

Table of Contents

Lab 1: Building Your First React App (45 minutes).....	3
Lab 2: Components in React (60 minutes).....	11
Lab 3: Styling in React (45 minutes)	23
Lab 4: Creating Complex Components (45 minutes)	37
Lab 5: Transferring Properties (Props) (30 minutes).....	53
Lab 6: Meet JSX—Again! (30 minutes)	65
Lab 7: Dealing with State (30 minutes)	73
Lab 8: Going from Data to UI (30 minutes).....	87
Lab 9: Working with Events (45 minutes).....	95
Lab 10: The Component Lifecycle (15 minutes).....	109
Lab 11: Accessing DOM Elements (30 minutes)	115
Lab 12: Creating a Single-Page App Using React Router (60 minutes).....	125
Lab 13: Building a Todo List App (60 minutes)	147
Lab 14: Setting Up Your React Development Environment (45 minutes)	167

Lab 1: Building Your First React App (45 minutes)

TASK 1: React Starting Point

1. First, we will need a blank HTML page that will act as our starting point. If you don't have a blank HTML page handy, feel free to use the following:

```
<!DOCTYPE html>
<html>

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

  <body>
    <script>

      </script>
    </body>

  </html>
```

2. Let's add a reference to the React library. Just below the title, add these two lines:

```
<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>
```

EXPLANATION: These two lines bring in both the core React library as well as the various things React needs to work with the DOM. Without them, you aren't building a React app at all.

3. There is one more library we need to reference. Just below the previous two script tags, add the following line:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
```

EXPLANATION: What we are doing here is adding a reference to the Babel JavaScript compiler (<http://babeljs.io/>). Babel does many cool things, but the one we care about is its capability to turn JSX into JavaScript.

4. At this point, our HTML page should look something 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>
</head>

<body>
  <script>

  </script>
</body>

</html>
```

5. If you preview your page right now, you'll notice that this page is still blank with nothing visible going on. That's OK. We are going to fix that next.

TASK 2: Displaying Your Name

The first thing we are going to do is use React to display our name on screen. The way we do that is by using a method called **render**.

1. Inside your script tag, add the following:

```
ReactDOM.render(  
  <h1>Sherlock Holmes</h1>,  
  document.body  
);
```

EXPLANATION: Don't worry if none of this makes sense at this point. Our goal is to get something to display on screen first, and we'll make sense of what we did shortly afterwards.

2. Now, before we preview this in our page to see what happens, we need to designate this script block as something that Babel can do its magic on.

- a. The way we do that is by setting the **type attribute** on the script tag to a value of **text/babel**:

```
<script type="text/babel">  
  ReactDOM.render(  
    <h1>Sherlock Holmes</h1>,  
    document.body  
  );  
</script>
```

- b. Once you've made that change, now preview what you have in your browser. What you'll see are the words Sherlock Holmes printed in giant letters. Congratulations! You just built an app using React.

EXPLANATION: The ReactDOM.render method takes two arguments:

1. The HTML-like elements (aka JSX) you wish to output
2. The location in the DOM that React will render the JSX into

- c. **INFORMATION ONLY: Here is what our render method looks like:**

```
ReactDOM.render(  
  <h1>Sherlock Holmes</h1>,  
  document.body  
);
```

EXPLANATION:

Our first argument is the text Sherlock Holmes wrapped inside some h1 tags. The second argument is document.body. It simply specifies where the converted markup from the JSX will end up living in our DOM. In our example, when the render method runs, the h1 tag (and everything inside it) is placed in our document's body element.

3. Now, the goal of this exercise wasn't to display a name on the screen. It was to display your name.

a. Go ahead and modify your code to do that. In my case, the render method will look as follows:

```
ReactDOM.render(  
  <h1>Batman</h1>,  
  document.body  
)
```

4. Well—it would look like that if my name was Batman! Anyway, if you preview your page now, you will see your name displayed instead of Sherlock Holmes.

TASK 3: Changing the Destination

The first thing we'll do is change where our JSX gets output. Using JavaScript to place things directly in your body element is never a good idea. A lot can go wrong—especially if you are going to be mixing React with other JS libraries and frameworks. The recommended path is to create a separate element that you will treat as a new root element. This element will serve as the destination our render method will use.

1. To make this happen, go back to the HTML and add a div element with an id value of container.
 - a. **INFORMATION ONLY:** Instead of showing you the full HTML for this one minor change, here is what just our body element looks like:

```
<body>
  <div id="container"></div>
  <script type="text/babel">
    ReactDOM.render(
      <h1>Batman</h1>,
      document.body
    );
  </script>
</body>
```

2. With our container div element safely defined, let's **modify the render method** to use it instead of document.body. Here is one way of doing this:

```
ReactDOM.render(
  <h1>Batman</h1>,
  document.querySelector("#container")
);
```

3. Another way of doing this is by doing some things outside of the render method itself. Go ahead and make these highlighted modifications as it is the proper way to handle this:

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

ReactDOM.render(
  <h1>Batman</h1>,
  destination
);
```

TASK 4: Styling It Up!

1. Time for our last change. Right now, our names show up in whatever default h1 styling our browser provides. Let's fix it by adding some CSS.
 - a. Inside your head tag, add a style block with the following CSS:

```
#container {  
  padding: 50px;  
  background-color: #EEE;  
}  
#container h1 {  
  font-size: 48px;  
  font-family: sans-serif;  
  color: #0080A8;  
}
```

2. After you have added all of this, preview your page. Notice that our text appears with a little more purpose than it did earlier when it relied entirely on the browser's default styling (see Figure 2-1).

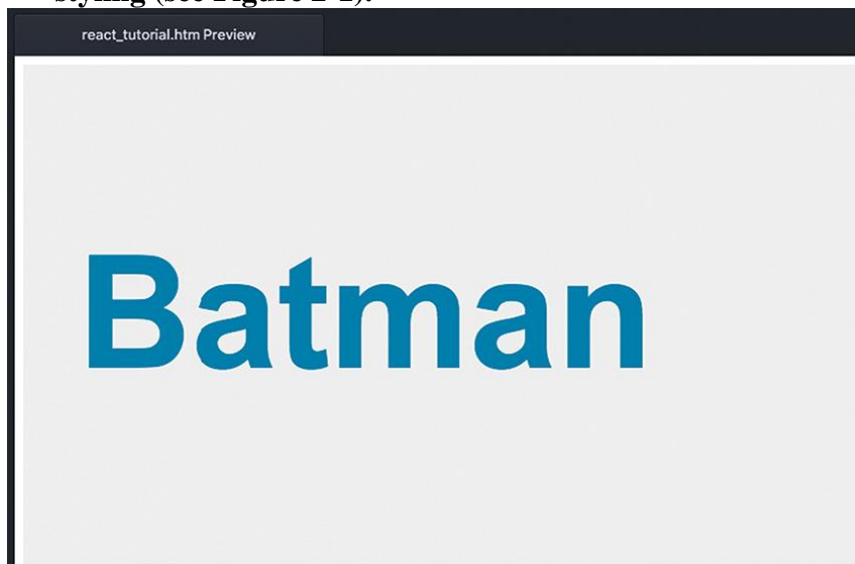


Figure 2-1 The result of adding the CSS.

3. INFORMATION ONLY: The end result is that your React app is still going to be made up of some 100% organic HTML, CSS, and JavaScript:

```
<!DOCTYPE html>
<html>

<head>
  <title>React! React! React!</title>
  <script src="https://fb.me/react-15.1.0.js"></script>
  <script src="https://fb.me/react-dom-15.1.0.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: #EEE;
    }
    #container h1 {
      font-size: 144px;
      font-family: sans-serif;
      color: #0080a8;
    }
  </style>
</head>

<body>
  <div id="container"></div>
  <script type="text/babel">
    var destination = document.querySelector("#container");

    ReactDOM.render(React.createElement(
      "h1",
      null,
      "Batman"
    ), destination);
  </script>
</body>

</html>
```

Notice that there is not a trace of React-like code in sight.

Conclusion

If this is your first time building a React app, we covered a lot of ground here. One of the biggest takeaways is that React is different than other libraries because it uses a whole new language called JSX to define what the visuals will look like. We got a very small glimpse of that here when we defined the `<h1>` tag inside the render method.

JSX's impact goes beyond how you define your UI elements. It also alters how you build your app as a whole. Because your browser can't understand JSX in its native representation, you need to use an intermediate step to convert that JSX into JavaScript. One approach is to build your app to generate the transpiled JavaScript output to correspond to the JSX source. Another approach (aka the one we used here) is to use the Babel library to translate the JSX into JavaScript on the browser itself. While the performance hit of doing that is not recommended for live/production apps, when familiarizing yourself with React, you can't beat the convenience.

In future modules, we'll spend some time diving deeper into JSX and going beyond the render method as we look at all the important things that make React tick.

Lab 2: Components in React (60 minutes)

TASK 1: Changing How We Deal with the UI

1. We are going to go back and look at the render method we used in the previous module:

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

ReactDOM.render(
  <h1>Batman</h1>,
  destination
);
```

EXPLANATION: What you see on the screen is the word Batman printed in giant letters—thanks to the h1 element.

2. Let's change things up a bit and say that we want to print the names of several other superheroes. To do this, let's modify our render method to now look as follows:

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

ReactDOM.render(
  <div>
    <h1>Batman</h1>
    <h1>Iron Man</h1>
    <h1>Nicolas Cage</h1>
    <h1>Mega Man</h1>
  </div>,
  destination
);
```

EXPLANATION: Notice what you see here. We emit a div that contains the four h1 elements with our superhero names.

TASK 2: JSX Gotcha: Outputting Multiple Elements

There is an important JSX detail to call out here. The div that wraps our h1 elements isn't there because it looks like a good idea. **It is there because it has to be there.**

1. **INFORMATION ONLY:** In React, you can't output multiple adjacent elements as shown in the following:

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

```
ReactDOM.render(  
  <h1>Batman</h1>  
  <h1>Iron Man</h1>  
  <h1>Nicolas Cage</h1>  
  <h1>Mega Man</h1>,  
  destination  
);
```

2. **Ok, so what we have now are four h1 elements that each contain the name of a superhero. What if we want to change our h1 element to something like an h3 instead? We can manually update all of these elements as follows:**

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

```
ReactDOM.render(  
  <div>  
    <h3>Batman</h3>  
    <h3>Iron Man</h3>  
    <h3>Nicolas Cage</h3>  
    <h3>Mega Man</h3>  
  </div>,  
  destination  
);
```

3. If you preview what we have, you'll see something that looks a bit unstyled and plain (see Figure 3-3).

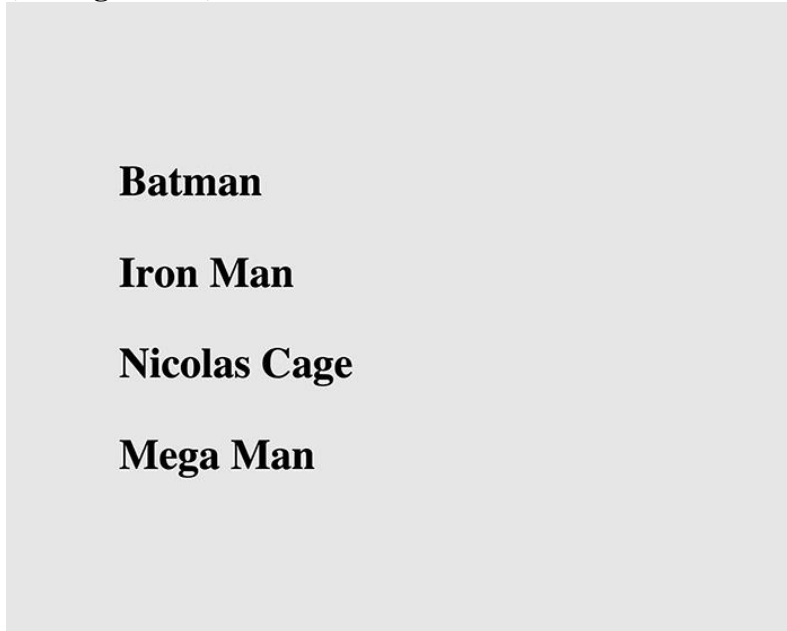


Figure 3-3 Plain vanilla super hero names.

4. We don't want to go crazy with the styling here. All we want to do is just italicize all of these names by using the `<i>` tag, so let's manually update what we render by making this change:

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

```
ReactDOM.render(  
  <div>  
    <h3><i>Batman</i></h3>  
    <h3><i>Iron Man</i></h3>  
    <h3><i>Nicolas Cage</i></h3>  
    <h3><i>Mega Man</i></h3>  
  </div>,  
  destination  
)
```

EXPLANATION: We went through each h3 element and wrapped the content inside some i tags. Can you start to see the problem here?

Every change we want to make to our h1 or h3 elements needs to be duplicated for every instance of it. What if we want to do something even more complex than just modifying the appearance of our elements? What if we want to represent something more complex than the simple examples we are using so far? What we are doing right now won't scale because manually updating every copy of what we want to modify is time consuming.

TASK 3: Meet the React Component

The solution to all of our problems (even the existential ones we grapple with!) can be found in React components. React components are reusable chunks of JavaScript that output (via JSX) HTML elements.

1. Let's start by building a couple of components together. To follow along, start with a blank React document:

```
<!DOCTYPE html>
<html>

<head>
  <title>React Components</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">

    </script>
</body>

</html>
```

EXPLANATION: There is nothing exciting going on this page. Nearly identical to what we had in our earlier module, this page is pretty barebones, with just a reference to the React and Babel libraries and a div element which has an id value of container.

TASK 4: Creating a Hello, World! Component

1. We are going to start really simple. What we want to do is use a component to help us print the famous “Hello, world!” text to the screen. Just add the render method of ReactDOM as follows inside of the script tag:

```
ReactDOM.render(  
  <div>  
    <p>Hello, world!</p>  
  </div>,  
  document.querySelector("#container")  
)
```

2. Let’s recreate all of this by using a component. You have several ways of creating components in React, but the way we are going to create them initially is by using **React.createClass**.
 - a. Go ahead and add the following **highlighted** code just above our existing render method:

```
var HelloWorld = React.createClass({
```

```
});
```

```
ReactDOM.render(  
  <div>  
    <p>Hello, world!</p>  
  </div>,  
  document.querySelector("#container")  
)
```

EXPLANATION: What we have done is create a new component called HelloWorld. This HelloWorld component doesn’t do anything right now. In fact, it is basically an empty JavaScript object at this point.

3. Go ahead and modify our HelloWorld component by adding a render property as shown in the following:

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

```
  }
```

```
});
```

EXPLANATION: Just like the render method of we saw a few moments earlier as part of ReactDOM.render, the render method inside a component is also responsible for dealing with JSX.

4. Let's modify our render method to return "Hello, componentized world!", so go ahead and add the following **highlighted** lines:

```
var HelloWorld = React.createClass({  
  render: function() {  
    return (  
      <p>Hello, componentized world!</p>  
    );  
  }  
});
```

EXPLANATION: What we've done is told our render method to return the JSX that represents our Hello, componentized world! text. All that remains is to actually use this component.

5. The way you use a component, once you've defined it, is by calling it. We are going to call it from our old friend, the ReactDOM.render method:

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

EXPLANATION: That isn't a typo! The JSX we use for calling our HelloWorld component is the very HTML-like <HelloWorld/>.

6. If you preview your page in your browser, you'll see the text "Hello, componentized world!" showing up on your screen.
7. Ok, back to reality. What we've done so far might seem crazy, but simply think of your <HelloWorld/> component as a cool and new HTML tag whose functionality you have full control over. This means you can do all sorts of HTML-ey things to it.
 - a. For example, go ahead and modify our ReactDOM.render method to look as follows:

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

EXPLANATION: We wrapped our call to the HelloWorld component inside a div element, and if you preview this in your browser, everything still works.

8. Let's go one step further! Instead of having just a single call to HelloWorld, let's make a bunch of calls.
 - a. Modify our ReactDOM.render method to now look as follows:

```
ReactDOM.render(  
  <div>  
    <HelloWorld/>  
    <HelloWorld/>  
    <HelloWorld/>  
    <HelloWorld/>  
    <HelloWorld/>  
    <HelloWorld/>  
  </div>,  
  document.querySelector("#container")  
)
```

EXPLANATION: What you will see now is a bunch of “Hello, componentized world!” text instances appear.

9. Let's do one more thing before we move on to something shinier. Go back to our HelloWorld component declaration, and change the text we return to the more traditional Hello, world! value:

```
var HelloWorld = React.createClass({  
  render: function() {  
    return (  
      <p>Hello, world!</p>  
    );  
  }  
});
```

10. Just make this one change and preview your code. This time around, all of the various HelloWorld calls we specified earlier now return “Hello, world!” to the screen. There was no manually modifying every HelloWorld call. That's a good thing!

TASK 5: First Part: Updating the Component Definition

Right now, our HelloWorld component is hard coded to always send out Hello, world! as part of its return value. The first thing we are going to do is change that behavior by having return print out the value passed in by a property. We need a name to give our property, and for this example, we are going to call our property `greetTarget`.

1. To specify the value of `greetTarget` as part of our component, here is the modification we need to make:

```
var HelloWorld = React.createClass({
  render: function() {
    return (
      <p>Hello, {this.props.greetTarget}!</p>
    );
  }
});
```

EXPLANATION: The way you access a property is by calling it via the props property that every component has access to. Notice how we specify this property. We place it inside curly brackets { and }. In JSX, if you want something to get evaluated as an expression, you need to wrap that something inside curly brackets. If you don't do that, you'll see the raw text `this.props.greetTarget` printed out.

Task 6: Second Part: Modifying the Component Call

Once you've updated the component definition, all that remains is to pass in the property value as part of the component call. That is done by **adding an attribute with the same name as our property, followed by the value you want to pass in**. In our code, that would involve modifying the HelloWorld call with the `greetTarget` attribute and the value we want to give it.

1. Go ahead and modify our HelloWorld calls as follows:

```
ReactDOM.render(
  <div>
    <HelloWorld greetTarget="Batman"/>
    <HelloWorld greetTarget="Iron Man"/>
    <HelloWorld greetTarget="Nicolas Cage"/>
    <HelloWorld greetTarget="Mega Man"/>
    <HelloWorld greetTarget="Bono"/>
    <HelloWorld greetTarget="Catwoman"/>
  </div>,
  document.querySelector("#container")
);
```

EXPLANATION: Each of our HelloWorld calls now has the `greetTarget` attribute along with the name of a superhero that we wish to greet.

2. If you preview this example in the browser, you'll see the greetings happily printed out on screen.

TASK 7: Dealing with Children

There is one more thing you can do with components just like you can with many HTML elements. Your components can have children.

1. To make sense of all this, let's fiddle with another really simple example. This time around, create a component called **Buttonify** that wraps its children inside a button. The component looks like this:

```
var Buttonify = React.createClass({
  render: function() {
    return (
      <div>
        <button type={this.props.behavior}>{this.props.children}</button>
      </div>
    );
  }
});
```

2. The way you can use this component is by just calling it via the ReactDOM.render method as shown here:

```
ReactDOM.render(
  <div>
    <Buttonify behavior="Submit">SEND DATA</Buttonify>
  </div>,
  document.querySelector("#container")
);
```

3. When this code runs, given what the JSX in the Buttonify component's render method looked like, what you will see are the words "SEND DATA" wrapped inside a button element. With the appropriate styling, the result could look comically large like in Figure 3-4.



Figure 3-4 A large send data button.

EXPLANATION: In the JSX, notice that we specify a custom property called behavior. This property enables us to specify the button element's type attribute, and you can see us accessing it via `this.props.behavior` in the component definition's render method.

Conclusion

If you want to build an app using React, you can't wander too far without having to use a component.

Lab 3: Styling in React (45 minutes)

Well, don't tell React that. While React doesn't actively hate CSS, it has a different view when it comes to styling content. As we've seen so far, one of React's core ideas is to have our app's visual pieces be self-contained and reusable. That is why the HTML elements and the JavaScript that impacts them are in the same bucket we call a component.

TASK 1: Displaying Some Vowels

To learn how to style our React content, let's work together on an example that simply displays vowels on a page.

1. First, you'll need a blank HTML page that will host our React content. If you don't have one, feel free to use the following markup:

```
<!DOCTYPE html>
<html>

<head>
  <title>Styling in 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>

</body>

</html>
```

EXPLANATION: All this markup does is load in our React and Babel libraries and specify a div with an id value of container.

2. To display the vowels, we're going to add some React-specific code.
 - a. Just below the container div element, add the following:

```
<script type="text/babel">

var Letter = React.createClass({
  render: function() {
    return (
      <div>
        {this.props.children}
      </div>
    );
  }
});

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

ReactDOM.render(
  <div>
    <Letter>A</Letter>
    <Letter>E</Letter>
    <Letter>I</Letter>
    <Letter>O</Letter>
    <Letter>U</Letter>
  </div>,
  destination
);

</script>
```

EXPLANATION: From what we learned about components, nothing here should be a mystery. We create a component called Letter that is responsible for wrapping our vowels inside a div element. All of this is anchored in our HTML via a script tag whose type designates it as something Babel will know what to do with.

3. If you preview your page, you'll see something that looks like Figure 4-1.

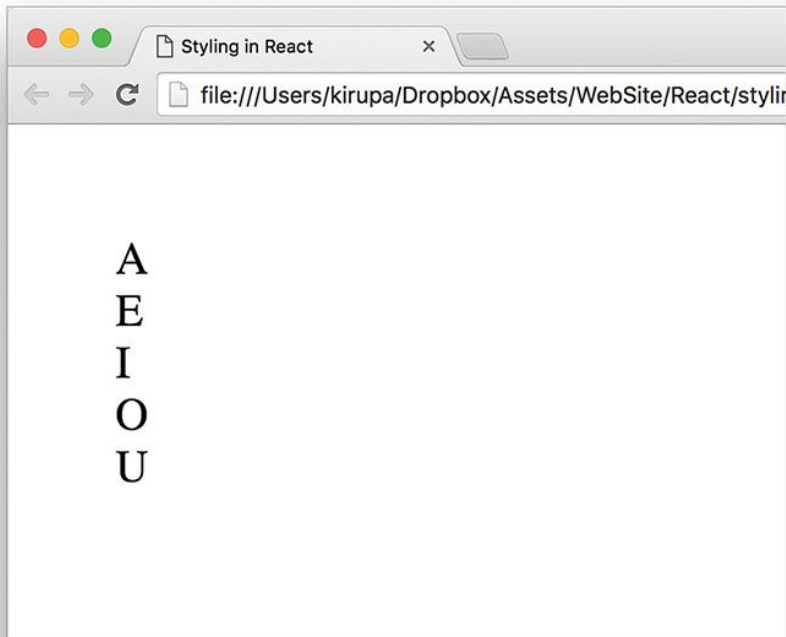


Figure 4-1 An output of what you see.

EXPLANATION: Don't worry, we'll make it look a little less mundane in a few moments. After we've had a run at these letters, you will see something that looks more like Figure 4-2.



Figure 4-2 The letters arranged horizontally and with a yellow background.

Our vowels will be wrapped in a yellow background, aligned horizontally, and sport a fancy monospace font. Let's look at how to do all of this in both CSS as well as React's new-fangled approach.

Styling React Content Using CSS

Using CSS to style our React content is actually as straightforward as you can imagine it to be. Because React ends up spitting out regular HTML tags, all of the various CSS tricks you've learned over the years to style HTML still apply. There are just a few minor things to keep in mind.

TASK 2: Understand the Generated HTML

Before you can use CSS, you need to first get a feel for what the HTML that React spits out is going to look like. You can easily figure that out by looking at the JSX defined inside the render methods.

1. **INFORMATION ONLY** - The parent render method is our ReactDOM based one, and it looks as follows:

```
<div>
  <Letter>A</Letter>
  <Letter>E</Letter>
  <Letter>I</Letter>
  <Letter>O</Letter>
  <Letter>U</Letter>
</div>
```

EXPLANATION: We have our various Letter components wrapped inside a div. Nothing too exciting here. The render method inside our Letter component isn't that much different either:

```
<div>
  {this.props.children}
</div>
```

2. As you can see, each individual vowel is wrapped inside its own set of div tags. If you had to play this all out (such as, previewing our example in a browser), the final DOM structure for our vowels looks like Figure 4-3.

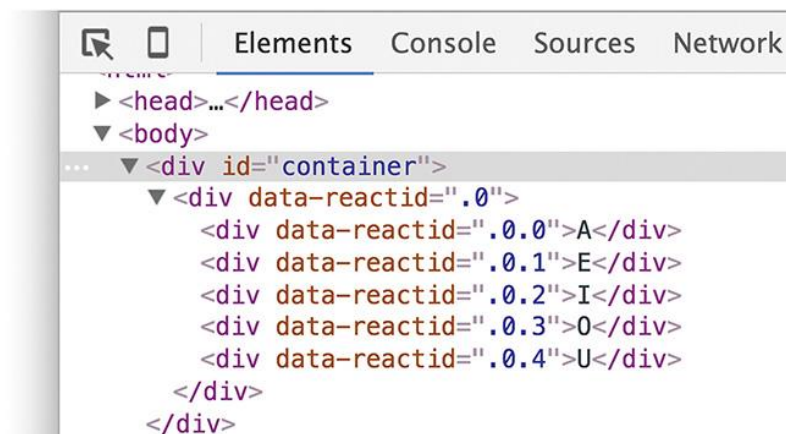


Figure 4-3 The preview from inside the browser.

EXPLANATION: Ignore the data-reactroot attribute (that you may not even see depending on your version of React!) on the container div, but pay attention to the rest of the things you see. What we have is simply an HTML-ized expansion of the various JSX fragments we saw in the render method a few moments ago with our vowels nested inside a bunch of div elements.

TASK 3: Just Style It Already!

Once you understand the HTML arrangement of the things you want to style, the hard part is done. Now comes the fun and familiar part of defining style selectors and specifying the properties you want to set.

1. To affect our inner div elements, add the following inside our style tag:

```
div div div {  
  padding: 10px;  
  margin: 10px;  
  background-color: #ffde00;  
  color: #333;  
  display: inline-block;  
  font-family: monospace;  
  font-size: 32px;  
  text-align: center;  
}
```

EXPLANATION: The div div div selector will ensure we style the right things. The end result will be our vowels styled to look exactly like we saw earlier. With that said, a style selector of div div div looks a bit odd, doesn't it? It is too generic. In apps with more than three nested div elements (which will be very common), you may end up styling the wrong things. It is at times like this where you will want to change the HTML that React generates to make our content more easily style-able.

2. The way we are going to address this is by giving our inner div elements a class value of letter. Here is where JSX differs from HTML.
 - a. Make the following **highlighted** change:

```
var Letter = React.createClass({  
  render: function() {  
    return (  
      <div className="letter">  
        {this.props.children}  
      </div>  
    );  
  }  
});
```

EXPLANATION: Notice that we designate the class value by using the `className` attribute instead of the `class` attribute. The reason has to do with the word `class` being a special keyword in JavaScript.

3. Anyway, once you've given your `div` a **className** attribute value of `letter`, there is just one more thing to do.
 - a. Modify the CSS selector to target our `div` elements more cleanly:

```
.letter {  
  padding: 10px;  
  margin: 10px;  
  background-color: #ffde00;  
  color: #333;  
  display: inline-block;  
  font-family: monospace;  
  font-size: 32px;  
  text-align: center;  
}
```

EXPLANATION: As you can see, using CSS is a perfectly viable way to style the content in your React-based apps. In the next section, we'll look at how to style our content using the approach preferred by React.

TASK 4: Styling Content the React Way

React favors an inline approach for styling content that doesn't use CSS. While that seems a bit strange at first, it is designed to help make your visuals more reusable. The goal is to have your components be little black boxes where everything related to how your UI looks and works gets stashed there. Let's see this for ourselves.

1. Continuing our code from earlier, remove the `.letter` style rule. Once you have done this, your vowels will return to their unstyled state when you preview your app in the browser.
 - a. For completeness, you should remove the `className` declaration from our `Letter` component's render function as well. There is no point having our markup contain things we won't be using.
2. Right now, our `Letter` component is back to its original state:

```
var Letter = React.createClass({
  render: function() {
    return (
      <div>
        {this.props.children}
      </div>
    );
  }
});
```

EXPLANATION: The way you specify styles inside your component is by defining an object whose content is the CSS properties and their values. Once you have that object, you assign that object to the JSX elements you wish to style by using the `style` attribute.

TASK 5: Creating a Style Object

1. Let's get right to it by defining our object that contains the styles we wish to apply:

```
var Letter = React.createClass({
  render: function() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: "#ffde00",
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
      fontSize: 32,
      textAlign: "center"
    };

    return (
      <div>
        {this.props.children}
      </div>
    );
  }
});
```

EXPLANATION: We have an object called `letterStyle`, and the properties inside it are just CSS property names and their value. If you've never defined CSS properties in JavaScript before (i.e., by setting `object.style`), the formula for converting them into something JavaScript-friendly is pretty simple:

- Single word CSS properties (like `padding`, `margin`, `color`) remain unchanged.
- Multi-word CSS properties with a dash in them (like `background-color`, `font-family`, `border-radius`) are turned into one camelCase word with the dash removed and the words following the dash capitalized. For example, using our example properties, `background-color` would become `backgroundColor`, `font-family` would become `fontFamily`, and `border-radius` would become `borderRadius`.

TASK 6: Actually Styling Our Content

Now that we have our object containing the styles we wish to apply, the rest is very easy. Find the element we wish to apply the style to and set the style attribute to refer to that object. In our case, that will be the div element returned by our Letter component's render function.

1. Take a look at the **highlighted** line to see how this is done for our code:

```
var Letter = React.createClass({
  render: function() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: "#ffde00",
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
      fontSize: "32",
      textAlign: "center"
    };

    return (
      <div style={letterStyle}>
        {this.props.children}
      </div>
    );
  }
});
```

EXPLANATION: Our object is called letterStyle, so that is what we specify inside the curly brackets to let React know to evaluate the expression.

2. Go ahead and run the example in the browser to ensure everything works properly and all of our vowels are properly styled.

3. For some extra validation, if you inspect the styling applied to one of the vowels using your browser developer tool of choice, you'll see that the styles are in fact applied inline (see Figure 4-4).

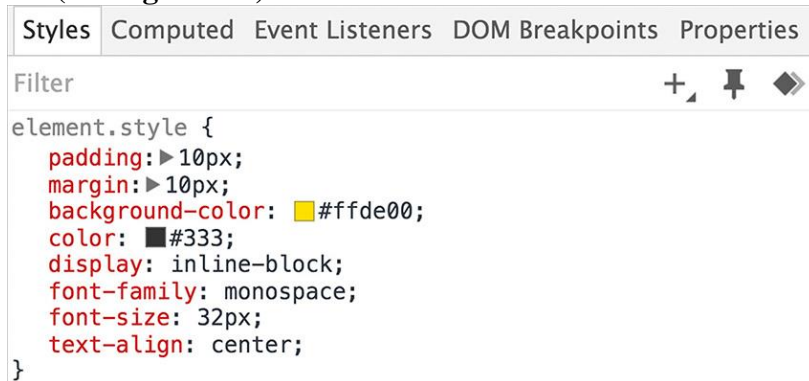


Figure 4-4 The styles are applied inline.

TASK 7: You Can Omit the “px” Suffix

1. **INFORMATION ONLY** - To help with this, React allows you to omit the px suffix for a bunch of CSS properties. If you recall, our letterStyle object looks as follows:

```
var letterStyle = {  
  padding: 10,  
  margin: 10,  
  backgroundColor: "#ffde00",  
  color: "#333",  
  display: "inline-block",  
  fontFamily: "monospace",  
  fontSize: "32",  
  textAlign: "center"  
};=
```

EXPLANATION: Notice that for some of the properties with a numerical value such as padding, margin, and fontSize, we didn't specify the px suffix at all. That is because, at runtime, React will add the px suffix automatically.

TASK 8: Making the Background Color Customizable

The last thing we are going to do before we wrap things up is take advantage of how React works with styles. By having our styles defined in the same vicinity as the JSX, we can make the various style values easily customizable by the parent (aka the consumer of the component).

1. Right now, all of our vowels have a yellow background. Wouldn't it be nice if we could specify the background color as part of each Letter declaration?
 - a. To do this, in our ReactDOM.render method, first **add a bgcolor attribute** and specify some colors as shown in the following **highlighted** lines:

```
ReactDOM.render(  
  <div>  
    <Letter bgcolor="#58B3FF">A</Letter>  
    <Letter bgcolor="#FF605F">E</Letter>  
    <Letter bgcolor="#FFD52E">I</Letter>  
    <Letter bgcolor="#49DD8E">O</Letter>  
    <Letter bgcolor="#AE99FF">U</Letter>  
  </div>,  
  destination  
)
```

2. Next, we need to use this property.
 - a. In our letterStyle object, set the value of **backgroundColor** to **this.props.bgColor**:

```
var letterStyle = {  
  padding: 10,  
  margin: 10,  
  backgroundColor: this.props.bgcolor,  
  color: "#333",  
  display: "inline-block",  
  fontFamily: "monospace",  
  fontSize: "32",  
  textAlign: "center"  
};
```

EXPLANATION: This will ensure that the backgroundColor value is inferred from what we set via the bgColor attribute as part of the Letter declaration.

3. If you preview this in your browser, you will now see our same vowels sporting some background colors as shown in Figure 4-5.

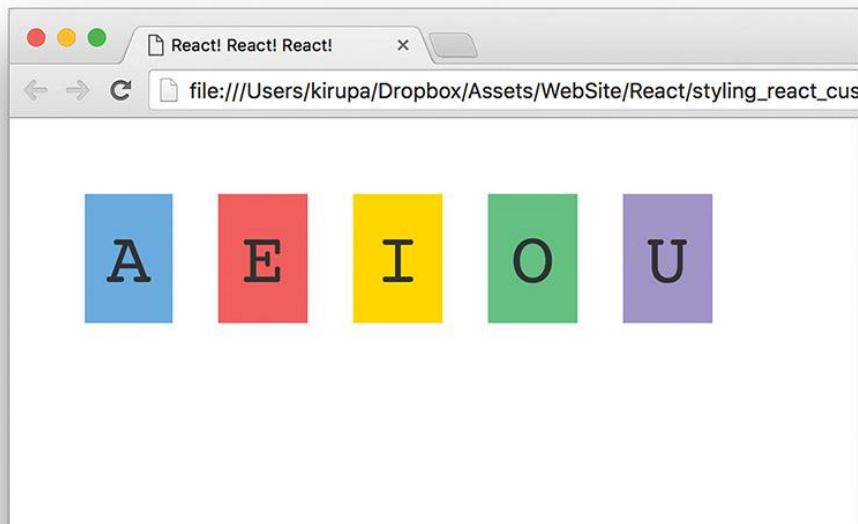


Figure 4-5 Our vowels with background colors!

Conclusion

As we dive in further and learn more about React, you'll see several more cases where React does things quite differently than what we've been told is the correct way of doing things on the web. In this lab, we saw React promoting inline styles in JavaScript as a way to style content as opposed to using CSS style rules. Earlier, we looked at JSX and how the entirety of your UI can be declared in JavaScript using an XML-like syntax that resembles like HTML.

In all of these cases, if you look deeper beneath the surface, the reasons for why React diverges from conventional wisdom makes a lot of sense. Building apps with their very complex UI requirements requires a new way of solving the challenges associated with complex UIs. HTML, CSS, and JavaScript techniques that probably made a lot of sense when dealing with web pages and documents may not be applicable in the web app world where components are re-used inside other components.

With that said, you should pick and choose the techniques that make the most sense for your situation. While I am biased towards React's way of solving our UI development problems, I'll do my best to highlight alternate or conventional methods as well. Tying that back to what we saw here, using CSS style rules with your React content is OK as long as you make the decision knowing the things you gain as well as lose by doing so.

Lab 4: Creating Complex Components (45 minutes)

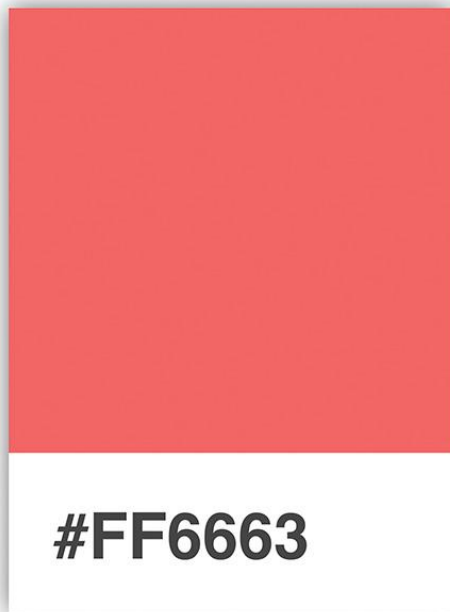
Beyond reusability, there is another major advantage components bring to the table. They make possible composability. You can combine components to create more complex components.

In this lab, we look at what all of this means. More specifically, we look at two things:

- The technical stuff that you need to know.
- The stuff you need to know about how to identify components when you look at a bunch of visual elements.

TASK 1:

What we are going to do is build a simple color palette card (see Figure 5-2).



Hi! I am a simple color palette card :P

Figure 5-2 A simple color palette card.

If you are not sure what these are, these are small rectangular cards that help you match a color with a particular type of paint. You'll frequently see them in home improvement stores or anywhere paint is sold. Your designer friend probably has a giant closet dedicated to them in their place. Anyway, our mission is to recreate one of these cards using React.

There are several ways to go about this, but I am going to show you a very systematic approach that will help you simplify and make sense of even the most complex user interfaces. This approach involves two steps:

1. Identify the major visual elements
2. Figure out what the components will be

Identifying the Major Visual Elements

1. The first step is to identify all of the visual elements we are dealing with. No visual element is too minor to omit—at least not initially. The easiest way to start identifying the relevant pieces is to start with the obvious visual elements and then dive into the less obvious ones.

We have identified our three components, and the component hierarchy looks as in Figure 5-8.

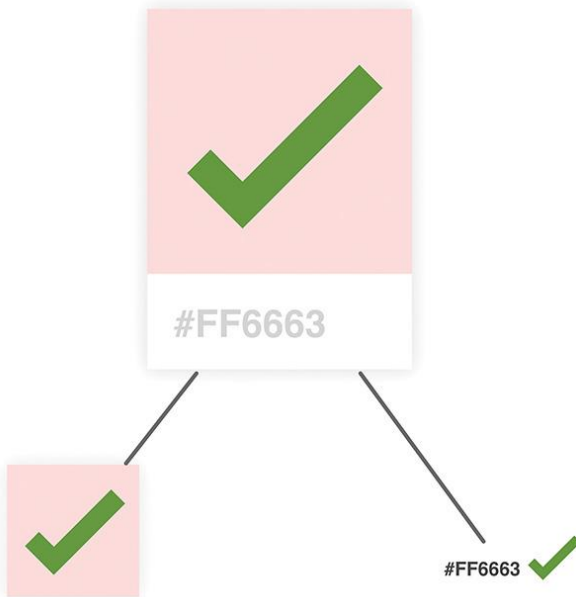


Figure 5-8 The three components.

EXPLANATION: An important thing to note is that the component hierarchy has more to do with helping us define our code than it does with how the finished product will look. You'll notice that it looks a bit different than the visual hierarchy we started off with. For visual details, you should always refer to your source material (aka your visual comps, redlines, screenshots, and other related items). For figuring out which components to create, you should use the component hierarchy.

TASK 2: Creating the Components

It is time for us to start writing some code.

1. The first thing we need is a mostly-empty HTML page that will serve as our starting point:

```
<!DOCTYPE html>
<html>

<head>
  <title>More Components!</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">

    ReactDOM.render(
      <div>

        </div>,
      document.querySelector("#container")
    );
  </script>
</body>

</html>
```

- a. Take a moment to see what this page has going on. There isn't much—just the bare minimum needed to have React render an empty div into our container element.

2. It is now time to define our three components. The names we will go with for our components will be **Card, Label, and Square**.

a. Go ahead and add the following **highlighted** lines just above the ReactDOM.render function:

```
var Square = React.createClass({
  render: function() {
    return(
      <p>Nothing</p>
    );
  }
});
```

```
var Label = React.createClass({
  render: function() {
    return (
      <p>Nothing</p>
    );
  }
});
```

```
var Card = React.createClass({
  render: function() {
    return (
      <p>Nothing</p>
    );
  }
});
```

```
ReactDOM.render(
  <div>

  </div>,
  document.querySelector("#container")
);
```

EXPLANATION: Within our three components, we also threw in the render function that each component absolutely needs to function. Other than that, our components are empty. In the following sections, we will fix that by filling them in.

TASK 3: The Card Component

1. We are going to start at the top of our component hierarchy and focus on our Card component first. This component will act as the container that our Square and Label components will live in.
 - a. To implement it, go ahead and make the following **highlighted** modifications:

```
var Card = React.createClass({
  render: function() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      WebkitFilter: "drop-shadow(0px 0px 5px #666)",
      filter: "drop-shadow(0px 0px 5px #666)"
    };

    return (
      <div style={cardStyle}>

    </div>
    );
  }
});
```

EXPLANATION: While this seems like a lot of changes, the bulk of the lines are going into styling the output of our Card component via the cardStyle object. Inside the object, notice that we specify a vendor-prefixed version of the CSS filter property with WebkitFilter. That's not the interesting detail. The interesting detail is the capitalization. Instead of the first letter being camelcased as webkitFilter, the W is actually capitalized. That isn't how other normal CSS properties are represented, so keep that in mind if you ever need to specify a vendor-prefixed property.

The rest of the changes are pretty unimpressive. We return a div element, and that element's style attribute is set to our cardStyle object.

2. Now, to see our Card component in action, we need to display it in our DOM as part of the ReactDOM.render function.
 - a. To make that happen, go ahead and make the following **highlighted** change:

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

EXPLANATION: All we are doing is telling the ReactDOM.render function to render the output of our Card component by invoking it.

3. If everything worked out properly, you'll see the same thing as in Figure 5-9 if you test your app.

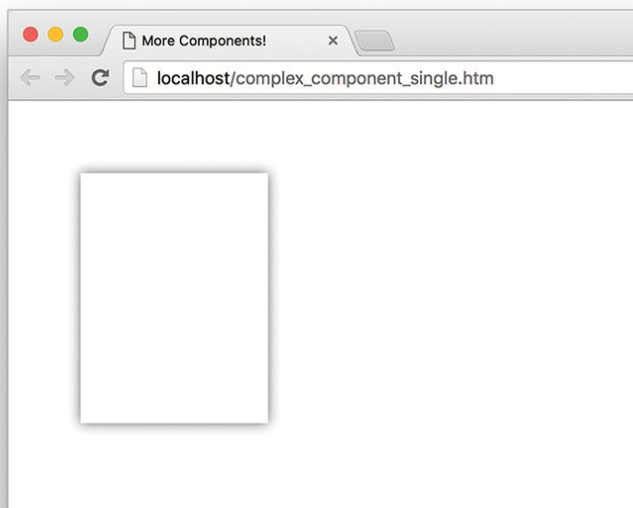


Figure 5-9 The result of your test—the outline of the color palette card.

EXPLANATION: Yes, it is just the outline of our color palette card, but that is definitely more than what we started out with just a few moments ago!

TASK 4: The Square Component

1. It's time to go one level down in our component hierarchy and look at our Square component.

- a. This is pretty straightforward, so make the following **highlighted** changes:

```
var Square = React.createClass({
  render: function() {
    var squareStyle = {
      height: 150,
      backgroundColor: "#FF6663"
    };
    return(
      <div style={squareStyle}>

        </div>
    );
  }
});
```

EXPLANATION: Just like with our Card component, we are returning a div element whose style attribute is set to a style object that defines how this component looks.

2. To see our Square component in action, we need to get it onto our DOM just like we did with the Card component earlier. The difference this time around is that we won't be calling the Square component via our ReactDOM.render function. Instead, we'll call the Square component from inside the Card component.

- a. To see this, go back to our Card component's render function, and make the following change:

```
var Card = React.createClass({
  render: function() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      WebkitFilter: "drop-shadow(0px 0px 5px #666)",
      filter: "drop-shadow(0px 0px 5px #666)"
    };
    return (
      <div style={cardStyle}>
        <Square/>
      </div>
    );
  }
});
```

3. At this point, if you preview our app, you'll see a colorful square making an appearance (see Figure 5-10).

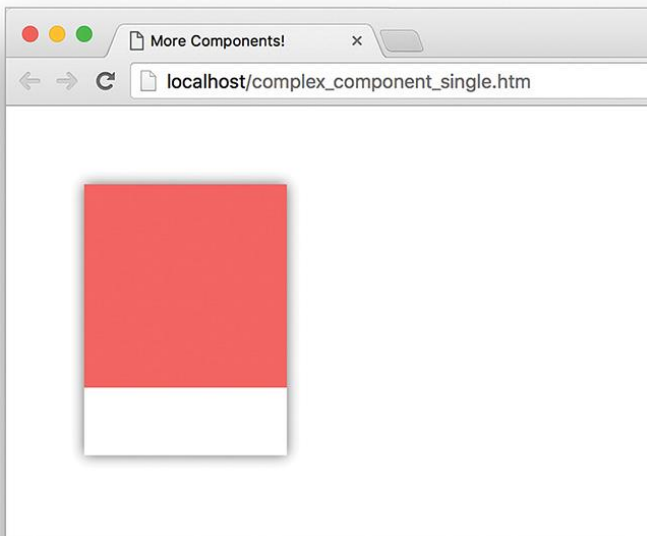


Figure 5-10 The red portion appears.

EXPLANATION: The cool thing to call out is that we called our Square component from inside the Card component! This is an example of component composability where one component relies on the output of another component. The final thing you see is the result of these two components colluding with each other.

TASK 5: The Label Component

1. The last component that remains is our Label. Go ahead and make the following **highlighted** changes:

```
var Label = React.createClass({
  render: function() {
    var labelStyle = {
      fontFamily: "sans-serif",
      fontWeight: "bold",
      padding: 13,
      margin: 0
    };

    return (
      <p style={labelStyle}>#FF6663</p>
    );
  }
});
```

EXPLANATION: We have a style object that we assign to what we return. What we return is a p element whose content is the string #FF6663.

2. To have what we return ultimately make it to our DOM, we need to call our Label component via our Card component. Go ahead and make the following **highlighted** change:

```
var Card = React.createClass({
  render: function() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      WebkitFilter: "drop-shadow(0px 0px 5px #666)",
      filter: "drop-shadow(0px 0px 5px #666)"
    };

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

EXPLANATION: Notice that our Label component lives just under the Square component we added to our Card component's return function earlier.

3. If you preview your app in the browser now, you should see something that looks like Figure 5-11.

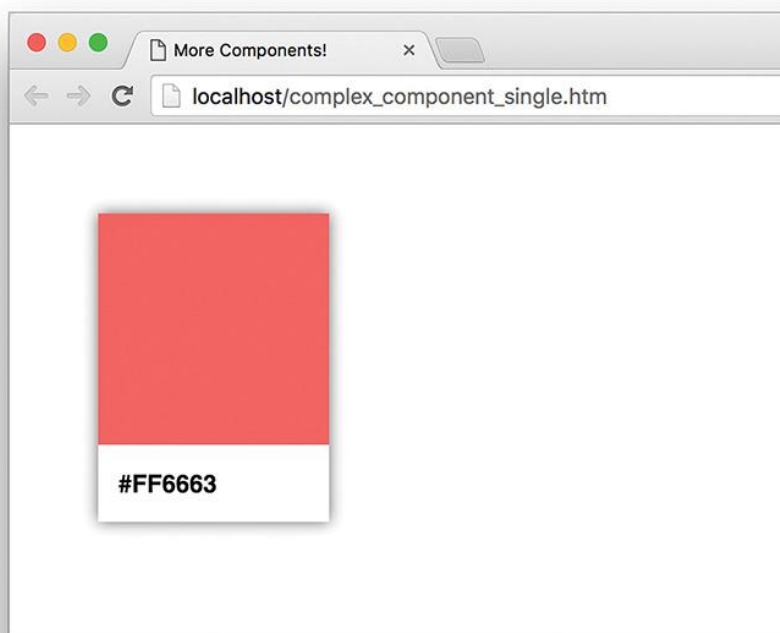


Figure 5-11 The label appears.

EXPLANATION: Our color palette card is done and visible, thanks to the efforts of our Card, Square, and Label components. That doesn't mean we are done yet, though. There are a few more things to cover.

TASK 6: Passing Properties, Again!

In our current code, we hard-coded the color value that is used by our Square and Label components. That is an odd thing to do, but fixing it is straightforward. It just involves us specifying a property name and accessing it via **this.props**. We've seen all this before. What is different is the number of times we will have to do this.

There is no way to properly specify a property on a parent component and have all descendants automatically gain access to that property.

1. **The proper way to pass a property value to a child component is to have each intermediate parent component pass on the property as well.**
 - a. **To see this in action, take a look at the **highlighted** changes to our current code where we move away from a hard-coded color and define our card's color using a color property instead:**

```
var Square = React.createClass({
  render: function() {
    var squareStyle = {
      height: 150,
      backgroundColor: this.props.color
    };
    return(
      <div style={squareStyle}>

        </div>
      );
  }
});

var Label = React.createClass({
  render: function() {
    var labelStyle = {
      fontFamily: "sans-serif",
      fontWeight: "bold",
      padding: 13,
      margin: 0
    };

    return (
      <p style={labelStyle}>{this.props.color}</p>
    );
  }
});

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



```

var cardStyle = {
  height: 200,
  width: 150,
  padding: 0,
  backgroundColor: "#FFF",
  WebkitFilter: "drop-shadow(0px 0px 5px #666)",
  filter: "drop-shadow(0px 0px 5px #666)"
};

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

ReactDOM.render(
  <div>
    <Card color="#FF6663"/>
  </div>,
  document.querySelector("#container")
);

```

2. Once you have made this change, you can specify any hex color you want as part of calling the Card component:

```

ReactDOM.render(
  <div>
    <Card color="#FFA737"/>
  </div>,
  document.querySelector("#container")
);

```

3. The resulting color palette card will feature the color you specified (see Figure 5-12).



Figure 5-12 The color for hex value #FFA737.

EXPLANATION: Now, let's go back to the changes we made. Even though the color property is only consumed by the Square and Label components, the parent Card component is responsible for passing the property on to them.

For even more deeply nested situations, you'll have more intermediate components that will be responsible for transferring properties. It gets worse. When you have multiple properties that you would like to pass around multiple levels of components, the amount of typing (or copying/pasting) you do increases a lot as well.

There are ways to mitigate this, and we'll look at those mitigation strategies in much greater detail in a future module.

TASK 7: Why Component Composability Rocks

When we are heads-down in React, we often tend to forget that what we are ultimately creating is just plain old HTML, CSS, and JavaScript.

1. INFORMATION ONLY: The generated HTML for our color palette card looks as follows:

```
<div id="container">
  <div data-reactid=".0">
    <div style="height:200px;
      width:150px;
      padding:0;
      background-color:#FFF;
      -webkit-filter:drop-shadow(0px 0px 5px #666);
      filter:drop-shadow(0px 0px 5px #666);">
      <div style="height:150px;
        background-color:#FF6663;"></div>
      <p style="font-family:sans-serif;
        font-weight:bold;
        padding:13px;
        margin:0;">#FF6663</p>
    </div>
  </div>
</div>
```

SIMPLE EXPLANATION:

This markup has no idea of how it got there. It doesn't know about which components were responsible for what. It doesn't care about component composability or the frustrating way we had to transfer the color property from parent to child. That brings up an important point to make.

If we had to generalize the end result of what components do, **all they do is return blobs of HTML to whatever called it**. Each component's render function returns some HTML to another component's render function. All of this HTML keeps accumulating until a giant blob of HTML is pushed (very efficiently) to our DOM. That simplicity is why component re-use and composability works so well. Each blob of HTML works independently from other blobs of HTML—especially if you specify inline styles as React recommends. This enables you to easily create visual elements from other visual elements without having to worry about anything. ANYTHING!

Conclusion

As you may have realized by now, we are slowly shifting focus towards the more advanced scenarios that React thrives in. Actually, advanced isn't the right word. The correct word is *realistic*. In this module, we started by learning how to look at a piece of UI and identify the components in a way that you can later implement. That is a situation you will find yourself in all the time. While the approach we employed seemed really formal, as you get more experienced with creating things in React, you can ratchet down the formality. If you can quickly identify the components and their parent/child relationships without creating a visual and component hierarchy, then that is one more sign that you are getting really good at working with React!

Identifying the components is only one part of the equation. The other part is bringing those components to life. Most of the technical stuff we saw here was just a minor extension of what we've already seen earlier. We looked at one level of components in an earlier module, and here we looked at how to work with multiple levels of components. We looked at how to pass properties between one parent and one child in an earlier module, and here we looked at how to pass properties between multiple parents and multiple children. Maybe in a future module we'll do something groundbreaking like drawing multiple color palette cards to the screen! Or, we can maybe specify two properties instead of just a single one.

Lab 5: Transferring Properties (Props) (30 minutes)

There is a frustrating side to working with properties. We partially saw this side in the previous module. Passing properties from one component to another is nice and simple when you are dealing with only one layer of components. When you wish to send a property across multiple layers of components, things start getting complicated.

Things getting complicated is never a good thing, so in this module, let's see what we can do to make working with properties across multiple layers of components easy.

TASK 1: Problem Overview -**INFORMATION ONLY**

1. Let's say that you have a deeply nested component, and its hierarchy (modeled as colored circles) looks like Figure 6-1.

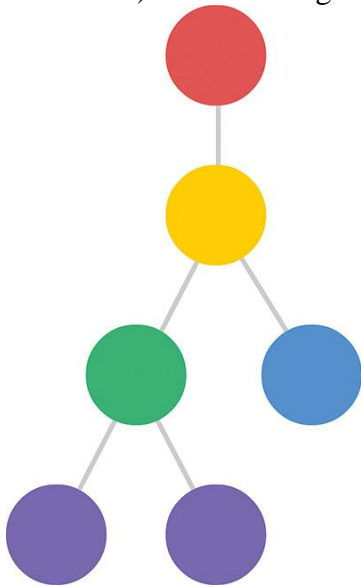


Figure 6-1 The component hierarchy.

2. What you want to do is pass a property from your red circle all the way down to our purple circles where it will be used. What we **can't do** is the very obvious and straightforward thing shown in Figure 6-2.

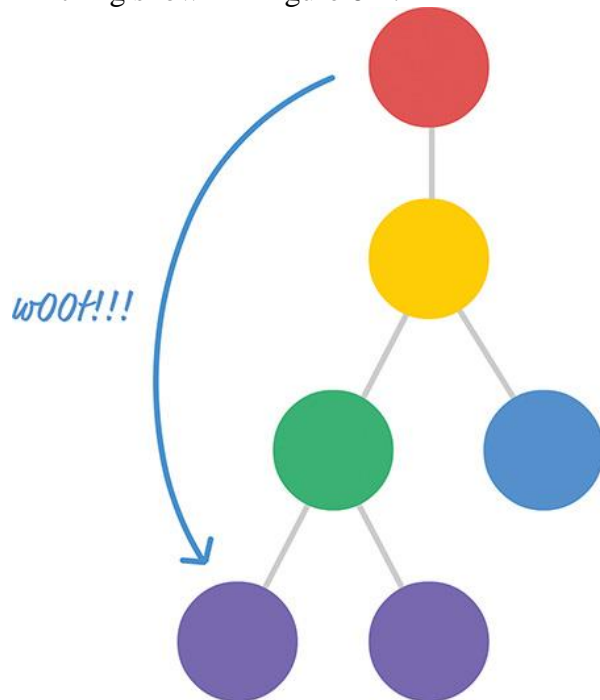


Figure 6-2 Can't do this.

3. You can't pass a property directly to the component or components that you wish to target. The reason has to do with how React works. React enforces a chain of command where properties have to flow down from a parent component to an immediate child component. This means you can't skip a layer of children when sending a property. This also means your children can't send a property back up to a parent. All communication is one-way from the parent to the child.
 - a. Under these guidelines, passing a property from our red circle to our purple circle looks a little bit like Figure 6-3.

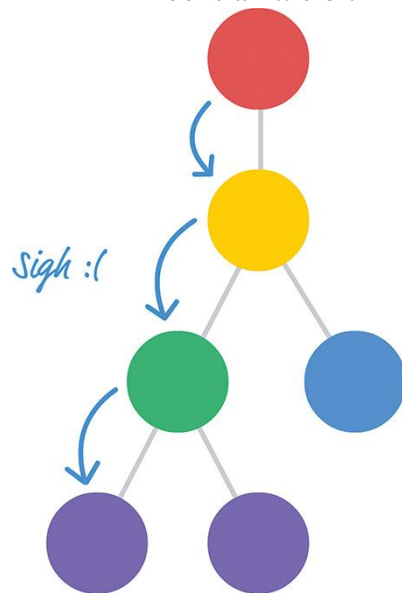


Figure 6-3 The property is passed from parent to child.

4. Every component that lies on the intended path has to receive the property from its parent and then re-send that property to its child. This process repeats until your property reaches its intended destination. The problem is in this receiving and re-sending step.
 - a. If we had to send a property called `color` from the component representing our red circle to the component representing our purple circle, its path to the destination would look something like Figure 6-4.

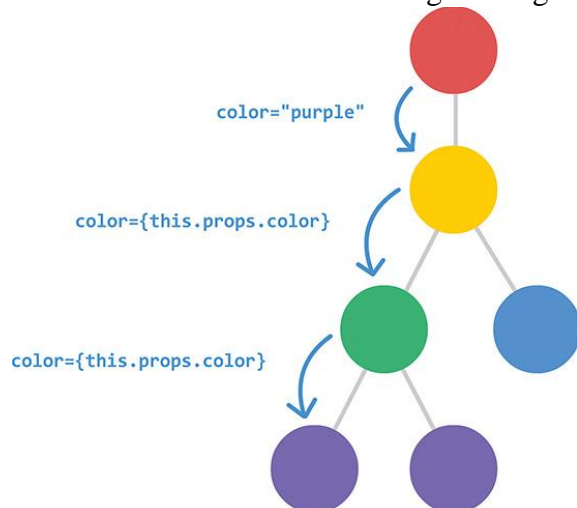


Figure 6-4 Sending the color property.

5. Now, imagine we have two properties that we need to send, as in Figure 6-5.

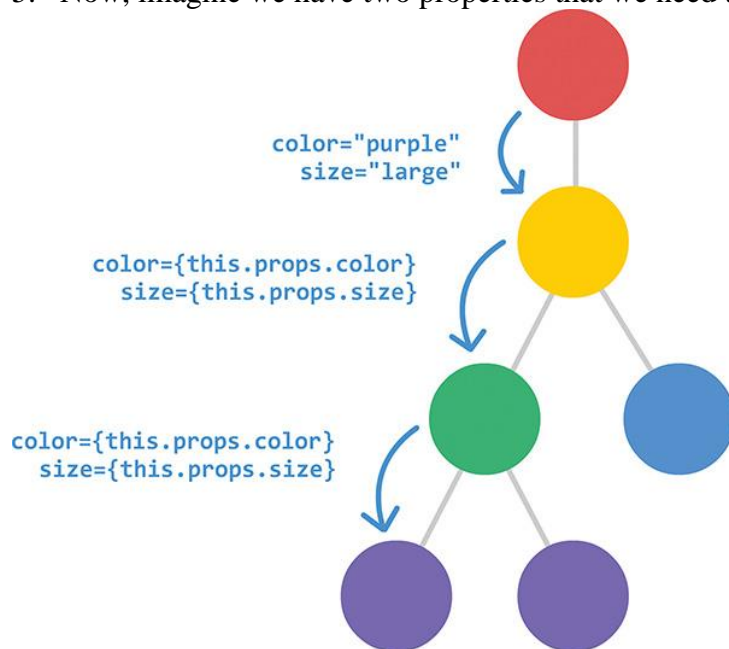


Figure 6-5 Sending two properties.

6. What if we wanted to send three properties? Or four?
7. We can quickly see that this approach is neither scalable nor maintainable. For every additional property we need to communicate, we are going to have to add an entry for it as part of declaring each component. If we decide to rename our properties at some point, we will have to ensure that every instance of that property is renamed as well. If we remove a property, we need to remove the property from every component that relied on it. Overall, these are the kinds of situations we try to avoid when writing code. What can we do about this?

TASK 2: Detailed Look at the Problem

In the previous TASK, we talked at a high level about what the problem is. Before we can dive into figuring out a solution, we need to go beyond diagrams and look at a more detailed example with real code.

1. We need to create a component to take a look at something like the following:

```
var Display = React.createClass({
  render: function() {
    return(
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
  }
});

var Label = React.createClass({
  render: function() {
    return (
      <Display color={this.props.color}
        num={this.props.num}
        size={this.props.size}/>
    );
  }
});

var Shirt = React.createClass({
  render: function() {
    return (
      <div>
        <Label color={this.props.color}
          num={this.props.num}
          size={this.props.size}/>
      </div>
    );
  }
});

ReactDOM.render(
  <div>
    <Shirt color="steelblue" num="3.14" size="medium"/>
  </div>,
  document.querySelector("#container")
);
```

- a. Take a few moments to understand what is going on. Once you have done that, let's walk through this example together.
- b. What we have is a **Shirt** component that relies on the output of the **Label** component which relies on the output of the **Display** component.

2. The component hierarchy can be seen in Figure 6-6.

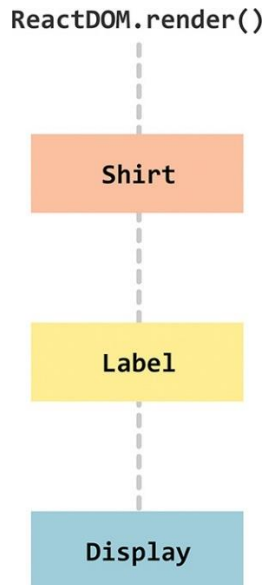


Figure 6-6 The component hierarchy.

3. When you run this code, what gets output is nothing special. It is just three lines of text (see Figure 6-7).

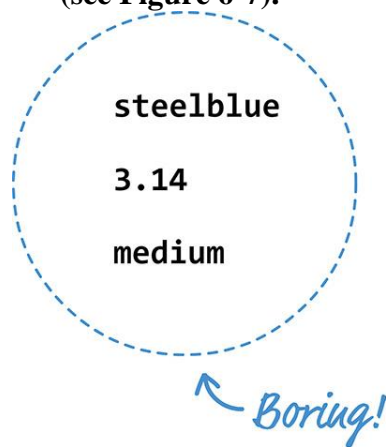


Figure 6-7 The three lines of text.

4. **INFORMATION ONLY:** The interesting part is how the text gets there. Each of the three lines of text that you see maps to a property we specified at the very beginning inside `ReactDOM.render`:

```
<Shirt color="steelblue" num="3.14" size="medium"/>
```

EXPLANATION: The color, num, and size properties (and their values) make a journey all the way to the Display component that would make even the most seasoned world traveler jealous. Let's follow these properties from their inception to when they get consumed.

- a. Life for our properties starts inside `ReactDOM.render` when our `Shirt` component gets called with the color, num, and size properties specified:

```
ReactDOM.render(  
  <div>  
    <Shirt color="steelblue" num="3.14" size="medium"/>  
  </div>,  
  document.querySelector("#container")  
);
```

EXPLANATION: We not only define the properties, we also initialize them with the values they will carry. Inside the `Shirt` component, these properties are stored inside the props object. To transfer these properties on, we need to explicitly access these properties from the props object and list them as part of the component call.

- b. The following is an example of what that looks like when our `Shirt` component calls our `Label` component:

```
var Shirt = React.createClass({  
  render: function() {  
    return (  
      <div>  
        <Label color={this.props.color}  
          num={this.props.num}  
          size={this.props.size}/>  
      </div>  
    );  
  }  
});
```

EXPLANATION: Notice that the color, num, and size properties are listed again. The only difference from what we saw with the `ReactDOM.render` call is that the values for each property are taken from their respective entry in the props object as opposed to being manually entered. When our `Label` component goes live, it has its props object properly filled out with the color, num, and size properties stored.

5. The Label component continues the tradition by repeating the same steps and calling the Display component:

```
var Label = React.createClass({  
  render: function() {  
    return (  
      <Display color={this.props.color}  
        num={this.props.num}  
        size={this.props.size}/>  
    );  
  }  
});
```

PROBLEM: All we wanted to do was have our Display component display some values for color, num, and size. The only complication was that the values we wanted to display were originally defined as part of ReactDOM.render. The annoying solution is the one you see here where every component along the path to the destination needs to access and re-define each property as part of passing it along. That's just terrible. We can do better than this, and you will see how in a few moments!

TASK 3: Properly Transferring Properties

1. **INFORMATION ONLY:** The situation we are facing with transferring properties across components is very similar to our problem of accessing each array item individually. Allow me to elaborate.

Inside a component, our props object looks as follows:

```
var props = {  
  color: "steelblue",  
  num: "3.14",  
  size: "medium"  
}
```

- a. As part of passing these property values to a child component, we manually access each item from our props object:

```
<Display color={this.props.color}  
  num={this.props.num}  
  size={this.props.size}/>
```

- b. Wouldn't it be great if there was a way to unwrap an object and pass on the property/value pairs just like we were able to unwrap an array using the spread operator?

2. **INFORMATION ONLY:** As it turns out, there is a way. It actually involves the spread operator as well. I'll explain how later, but what this means is that we can call our Display component by using **...props**:

```
<Display {...props}/>
```

3. By using **...props**, the runtime behavior is the same as specifying the color, num, and size properties manually. This means our earlier example can be simplified as follows (pay attention to the **highlighted** lines):

```
var Display = React.createClass({
  render: function() {
    return(
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
  }
});

var Label = React.createClass({
  render: function() {
    return (
      <Display {...this.props}/>
    );
  }
});

var Shirt = React.createClass({
  render: function() {
    return (
      <div>
        <Label {...this.props}/>
      </div>
    );
  }
});

ReactDOM.render(
  <div>
    <Shirt color="steelblue" num="3.14" size="medium"/>
  </div>,
  document.querySelector("#container")
);
```

4. If you run this code, the end result is going to be unchanged from what we had earlier.

EXPLANATION: 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 where you define your property. You make another change at the spot you consume the property. That's it. All of the intermediate components that merely transfer the properties on will remain untouched, for the `{...this.props}` expression contains no details of what goes on inside it.

Conclusion

As designed by the ES6/ES2015 committee, the spread operator is designed to work only on arrays and array-like creatures (aka that which has a `Symbol.iterator` property). The fact that it works on object literals like our `props` object is due to React extending the standard. As of now, no browser currently supports using the spread operator on object literals. The reason our example works is because of Babel. Besides turning all of our JSX into something our browser understands, Babel also turns cutting-edge and experimental features into something cross-browser friendly. That is why we are able to get away with using the spread operator on an object literal, and that is why we are able to elegantly solve the problem of transferring properties across multiple layers of components!

Lab 6: Meet JSX—Again! (30 minutes)

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

JSX Quirks to Remember

As we've been working with JSX, you probably noticed that we ran into some arbitrary rules and exceptions to what you can and can't do. In this section, let's put all of those quirks together in one area and maybe even run into some new ones!

Task 1: You Can Only Return A Single Root Node (**chapter4.html**)

1. Try entering the following code in a copy of a previous lab (**chapter4.html**). In JSX, what you return or render can't be made up of multiple root elements:

```
ReactDOM.render(  
  <Letter>B</Letter>  
  <Letter>E</Letter>  
  <Letter>I</Letter>  
  <Letter>O</Letter>  
  <Letter>U</Letter>,  
  document.querySelector("#container")  
);
```

2. Now try this code. If you want to do something like this, you need to wrap all of your elements into a single parent element first:

```
ReactDOM.render(  
  <div>  
    <Letter>A</Letter>  
    <Letter>E</Letter>  
    <Letter>I</Letter>  
    <Letter>O</Letter>  
    <Letter>U</Letter>  
  </div>,  
  document.querySelector("#container")  
);
```

EXPLANATION: This seemed like a bizarre requirement when we looked at it before, but you can blame `createElement` for why we do this. With the `render` and `return` functions, what you are ultimately returning is a single `createElement` call (which in turn might have many nested `createElement` calls). Here is what our earlier JSX looks like when turned into JavaScript:

```
ReactDOM.render(React.createElement(
  "div",
  null,
  React.createElement(
    Letter,
    null,
    "A"
  ),
  React.createElement(
    Letter,
    null,
    "E"
  ),
  React.createElement(
    Letter,
    null,
    "I"
  ),
  React.createElement(
    Letter,
    null,
    "O"
  ),
  React.createElement(
    Letter,
    null,
    "U"
  )
), document.querySelector("#container"));
```

Having multiple root elements would break how functions return values and how `createElement` works, so that is why you can specify only one return (root) element!

TASK 2: You Can't Specify CSS Inline (**chapter4.html**)

As we saw in Module 4, the style attribute in your JSX behaves differently from the style attribute in HTML.

1. In the copy of the **chapter4.html**, try to specify CSS properties directly as values on your style attribute:

```
<div style="font-family:Arial;font-size:24px">
  <p>Blah!</p>
</div>
```

2. Fix it by using JSX where the style attribute can't contain CSS inside it. Instead, it needs to refer to an object that contains styling information instead:

```
var Letter = React.createClass({
  render: function() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: this.props.bgcolor,
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
      fontSize: "32",
      textAlign: "center"
    };

    return (
      <div style={letterStyle}>
        {this.props.children}
      </div>
    );
  }
});
```

EXPLANATION: Notice that we have an object called `letterStyle` that contains all of the CSS properties (in camelCase JavaScript form) and their values. That object is what we then specify to the style attribute.

TASK 3: Reserved Keywords and className (chapter7.html)

1. Take a look at the following code and try to run in Chrome:

```
ReactDOM.render(  
  <div class="slideIn">  
    <p class="emphasis">Gabagool!</p>  
    <Label/>  
  </div>,  
  document.querySelector("#container")  
);
```

2. Change the “class” to “className” to use the DOM API version of the class attribute called **className** instead:

```
ReactDOM.render(  
  <div className="slideIn">  
    <p className="emphasis">Gabagool!</p>  
    <Label/>  
  </div>,  
  document.querySelector("#container")  
);
```

EXPLANATION: You can see the full list of supported tags and attributes at the following Facebook article (<https://facebook.github.io/react/docs/tags-and-attributes.html>), and notice that all of the attributes are camelCase. That detail is important, for using the lowercase version of an attribute won't work. If you are ever pasting a large chunk of HTML that you want JSX to deal with, be sure to go back to your pasted HTML and make these minor adjustments to turn it into valid JSX.

TASK 4: Comments (**chapter7.html**) – **INFORMATION ONLY**

Just like it is a good idea to comment your HTML, CSS, and JavaScript, it is a good idea to provide comments inside your JSX as well. Specifying comments in JSX is very similar to how you would comment in JavaScript ...except for one exception.

1. **If you are specifying a comment as a child of a tag, you need to wrap your comment by the { and } curly brackets to ensure it is parsed as an expression:**

```
ReactDOM.render(  
  <div class="slideIn">  
    <p class="emphasis">Gabagool!</p>  
    { /* I am a child comment */ }  
    <Label/>  
  </div>,  
  document.querySelector("#container")  
);
```

2. **Our comment in this case is a child of our div element. If you specify a comment wholly inside a tag, you can just specify your single-or multi-line comment without having to use the { and } angle brackets:**

```
ReactDOM.render(  
  <div class="slideIn">  
    <p class="emphasis">Gabagool!</p>  
    <Label  
      /* This comment  
         goes across  
         multiple lines */  
      className="colorCard" // end of line  
    />  
  </div>,  
  document.querySelector("#container")  
);
```

In this snippet, you can see an example of what both multi-line comments and a comment at the end of a line look like.

TASK 5: Capitalization, HTML Elements, and Components

INFORMATION ONLY

1. Capitalization is important. To represent HTML elements, ensure the HTML tag is lower-case:

```
ReactDOM.render(  
  <div>  
    <section>  
      <p>Something goes here!</p>  
    </section>  
  </div>,  
  document.querySelector("#container")  
);
```

2. When wishing to represent components, the component name must be capitalized, both in JSX as well as when you define them:

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

EXPLANATION: If you get the capitalization wrong, React will not render your content properly. The component will not be found. Trying to identify capitalization issues is probably the last thing you'll think about when things aren't working, so keep this little tip in mind.

Conclusion

With this module, we've finally pieced together in one location the various bits of JSX information that the previous modules introduced. The most important thing to remember is that JSX is not HTML. It looks like HTML and behaves like it in many common scenarios, but it is ultimately designed to be translated into JavaScript. This means you can do things that you could never imagine doing using just plain HTML. Being able to evaluate expressions or programmatically manipulate entire chunks of JSX is just the beginning. In upcoming modules, we'll explore this intersection of JavaScript and JSX further.

Lab 7: Dealing with State (30 minutes)

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 lab you learn all about it and how you can use it to create stateful components.

TASK 1: Using State

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



Figure 8-1 The app you will be building.

- a. 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.

TASK 2: Our Starting Point

The primary focus of this lab is to see how we can work with state. There is no point in us spending a bunch of time creating the lab from scratch and retracing paths that we've walked many times already.

1. **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>
        );
      }
    });
  </script>

```

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

- a. **INFORMATION ONLY:** 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>
    );
  }
});
```

EXPLANATION: 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.

- b. **INFORMATION ONLY:** The `LightningCounter` component is where all the action is going to be taking place:

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

EXPLANATION: 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.

- c. **INFORMATION ONLY:** The last thing to look at is our **ReactDOM.render** method:

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

EXPLANATION: 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.

TASK 3: Turning 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.

1. Setting the Initial State Value

- a. 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>
    );
  }
});
```

EXPLANATION: The `getInitialState` method automatically runs waaaaay 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.

2. 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
}
```

3. 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>
    );
  }
});
```

EXPLANATION: What we've done is replaced our default Hello! text with an expression that displays the value stored by the this.state.strikes property.

4. If you preview your example in the browser, you will see a value of 0 displayed. That's a start!

TASK 4: 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.

1. 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>
    );
  }
});
```

- a. 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).

2. 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>
    );
  }
});
```

EXPLANATION: 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.

TASK 5: Rendering the State Change

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

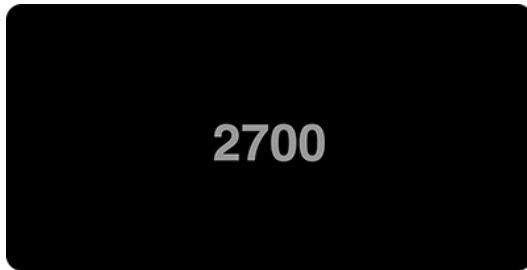


Figure 8-2 The strikes value increments by 100 every second.

EXPLANATION: 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 is 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.

INFORMATION ONLY: 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")
);

```

EXPLANATION: 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 module, 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!

Lab 8: Going from Data to UI (30 minutes)

When you are building your apps, thinking in terms of props, state, components, JSX tags, render methods, and other React-isms may be the last thing on your mind. Most of the time, you are 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! This module helps reduce some of those frustrating moments by running through some common scenarios you'll encounter!

TASK 1: The Example

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

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

```

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

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

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

</html>

```

2. Once you have your document set up, go ahead and preview what you have in your browser. If everything went well, you will be greeted by a happy yellow circle (see Figure 9-1).

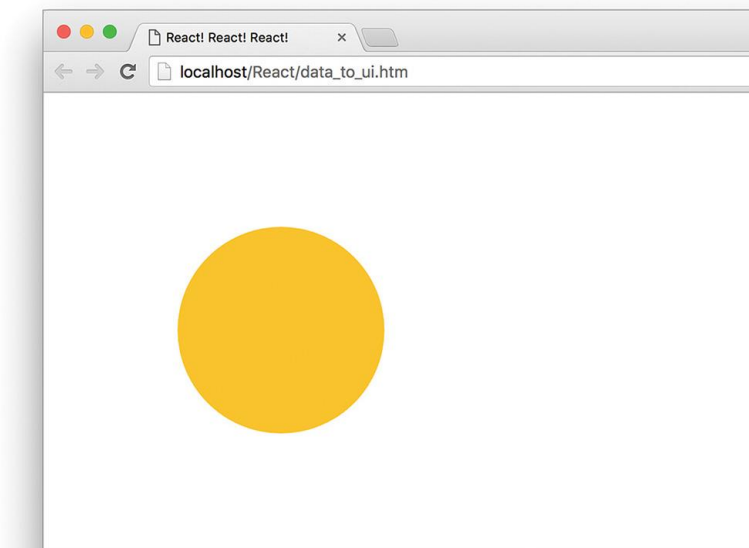


Figure 9-1 If everything went well, you will get this yellow circle.

EXPLANATION OF CODE:

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

```
var Circle = React.createClass({
  render: function() {
    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 is mostly made up of our `circleStyle` object that contains the inline style properties that turn our div into an awesome circle. All the style values are hard-coded except for the `backgroundColor` property. It takes its value from the `bgColor` prop that gets passed in.

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

```
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 in. Now, having our Circle component be defined as-is inside our render method is a bit limiting - especially if you are going to be dealing with data that could affect what our Circle component does. In the next couple of sections, we'll look at the ways we have for solving that.

TASK 2: Your JSX Can Be Anywhere—Part II

In the “Meet JSX—Again”! module (Module 7), we learned that our JSX can actually live outside of a render function and can be used as a value assigned to a variable or property.

1. For example, we can fearlessly do something like this:

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

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

EXPLANATION: 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 having our Circle component instantiation freed from the shackles of the render method gives us more options.

2. You can go further and create a function that returns a Circle component:

```
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 with the value for the bgColor prop set to a random color value (awesomesauce!).

3. To have our example use the showCircle function, all you have to do is evaluate it inside ReactDOM.render:

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

TASK 3: Dealing with Arrays in the Context of JSX

Now we are going to get to some fun stuff! When you are displaying multiple components, you won't always be able to manually specify them:

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

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

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

What we want to do is create a `Circle` component for each item in this array (and set the `bgColor` prop to the value of each array item).

1. The way we are going to do this is by creating an array of `Circle` components:

```
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]}/>);  
}
```

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

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

3. 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 seeing something that looks like Figure 9-2 when you preview in your browser.

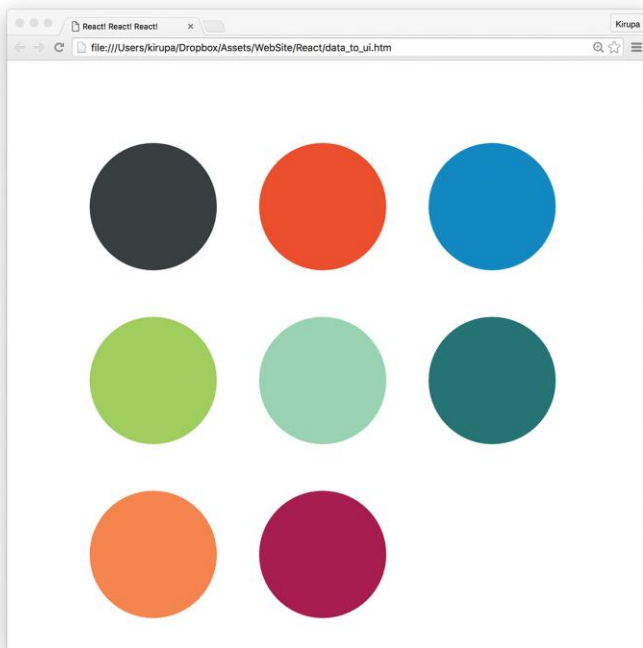


Figure 9-2 What you should see in your browser.

Ok, while our example seems to work, we aren't done yet! There is actually one more thing we need to do, and this is a subtle one. The way React makes UI updates really fast is 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. This "marking" happens automatically when you explicitly specify elements in your JSX.

When you create elements dynamically (such as what we are 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.

4. For our example, we can do something like this:

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

EXPLANATION: 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. Now, we could just use the index position as the identifier, but if you have multiple blocks of code where you are dynamically generating elements, you may get multiple elements with duplicate index values.

Conclusion

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

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

Even though we are 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:

```
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 with our data and still get the end result we want! Because, in the end, it's all just JavaScript.

Lab 9: Working with Events (45 minutes)

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.

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.

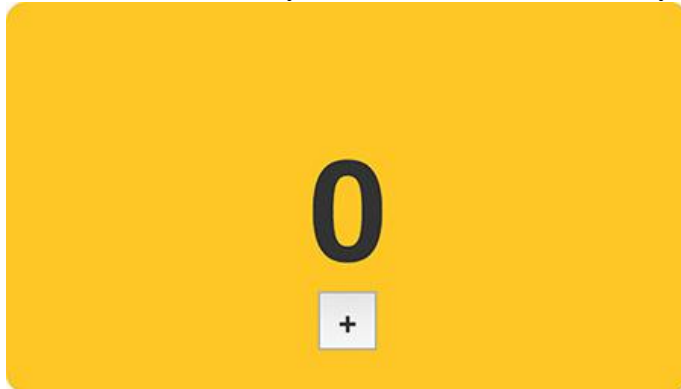


Figure 10-1 Our example.

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 almost 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 things to get the counter to update when this event gets overheard.

TASK 1: Starting Point

To save all of us some time, we aren't going to be creating everything in our example from scratch. 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.

1. 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>
```

2. 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 = {
            fantasize: "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  
)
```

3. **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.**
 - a. **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.**

TASK 2: 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.

1. 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>  
);
```

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

2. 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>
    );
  }
});
```

EXPLANATION: 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.

- 3. 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!**

TASK 3: 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. 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. Each Event type contains its own set of properties that you can access via the event handler for that event!

1. 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  
  });  
},
```

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

Explanation of Code:

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.

TASK 4: You Can't Directly Listen to Events on Components

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.

1. 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>
    );
  }
});
```

EXPLANATION: In this example, we create a property called **clickHandler** whose value is the increase event handler.

2. Inside our PlusButton component, we can then do something like this:

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

Explanation of Code:

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 code!

TASK 5: Listening to Regular DOM Events

1. **INFORMATION ONLY:** 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.

2. **INFORMATION ONLY:** 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>
    );
  }
});
```


3. **INFORMATION ONLY:** 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>
    );
  }
});
```

4. **INFORMATION ONLY:** 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.**

(OPTIONAL) TASK 6: The Meaning of this Inside the Event Handler

1. **INFORMATION ONLY:** 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);
```

2. 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
 - a. Use the following code example to change how the “increase” button event is handled:

```
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>  
    );  
  }  
});
```

EXPLANATION: In this code, 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`.

3. **INFORMATION ONLY:** 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>
```

EXPLANATION: 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.

Conclusion

You'll spend a lot of time dealing with events, and this module 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 unique cases that you might also run into.

Lab 10: The Component Lifecycle (15 minutes)

In the beginning, we started off with a very simple view of components and what they do. As we learned more about React and did cooler and more involved things, it turns out our components aren't all that simple. They help deal with properties, state, events, and often are responsible for the well-being of other components as well. Keeping track of everything components do sometimes can be tough.

To help with this, React provides us with something known as lifecycle methods. Lifecycle methods are (unsurprisingly) special methods that automatically get called as our component goes about its business. They notify us of important milestones in our component's life, and we can use these notifications to simply pay attention or change what our component is about to do.

In this lab, we look at these lifecycle methods and learn all about what we can do with them.

TASK 1: See the Lifecycle Methods in Action

Learning about these lifecycle methods is about as exciting as memorizing names for foreign places (or distant star systems!) you have no plans to visit. To help make all of this more bearable, I am going to first have you play with them through a simple example before we get all academic and read about them.

1. To play with this example, go to the following URL:

https://www.kirupa.com/react/lifecycle_example.htm Once this page loads, you'll see a variation of the counter example we saw earlier (see Figure 11-1).

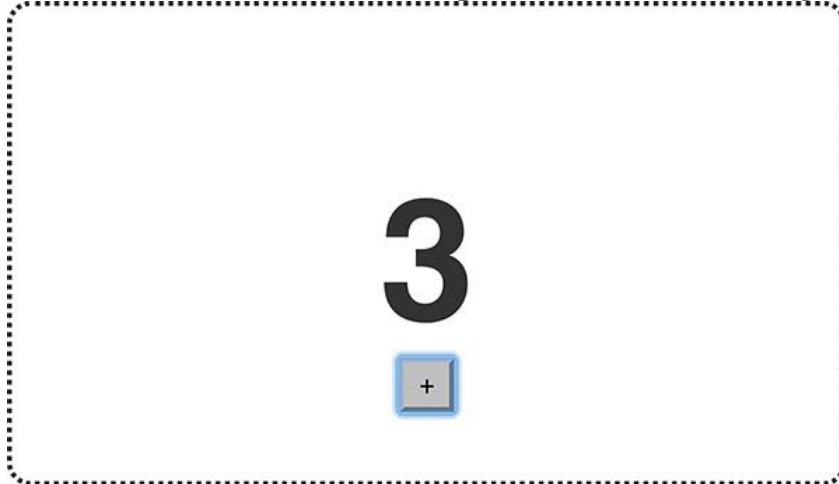


Figure 11-1 A variation on the counter example.

- a. **Don't click on the button or anything just yet.**
 - b. **If you have already clicked on the button, just refresh the page to start the example from the beginning.**
2. Now, bring up your browser's developer tools and take a look at the Console tab. In Chrome, you'll see something that looks like Figure 11-2.

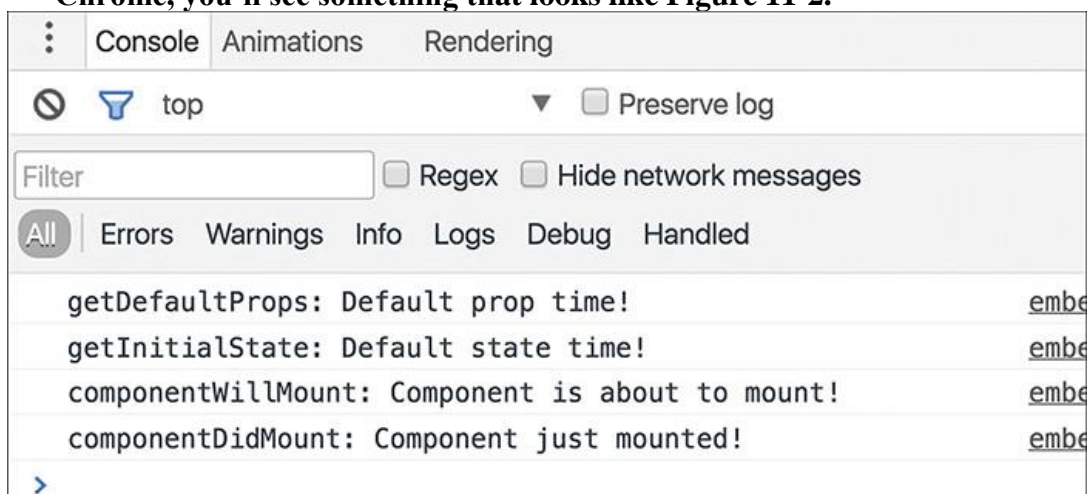


Figure 11-2 The Console view in Chrome.

3. Notice what you see printed.
 - a. You will see some messages, and these messages start out with the name of what looks like a lifecycle method.
 - b. If you click on the plus button once, notice that your Console will show more lifecycle methods getting called (see Figure 11-3).

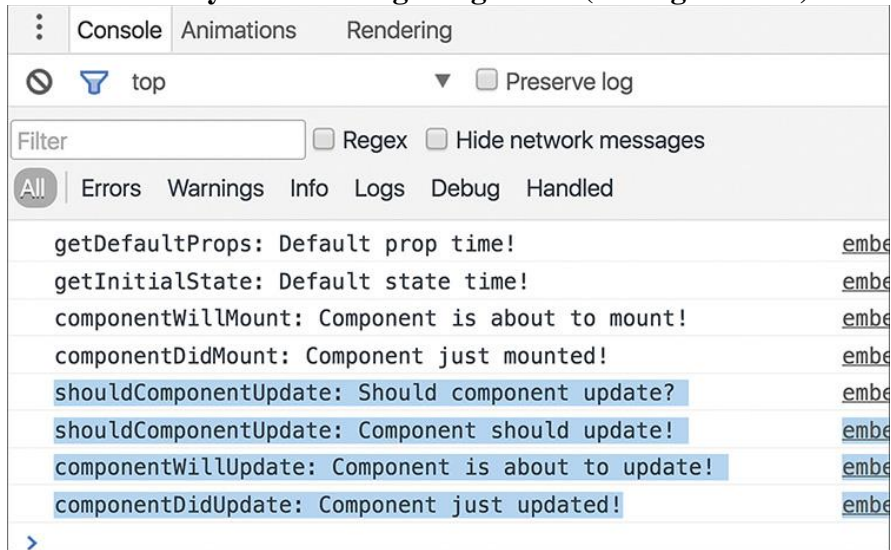


Figure 11-3 More lifecycle methods getting called.

4. Play with this example for a bit. What this example does is allow you to place all of these lifecycle methods in the context of a component that we've already seen earlier.
 - a. As you keep hitting the plus button, more lifecycle method entries will show up.
 - b. Eventually, once your counter approaches a value of 5, your example will just disappear with the following entry showing up in your console:
componentWillUnmount: Component is about to be removed from the DOM!
 - c. At this point, you have reached the end of this example.
 - d. Of course, to start over, you can just refresh the page!

5. Now that you've seen the example, let's take a quick look at the component that is responsible for all of this: [\(chapter11.html\)](#)

```
var CounterParent = React.createClass({
  getDefaultProps: function(){
    console.log("getDefaultProps: Default prop time!");
    return {};
  },
  getInitialState: function() {
    console.log("getInitialState: Default state time!");
    return {
      count: 0
    };
  },
  increase: function() {
    this.setState({
      count: this.state.count + 1
    });
  },
  componentWillUpdate: function(newProps, newState) {
    console.log("componentWillUpdate: Component is about to update!");
  },
  componentDidUpdate: function(currentProps, currentState) {
    console.log("componentDidUpdate: Component just updated!");
  },
  componentWillMount: function() {
    console.log("componentWillMount: Component is about to mount!");
  },
  componentDidMount: function() {
    console.log("componentDidMount: Component just mounted!");
  },
  componentWillUnmount: function() {
    console.log("componentWillUnmount: Component is about to be removed from the
DOM!");
  },
  shouldComponentUpdate: function(newProps, newState) {
    console.log("shouldComponentUpdate: Should component update?");

    if (newState.count < 5) {
      console.log("shouldComponentUpdate: Component should update!");
      return true;
    } else {
      ReactDOM.unmountComponentAtNode(destination);
      console.log("shouldComponentUpdate: Component should not update!");
      return false;
    }
  }
});
```



```

    },
    componentWillReceiveProps: function(newProps){
      console.log("componentWillReceiveProps: Component will get new props!");
    },
    render: function() {
      var backgroundStyle = {
        padding: 50,
        border: "#333 2px dotted",
        width: 250,
        height: 100,
        borderRadius: 10,
        textAlign: "center"
      };

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

```

EXPLANATION OF CODE:

Take a few moments to look what all of this code does. It seems lengthy, but a bulk of it is just each lifecycle method listed with a `console.log` statement defined. Once you've gone through this code, play with the example one more time. Trust me. The more time you spend in the example and figure out what is going on, the more fun you are going to have.

Conclusion

React is constantly watching and notifying your component every time something happens. All of this is done via the lifecycle methods that we spent this entire lab looking at.

Lab 11: Accessing DOM Elements (30 minutes)

There will be times when you want to access properties and methods on an HTML element directly.

To highlight one such situation, take a look at the Colorizer example in Figure 12-1.

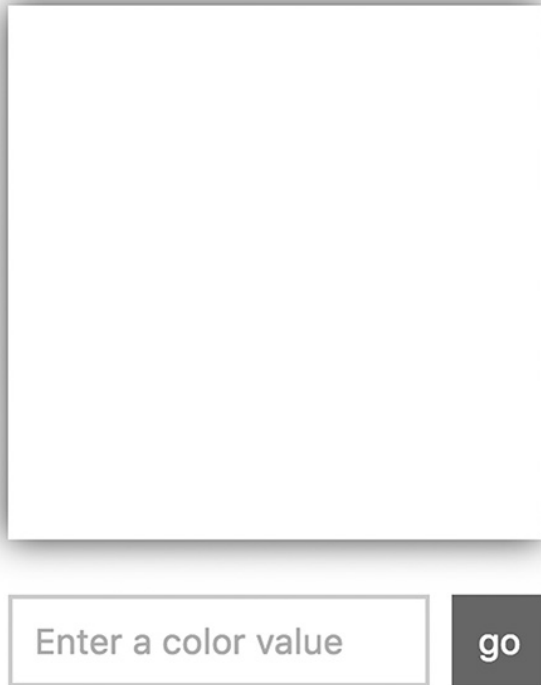


Figure 12-1 Colorizer example.

1. **If you have access to a browser, you can view it live at the following location:**
<https://www.kirupa.com/react/examples/colorizer.htm>

EXPLANATION: The Colorizer colorizes the (currently) white square with whatever color you provide it.

2. To see it in action, enter a color value inside the text field and click/tap on the go button.
 - a. If you don't have any idea of what color to enter, yellow is a good one!
 - b. Once you have provided a color and submitted it, the white square will turn whatever color value you provided (see Figure 12-2).

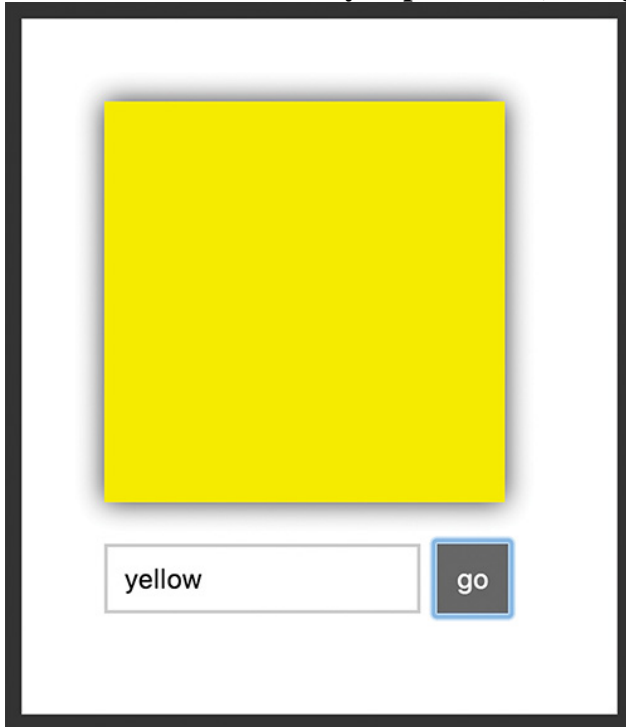


Figure 12-2 The white square turns yellow.

EXPLANATION: That the square changes color for any valid color value you submit is pretty awesome, but it isn't what I want you to focus on. Instead, pay attention to the text field and the button after you submit a value. Notice that the button gets focus, and the color value you just submitted is still displayed inside the form. If you want to enter another color value, you need to explicitly return focus to the text field and clear out whatever current value is present.

3. Now, wouldn't it be great if we could clear both the existing color value and return focus to the text field immediately after you submit a color? That would mean that if we submitted a color value of purple, what we would see afterwards would look like Figure 12-3.

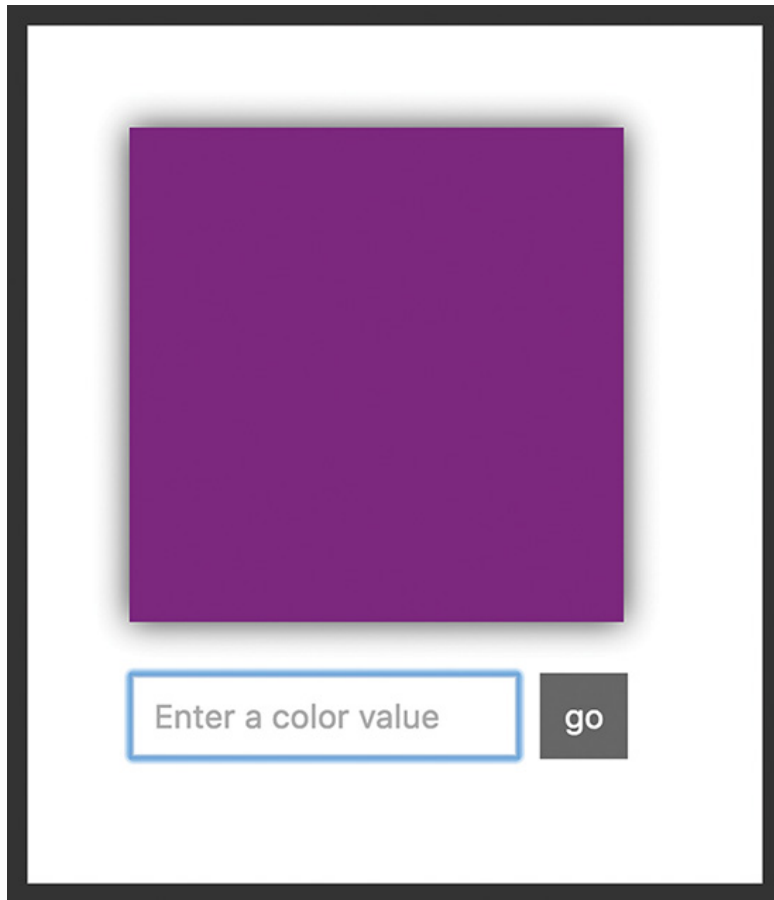


Figure 12-3 We get purple and the text field is ready for the next color.

EXPLANATION: The entered value of purple is cleared, and the focus is returned to the text field. This allows us to enter additional color values and submit them easily without having to manually keep jumping focus back and forth between the text field and the button. Isn't that much nicer?

TASK 1: Meet Refs

As you know very well by now, inside our various render methods, we've been writing HTML-like things known as JSX. Our JSX is simply a description of what the DOM should look like. It doesn't represent actual HTML—despite looking a whole lot like it. Anyway, to provide a bridge between JSX and the final HTML elements in the DOM, React provides us with something known as refs (short for references).

1. 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>
```

2. Let's say we have a render method from our Colorizer example that looks as follows:

```
render: function() {  
  var squareStyle = {  
    backgroundColor: this.state.bgColor  
  };  
  
  return (  
    <div className="colorArea">  
      <div style={squareStyle} className="colorSquare"></div>  
  
      <form onSubmit={this.setNewColor}>  
        <input  
          onChange={this.colorValue}  
          placeholder="Enter a color value">  
        </input>  
        <button type="submit">go</button>  
      </form>  
    </div>  
  );  
}
```

EXPLANATION: Inside this render method, we are returning a big chunk of JSX representing (among other things) the input element where we enter our color value. What we want to do is access the input element's DOM representation so that we can call some APIs on it using JavaScript.

3. The way we reference the input element's DOM using refs is by setting the ref attribute on the element we would like to reference the HTML of:

```
render: function() {  
  var squareStyle = {  
    backgroundColor: this.state.bgColor  
  };  
  
  return (  
    <div className="colorArea">  
      <div style={squareStyle} className="colorSquare"></div>  
  
      <form onSubmit={this.setNewColor}>  
        <input  
          ref={}  
          onChange={this.colorValue}  
          placeholder="Enter a color value">  
        </input>  
        <button type="submit">go</button>  
      </form>  
    </div>  
  );  
}
```

EXPLANATION: Because we are interested in the input element, our ref attribute is attached to it. Right now, our ref attribute is empty. What you typically set as the ref attribute's value is a JavaScript callback function. This function gets called automatically when the component housing this render method gets mounted.

4. If we set our `ref` attribute's value to a simple JavaScript function that stores a reference to the referenced DOM element, it would look something like the following **highlighted** lines:

```
render: function() {  
  var squareStyle = {  
    backgroundColor: this.state.bgColor  
  };  
  
  var self = this;  
  
  return (  
    <div className="colorArea">  
      <div style={squareStyle} className="colorSquare"></div>  
  
      <form onSubmit={this.setNewColor}>  
        <input  
          ref={  
            function(el) {  
              self._input = el;  
            }  
          }  
          onChange={this.colorValue}  
          placeholder="Enter a color value">  
        </input>  
        <button type="submit">go</button>  
      </form>  
    </div>  
  );  
}
```

The end result of this code running once our component mounts is simple: we can access the HTML representing our input element from anywhere inside our component by calling `this._input`. Take a few moments to see how the **highlighted** lines of code help do that. Once you are done, we'll walk through this code together.

Explanation of Code:

First, our callback function looks as follows:

```
function(el) {  
  self._input = el;  
}
```

This anonymous function gets called when our component mounts, and a reference to the final HTML DOM element is passed in as an argument. We capture this argument using the “el” identifier, but you can use any name for this argument that you want.

The body of this callback function simply sets a custom property called “_input” to the value of our DOM element. To ensure we create this property on our component, we use the self variable to create a closure where the this in question refers to our component as opposed to the callback function itself. (Autobinding doesn’t happen automatically this time around!)

5. **Taking a step back and looking at the bigger picture that ties everything together including the render method we just saw, let’s look at the full Colorizer component with all of the ref-related code highlighted:**

```
var Colorizer = React.createClass({  
  getInitialState: function() {  
    return {  
      color: '',  
      bgColor: ''  
    }  
  },  
  colorValue: function(e) {  
    this.setState({color: e.target.value});  
  },  
  setNewColor: function(e){  
    this.setState({bgColor: this.state.color});  
  
    this._input.value = "";  
    this._input.focus();  
  
    e.preventDefault();  
  },  
  render: function() {  
    var squareStyle = {  
      backgroundColor: this.state.bgColor  
    };  
  
    var self = this;  
  
    return (  

```

```

    <div className="colorArea">
      <div style={squareStyle} className="colorSquare"></div>

      <form onSubmit={this.setNewColor}>
        <input
          ref={
            function(el) {
              self._input = el;
            }
          }
          onChange={this.colorValue}
          placeholder="Enter a color value">
        </input>
        <button type="submit">go</button>
      </form>
    </div>
  );
}
});

```

Focusing just on what happens to our input element, when the form gets submitted and the `setNewColor` method gets called, we clear the contents of our input element by calling `this._input.value = ""`. We set focus to our input element by calling `this._input.focus()`. All of our `ref`-related work was simply to enable these two lines where we needed some way to have `this._input` point to the HTML element representing our input element that we define in JSX. Once we figured that out, we just call the `value` property and `focus` method the DOM API exposes on this element.

Simplifying Further with ES6 Arrow Functions

Learning React is hard enough, so we have tried to shy away from forcing you to use ES6 techniques by default. **When it comes to working with the `ref` attribute, using arrow functions to deal with the callback function does simplify matters a bit.** This is one of those cases where we recommend you use an ES6 technique.

As you saw a few moments ago, to assign a property on our component to the referenced HTML element, we did something like this:

```

<input
  ref={
    function(el) {
      self._input = el;
    }
  }
  onChange={this.colorValue}
  placeholder="Enter a color value">
</input>

```

To deal with scoping issues, we created a self variable initialized to this to ensure we created the `_input` property on our component. That seems unnecessarily messy. Using arrow functions, we can simplify all of this down to just the following:

```
<input
  ref={
    (el) => this._input = el
  }
  onChange={this.colorValue}
  placeholder="Enter a color value">
</input>
```

The end result is identical to what we spent all of this time looking at, and because of how arrow functions deal with scope, you can use `this` inside the function body and reference the component without doing any extra work. No need for an outer self variable equivalent!

Conclusion

In this lab, we saw how “easy” it is to access a DOM element directly.

Lab 12: Creating a Single-Page App Using React Router (60 minutes)

Now that you've familiarized yourself with the basics of how to work with React, let's kick things up a few notches. What we are going to do is use React to build a simple, **single-page app**. Single-page apps are different from the more traditional multi-page apps that you see everywhere. The biggest difference is that navigating a single-page app doesn't involve going to an entirely new page. Instead, your pages (commonly known as **views** in this context) typically load inline within the same page as illustrated in Figure 13-1.

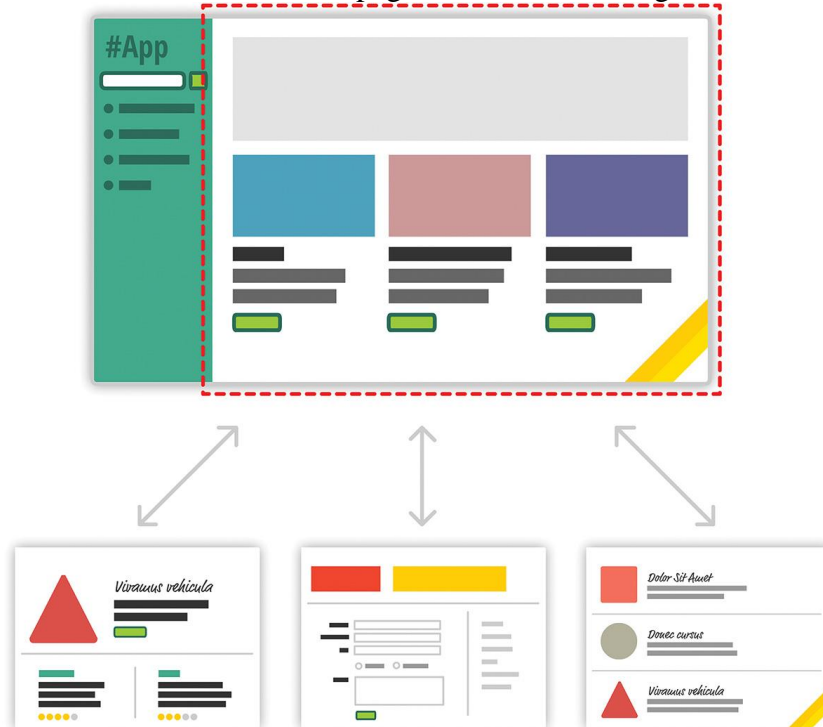


Figure 13-1 Single-page apps use load views inline rather than load new pages.

The Example

Before we go further, let's take a look at the lab solution (see Figure 13-2).

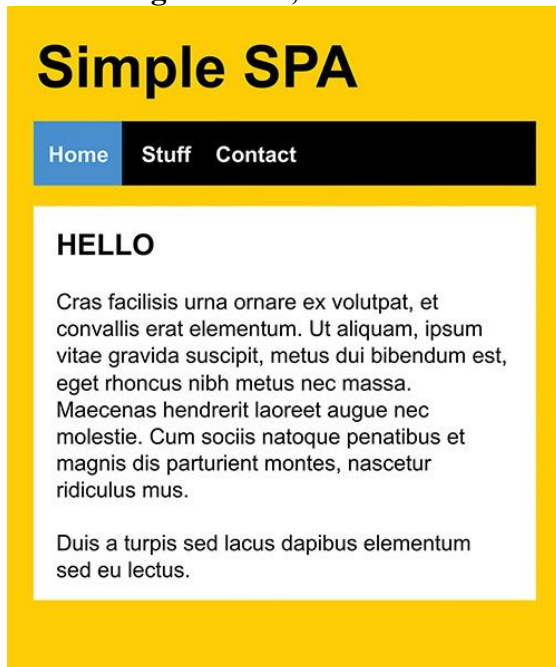


Figure 13-2 A simple React app that uses React Router.

What you have here is a simple React app that uses React Router to provide all of the navigation and view-loading goodness! While the screenshot of the app looks nice and all, this is one of those cases where you want to play with the app to see more of what it does. Go ahead and open this page (https://www.kirupa.com/react/examples/react_router_final.htm) in its own browser window, click on the various navigation tabs to see the different views, and use the back and forward buttons to see them working.

In the following sections, we are going to be building this app in pieces. By the end, not only will you have recreated this app, you'll hopefully have learned enough about React Router to build cooler and more awesome things.

TASK 1: Building the App

1. The first thing we need to do is get the boilerplate markup and code for our app up and running. Create a new HTML document and add the following content into it:

```
<!DOCTYPE html>
<html>

<head>
  <title>React! React! React!</title>
  <script src="https://npmcdn.com/react@15.3.0/dist/react.js"></script>
  <script src="https://npmcdn.com/react-dom@15.3.0/dist/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>

  <style>

  </style>
</head>

<body>

  <div id="container">

  </div>

  <script type="text/babel">
    var destination = document.querySelector("#container");

    ReactDOM.render(
      <div>
        Hello!
      </div>,
      destination
    );
  </script>
</body>

</html>
```

EXPLANATION: This starting point is almost the same as what you've seen for all of our other examples. This is just a nearly blank app that happens to load the React and React-DOM libraries. If you preview what you have in your browser, you'll see a very lonely **Hello!** displayed.

2. Next, because React Router isn't a part of React itself, we need to add a reference to it.
 - a. In our markup, find where we have our existing script references and add the following **highlighted** line:

```
<script src="https://npmcdn.com/react@15.3.0/dist/react.js"></script>  
<script src="https://npmcdn.com/react-dom@15.3.0/dist/react-dom.js"></script>  
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>  
<script src="https://npmcdn.com/react-router/umd/ReactRouter.min.js"></script>
```

EXPLANATION: By adding the highlighted line, we ensure the React Router library is loaded alongside the core React, ReactDOM, and Babel libraries. At this point, we are in a good state to start building our app and taking advantage of the extra functionality React Router brings to the table.

TASK 2: Displaying the Initial Frame

When building a single-page app, there will always be a part of your page that will remain static. This static part, also referred to as an **app frame**, could just be one invisible HTML element that acts as the container for all of your content, or it could include some additional visual tags like a header, footer, navigation, etc.

In our case, our app frame will involve our navigation header and an empty area for content to load in. To display this, we are going to create a component that is going to be responsible for this.

1. Inside your script tag just above your `ReactDOM.render` call, go ahead and add the following chunk of JSX and JavaScript:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li>Home</li>
          <li>Stuff</li>
          <li>Contact</li>
        </ul>
        <div className="content">

      </div>
    </div>
  )
});
```

2. Once you have pasted this, take a look at what we have here. What we have is a component called `App` that returns some HTML.
 - a. To see what this HTML looks like, modify your `ReactDOM.render` call to reference this component instead of displaying the word `Hello`! Go ahead and make the following **highlighted** change:

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

3. Once you have done this, preview your app in the browser. You should see an un-styled version of an app title and some list items (see Figure 13-3).

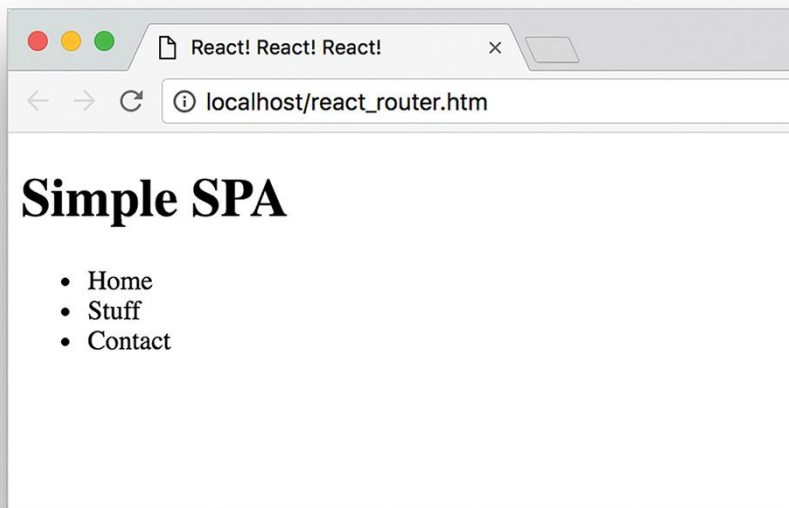


Figure 13-3 Unstyled version.

EXPLANATION: This doesn't look all fancy and styled, but that's OK for now. We will deal with that later. Going a bit deeper, what we've done is just create a component called App and display it via our **ReactDOM.render** call. The important thing to call out is that there is nothing React Router - specific here. Let's fix that by throwing React Router into the mix.

4. Replace the contents of your ReactDOM.render call with the following:

```
ReactDOM.render(  
  <ReactRouter.Router>  
    <ReactRouter.Route path="/" component={App}>  
  
    </ReactRouter.Route>  
  </ReactRouter.Router>,  
  destination  
)
```

5. Ignore how strange everything looks for a moment, and just preview your app in the browser after you've made this change. If everything worked out properly, you will see your App component displayed just like you saw earlier.

EXPLANATION: Now, let's figure out why that is the case by learning more about what exactly is going on here. This is where we deviate a bit from core React concepts and learn things specific to React Router itself.

- a. **INFORMATION ONLY:** First, what we did is specify our Router component:

```
ReactDOM.render(  
  <BrowserRouter>  
    <BrowserRouter.Route path="/" component={ App }>  
  
    </BrowserRouter.Route>  
  </BrowserRouter>  
  ,  
  destination  
);
```

EXPLANATION: The Router component is part of the React Router API, and its job is to deal with all of the routing-related logic our app will need. Inside this component, we specify what is known as the routing configuration. That is a fancy term that people use to describe the mapping between URLs and the views.

- b. **INFORMATION ONLY:** The specifics of that are handled by another component called Route:

```
ReactDOM.render(  
  <BrowserRouter>  
    <BrowserRouter.Route path="/" component={ App }>  
  
    </BrowserRouter.Route>  
  </BrowserRouter>  
  ,  
  destination  
);
```

EXPLANATION: The Route component takes several props that help define what to display at what URL. The path prop specifies the URL we are interested in matching. In this case, it is the root, aka /. The component prop allows you to specify the name of the component you wish to display. For this example, it is our App component. Putting this all together, what this Route says is as follows: If the URL you are on contains the root, go ahead and display the App component. Because this condition is true when you preview your app, you see the result of what happens when your App component renders.

TASK 3: Displaying the Home Page

As you can see, the way React Router provides you with all of this routing functionality is by using concepts in React you are already familiar with—namely components, props, and JSX. What we have right now for displaying our app’s frame is a great example of this. Now, it’s time to go even further. What we want to do next is define the content that we will display as part of our home view.

1. To do this, we are going to create a component called **Home** that is going to contain the markup we want to display.

- a. Just above where you have your **App** component defined, add the following:

```
var Home = React.createClass({
  render: function() {
    return (
      <div>
        <h2>HELLO</h2>
        <p>Cras facilisis urna ornare ex volutpat, et
        convallis erat elementum. Ut aliquam, ipsum vitae
        gravida suscipit, metus dui bibendum est, eget rhoncus nibh
        metus nec massa. Maecenas hendrerit laoreet augue
        nec molestie. Cum sociis natoque penatibus et magnis
        dis parturient montes, nascetur ridiculus mus.</p>

        <p>Duis a turpis sed lacus dapibus elementum sed eu lectus.</p>
      </div>
    );
  }
});
```

EXPLANATION: As you can see, our Home component doesn’t do anything special. It just returns a blob of HTML. Now, what we want to do is display the contents of our Home component when the page loads. This component is the equivalent of our app’s “home page.” The way we do this is simple. Inside our App component, we have a div with a class value of content. We are going to load our Home component inside there.

2. The obvious solution might look something like this:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li>Home</li>
          <li>Stuff</li>
          <li>Contact</li>
        </ul>
        <div className="content">
          <Home/>
        </div>
      </div>
    );
  }
});
```

Notice that we define our Home component inside that content div.

3. If you preview your app, things will even seem to work as expected (see Figure 13-4).



Figure 13-4 Increased functionality.

EXPLANATION: You see your navigation header, and then you see the contents of our Home component. While this approach works, it is actually the wrong thing to do. It is wrong because it complicates our desire to load other pieces of content as the user is navigating around our app. **We've essentially hard-coded our app to only display the Home component.** That's a problem, but we'll come back to that in a little bit.

TASK 4: Interim Cleanup Time

Before we continue making progress on our app, let's take a short break and make some stylistic improvements to what we have so far.

Adding the CSS

Right now, our app looks very plain. To fix this, we are going to rely on CSS.

1. Inside the style tag, go ahead and add the following style rules:

```
body {
  background-color: #FFCC00;
  padding: 20px;
  margin: 0;
}
h1, h2, p, ul, li {
  font-family: Helvetica, Arial, sans-serif;
}
ul.header li {
  display: inline;
  list-style-type: none;
  margin: 0;
}
ul.header {
  background-color: #111;
  padding: 0;
}
ul.header li a {
  color: #FFF;
  font-weight: bold;
  text-decoration: none;
  padding: 20px;
  display: inline-block;
}
.content {
  background-color: #FFF;
  padding: 20px;
}
.content h2 {
  padding: 0;
  margin: 0;
}
.content li {
  margin-bottom: 10px;
}
```

EXPLANATION: Yes, we are using CSS in its markup form. We aren't doing the inline style object approach that we saw in Module 4. The reason has to do with convenience. Our components aren't going to be re-used outside of our particular app, and we really want to take advantage of CSS inheritance to minimize duplicated markup. Otherwise, if we didn't use regular CSS, we'll end up with a bunch of giant style objects defined for almost every element in our markup. That would make even the most patient among us annoyed when reading the code.

2. Once you have added all of this CSS, our app will start to look much better (see Figure 13-5).



Figure 13-5 CSS styling added.

EXPLANATION: There is still some more work to be done (for example, our navigation links disappeared behind the black banner), but we'll fix all of those up in a little bit.

TASK 5: Avoiding the ReactRouter Prefix

We have just one more cleanup related task before we return to our regularly scheduled programming. Have you noticed that every single time we call something defined by the React Router API, we prefix that something with the word `ReactRouter`?

```
<ReactRouter.Router>
  <ReactRouter.Route path="/" component={App}>

    </ReactRouter.Route>
  </ReactRouter.Router>
```

That is a bit verbose to have to repeat for every API call we make, and this is going to be more of a problem as we dive further into the React Router API and use more things from inside it.

1. The fix for this involves using a new ES6 trick where you can manually specify which values will automatically get prefixed.
 - a. Towards the top of your script tag, add the following:

```
var { Router,  
    Route,  
    IndexRoute,  
    IndexLink,  
    Link } = ReactRouter;
```

EXPLANATION: Once you've added this code, every time you use one of the values defined inside the brackets, the prefix **ReactRouter** will automatically be added for you when your app runs.

2. This means, you can now go back to your ReactDOM.render method and **remove** the **ReactRouter** prefix from our Router and Route component instances:

```
ReactDOM.render(  
  <Router>  
    <Route path="/" component={App}>  
  
    </Route>  
  </Router>,  
  destination  
)
```

3. If you preview your app now, nothing really should change. The end result is identical to what you had before. The only difference is that our markup is a bit more compact.

EXPLANATION: Now, before we move on, you are probably wondering why the list of values that will automatically be prefixed with **ReactRouter** contains a whole bunch of things beyond the **Router** and **Route** values that we have used in our code so far. Think of these additional values as a preview of the other parts of the React Router API we will be using shortly.

TASK 6: Displaying the Home Page Correctly

We ended a few sections ago by saying that the way we currently have our home page displayed is incorrect. Although you get the desired result when our page loads, this approach doesn't really make it easy for us to load anything other than the home page when users navigate around. The call to our Home component is hard-coded inside App.

The correct solution involves letting React Router handle which component to call depending on what your current URL structure is. This involves nesting Route components inside Route components to better define the URL-to-view mapping.

1. Go back to our ReactDOM.render method, and make the following **highlighted** change:

```
ReactDOM.render(  
  <Router>  
    <Route path="/" component={App}>  
      <IndexRoute component={Home}/>  
    </Route>  
  </Router>  
,  
destination  
);
```

EXPLANATION: Inside our root Route element, we are defining another Route element of type IndexRoute (more on who this is in a second!) and setting its view to be our Home component.

2. There is one more change we need to make.
 - a. Inside our App component, remove the call to the Home component and replace it with the following **highlighted** line:

```
var App = React.createClass({  
  render: function() {  
    return (  
      <div>  
        <h1>Simple SPA</h1>  
        <ul className="header">  
          <li>Home</li>  
          <li>Stuff</li>  
          <li>Contact</li>  
        </ul>  
        <div className="content">  
          {this.props.children}  
        </div>  
      </div>  
    )  
  }  
});
```

3. **If you preview your page now, you will still see your Home content displayed. The difference this time is that we are displaying the Home content properly in a way that doesn't prevent other content from being displayed instead.**

EXPLANATION: This is because of two things:

1. What gets displayed inside App is controlled by the result of `this.props.children` instead of a hard-coded component.
2. Our Route element inside `ReactDOM.render` contains an `IndexRoute` element whose sole purpose for existing is to declare which component will be displayed when your app initially loads.

All of this may seem even more bizarre than what you expected a few moments ago, but things will make more sense as we use these various APIs more in the following sections.

TASK 7: Creating the Navigation Links

Right now, we just have our frame and home view setup. There isn't really anything else for a user to do here outside of just seeing what we have set as the home page. Let's fix that by creating some navigation links.

1. More specifically, let's linkify the navigation elements we already have:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li>Home</li>
          <li>Stuff</li>
          <li>Contact</li>
        </ul>
        <div className="content">
          {this.props.children}
        </div>
      </div>
    )
  }
});
```

EXPLANATION: If you aren't sure why these elements aren't visible when you preview your page, that's because they blended in with the black background once we added the CSS in. No biggie there. We'll fix that in a few, but first let's talk about how we are going to turn these elements into links.

The way you specify navigation links in React Router isn't by **directly** using the tried and tested `a` tag and throwing in a path via the `href` attribute. Instead, you specify your navigation link using React Router's Link components that are similar to `anchor(a)` tags but offer a lot more functionality.

2. To see the Link component in action, go ahead and modify our existing navigation elements to look like the following **highlighted** lines:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li><Link to="/">Home</Link></li>
          <li><Link to="/stuff">Stuff</Link></li>
          <li><Link to="/contact">Contact</Link></li>
        </ul>
        <div className="content">
          {this.props.children}
        </div>
      </div>
    )
  }
});
```

EXPLANATION: Notice what we have done here. Our Link components specify a prop called `to`. This prop **specifies the value of the URL we will display in the address bar**. Indirectly, it also specifies the location we will be telling React Router we are virtually navigating to. Our Home link takes users to the root (`/`), the Stuff link takes users to a location called **stuff**, and the Contact link takes users to a location called **contact**.

3. If you preview your page and click on the links (which will now be visible because the CSS for them will have kicked in), you won't see anything new to display.
 - a. You will just see your Home content because that is all that we had specified earlier. With that said, you can see the URLs updating in the address bar. You'll see your current page followed by a **`#/contact`, `#/stuff`, or `#/`** depending on which of the links you clicked. Oh, you'll also see a random hash added after the URL.

TASK 8: Adding the Stuff and Contact Views

Our app is slowly taking its final shape...or it will get really close by the time we are done with this section! What we are going to do next is define the components for our Stuff and Contact views that we linked to earlier.

1. In your code, just below where you have your Home component, go ahead and add in the following:

```
var Contact = React.createClass({
  render: function() {
    return (
      <div>
        <h2>GOT QUESTIONS?</h2>
        <p>The easiest thing to do is post on
        our <a href="http://forum.kirupa.com">forums</a>.
        </p>
      </div>
    );
  }
});

var Stuff = React.createClass({
  render: function() {
    return (
      <div>
        <h2>STUFF</h2>
        <p>Mauris sem velit, vehicula eget sodales vitae,
        rhoncus eget sapien:</p>
        <ol>
          <li>Nulla pulvinar diam</li>
          <li>Facilisis bibendum</li>
          <li>Vestibulum vulputate</li>
          <li>Eget erat</li>
          <li>Id porttitor</li>
        </ol>
      </div>
    );
  }
});
```

EXPLANATION: What we have just added are the Stuff and Contact components that simply render out HTML. All that remains is for us to update our routing configuration to include these two components and display them at the appropriate URL.

2. In our ReactDOM.render method, go ahead and add the following two highlighted lines:

```
ReactDOM.render(  
  <Router>  
    <Route path="/" component={ App }>  
      <IndexRoute component={ Home }/>  
      <Route path="stuff" component={ Stuff } />  
      <Route path="contact" component={ Contact } />  
    </Route>  
  </Router>,  
  destination  
)
```

EXPLANATION: All we are doing here is updating our routing logic to display the Stuff component if the URL contains the word stuff and to display the Contact component if the URL contains the word contact.

3. If you preview your page now, click on the Stuff and Contact links. If everything worked out fine, you'll see these views get loaded inside our app frame when you navigate to them.

TASK 9: Creating Active Links

The last thing we are going to tackle is something that greatly increases the usability of our app. Depending on which page you are currently displaying, we are going to highlight that link with a blue background. For example, Figure 13-6 is what our app will look like when the Stuff content is being displayed.

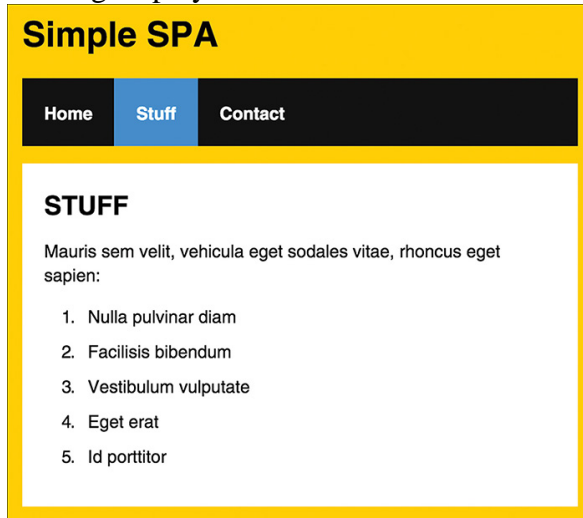


Figure 13-6 The Stuff content.

The way you accomplish this in React Router is by setting a prop called `activeClassName` on your Link instances with the name of the CSS class that will get set when that link is currently active.

1. To make this happen, go back to your App component and make the **highlighted changes:**

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li><Link to="/" activeClassName="active">Home</Link></li>
          <li><Link to="/stuff" activeClassName="active">Stuff</Link></li>
          <li><Link to="/contact" activeClassName="active">Contact</Link></li>
        </ul>
        <div className="content">
          {this.props.children}
        </div>
      </div>
    )
  }
});
```

EXPLANATION: We specify the `activeClassName` prop and set it to a value of `active`. This ensures that whenever a link is clicked (and its path becomes active), the link element's class attribute at runtime gets set to a value of `active`.

2. To ensure our active links are styled differently, go ahead and add the following CSS:

```
.active {  
  background-color: #0099FF;  
}
```

3. If you preview your app now, click on any of the links. Notice that the active link (and the Home link) displays with a blue background.

EXPLANATION: We aren't done just yet, though. Our Home link is always highlighted. It should only be highlighted when we load our Home page for the first time or explicitly navigate to the Home link itself. To fix this, we need to change how we link to our Home content.

4. Instead of specifying our Home content with a Link element, we are going to replace it with an **IndexLink** element instead. Go ahead and make this change:

```
var App = React.createClass({  
  render: function() {  
    return (  
      <div>  
        <h1>Simple SPA</h1>  
        <ul className="header">  
          <li><IndexLink to="/" activeClassName="active">Home</IndexLink></li>  
          <li><Link to="/stuff" activeClassName="active">Stuff</Link></li>  
          <li><Link to="/contact" activeClassName="active">Contact</Link></li>  
        </ul>  
        <div className="content">  
          {this.props.children}  
        </div>  
      </div>  
    )  
  }  
});
```

EXPLANATION: Once your Home navigation element is represented by an IndexLink instead of a Link, preview your app again.

5. This time, when the app loads, you'll notice that your Home link has the cool blue background by default. When you navigate to the Stuff or Contact pages, the Home link no longer has the highlight applied. And with this, your app is mostly good to go!

Conclusion

By now, we've covered a good chunk of the awesome functionality React Router has for helping you build your single-page apps. There is a whole lot more that React Router provides you can find the full React Router documentation and examples at <https://github.com/reactjs/react-router>.

Lab 13: Building a Todo List App (60 minutes)

If creating the Hello, World! example was a celebration of you getting your feet wet with React, creating the quintessential Todo List app is a celebration of you approaching React mastery! In this lab, we tie together a lot of the concepts and techniques you've learned to create something that works as follows: <https://www.kirupa.com/react/examples/todo.htm>

1. You start off with a blank app that allows you to enter tasks for later (see Figure 14-1).

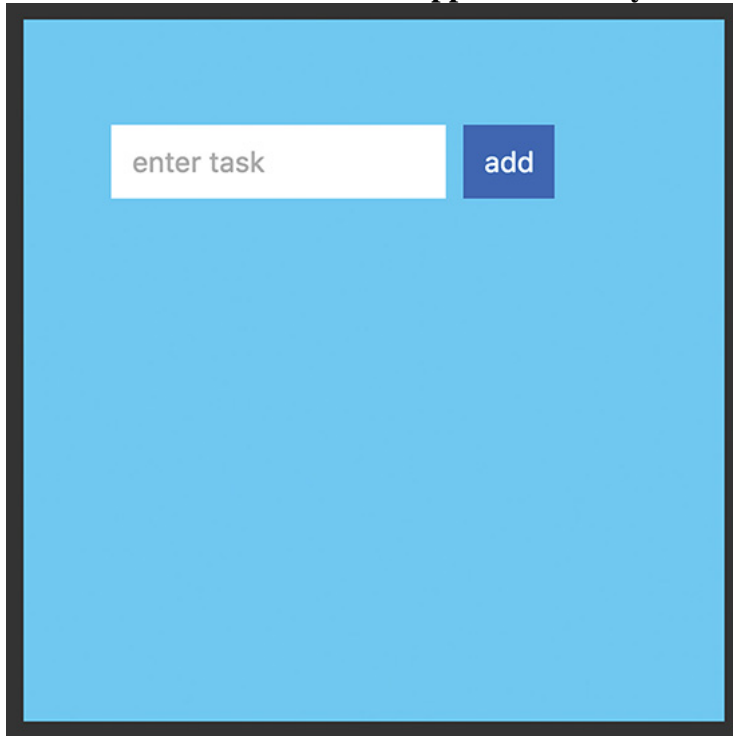


Figure 14-1 A blank app with task entry.

EXPLANATION: The way this Todo List app works is pretty simple. Type in a task or whatever you want into the text field and press Add (or hit Enter/Return). Once you've submitted your task, you will see it appear as an entry. You can keep adding tasks to add additional entries and have them all show up (see Figure 14-2).

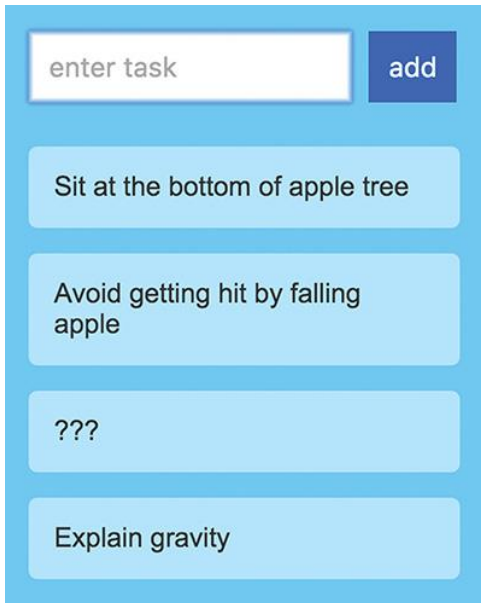
A screenshot of a mobile application interface for a todo list. At the top, there is a light blue header bar. Inside this bar, on the left, is a white text input field with the placeholder text "enter task". To the right of the input field is a dark blue button with the word "add" in white. Below the header bar, the background is a solid light blue. There are four rounded rectangular task entries stacked vertically. The first entry is light blue with the text "Sit at the bottom of apple tree". The second entry is light blue with the text "Avoid getting hit by falling apple". The third entry is light blue with the text "???". The fourth entry is light blue with the text "Explain gravity".

Figure 14-2 You can add tasks and have them show up.

Task 1: Getting Started

1. We need a starting point, so go ahead and create a new HTML document. Inside it, add the following content into it:

```
<!DOCTYPE html>
<html>

<head>
  <title>React! React! React!</title>
  <script src="https://npmcdn.com/react@15.3.0/dist/react.js"></script>
  <script src="https://npmcdn.com/react-dom@15.3.0/dist/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>

  <style>

  </style>
</head>

<body>

  <div id="container">

  </div>

  <script type="text/babel">
    var destination = document.querySelector("#container");

    ReactDOM.render(
      <div>
        Hello!
      </div>,
      destination
    );
  </script>
</body>

</html>
```

2. If you preview all of this in the browser, you will see the word **Hello!** appear. If you see that, then you are in good shape. It's time to start building our **Todo List** app!

TASK 2: Creating the UI

Right now, our app doesn't do a whole lot. We'll fix that by first getting the various UI elements up and running. That isn't very complicated for our app! The first thing we are going to do to is get our input field and button to appear. This is all done by using the `div`, `form`, `input`, and `button` elements!

1. All of that will live inside a component we are going to call **TodoList**. Go ahead and add the following code above where you have your `ReactDOM.render` method:

```
var TodoList = React.createClass({
  render: function() {
    return (
      <div className="todoListMain">
        <div className="header">
          <form>
            <input placeholder="enter task">
          </input>
          <button type="submit">add</button>
        </form>
      </div>
    </div>
    );
  }
});
```

2. Inside your `ReactDOM.render` method, we need to call our newly added **TodoList** component to render it. Go ahead and replace your existing JSX with the following:

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

3. Save your changes and preview what you have right now in your browser. You'll see something that looks like Figure 14-3.

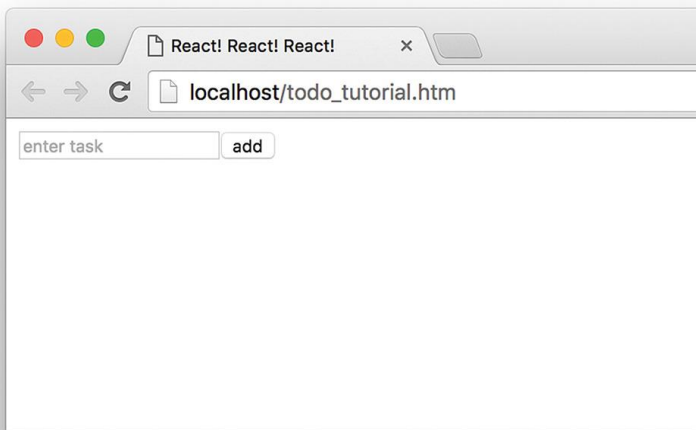


Figure 14-3 What you should see in the browser.

EXPLANATION: If you are surprised at what you see, take a few moments to look at the JSX we defined inside the `TodoList` component. There shouldn't be anything surprising there. We just defined a handful of HTML elements that look really plain. So, let's make our HTML elements look less plain by introducing them to some CSS!

4. Inside your style block, add the following:

```
body {  
  padding: 50px;  
  background-color: #66CCFF;  
  font-family: sans-serif;  
}  
.todoListMain .header input {  
  padding: 10px;  
  font-size: 16px;  
  border: 2px solid #FFF;  
}  
.todoListMain .header button {  
  padding: 10px;  
  font-size: 16px;  
  margin: 10px;  
  background-color: #0066FF;  
  color: #FFF;  
  border: 2px solid #0066FF;  
}  
  
.todoListMain .header button:hover {  
  background-color: #003399;  
  border: 2px solid #003399;  
  cursor: pointer;  
}
```

5. Once you've added all of this, preview your app now.
 - a. Because our HTML elements had the appropriate `className` values set on them, our CSS will kick in and our example will now look like Figure 14-4.

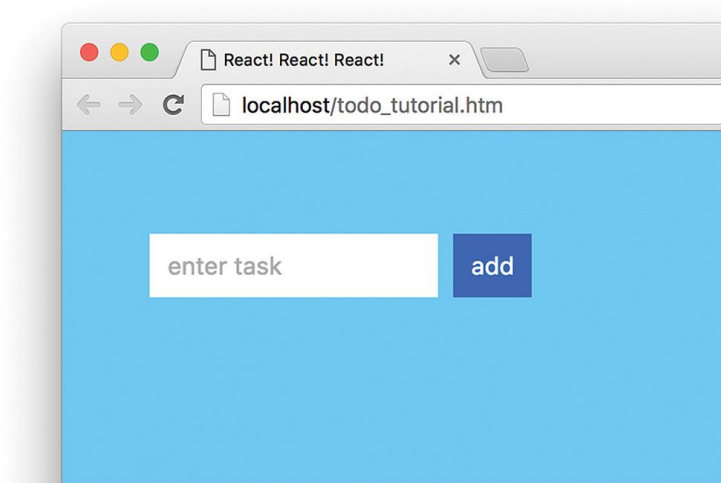


Figure 14-4 The improved example.

TASK 3: Creating the Functionality

INFORMATION ONLY:

The actual implementation of our Todo List app functionality is not as crazy as you might think. Let's take a high-level view of how it works. The most important piece of data is the text you enter into the text field. Each time you enter some text and submit the form, that text gets visually displayed in a list below any previous pieces of text you submitted. So far, this makes sense, right?

All of this is done by simply taking advantage of React's state functionality. Inside our state object, we have an array that is responsible for storing everything you enter (see Figure 14-5).

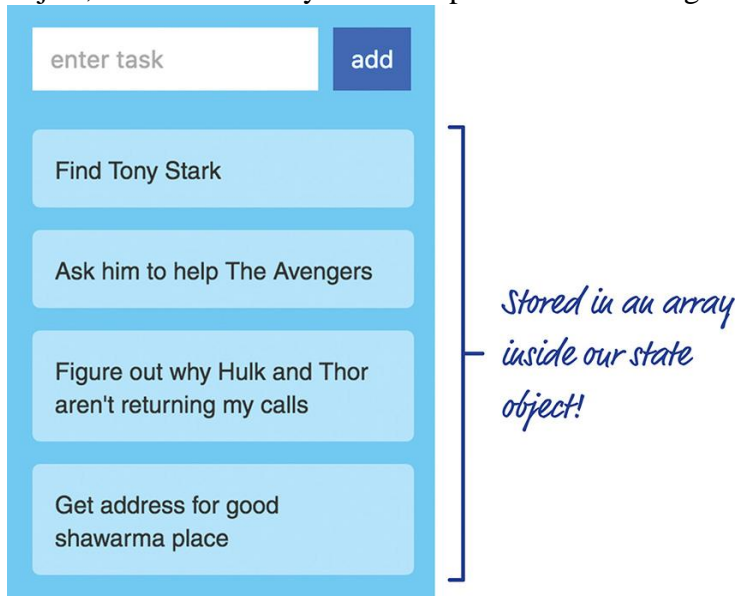


Figure 14-5 Our tasks are stored in an array. I know. Not very exciting :-)

EXPLANATION: Each time this array of items gets updated with new text that you submit, we update what you see with the newly submitted text. The rest of the work is just around setting up events and event handlers to ensure we can submit the form and know exactly what text to add to our array of items. In the following sections, we are going to turn all of this English we've seen here into React-flavored JavaScript and JSX!

TASK 4: Initializing our State Object

1. The first thing we are going to do is initialize our state object with the array that will be responsible for storing all of the submitted text.
 - a. Inside our `ToDoList` component, add the following **highlighted** lines:

```
var ToDoList = React.createClass({
  getInitialState: function() {
    return {
      items: []
    };
  },
  render: function() {
    return (
      <div className="todoListMain">
        <div className="header">
          <form>
            <input placeholder="enter task">
          </input>
          <button type="submit">add</button>
        </form>
      </div>
    </div>
    );
  }
});
```

EXPLANATION: What we are doing here is specifying the `getInitialState` method that gets called before our component renders. Inside that method, we create an empty array called `items` that we can then access via **`this.state.items`** from anywhere inside this component.

TASK 5: Handling the Form Submit

We add new items to our todo list when you submit the form either by pressing the Add button or hitting Enter/Return on your keyboard.

This behavior is mostly built-in to HTML and our browsers know all about how to deal with this. We don't have to write any special code for dealing with the Enter/Return key or listening for a press on the Add button. The only thing we need to worry about is dealing with what happens when the form actually gets submitted.

To do that, we listen to the `onSubmit` event on our form element. This event is fired every time the form is submitted, and that includes hitting the Enter/Return key or fiddling with any element that has a `type` attribute of `submit` on it. When the form is submitted and that event gets overheard, we will need to call an event handler. Let's give that event handler a name of `addItem`.

1. Putting all of this together, inside your `TodoList` component's render function, make the following **highlighted** change:

```
render: function() {  
  return (  
    <div className="todoListMain">  
      <div className="header">  
        <form onSubmit={this.addItem}>  
          <input placeholder="enter task">  
          </input>  
          <button type="submit">add</button>  
        </form>  
      </div>  
    </div>  
  );  
}
```

2. As we had hoped to do, we just linked our form element's `onSubmit` event to the `addItem` event handler. This event handler doesn't exist, but we are going to fix that by adding the following **highlighted** lines:

```
var TodoList = React.createClass({
  getInitialState: function() {
    return {
      items: []
    };
  },
  addItem: function(e) {
  },
  render: function() {
    return (
      <div className="todoListMain">
        <div className="header">
          <form onSubmit={this.addItem}>
            <input placeholder="enter task">
              </input>
            <button type="submit">add</button>
          </form>
        </div>
      </div>
    );
  }
});
```

EXPLANATION: Our `addItem` event handler/function doesn't do a whole lot right now, but the important thing is that it exists! Next, we'll fix the part where it doesn't do a whole lot.

TASK 6: Populating Our State

Right now, our `TodoList` component's state object contains the `items` array. What we need to do is populate this array with the text that you enter into the input field. That means we need a way to access our input element from within React. The way we are going to do that is by **setting a `ref` attribute** (as you saw in Module 12) on our input element and storing the reference to the HTML element that gets generated.

1. Inside our `TodoList` component's render method, add the following line:

```
render: function() {  
  return (  
    <div className="todoListMain">  
      <div className="header">  
        <form onSubmit={this.addItem}>  
          <input ref={(a) => this._inputElement = a}  
            placeholder="enter task">  
            </input>  
            <button type="submit">add</button>  
          </form>  
        </div>  
      </div>  
    );  
  }  
}
```

EXPLANATION: When this highlighted code runs, which is immediately after this component mounts, the `_inputElement` property will store a reference to the generated input element. Now that we have done this, we can treat this element like we would any DOM element we might have found using `querySelector` or equivalent function in the non-React world. What we are going to do next is populate our `items` array!

2. Go ahead and modify the `addItem` method by adding the following lines:

```
addItem: function(e) {  
  var itemArray = this.state.items;  
  
  itemArray.push(  
    {  
      text: this._inputElement.value,  
      key: Date.now()  
    }  
  );  
  this.setState({  
    items: itemArray  
  });  
  
  e.preventDefault();  
}
```

EXPLANATION OF CODE:

This looks like a lot of code you just added, but all we are doing here is putting into JavaScript our earlier stated goal of populating our items array with text from our input field. Let's walk through this code in greater detail.

- The first thing we do is create an array called `itemArray` that stores a reference to our state object's `items` property:

```
var itemArray = this.state.items;
```

- Once we have this array, we add to it our recently submitted text entry from our input element:

```
itemArray.push(  
  {  
    text: this._inputElement.value,  
    key: Date.now()  
  }  
);
```

Notice that we aren't just adding the text entry from our input element. We are instead adding an object made up of the text and key properties. The text property stores our input element's text value. The key property stores the current time. This sounds like a bizarre thing to do, but as you recall from Module 9, the goal is to have this key value be unique for every entry that gets submitted. This is important because we will be using the data in this array to eventually generate some UI elements. This key value is what React will use to uniquely identify each generated UI element, so by generating the key using `Date.now()`, we ensure a certain level of uniqueness. Because this is an important (yet easy to overlook) detail, we will revisit all of this again in a few moments.

- Once we are done with the `itemArray`, all that remains is to set our state object's `items` property to it:

```
this.setState({  
  items: itemArray  
});
```

- Almost done here! The last thing we do in this method is the following:

```
e.preventDefault();
```

The `preventDefault` method ensures we override the default `onSubmit` event. The reason we do this is a bit obscure, but it is to ensure the following: all we want to do when we submit the form is call the `addItem` method. If we didn't stop the default behavior, our app will correctly call `addItem` as desired when we submit the form. It will also trigger our browser's default POST behavior—which we definitely don't want. By stopping the `onSubmit` event from performing the default behavior, we get our desired behavior of calling the `addItem` method without any of the unwanted side effects like an unnecessary POST action that might refresh your page.

TASK 7: Displaying the Tasks

We are almost done here! The next to the last thing we are going to do is visualize the tasks that currently live inside our state object's items array. This is going to involve creating a whole new component called `TodoItems`, passing around some props, using the `map` function, and doing other awesome coding things.

1. The first thing we are going to do is define our `TodoItems` component.
 - a. In your code, just above where you have the `TodoList` component defined, go ahead and add the following in:

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

  }
});
```

EXPLANATION: There is nothing going on right now, but that's OK.

2. Next, call this component from inside the `TodoList` component's render method.
 - a. Then specify a prop and pass in our `TodoList` component's state object that contains our items array.
 - b. Doing all of this is really simple, so go ahead and add the following **highlighted** line to your `TodoList` component's render method:

```
render: function() {
  return (
    <div className="todoListMain">
      <div className="header">
        <form onSubmit={this.addItem}>
          <input ref={ (a) => this._inputElement = a }
            placeholder="enter task">
          </input>
          <button type="submit">add</button>
        </form>
      </div>
      <TodoItems entries={this.state.items}/>
    </div>
  );
}
```

EXPLANATION: All we did here is instantiate our `TodoItems` component and pass in our items state property to a prop called `entries`. At this point, if you run our app in the browser, nothing visible will happen. Our `TodoItems` component is ready to render, and it has access to all of the tasks that were submitted. The only problem is that it doesn't really do anything with all of that, but we are going to fix that up next.

3. Getting back to our `TodoItems` component, the first thing we are going to do is create a new variable to store our passed in array of tasks. To do that, add the following **highlighted** line:

```
var TodoItems = React.createClass({  
  render: function() {  
    var todoEntries = this.props.entries;  
  
  }  
});
```

EXPLANATION: We just added a variable called `todoEntries`, and it stores the value from the `entries` prop that we passed in based on the `TodoList` component's `this.state.items` value. Sweet! Now, our `todoEntries` variable stores an array containing a bunch of objects that each store a task and a key.

4. All that remains is to create the HTML elements that will be used to display our data. In the first step towards accomplishing that, add the following **highlighted** lines of code to create the li elements:

```
var TodoItems = React.createClass({
  render: function() {
    var todoEntries = this.props.entries;

    function createTasks(item) {
      return <li key={item.key}>{item.text}</li>
    }

    var listItems = todoEntries.map(createTasks);
  }
});
```

EXPLANATION OF CODE:

- We are using the map function to iterate every item inside todoEntries and call the createTasks function to create a list element for each entry:

```
function createTasks(item) {
  return <li key={item.key}>{item.text}</li>
}
```

- To reiterate a point we made earlier, since these list elements are dynamically created, we need to help React keep track of them by specifying the key attribute and giving each a unique value. We already solved this part of the problem when we stored our tasks initially, as you recall:

```
itemArray.push(
  {
    text: this._inputElement.value,
    key: Date.now()
  }
);
```

- Because of our earlier planning, we take the easy street right now by assigning our key attribute the item.key value that each item in our todoEntries array already contains. Our list element's visible content is simply the text value stored by item.text.

5. Putting all of this together, this collection of list elements is fully processed and stored by our `listItems` variable. All that remains at this point is to go from list elements inside an array to list elements rendered on the screen.

a. To accomplish that, go ahead and add the following **highlighted** lines:

```
var TodoItems = React.createClass({
  render: function() {
    var todoEntries = this.props.entries;

    function createTasks(item) {
      return <li key={item.key}>{item.text}</li>
    }

    var listItems = todoEntries.map(createTasks);

    return (
      <ul className="theList">
        {listItems}
      </ul>
    );
  }
});
```

EXPLANATION: What we are doing is returning an `` element whose contents are the list elements stored by `listItems`.

6. After you've added this, save your document and preview your app. You'll see something that looks like Figure 14-7 after entering a few tasks.

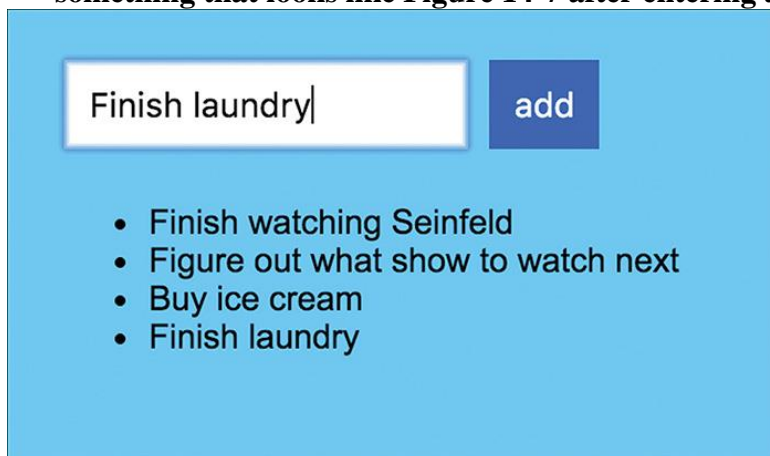


Figure 14-7 List element for the list items.

EXPLANATION: Our app works! Every task you submit shows up in its own list item. This is awesome progress, and all we have left are a few little things here and there that need to be wrapped up.

TASK 8: Adding the Finishing Touches

We are almost done here! First, what we have right now doesn't look exactly like the code we started out with. Our list of tasks looks a bit plain, but that can be fixed with some CSS.

1. Inside your style block, add the following style rules just below where your existing style rules live:

```
.todoListMain .theList {  
  list-style: none;  
  padding-left: 0;  
  width: 255px;  
}  
  
.todoListMain .theList li {  
  color: #333;  
  background-color: rgba(255,255,255,.5);  
  padding: 15px;  
  margin-bottom: 15px;  
  border-radius: 5px;  
}
```

2. If you preview your app now, you'll see that the entered tasks look exactly as you expected them to:



3. Next, have you noticed that whatever you enter into the input field doesn't go away after you submit the form? You have to manually clear out the field each time after submitting a task. That is annoying, but the fix for it is quite simple.
 - a. Inside our `ToDoList` component's `addItem` method, add the following highlighted line:

```
addItem: function(e) {  
  var itemArray = this.state.items;  
  
  itemArray.push(  
    {  
      text: this._inputElement.value,  
      key: Date.now()  
    }  
  );  
  
  this.setState({  
    items: itemArray  
  });  
  
  this._inputElement.value = "";  
  
  e.preventDefault();  
}
```

EXPLANATION: All we are doing here is clearing our input element's value property when the form is submitted and the `addItem` method gets called. This ensures that we no longer have to manually clear out our input field between each task we would like to submit.

Conclusion

Our Todo app is pretty simple in what it does, but by building it from scratch, we covered almost every little interesting detail React brings to the table. More importantly, we created code that shows how the various concepts we learned individually work together.

Lab 14: Setting Up Your React Development Environment (45 minutes)

We are now going to set up your development environment where your JSX to JS conversion is handled prior to the user loading the page (see Figure 15-2).

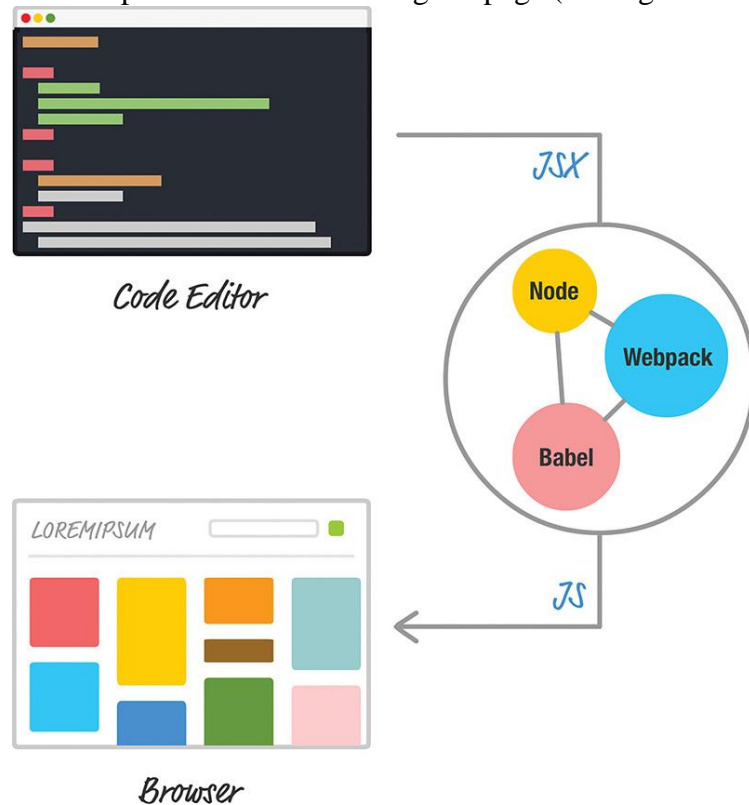


Figure 15-2 JSX to JavaScript conversion as part of your app building process.

EXPLANATION:

With this solution, your browser is loading your app and dealing with an already converted (and potentially optimized) JavaScript file. Now, the only reason why we delayed talking about all of this until now is for simplicity. Learning React is difficult enough. Adding the complexity of build tools and setting up your environment as part of learning React is just not cool. Now that you have a solid grasp of everything React does, it's time to change that with this module.

In the following sections, we look at one way to set up your development environment using a combination of Node, Babel, and webpack.

TASK 1: It Is Environment Setup Time!

Setting up our Initial Project Structure

1. The first thing we are going to do is set up our project.
 - a. Go to your Desktop and create a new folder called **MyTotallyAwesomeApp**.
2. Inside this folder, create two more folders called **dev** and **output**.
 - a. Your folder arrangement will look a little bit like Figure 15-5.



Figure 15-5 Our current folder arrangement.

EXPLANATION: What we are doing here is pretty simple. Inside our **dev** folder, we will place all of our unoptimized and unconverted JSX, JavaScript, and other script-related content. In other words, this is where the code you are writing and actively working on will live. Inside our **output** folder, we will place the result of running our various build tools on the script files found inside the **dev** folder. This is where Babel will convert all of our JSX files into JS. This is also where **webpack** will resolve any dependencies between our script files, and place all of the important script content into a single JavaScript file.

3. The next thing we are going to do is create the **HTML** file that we will point our browser to.
 - a. Inside the **MyTotallyAwesomeApp** folder, use your code editor to create a new **HTML** file called **index.html** with the following contents:

```
<!DOCTYPE html>
<html>
<head>
  <title>React! React! React!</title>
</head>

<body>
  <div id="container"></div>

  <script src="output/myCode.js"></script>
</body>
</html>
```

EXPLANATION: Be sure to save your file after adding this content in. Now, speaking of the content, our markup is pretty simple. Our document's body is just an empty div element with an id value of **container** and a script tag that points to the final JavaScript file (**myCode.js**) that will get generated inside the output folder:

```
<script src="output/myCode.js"></script>
```


4. INFORMATION ONLY:

Besides those two things, our HTML file doesn't have a whole lot going for it. If we had to visualize the relationship of everything right now, it looks a bit like Figure 15-6.

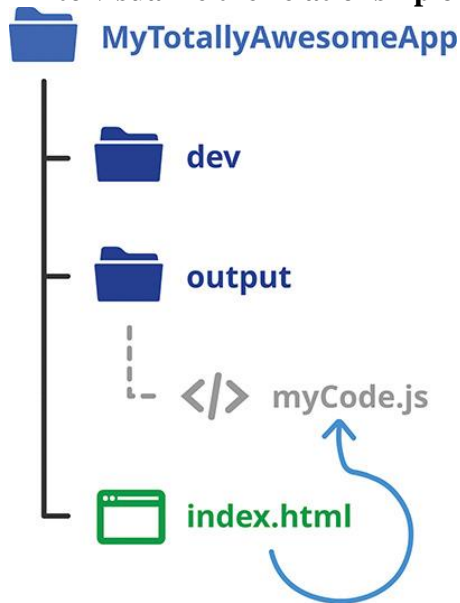


Figure 15-6 What your current project structure looks like.

EXPLANATION: I've dotted the line to the `myCode.js` file in our `output` folder because that file doesn't exist there yet. We are pointing to something in our HTML that currently is non-existent, but that won't stay that way for long.

TASK 2: Installing and Initializing Node.js

1. Our next step is to install Node.js. Visit the Node.js site (<https://nodejs.org/>) to install the version that is appropriate for your operating system (see Figure 15-7).

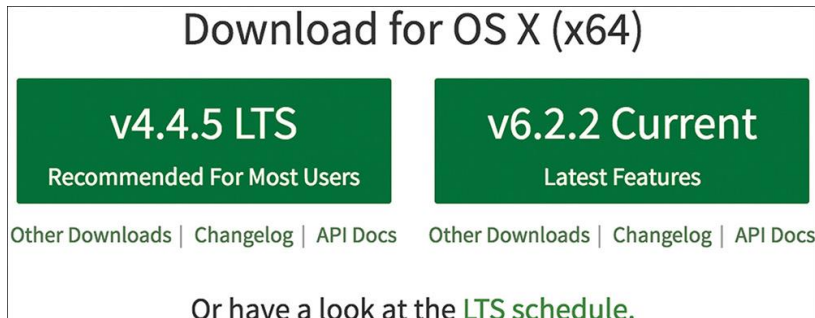


Figure 15-7 The download buttons on the Node.js site.

- a. I tend to always install the latest version, so you should go with that as well. The download and installation procedure isn't particularly exciting.
2. Once you have Node.js installed, test to make sure it is truly installed by launching the Terminal (on Mac), Command Prompt (on Windows), or equivalent tool of choice and typing in the following and pressing Enter:

node -v
 3. If everything worked out properly, you will see a version number displayed that typically corresponds to the version of Node.js you just installed.
 - a. If you are getting an error for whatever reason, follow the troubleshooting steps listed here (<https://github.com/npm/npm/wiki/Troubleshooting>).
 4. Next, we are going to initialize Node.js on our MyTotallyAwesomeApp folder. To do this, first navigate to the MyTotallyAwesomeApp folder using your Terminal or Command Prompt. On OS X, this will look like Figure 15-8.



Figure 15-8 Navigate to the MyTotallyAwesomeApp folder.

5. Now, go ahead and initialize Node.js by entering the following:

npm init

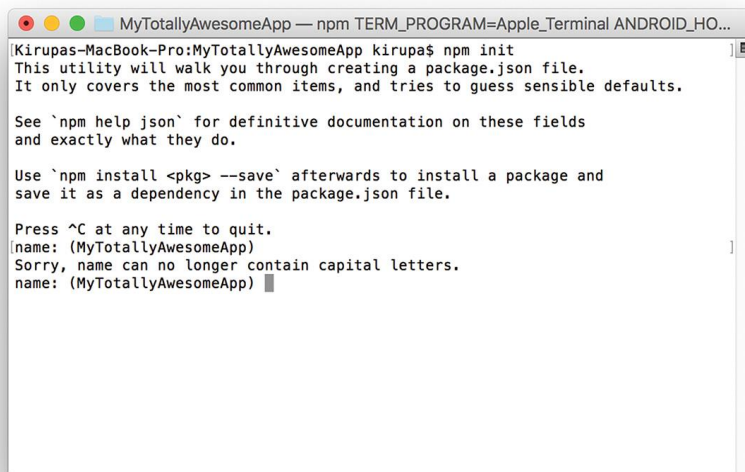
This will kick off a series of questions that will help set up Node.js on our project.

- a. The first question will ask you to specify your project name. Hitting Enter will allow you to specify the default value that has already been selected for you.

That is all great, but the default name is our project folder, which is

MyTotallyAwesomeApp.

- i. If you hit Enter, because it contains capital letters, it will throw an error (see Figure 15-9).



```
Kirupas-MacBook-Pro:MyTotallyAwesomeApp kirupa$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (MyTotallyAwesomeApp)
Sorry, name can no longer contain capital letters.
name: (MyTotallyAwesomeApp)
```

Figure 15-9 Our project folder name includes capital letters, triggering an error.

- b. Go ahead and enter the lowercase version of the name, **mytotallyawesomeapp**. Once you've done that, press Enter.
- c. For the remaining questions, just hit Enter to accept all the default values.

- d. The end result of all of this is a new file called **package.json** that will be created in your **MyTotallyAwesomeApp** folder (see Figure 15-10).

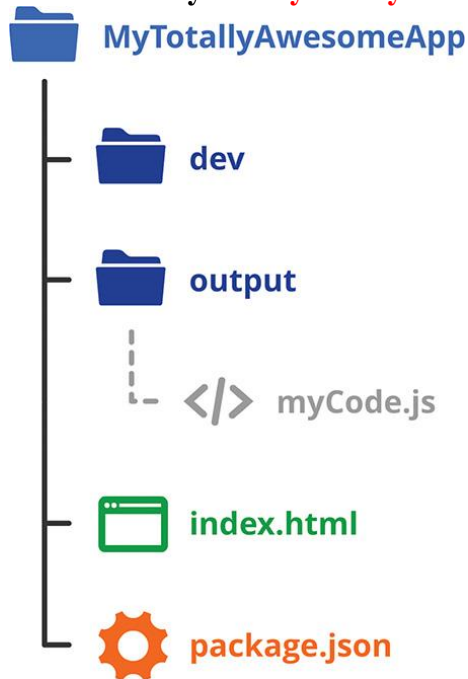


Figure 15-10 The package.json file shows up in your folder.

6. If you open the contents of **package.json** in your code editor, you'll see something that looks similar to the following:

```
{
  "name": "mytotallyawesomeapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

EXPLANATION: Don't worry too much about the contents of this file, but just know that one of the results of you calling `npm init` is that you have a **package.json** file created with some properties and values that Node.js knows what to do with.

Task 3: Installing the React Dependencies

What we are going to do next is install our React dependencies so that we can use the React and React DOM libraries in our code.

1. In your Command Prompt, enter the following to install our React dependencies:

npm install react react-dom --save

EXPLANATION: Once you Enter this, a lot of weird stuff will show up on your screen. You may even see a bunch of warnings, but they should be safe to ignore. What is happening is that the React and React-DOM libraries (and code that they depend on) is getting downloaded from a giant repository of Node.js packages found here:

<https://www.npmjs.com/>

2. If you take a look at your **MyTotallyAwesomeApp** folder, you'll see a folder called **node_modules**.
- Inside that folder, you'll see a bunch of various modules.
 - Let's update our visualization of our current file/folder structure to look like Figure 15-11.

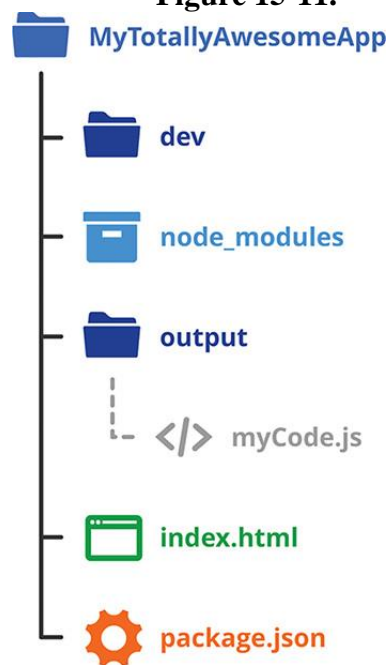


Figure 15-11 The updated folder structure.

EXPLANATION: The list of modules you see right now is just the beginning. We'll be adding a few more by the time you reach the end of this.

Task 4: Adding our JSX File

Now that we've told Node.js all about our interest in React, we are one step closer towards building a React app. We are going to go further by adding a JSX file that is a modified version of the example we saw in Module 3 when looking at Components.

1. Inside our **dev** folder, using the code editor, create a file called **index.jsx** with the following code as its contents:

```
import React from "react";
import ReactDOM from "react-dom";

var HelloWorld = React.createClass({
  render: function() {
    return (
      <p>Hello, {this.props.greetTarget}!</p>
    );
  }
});

ReactDOM.render(
  <div>
    <HelloWorld greetTarget="Batman"/>
    <HelloWorld greetTarget="Iron Man"/>
    <HelloWorld greetTarget="Nicolas Cage"/>
    <HelloWorld greetTarget="Mega Man"/>
    <HelloWorld greetTarget="Bono"/>
    <HelloWorld greetTarget="Catwoman"/>
  </div>,
  document.querySelector("#container")
);
```

EXPLANATION: Notice that the bulk of the JSX we added is pretty much unmodified from what we had earlier. **The only difference is that what used to be script references for getting the React and React DOM libraries into our app has now been replaced with import statements pointing to our react and react-dom Node.js packages** we added a few moments ago:

```
import React from "react";
import ReactDOM from "react-dom";
```

2. Now, you are probably wondering when we can build our app and get it all working in our browser. Well, there are still a few more steps left. Figure 15-12 shows what the current visualization of our project looks like.

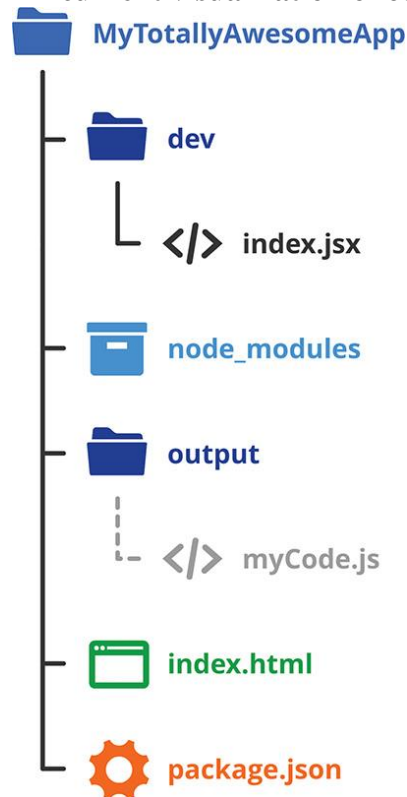


Figure 15-12 The current project.

EXPLANATION: Our **index.html** file is looking for code from the **myCode.js** file which still doesn't exist. We added our JSX file, but we know that our browser doesn't know what to do with JSX. We need to go from **index.jsx** in our **dev** folder to **myCode.js** in the **output** folder. Guess what we are going to do next?

Going from JSX to JavaScript

The missing step right now is turning our JSX into JavaScript that our browser can understand. This involves both webpack and Babel, and we are going to configure both of them to make this all work.

Task 5: Setting up webpack

Since we are in Node.js territory and both webpack and Babel exist as Node packages, we need to install them both just like we installed the React-related packages.

1. To install webpack, enter the following in your Terminal / Command Prompt:

npm install webpack --save

- a. This will take a few moments while the webpack package (and its large list of dependencies) gets downloaded and placed into our **node_modules** folder.
2. After you've done this, we need to add a configuration file to specify how webpack will work with our current project. Using your code editor, add a file called **webpack.config.js** inside our **MyTotallyAwesomeApp** folder (see Figure 15-13).

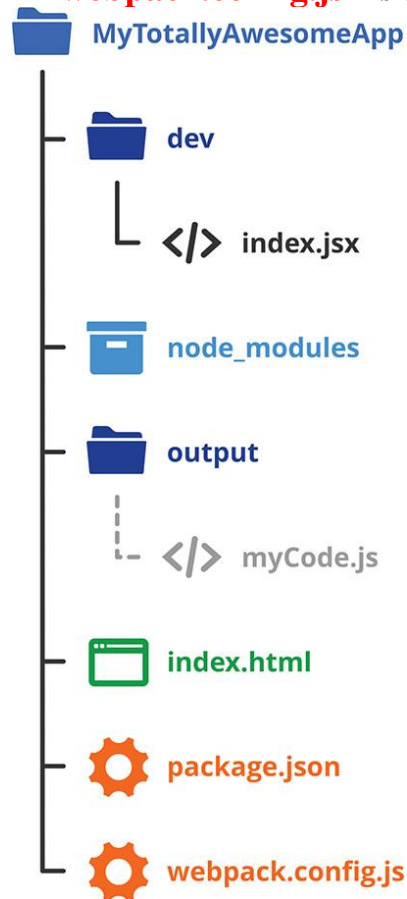


Figure 15-13 Adding webpack.config.js.

3. Inside this file, we will specify a bunch of JavaScript properties to define where our original, unmodified source files live and where to output the final source files. Go ahead and add the following JavaScript into **webpack.config.js**:

```
var webpack = require("webpack");
var path = require("path");

var DEV = path.resolve(__dirname, "dev");
var OUTPUT = path.resolve(__dirname, "output");

var config = {
  entry: DEV + "/index.jsx",
  output: {
    path: OUTPUT,
    filename: "myCode.js"
  }
};

module.exports = config;
```

4. Take a few moments to see what this code is doing.
 - a. We defined two variables called DEV and OUTPUT that refer to folders of the same name in our project.
 - b. Inside the config object, we have two properties called entry and output that use our DEV and OUTPUT variables to help map our **index.jsx** file to become **myCode.js**.

TASK 6: Setting up Babel

The last piece in our current setup is to transform our **index.jsx** file to become regular JavaScript in the form of **myCode.js**. This is where Babel comes in.

1. To install Babel, let's go back to our trusty Command Prompt and enter the following Node.js command:

```
npm install babel-loader babel-preset-es2015 babel-preset-react --save
```

EXPLANATION: With this command, we install the **babel-loader**, **babel-preset-es2015**, and **babel-preset-react** packages.

Now we need to configure Babel to work with our project. This is a two-step process.

2. The **first step** is to specify which Babel presets we want to use. There are several ways of doing this, but the preferred way is to modify **package.json** and add the following **highlighted** content:

```
{
  "name": "mytotallyawesomeapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "babel-loader": "^6.2.4",
    "babel-preset-es2015": "^6.9.0",
    "babel-preset-react": "^6.5.0",
    "react": "^15.1.0",
    "react-dom": "^15.1.0",
    "webpack": "^1.13.1"
  },
  "babel": {
    "presets": [
      "es2015",
      "react"
    ]
  }
}
```

EXPLANATION: In the **highlighted** lines, we specify our babel object and specify the **es2015** and **react** preset values.

3. The **second step** is to tell webpack about Babel. In our **webpack.config.js** file, go ahead and add the following **highlighted** lines:

```
var webpack = require("webpack");
var path = require("path");

var DEV = path.resolve(__dirname, "Dev");
var OUTPUT = path.resolve(__dirname, "output");

var config = {
  entry: DEV + "/index.jsx",
  output: {
    path: OUTPUT,
    filename: "myCode.js"
  },
  module: {
    loaders: [{
      include: DEV,
      loader: "babel",
    }]
  }
};

module.exports = config;
```

EXPLANATION: We added the module and loaders objects that tell webpack to pass the **index.jsx** file defined in our entry property to turn into JavaScript through Babel. With this change, we've pretty much gotten our development environment setup for building a React app.

TASK 7: Building and Testing Our App

The last step in all of this is building our app and having the end-to-end workflow work.

1. To build our app, what you type varies on whether you are on the Terminal or on the Command Prompt.

a. ~~For the Terminal on the Mac, enter the following:~~

~~`./node_modules/.bin/webpack`~~

b. In the Command Prompt on Windows, enter this instead:

`node_modules\.bin\webpack.cmd`

EXPLANATION: This command runs webpack and does all the things we've specified in our `webpack.config.js` and `package.json` configuration files.

- c. Your output in your Terminal / Command Prompt will look something like Figure 15-14.

A screenshot of a macOS terminal window titled "MyTotallyAwesomeApp -- bash -- 80x24". The terminal shows the command `./node_modules/.bin/webpack` being executed. The output includes a hash, version, time, and a table of assets. The asset `myCode.js` is 701 kB and consists of 0 chunks, with a status of "[emitted]". The chunk is named "main". Below the table, it says "+ 168 hidden modules". The prompt is `Kirupas-MacBook-Pro:MyTotallyAwesomeApp kirupa$`.

```
Kirupas-MacBook-Pro:MyTotallyAwesomeApp kirupa$ ./node_modules/.bin/webpack
Hash: dd8d2042676816001203
Version: webpack 1.13.1
Time: 1022ms
   Asset      Size  Chunks             Chunk Names
myCode.js  701 kB      0 [emitted]        main
+ 168 hidden modules
Kirupas-MacBook-Pro:MyTotallyAwesomeApp kirupa$
```

Figure 15-14 The webpack output.

2. Besides seeing something that vaguely looks like a successful build displayed in cryptic text form, navigate to your `MyTotallyAwesomeApp` folder.

3. **Open your `index.html` file in your browser. If everything was set up properly, you'll see our simple React app displaying (see Figure 15-15).**

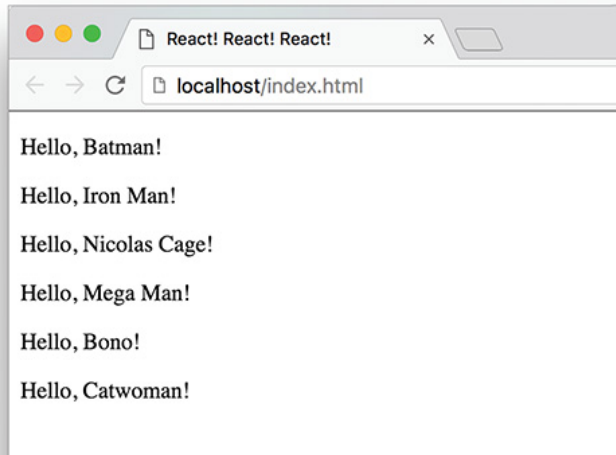


Figure 15-15 The simple React app displaying.

4. **If you venture into the **Output** folder and look at `myCode.js`, you'll see a fairly hefty (~700Kb) file with a lot of JavaScript made up of the relevant React, ReactDOM, and your app code all organized there.**
5. **From this point, you can build your app, add new assets, and make the typical changes you normally would. The only difference between what we had been doing throughout this book and what we are doing now is simple—what your browser cares about is generated for you by the various build tools and packager.**
6. **Your browser is no longer taking all of this React JSX/ES6/etc. stuff and converting it into normal HTML/CSS/JS on the fly during page load.**

Conclusion

What we've just seen is a very small part of everything you can do when you put Node, Babel, and webpack together. The unfortunate thing is that covering all of that goes well beyond the scope of learning React, but if you want to dive deeper, there are many resources available to assist you in learning the ins and outs of all of these build tools.