep in mind is removing the event listener when the component is about to be destroyed. To do that, you can use the opposite of the componentDidMount method, the **componentWillUnmount** method. Inside that method, put your removeEventListener call to ensure no trace of our event listening takes place after our component goes away.

The Meaning of this Inside the Event Handler

When dealing with events in React, the value of this inside your event handler is different from what you would normally see in the non-React DOM world. In the non-React world, the value of this inside an event handler refers to the element that your event is listening on:

```
function doSomething(e) {
  console.log(this); //button element
}

var foo = document.querySelector("button");
foo.addEventListener("click", doSomething, false);
```

In the React world (when your components are created using React.createClass), the value of this inside your event handler ALWAYS refers to the component the event handler lives in:

```
var CounterParent = React.createClass({
  getInitialState: function() {
    return {
      count: 0
    };
  },
  increase: function(e) {
    console.log(this); // CounterParent component

    this.setState({
      count: this.state.count + 1
    });
  },
  render: function() {
     return (
       <div>
         <Counter display={this.state.count}/>
         <button onClick={this.increase}>+</button>
       </div>
     );
    }
});
```

In this example, the value of **this** inside the increase event handler refers to the CounterParent component. It doesn't refer to the element that triggered the event. You get this behavior because React automatically binds all methods inside a component to this. This autobinding behavior only applies when your component is created using React.createClass.

If you are using ES6 classes to define your components, the value of this inside your event handler is going to be undefined unless you explicitly bind it yourself:

```
<button onClick={this.increase.bind(this)}>+</button>
```

There is no autobinding magic that happens with the new class syntax, so be sure to keep that in mind if you aren't using React.createClass to create your components.

React…Why? Why?!

Before we call it a day, let's use this time to talk about why React decided to deviate from how we've worked with events in the past. There are two reasons:

- Browser Compatibility
- Improved Performance

Let's elaborate on these two reasons a little bit.

Browser Compatibility

Event handling is one of those things that mostly works consistently in modern browsers, but once you go back to older browser versions, things get really bad really quickly. By wrapping all of the native events as an object of type SyntheticEvent, React frees you from dealing with event handling quirks that you would end up having to deal with otherwise.

Improved Performance

In complex UIs, the more event handlers you have, the more memory your app takes up. Manually dealing with that isn't difficult, but it is a bit tedious as you try to group events under a common parent. Sometimes, that just isn't possible. Sometimes, the hassle doesn't outweigh the benefits. What React does is pretty clever.

React never attaches event handlers to the DOM elements directly. It uses one event handler at the root of your document that is responsible for listening to all events and calling the appropriate event handler as necessary (see Figure 10-3).

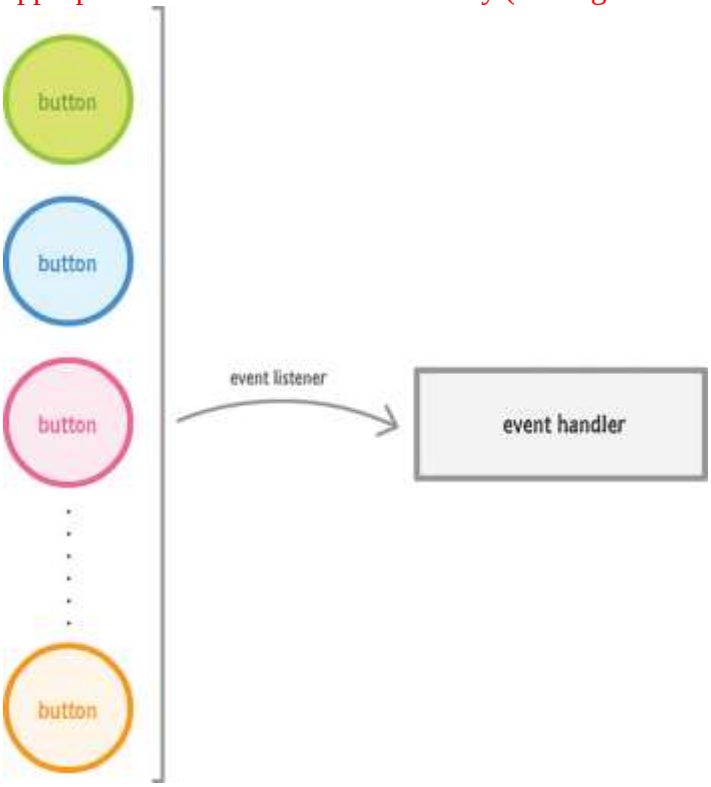

Figure 10-3 React uses one event handler at the root of your document.

This frees you from having to deal with optimizing your event handler-related code yourself. If you've manually had to do that in the past, you can relax knowing that React takes care of that tedious task for you. If you've never had to optimize event handler-related code yourself, consider yourself lucky

Conclusion

You'll spend a lot of time dealing with events, and this chapter threw a lot of things at you. We started by learning the basics of how to listen to events and specify the event handler. Towards the end, we were fully invested and looking at eventing corner cases that you will bump into if you aren't careful enough. You don't want to bump into corners. That is never fun.