**Chapter 4. Pure React**

In order to understand how React runs in the browser, we will be working purely with React in this chapter. We will not introduce JSX, or JavaScript as XML, until the next chapter. You may have worked with React in the past without ever looking at the pure React code that is generated when we transpile JSX into React. You can successfully use React without looking at pure React. However, if you take the time to understand what is going on behind the scenes, you will be more efficient, especially when it comes time to debug. That is our goal in this chapter: to look under the hood and understand how React works.

**Page Setup**

In order to work with React in the browser, we need to include two libraries: React and ReactDOM. React is the library for creating views. ReactDOM is the library used to actually render the UI in the browser.

**REACTDOM**

React and ReactDOM were split into two packages for version 0.14. The release notes state: “The beauty and the essence of React has nothing to do with browsers or the DOM... This [splitting into two packages] paves the way to writing components that can be shared between the web version of React and React Native.”[1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391606008080)Instead of assuming that React will render only in the browser, future releases will aim to support rendering for a variety of platforms.

We also need an HTML element that ReactDOM will use to render the UI. You can see how the scripts and HTML elements are added in [Example 4-1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0401). Both libraries are available as scripts from the Facebook CDN.

*Example 4-1. HTML document setup with React*

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** charset="utf-8"**>**

**<title>**Pure React Samples**</title>**

**</head>**

**<body>**

*<!-- Target container -->*

**<div** class="react-container"**></div>**

*<!-- React library & ReactDOM-->*

**<script** src="https://unpkg.com/react@15.4.2/dist/react.js"**></script>**

**<script** src="https://unpkg.com/react-dom@15.4.2/dist/react-dom.js"**></script>**

**<script>**

*// Pure React and JavaScript code*

**</script>**

**</body>**

**</html>**

These are the minimum requirements for working with React in the browser. You can place your JavaScript in a separate file, but it must be loaded somewhere in the page after React has been loaded.

**The Virtual DOM**

HTML is simply a set of instructions that a browser follows when constructing the document object model, or DOM. The elements that make up an HTML document become DOM elements when the browser loads HTML and renders the user interface.

Let’s say that you have to construct an HTML hierarchy for a recipe. A possible solution for such a task might look something like this:

**<section** id="baked-salmon"**>**

**<h1>**Baked Salmon**</h1>**

**<ul** class="ingredients"**>**

**<li>**1 lb Salmon**</li>**

**<li>**1 cup Pine Nuts**</li>**

**<li>**2 cups Butter Lettuce**</li>**

**<li>**1 Yellow Squash**</li>**

**<li>**1/2 cup Olive Oil**</li>**

**<li>**3 cloves of Garlic**</li>**

**</ul>**

**<section** class="instructions"**>**

**<h2>**Cooking Instructions**</h2>**

**<p>**Preheat the oven to 350 degrees.**</p>**

**<p>**Spread the olive oil around a glass baking dish.**</p>**

**<p>**Add the salmon, garlic, and pine nuts to the dish.**</p>**

**<p>**Bake for 15 minutes.**</p>**

**<p>**Add the yellow squash and put back in the oven for 30 mins.**</p>**

**<p>**Remove from oven and let cool for 15 minutes.

Add the lettuce and serve.**</p>**

**</section>**

**</section>**

In HTML, elements relate to each other in a hierarchy that resembles a family tree. We could say that the root element has three children: a heading, an unordered list of ingredients, and a section for the instructions.

Traditionally, websites have consisted of independent HTML pages. When the user navigated these pages, the browser would request and load different HTML documents. The invention of AJAX brought us the single-page application, or [SPA](https://en.wikipedia.org/wiki/Single-page_application). Since browsers could request and load tiny bits of data using AJAX, entire web applications could now run out of a single page and rely on JavaScript to update the user interface.

In an SPA, the browser initially loads one HTML document. As users navigate through the site, they actually stay on the same page. JavaScript destroys and creates a new user interface as the user interacts with the application. It may feel as though you are jumping from page to page, but you are actually still on the same HTML page and JavaScript is doing the heavy lifting.

The [DOM API](https://mzl.la/2m1oQDJ) is a collection of objects that JavaScript can use to interact with the browser to modify the DOM. If you have used document.createElement or document.appendChild, you have worked with the DOM API. Updating or changing rendered DOM elements in JavaScript is relatively easy.[2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391605874640) However, the process of inserting new elements is painfully slow.[3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391605872960) This means if web developers are meticulous about how they make changes to UI, they can improve the performance of their applications.

Managing DOM changes with JavaScript efficiently can become very complicated and time-consuming. From a coding perspective, it is easier to clear all the children of a particular element and reconstruct them than it would be to leave those child elements in place and attempt to efficiently update them.[4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391605870576) The problem is that we may not have the time or the advanced knowledge of JavaScript to work efficiently with the DOM API every time we build a new application. The solution is React.

React is a library that is designed to update the browser DOM for us. We no longer have to be concerned with the complexities associated with building performant SPAs because React can do that for us. With React, we do not interact with the DOM API directly. Instead, we interact with a *virtual DOM*, or set of instructions that React will use to construct the UI and interact with the browser.[5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391605867456)

The virtual DOM is made up of React elements, which conceptually seem similar to HTML elements, but are actually JavaScript objects. It is much faster to work directly with JavaScript objects than it is to work with the DOM API. We make changes to a JavaScript object, the virtual DOM, and React renders those changes for us using the DOM API as efficiently as possible.

**React Elements**

The browser DOM is made up of DOM elements. Similarly, the React DOM is made up of React elements. DOM elements and React elements may look the same, but they are actually quite different. A React element is a description of what the actual DOM element should look like. In other words, React elements are the instructions for how the browser DOM should be created.

We can create a React element to represent an h1 using React.createElement:

React.createElement("h1", **null**, "Baked Salmon")

The first argument defines the type of element that we wish to create. In this case, we want to create a heading-one element. The third argument represents the element’s children, any nodes that are inserted between the opening and closing tag. The second argument represents the element’s properties. This h1 currently does not have any properties.

During rendering, React will convert this element to an actual DOM element:

**<h1>**Baked Salmon**</h1>**

When an element has attributes, they can be described with properties. Here is a sample of an HTML h1 tag that has id and data-type attributes:

React.createElement("h1",

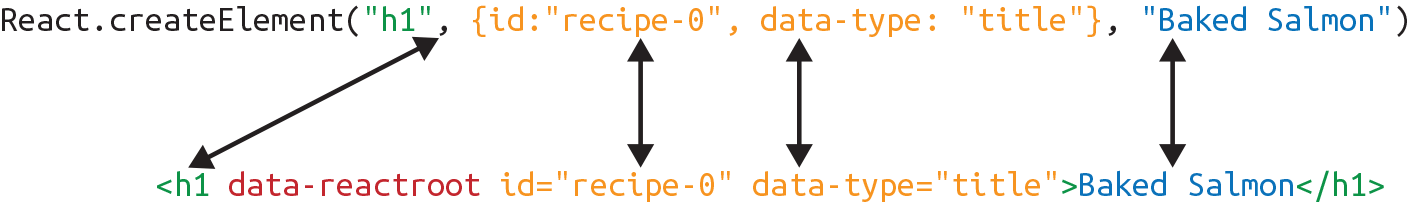
**{id: "recipe-0", 'data-type': "title"}**,

"Baked Salmon"

)

<h1 data-reactroot id="recipe-0" data-type="title">Baked Salmon</h1>

The properties are similarly applied to the new DOM element: the properties are added to the tag as attributes, and the child text is added as text within the element. You’ll also notice data-reactroot, which identifies that this is the root element of your React component ([Figure 4-1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#fig0401)).



*Figure 4-1. Relationship between createElement and the DOM element*

**DATA-REACTROOT**

data-reactroot will always appear as an attribute of the root element of your React component. Prior to version 15, React IDs were added to each node that was a part of your component. This helped with rendering and keeping track of which elements needed to be updated. Now, there is only an attribute added to the root, and rendering is kept track of based on the hierarchy of elements.

So, a React element is just a JavaScript literal that tells React how to construct the DOM element. [Example 4-2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0402)shows the element that createElement call actually creates.

*Example 4-2. Logging the title element*

{

$$typeof: Symbol(React.element),

"type": "h1",

"key": **null**,

"ref": **null**,

"props": {"children": "Baked Salmon"},

"\_owner": **null**,

"\_store": {}

}

This is a React element. There are fields that are used by React: \_owner, \_store, $$typeof. The key and reffields are important to React elements, but we’ll introduce those later, in [Chapter 5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#react_with_jsx). For now, let’s take a closer look at the type and props fields in [Example 4-2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0402).

The type property of the React element tells React what type of HTML or SVG element to create. The propsproperty represents the data and child elements required to construct a DOM element. The children property is for displaying other nested elements as text.

**A NOTE ON CREATING ELEMENTS**

We are taking a peek at the object that React.createElement returns. There is never a case where you would create elements by hand-typing literals that look like this. You must always create React elements with the React.createElement function or factories, which are discussed at the end of this chapter.

**ReactDOM**

ReactDOM contains the tools necessary to render React elements in the browser. ReactDOM is where we will find the render method as well as the renderToString and renderToStaticMarkup methods that are used on the server. These will be discussed in greater detail in [Chapter 12](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch12.html#react_and_the_server). All the tools necessary to generate HTML from the virtual DOM are found in this library.

We can render a React element, including its children, to the DOM with ReactDOM.render. The element that we wish to render is passed as the first argument and the second argument is the target node, where we should render the element:

**var** dish = React.createElement("h1", **null**, "Baked Salmon")

ReactDOM.render(dish, document.getElementById('react-container'))

Rendering the title element to the DOM would add a heading-one element to the div with the id of react-container, which we would already have defined in our HTML. In [Example 4-3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0404), we build this div inside the body tag.

*Example 4-3. React added the h1 element to the target: react-container*

**<body>**

**<div** id="react-container"**>**

**<h1>**Baked Salmon**</h1>**

**</div>**

**</body>**

All of the DOM rendering functionality in React has been moved to ReactDOM because we can use React to build native applications as well. The browser is just one target for React.

That’s all you need to do. You create an element, and then you render it to the DOM. In the next section, we’ll get an understanding of how to use props.children.

**Children**

ReactDOM allows you to render a single element to the DOM.[6](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391605484176) React tags this as data-reactroot. All other React elements are composed into a single element using nesting.

React renders child elements using props.children. In the previous section, we rendered a text element as a child of the h1 element, and thus props.children was set to "Baked Salmon". We could render other React elements as children too, creating a tree of elements. This is why we use the term *component tree*. The tree has one root component from which many branches grow.

Let’s consider the unordered list that contains ingredients in [Example 4-4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0405).

*Example 4-4. Ingredients list*

**<ul>**

**<li>**1 lb Salmon**</li>**

**<li>**1 cup Pine Nuts**</li>**

**<li>**2 cups Butter Lettuce**</li>**

**<li>**1 Yellow Squash**</li>**

**<li>**1/2 cup Olive Oil**</li>**

**<li>**3 cloves of Garlic**</li>**

**</ul>**

In this sample, the unordered list is the root element, and it has six children. We can represent this ul and its children with React.createElement ([Example 4-5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0406)).

*Example 4-5. Unordered list as React elements*

React.createElement(

"ul",

**null**,

React.createElement("li", **null**, "1 lb Salmon"),

React.createElement("li", **null**, "1 cup Pine Nuts"),

React.createElement("li", **null**, "2 cups Butter Lettuce"),

React.createElement("li", **null**, "1 Yellow Squash"),

React.createElement("li", **null**, "1/2 cup Olive Oil"),

React.createElement("li", **null**, "3 cloves of Garlic")

)

Every additional argument sent to the createElement function is another child element. React creates an array of these child elements and sets the value of props.children to that array.

If we were to inspect the resulting React element, we would see each list item represented by a React element and added to an array called props.children. Let’s do that now ([Example 4-6](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0407)).

*Example 4-6. Resulting React element*

{

"type": "ul",

"props": {

"children": [

{ "type": "li", "props": { "children": "1 lb Salmon" } … },

{ "type": "li", "props": { "children": "1 cup Pine Nuts"} … },

{ "type": "li", "props": { "children": "2 cups Butter Lettuce" } … },

{ "type": "li", "props": { "children": "1 Yellow Squash"} … },

{ "type": "li", "props": { "children": "1/2 cup Olive Oil"} … },

{ "type": "li", "props": { "children": "3 cloves of Garlic"} … }

]

...

}

}

We can now see that each list item is a child. Earlier in this chapter, we introduced HTML for an entire recipe rooted in a section element. To create this using React, we’ll use a series of createElement calls, as in [Example 4-7](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0408).

*Example 4-7. React Element tree*

React.createElement("section", {id: "baked-salmon"},

React.createElement("h1", **null**, "Baked Salmon"),

React.createElement("ul", {"className": "ingredients"},

React.createElement("li", **null**, "1 lb Salmon"),

React.createElement("li", **null**, "1 cup Pine Nuts"),

React.createElement("li", **null**, "2 cups Butter Lettuce"),

React.createElement("li", **null**, "1 Yellow Squash"),

React.createElement("li", **null**, "1/2 cup Olive Oil"),

React.createElement("li", **null**, "3 cloves of Garlic")

),

React.createElement("section", {"className": "instructions"},

React.createElement("h2", **null**, "Cooking Instructions"),

React.createElement("p", **null**, "Preheat the oven to 350 degrees."),

React.createElement("p", **null**,

"Spread the olive oil around a glass baking dish."),

React.createElement("p", **null**, "Add the salmon, garlic, and pine..."),

React.createElement("p", **null**, "Bake for 15 minutes."),

React.createElement("p", **null**, "Add the yellow squash and put..."),

React.createElement("p", **null**, "Remove from oven and let cool for 15 ....")

)

)

**CLASSNAME IN REACT**

Any element that has an HTML class attribute is using className for that property instead of class. Since class is a reserved word in JavaScript, we have to use className to define the classattribute of an HTML element.

This sample is what pure React looks like. Pure React is ultimately what runs in the browser. The virtual DOM is a tree of React elements all stemming from a single root element. React elements are the instructions that React will use to build a UI in the browser.

**Constructing Elements with Data**

The major advantage of using React is its ability to separate data from UI elements. Since React is just JavaScript, we can add JavaScript logic to help us build the React component tree. For example, ingredients can be stored in an array, and we can map that array to the React elements.

Let’s go back and think about the unordered list in [Example 4-8](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0409) for a moment.

*Example 4-8. Unordered list*

React.createElement("ul", {"className": "ingredients"},

React.createElement("li", **null**, "1 lb Salmon"),

React.createElement("li", **null**, "1 cup Pine Nuts"),

React.createElement("li", **null**, "2 cups Butter Lettuce"),

React.createElement("li", **null**, "1 Yellow Squash"),

React.createElement("li", **null**, "1/2 cup Olive Oil"),

React.createElement("li", **null**, "3 cloves of Garlic")

);

The data used in this list of ingredients can be easily represented using a JavaScript array ([Example 4-9](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0410)).

*Example 4-9. items array*

**var** items = [

"1 lb Salmon",

"1 cup Pine Nuts",

"2 cups Butter Lettuce",

"1 Yellow Squash",

"1/2 cup Olive Oil",

"3 cloves of Garlic"

]

We could construct a virtual DOM around this data using the Array.map function, as in [Example 4-10](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0411).

*Example 4-10. Mapping an array to li elements*

React.createElement(

"ul",

{ className: "ingredients" },

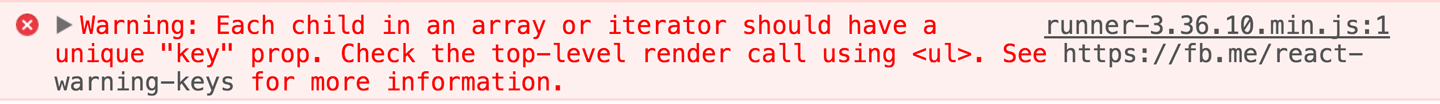
items.map(ingredient =>

React.createElement("li", **null**, ingredient)

)

This syntax creates a React element for each ingredient in the array. Each string is displayed in the list item’s children as text. The value for each ingredient is displayed as the list item.

When running this code, you’ll see a console error, as shown in [Figure 4-2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#fig04in01).



*Figure 4-2. Console warning*

When we build a list of child elements by iterating through an array, React likes each of those elements to have a key property. The key property is used by React to help it update the DOM efficiently. We will be discussing keys and why we need them in [Chapter 5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#react_with_jsx), but for now you can make this warning go away by adding a unique key property to each of the list item elements ([Example 4-11](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0412)). We can use the array index for each ingredient as that unique value.

*Example 4-11. Adding a key property*

React.createElement("ul", {className: "ingredients"},

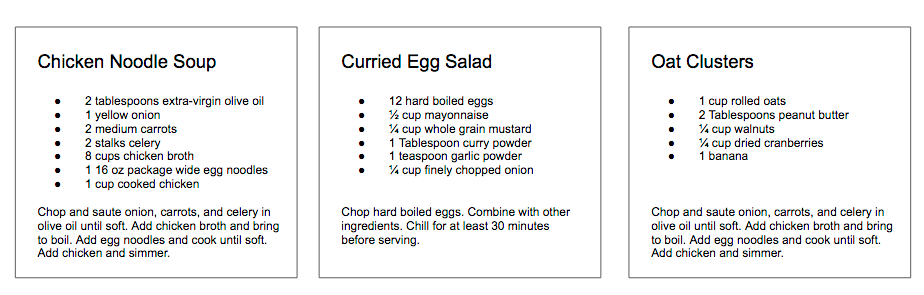
items.map((ingredient, i) =>

React.createElement("li", { key: i }, ingredient)

)

**React Components**

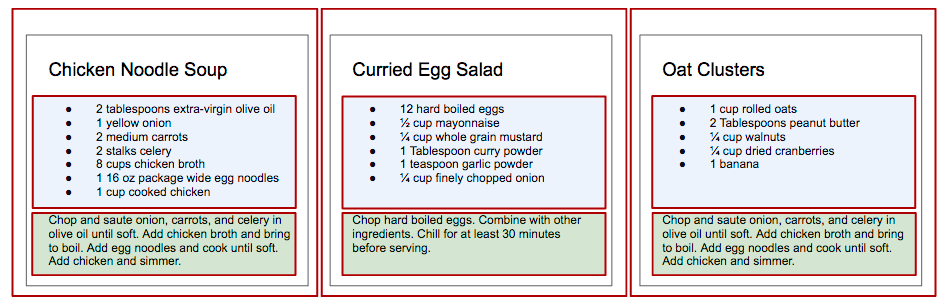
Every user interface is made up of parts. The recipe example we’ll use here has a few recipes, each made up of parts ([Figure 4-3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#fig0402)).



*Figure 4-3. Recipes app*

In React, we describe each of these parts as a *component*. Components allow us to reuse the same DOM structure for different recipes or different sets of data.

When considering a user interface that you want to build with React, look for opportunities to break down your elements into reusable pieces. For example, the recipes in [Figure 4-4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#fig0403) each have a title, ingredients list, and instructions. All are part of a larger recipe or app component. We could create a component for each of the highlighted parts: ingredients, instructions, and so on.



*Figure 4-4. Each component is outlined: App, IngredientsList, Instructions*

Think about how scalable this is. If we want to display one recipe, our component structure will support this. If we want to display 10,000 recipes, we’ll just create new instances of that component.

Let’s investigate the three different ways to create components: createClass, ES6 classes, and stateless functional components.

**React.createClass**

When React was first introduced in 2013, there was only one way to create a component: the createClassfunction.

New methods of creating components have emerged, but createClass is still used widely in React projects. The React team has indicated, however, that createClass may be deprecated in the future.

Let’s consider the list of ingredients that are included in each recipe. As shown in [Example 4-12](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0413), we can create a React component using React.createClass that returns a single unordered list element that contains a child list item for each ingredient in an array.

*Example 4-12. Ingredients list as a React component*

**const** IngredientsList = React.createClass({

displayName: "IngredientsList",

render() {

**return** React.createElement("ul", {"className": "ingredients"},

React.createElement("li", **null**, "1 lb Salmon"),

React.createElement("li", **null**, "1 cup Pine Nuts"),

React.createElement("li", **null**, "2 cups Butter Lettuce"),

React.createElement("li", **null**, "1 Yellow Squash"),

React.createElement("li", **null**, "1/2 cup Olive Oil"),

React.createElement("li", **null**, "3 cloves of Garlic")

)

}

})

**const** list = React.createElement(IngredientsList, **null**, **null**)

ReactDOM.render(

list,

document.getElementById('react-container')

)

Components allow us to use data to build a reusable UI. In the render function, we can use the this keyword to refer to the component instance, and properties can be accessed on that instance with this.props.

Here, we have created an element using our component and named it IngredientsList:

**<IngredientsList>**

**<ul** className="ingredients"**>**

**<li>**1 lb Salmon**</li>**

**<li>**1 cup Pine Nuts**</li>**

**<li>**2 cups Butter Lettuce**</li>**

**<li>**1 Yellow Squash**</li>**

**<li>**1/2 cup Olive Oil**</li>**

**<li>**3 cloves of Garlic**</li>**

**</ul>**

**</IngredientsList>**

Data can be passed to React components as properties. We can create a reusable list of ingredients by passing that data to the list as an array:

**const** IngredientsList = React.createClass({

displayName: "IngredientsList",

render() {

**return** React.createElement("ul", {className: "ingredients"},

**this**.props.items.map((ingredient, i) =>

React.createElement("li", { key: i }, ingredient)

)

)

}

})

**const** items = [

"1 lb Salmon",

"1 cup Pine Nuts",

"2 cups Butter Lettuce",

"1 Yellow Squash",

"1/2 cup Olive Oil",

"3 cloves of Garlic"

]

ReactDOM.render(

React.createElement(IngredientsList, {items}, **null**),

document.getElementById('react-container')

)

Now, let’s look at ReactDOM. The data property items is an array with six ingredients. Because we made the litags using a loop, we were able to add a unique key using the index of the loop:

**<IngredientsList** items=[...]**>**

**<ul** className="ingredients"**>**

**<li** key="0"**>**1 lb Salmon**</li>**

**<li** key="1"**>**1 cup Pine Nuts**</li>**

**<li** key="2"**>**2 cups Butter Lettuce**</li>**

**<li** key="3"**>**1 Yellow Squash**</li>**

**<li** key="4"**>**1/2 cup Olive Oil**</li>**

**<li** key="5"**>**3 cloves of Garlic**</li>**

**</ul>**

**</IngredientsList>**

The components are objects. They can be used to encapsulate code just like classes. We can create a method that renders a single list item and use that to build out the list ([Example 4-13](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0414)).

*Example 4-13. With a custom method*

**const** IngredientsList = React.createClass({

displayName: "IngredientsList",

renderListItem(ingredient, i) {

**return** React.createElement("li", { key: i }, ingredient)

},

render() {

**return** React.createElement("ul", {className: "ingredients"},

**this**.props.items.map(**this**.renderListItem)

)

}

})

This is also the idea of views in MVC languages. Everything that is associated with the UI for IngredientsList is encapsulated into one component; everything we need is right there.

Now we can create a React element using our component and pass it to the list of elements as a property. Notice that the element’s type is now a string—it’s the component class directly.

**COMPONENT CLASSES AS TYPES**

When rendering HTML or SVG elements, we use strings. When creating elements with components, we use the component class directly. This is why IngredientsList is not surrounded in quotation marks; we are passing the class to createElement because it is a component. React will create an instance of our component with this class and manage it for us.

Using the IngredientsList component with this data would render the following unordered list to the DOM:

**<ul** data-react-root class="ingredients"**>**

**<li>**1 lb Salmon**</li>**

**<li>**1 cup Pine Nuts**</li>**

**<li>**2 cups Butter Lettuce**</li>**

**<li>**1 Yellow Squash**</li>**

**<li>**1/2 cup Olive Oil**</li>**

**<li>**3 cloves of Garlic**</li>**

**</ul>**

**React.Component**

As discussed in [Chapter 2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch02.html#emerging_javascript), one of the key features included in the ES6 spec is React.Component, an abstract class that we can use to build new React components. We can create custom components through inheritance by extending this class with ES6 syntax. We can create IngredientsList using the same syntax ([Example 4-14](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0415)).

*Example 4-14. IngredientsList as an ES6 class*

**class** IngredientsList **extends** React.Component {

renderListItem(ingredient, i) {

**return** React.createElement("li", { key: i }, ingredient)

}

render() {

**return** React.createElement("ul", {className: "ingredients"},

**this**.props.items.map(**this**.renderListItem)

)

}

}

**Stateless Functional Components**

Stateless functional components are functions, not objects; therefore, they do not have a “this” scope. Because they are simple, pure functions, we’ll use them as much as possible in our applications. There may come a point where the stateless functional component isn’t robust enough and we must fall back to using class or createClass, but in general the more you can use these, the better.

Stateless functional components are functions that take in properties and return a DOM element. Stateless functional components are a good way to practice the rules of functional programming. You should strive to make each stateless functional component a pure function. They should take in props and return a DOM element without causing side effects. This encourages simplicity and makes the codebase extremely testable.

Stateless functional components will keep your application architecture simple, and the React team promises some performance gains by using them. If you need to encapsulate functionality or have a this scope, however, you can’t use them.

In [Example 4-15](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0416), we combine the functionality of renderListItem and render into a single function.

*Example 4-15. Creating a stateless functional component*

**const** IngredientsList = props =>

React.createElement("ul", {className: "ingredients"},

props.items.map((ingredient, i) =>

React.createElement("li", { key: i }, ingredient)

)

)

We would render this component with ReactDOM.render, the exact same way we render components created with createClass or ES6 class syntax. This is just a function. The function collects data through the props arguments and returns an unordered list for each item that is sent to the props data.

One way we can improve this stateless functional component is through destructuring the properties argument ([Example 4-16](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0417)). Using ES6 destructuring syntax, we can scope the list property directly to this function, reducing the repetitive dot syntax. Now we’d use the IngredientsList the same way we render component classes.

*Example 4-16. Destructuring the properties argument*

**const** IngredientsList = ({items}) =>

React.createElement("ul", {className: "ingredients"},

items.map((ingredient, i) =>

React.createElement("li", { key: i }, ingredient)

)

)

**CONST WITH STATELESS FUNCTIONAL COMPONENTS**

Each of these stateless functional components uses const instead of var when creating a component. This is a common practice but not a requirement. const declares this function as a constant and prevents us from redefining that variable later.

Aside from being slightly cleaner syntax, Facebook has hinted that in the future, stateless functional components might be faster than createClass or ES6 class syntax.

**DOM Rendering**

Since we are able to pass data to our components as props, we can separate our application’s data from the logic that is used to create the UI. This gives us an isolated set of data that is much easier to work with and manipulate than the document object model. When we change any of the values in this isolated dataset, we change the state of our application.

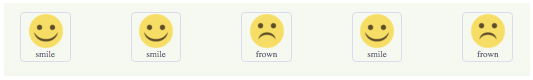
Imagine storing all of the data in your application in a single JavaScript object. Every time you made a change to this object, you could send it to a component as props and rerender the UI. This means that ReactDOM.renderis going to be doing a lot of heavy lifting.

In order for React to work in a reasonable amount of time, ReactDOM.render has to work smart, and it does. Instead of emptying and reconstructing the entire DOM, ReactDOM.render leaves the current DOM in place and only applies the minimal amount of changes required to mutate the DOM.

Let’s say we had an app that displayed the mood of our five team members using either a smiley face or a frowny face. We can represent the mood of all five individuals in a single JavaScript array:

["smile", "smile", "frown", "smile", "frown"];

This array of data may be used to construct a UI that looks something like this:



If something breaks and the team has to work all weekend, we can reflect the team’s new mood simply by changing the data in this array, producing the result shown in the image that follows:

["frown", "frown", "frown", "frown", "frown"];



How many changes do we have to make to the first array to make it look like the second array of all frowns?

[**"smile", "smile"**, "frown", **"smile"**, "frown"];

[**"frown", "frown"**, "frown", **"frown"**, "frown"];

We would need to change the first, second, and fourth values from a smile to a frown.

Therefore, we can say that it would take three mutations to change the first array of data to match the second.

Now consider how we can update the DOM to reflect these changes. One inefficient solution to applying these changes to the UI is to erase the entire DOM and rebuild it, as in [Example 4-17](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0418).

*Example 4-17. Start with the current list*

<ul>

<li **class**="smile">smile</li>

<li **class**="smile">smile</li>

<li **class**="frown">frown</li>

<li **class**="smile">smile</li>

<li **class**="frown">frown</li>

</ul>

This involves the following steps:

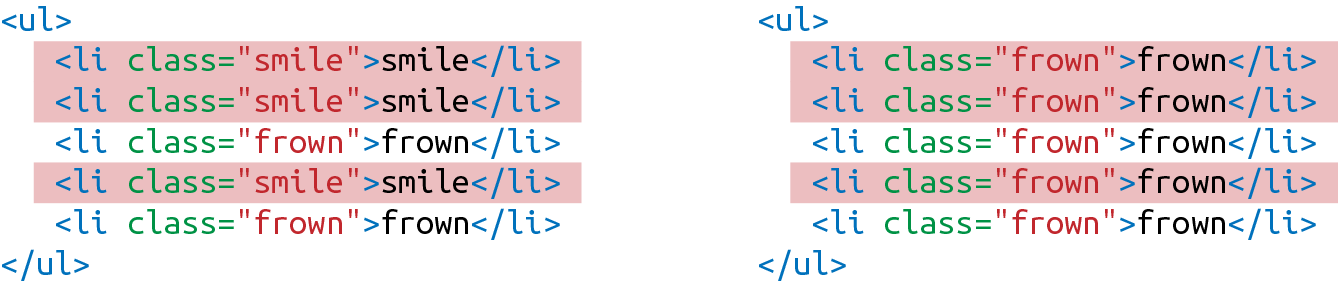
1. Empty the current data:
2. **<ul>**

**</ul>**

1. Begin looping through data and build the first list item:
2. **<ul>**
3. **<li** class="frown"**>**frown**</li>**
4. **</ul>**
5. Build and add the second list item:
6. **<ul>**
7. **<li** class="frown"**>**frown**</li>**
8. **<li** class="frown"**>**frown**</li>**
9. **</ul>**
10. Build and append the third list item:
11. **<ul>**
12. **<li** class="frown"**>**frown**</li>**
13. **<li** class="frown"**>**frown**</li>**
14. **<li** class="frown"**>**frown**</li>**
15. **</ul>**
16. Build and append the fourth list item:
17. **<ul>**
18. **<li** class="frown"**>**frown**</li>**
19. **<li** class="frown"**>**frown**</li>**
20. **<li** class="frown"**>**frown**</li>**
21. **<li** class="frown"**>**frown**</li>**
22. **</ul>**
23. Build and append the fifth list item:
24. **<ul>**
25. **<li** class="frown"**>**frown**</li>**
26. **<li** class="frown"**>**frown**</li>**
27. **<li** class="frown"**>**frown**</li>**
28. **<li** class="frown"**>**frown**</li>**
29. **<li** class="frown"**>**frown**</li>**
30. **</ul>**

If we change the UI by erasing and rebuilding the DOM, we are creating and inserting five new DOM elements. Inserting an element into the DOM is one of the most costly DOM API operations—it’s slow. In contrast, updating DOM elements that are already in place performs much more quickly than inserting new ones.

ReactDOM.render makes changes by leaving the current DOM in place and simply updating the DOM elements that need to be updated. In our example, there are only three mutations, so ReactDOM.render only needs to update three DOM elements (see [Figure 4-5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#fig0405)).



*Figure 4-5. Three DOM elements are updated*

If new DOM elements need to be inserted, ReactDOM will insert them, but it tries to keep DOM insertions (the most costly operation) to a minimum.

This smart DOM rendering is necessary for React to work in a reasonable amount of time because our application state changes a lot. Every time we change that state, we are going to rely on ReactDOM.render to efficiently rerender the UI.

**Factories**

So far, the only way we have created elements has been with React.createElement. Another way to create a React element is to use factories. A factory is a special object that can be used to abstract away the details of instantiating objects. In React, we use factories to help us create React element instances.

React has built-in factories for all commonly supported HTML and SVG DOM elements, and you can use the React.createFactory function to build your own factories around specific components.

For example, consider our h1 element from earlier in this chapter:

<h1>Baked Salmon</h1>

Instead of using createElement, we can create a React element with a built-in factory ([Example 4-18](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0419)).

*Example 4-18. Using createFactory to create an h1*

React.DOM.h1(**null**, "Baked Salmon")

In this case, the first argument is for the properties and the second argument is for the children. We can also use DOM factories to build an unordered list, as in [Example 4-19](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0420).

*Example 4-19. Building an unordered list with DOM factories*

React.DOM.ul({"className": "ingredients"},

React.DOM.li(**null**, "1 lb Salmon"),

React.DOM.li(**null**, "1 cup Pine Nuts"),

React.DOM.li(**null**, "2 cups Butter Lettuce"),

React.DOM.li(**null**, "1 Yellow Squash"),

React.DOM.li(**null**, "1/2 cup Olive Oil"),

React.DOM.li(**null**, "3 cloves of Garlic")

)

In this case, the first argument is for the properties, where we define the className. Additional arguments are elements that will be added to the children array of the unordered list. We can also separate out the ingredient data and improve the preceding definition using factories ([Example 4-20](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0421)).

*Example 4-20. Using map with factories*

**var** items = [

"1 lb Salmon",

"1 cup Pine Nuts",

"2 cups Butter Lettuce",

"1 Yellow Squash",

"1/2 cup Olive Oil",

"3 cloves of Garlic"

]

**var** list = React.DOM.ul(

{ className: "ingredients" },

items.map((ingredient, key) =>

React.DOM.li({key}, ingredient)

)

)

ReactDOM.render(

list,

document.getElementById('react-container')

)

**Using Factories with Components**

If you would like to simplify your code by calling components as functions, you need to explicitly create a factory ([Example 4-21](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#example0422)).

*Example 4-21. Creating a factory with IngredientsList*

**const** { render } = ReactDOM;

**const** IngredientsList = ({ list }) =>

React.createElement('ul', **null**,

list.map((ingredient, i) =>

React.createElement('li', {key: i}, ingredient)

)

)

**const** Ingredients = React.createFactory(IngredientsList)

**const** list = [

"1 lb Salmon",

"1 cup Pine Nuts",

"2 cups Butter Lettuce",

"1 Yellow Squash",

"1/2 cup Olive Oil",

"3 cloves of Garlic"

]

render(

Ingredients({list}),

document.getElementById('react-container')

)

In this example, we can quickly render a React element with the Ingredients factory. Ingredients is a function that takes in properties and children as arguments just like the DOM factories.

If you are not working with JSX, you may find using factories preferable to numerous React.createElementcalls. However, the easiest and most common way to define React elements is with JSX tags. If you use JSX with React, chances are you will never use a factory.

Throughout this chapter, we’ve used createElement and createFactory to build React components. In [Chapter 5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#react_with_jsx), we’ll take a look at how to simplify component creation by using JSX.

[1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391606008080-marker) Ben Alpert, [“React v0.14”](http://bit.ly/2nvPHEQ), React blog, October 7, 2015.

[2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391605874640-marker) Lindsey Simon, [“Minimizing Browser Reflow”](http://bit.ly/2m1pa58).

[3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391605872960-marker) Steven Luscher, [“Building User Interfaces with Facebook’s React”](http://bit.ly/2m1pEs3), Super VanJS 2013.

[4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391605870576-marker) Mark Wilton-Jones, [“Efficient JavaScript”](http://opr.as/2m1f5Fr), Dev.Opera, November 2, 2006.

[5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391605867456-marker) React Docs, [“Refs and the DOM”](http://bit.ly/2m1faJf).

[6](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html#idm140391605484176-marker) [Rendering Elements](http://bit.ly/2nvR2vf)

* [**Copy**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html)
* [**Add Highlight**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html)
* [**Add Note**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch04.html)

**Chapter 5. React with JSX**

In the last chapter, we looked at how the virtual DOM is a set of instructions that React follows when creating and updating a user interface. These instructions are made up of JavaScript objects called React elements. So far, we’ve learned two ways to create React elements: using React.createElement and using factories.

An alternative to typing out verbose React.createElement calls is JSX, a JavaScript extension that allows us to define React elements using syntax that looks similar to HTML. In this chapter, we are going to discuss how to use JSX to construct a virtual DOM with React elements.

**React Elements as JSX**

Facebook’s React team released JSX when they released React to provide a concise syntax for creating complex DOM trees with attributes. They also hoped to make React more readable, like HTML and XML.

In JSX, an element’s type is specified with a tag. The tag’s attributes represent the properties. The element’s children can be added between the opening and closing tags.

You can also add other JSX elements as children. If you have an unordered list, you can add child list item elements to it with JSX tags. It looks very similar to HTML (see [Example 5-1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0501)).

*Example 5-1. JSX for an unordered list*

**<ul>**

**<li>**1 lb Salmon**</li>**

**<li>**1 cup Pine Nuts**</li>**

**<li>**2 cups Butter Lettuce**</li>**

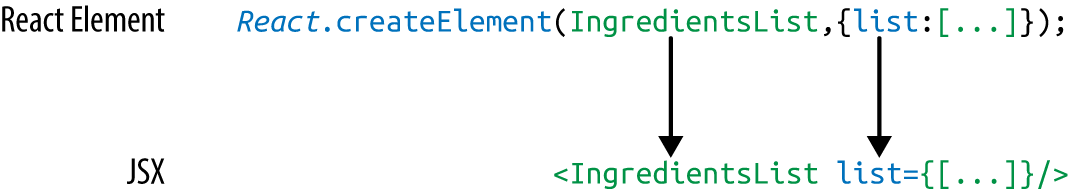
**<li>**1 Yellow Squash**</li>**

**<li>**1/2 cup Olive Oil**</li>**

**<li>**3 cloves of Garlic**</li>**

**</ul>**

JSX works with components as well. Simply define the component using the class name. In [Figure 5-1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#fig0501), we pass an array of ingredients to the IngredientsList as a property with JSX.



*Figure 5-1. Creating the IngredientsList with JSX*

When we pass the array of ingredients to this component, we need to surround it with curly braces. This is called a JavaScript *expression*, and we must use these when passing JavaScript values to components as properties. Component properties will take two types: either a string or a JavaScript expression. JavaScript expressions can include arrays, objects, and even functions. In order to include them, you must surround them in curly braces.

**JSX Tips**

JSX might look familiar, and most of the rules result in syntax that is similar to HTML. However, there are a few considerations that you should understand when working with JSX.

**NESTED COMPONENTS**

JSX allows you to add components as children of other components. For example, inside the IngredientsList, we can render another component called Ingredient multiple times ([Example 5-2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0502)).

*Example 5-2. IngredientsList with three nested Ingredient components*

**<IngredientsList>**

**<Ingredient** **/>**

**<Ingredient** **/>**

**<Ingredient** **/>**

**</IngredientsList>**

**CLASSNAME**

Since class is a reserved word in JavaScript, className is used to define the class attribute instead:

**<h1** className="fancy"**>**Baked Salmon**</h1>**

**JAVASCRIPT EXPRESSIONS**

JavaScript expressions are wrapped in curly braces and indicate where variables shall be evaluated and their resulting values returned. For example, if we want to display the value of the title property in an element, we can insert that value using a JavaScript expression. The variable will be evaluated and its value returned:

<h1>{**this**.props.title}</h1>

Values of types other than string should also appear as JavaScript expressions:

**<input** type="checkbox" defaultChecked={false} **/>**

**EVALUATION**

The JavaScript that is added in between the curly braces will get evaluated. This means that operations such as concatenation or addition will occur. This also means that functions found in JavaScript expressions will be invoked:

<h1>{"Hello" + **this**.props.title}</h1>

<h1>{**this**.props.title.toLowerCase().replace}</h1>

**function** appendTitle({**this**.props.title}) {

console.log(`${**this**.props.title} is great!`)

}

**MAPPING ARRAYS TO JSX**

JSX is JavaScript, so you can incorporate JSX directly inside of JavaScript functions. For example, you can map an array to JSX elements ([Example 5-3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0503)).

*Example 5-3. Array.map() with JSX*

**<ul>**

{this.props.ingredients.map((ingredient, i) =>

**<li** key={i}**>**{ingredient}**</li>**

)}

**</ul>**

JSX looks clean and readable, but it can’t be interpreted with a browser. All JSX must be converted into createElement calls or factories. Luckily, there is an excellent tool for this task: Babel.

**Babel**

Most software languages allow you to compile your source code. JavaScript is an interpreted language: the browser interprets the code as text, so there is no need to compile JavaScript. However, not all browsers support the latest ES6 and ES7 syntax, and no browser supports JSX syntax. Since we want to use the latest features of JavaScript along with JSX, we are going to need a way to convert our fancy source code into something that the browser can interpret. This process is called transpiling, and it is what [Babel](https://babeljs.io/) is designed to do.

The first version of the project was called 6to5, and it was released in September 2014. 6to5 was a tool that could be used to convert ES6 syntax to ES5 syntax, which is more widely supported by web browsers. As the project grew, it aimed to be a platform to support all of the latest changes in ECMAScript. It also grew to support transpiling JSX into pure React. The project was renamed to Babel in February 2015.

Babel is used in production at Facebook, Netflix, PayPal, Airbnb, and more. Previously, Facebook had created a JSX transformer that was their standard, but it was soon retired in favor of Babel.

There are many ways of working with Babel. The easiest way to get started is to include a link to the babel-core transpiler directly in your HTML, which will transpile any code in script blocks that have a type of “text/babel”. Babel will transpile the source code on the client before running it. Although this may not be the best solution for production, it is a great way to get started with JSX (see [Example 5-4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0504)).

*Example 5-4. Including babel-core*

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** charset="utf-8"**>**

**<title>**React Examples**</title>**

**</head>**

**<body>**

**<div** class="react-container"**></div>**

*<!-- React Library & React DOM -->*

**<script** src="https://unpkg.com/react@15.4.2/dist/react.js"**></script>**

**<script** src="https://unpkg.com/react-dom@15.4.2/dist/react-dom.js"**></script>**

**<script**

src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.29/browser.js"**>**

**</script>**

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

*// JSX code here. Or link to separate JavaScript file that contains JSX.*

**</script>**

**</body>**

**</html>**

**BABEL V5.8 REQUIRED**

To transpile code in the browser, use Babel v. 5.8. Babel 6.0+ will not work as an in-browser transformer.

[Later in the chapter](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#babel_presets), we’ll look at how we can use Babel with webpack to transpile our JavaScript files statically. For now, using the in-browser transpiler will do.

**Recipes as JSX**

One of the reasons that we have grown to love React is that it allows us to write web applications with beautiful code. It is extremely rewarding to create beautifully written modules that clearly communicate how the application functions. JSX provides us with a nice, clean way to express React elements in our code that makes sense to us and is immediately readable by the engineers that make up our community. The drawback of JSX is that it is not readable by the browser. Before our code can be interpreted by the browser, it needs to be converted from JSX into pure React.

The array in [Example 5-5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0505) contains two recipes, and they represent our application’s current state.

*Example 5-5. Array of recipes*

**var** data = [

{

"name": "Baked Salmon",

"ingredients": [

{ "name": "Salmon", "amount": 1, "measurement": "l lb" },

{ "name": "Pine Nuts", "amount": 1, "measurement": "cup" },

{ "name": "Butter Lettuce", "amount": 2, "measurement": "cups" },

{ "name": "Yellow Squash", "amount": 1, "measurement": "med" },

{ "name": "Olive Oil", "amount": 0.5, "measurement": "cup" },

{ "name": "Garlic", "amount": 3, "measurement": "cloves" }

],

"steps": [

"Preheat the oven to 350 degrees.",

"Spread the olive oil around a glass baking dish.",

"Add the salmon, garlic, and pine nuts to the dish.",

"Bake for 15 minutes.",

"Add the yellow squash and put back in the oven for 30 mins.",

"Remove from oven and let cool for 15 minutes. Add the lettuce and serve."

]

},

{

"name": "Fish Tacos",

"ingredients": [

{ "name": "Whitefish", "amount": 1, "measurement": "l lb" },

{ "name": "Cheese", "amount": 1, "measurement": "cup" },

{ "name": "Iceberg Lettuce", "amount": 2, "measurement": "cups" },

{ "name": "Tomatoes", "amount": 2, "measurement": "large"},

{ "name": "Tortillas", "amount": 3, "measurement": "med" }

],

"steps": [

"Cook the fish on the grill until hot.",

"Place the fish on the 3 tortillas.",

"Top them with lettuce, tomatoes, and cheese."

]

}

];

The data is expressed in an array of two JavaScript objects. Each object contains the name of the recipe, a list of the ingredients required, and a list of steps necessary to cook the recipe.

We can create a UI for these recipes with two components: a Menu component for listing the recipes and a Recipe component that describes the UI for each recipe. It’s the Menu component that we will render to the DOM. We will pass our data to the Menu component as a property called recipes ([Example 5-6](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0506)).

*Example 5-6. Recipe app code structure*

*// The data, an array of Recipe objects*

**var** data = [ ... ];

*// A stateless functional component for an individual Recipe*

**const** Recipe = (props) => (

...

)

*// A stateless functional component for the Menu of Recipes*

**const** Menu = (props) => (

...

)

*// A call to ReactDOM.render to render our Menu into the current DOM*

ReactDOM.render(

<Menu recipes={data} title="Delicious Recipes" />,

document.getElementById("react-container")

)

**ES6 SUPPORT**

We will be using ES6 in this file as well. When we transpile our code from JSX to pure React, Babel will also convert ES6 into common ES5 JavaScript that is readable by all browsers. Any ES6 features used have been discussed in [Chapter 2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch02.html#emerging_javascript).

The React elements within the Menu component are expressed as JSX ([Example 5-7](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0507)). Everything is contained within an article element. A header element, an h1 element, and a div.recipes element are used to describe the DOM for our menu. The value for the title property will be displayed as text within the h1.

*Example 5-7. Menu component structure*

**const** Menu = (props) =>

<article>

<header>

<h1>{props.title}</h1>

</header>

<div className="recipes">

</div>

</article>

Inside of the div.recipes element, we add a component for each recipe ([Example 5-8](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0508)).

*Example 5-8. Mapping recipe data*

<div className="recipes">

{props.recipes.map((recipe, i) =>

<Recipe key={i} name={recipe.name}

ingredients={recipe.ingredients}

steps={recipe.steps} />

)}

</div>

In order to list the recipes within the div.recipes element, we use curly braces to add a JavaScript expression that will return an array of children. We can use the map function on the props.recipes array to return a component for each object within the array. As mentioned previously, each recipe contains a name, some ingredients, and cooking instructions (steps). We will need to pass this data to each Recipe as props. Also remember that we should use the key property to uniquely identify each element.

Using the JSX spread operator can improve our code. The JSX spread operator works like the object spread operator discussed in [Chapter 2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch02.html#emerging_javascript). It will add each field of the recipe object as a property of the Recipecomponent. The syntax in [Example 5-9](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0509) accomplishes the same results.

*Example 5-9. Enhancement: JSX spread operator*

{props.recipes.map((recipe, i) =>

<Recipe key={i} {...recipe} />

)}

Another place we can make an ES6 improvement to our Menu component is where we take in the propsargument. We can use object destructuring to scope the variables to this function. This allows us to access the title and recipes variables directly, no longer having to prefix them with props ([Example 5-10](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0510)).

*Example 5-10. Refactored Menu component*

**const** Menu = ({ title, recipes }) => (

<article>

<header>

<h1>{title}</h1>

</header>

<div className="recipes">

{recipes.map((recipe, i) =>

<Recipe key={i} {...recipe} />

)}

</div>

</article>

)

Now let’s code the component for each individual recipe ([Example 5-11](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0511)).

*Example 5-11. Complete Recipe component*

**const** Recipe = ({ name, ingredients, steps }) =>

<section id={name.toLowerCase().replace(/ /g, "-")}>

<h1>{name}</h1>

<ul className="ingredients">

{ingredients.map((ingredient, i) =>

<li key={i}>{ingredient.name}</li>

)}

</ul>

<section className="instructions">

<h2>Cooking Instructions</h2>

{steps.map((step, i) =>

<p key={i}>{step}</p>

)}

</section>

</section>

This component is also a stateless functional component. Each recipe has a string for the name, an array of objects for ingredients, and an array of strings for the steps. Using ES6 object destructuring, we can tell this component to locally scope those fields by name so we can access them directly without having to use props.name, or props.ingredients, props.steps.

The first JavaScript expression that we see is being used to set the id attribute for the root section element. It is converting the recipe’s name to a lowercase string and globally replacing spaces with dashes. The result is that “Baked Salmon” will be converted to “baked-salmon” (and likewise, if we had a recipe with the name “Boston Baked Beans” it would be converted to “boston-baked-beans”) before it is used as the id attribute in our UI. The value for name is also being displayed in an h1 as a text node.

Inside of the unordered list, a JavaScript expression is mapping each ingredient to an li element that displays the name of the ingredient. Within our instructions section, we see the same pattern being used to return a paragraph element where each step is displayed. These map functions are returning arrays of child elements.

The complete code for the application should look like [Example 5-12](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0512).

*Example 5-12. Finished code for recipe app*

**const** data = [

{

"name": "Baked Salmon",

"ingredients": [

{ "name": "Salmon", "amount": 1, "measurement": "l lb" },

{ "name": "Pine Nuts", "amount": 1, "measurement": "cup" },

{ "name": "Butter Lettuce", "amount": 2, "measurement": "cups" },

{ "name": "Yellow Squash", "amount": 1, "measurement": "med" },

{ "name": "Olive Oil", "amount": 0.5, "measurement": "cup" },

{ "name": "Garlic", "amount": 3, "measurement": "cloves" }

],

"steps": [

"Preheat the oven to 350 degrees.",

"Spread the olive oil around a glass baking dish.",

"Add the salmon, garlic, and pine nuts to the dish.",

"Bake for 15 minutes.",

"Add the yellow squash and put back in the oven for 30 mins.",

"Remove from oven and let cool for 15 minutes. Add the lettuce and serve."

]

},

{

"name": "Fish Tacos",

"ingredients": [

{ "name": "Whitefish", "amount": 1, "measurement": "l lb" },

{ "name": "Cheese", "amount": 1, "measurement": "cup" },

{ "name": "Iceberg Lettuce", "amount": 2, "measurement": "cups" },

{ "name": "Tomatoes", "amount": 2, "measurement": "large"},

{ "name": "Tortillas", "amount": 3, "measurement": "med" }

],

"steps": [

"Cook the fish on the grill until hot.",

"Place the fish on the 3 tortillas.",

"Top them with lettuce, tomatoes, and cheese."

]

}

]

**const** Recipe = ({ name, ingredients, steps }) =>

<section id={name.toLowerCase().replace(/ /g, "-")}>

<h1>{name}</h1>

<ul className="ingredients">

{ingredients.map((ingredient, i) =>

<li key={i}>{ingredient.name}</li>

)}

</ul>

<section className="instructions">

<h2>Cooking Instructions</h2>

{steps.map((step, i) =>

<p key={i}>{step}</p>

)}

</section>

</section>

**const** Menu = ({ title, recipes }) =>

<article>

<header>

<h1>{title}</h1>

</header>

<div className="recipes">

{recipes.map((recipe, i) =>

<Recipe key={i} {...recipe} />

)}

</div>

</article>

ReactDOM.render(

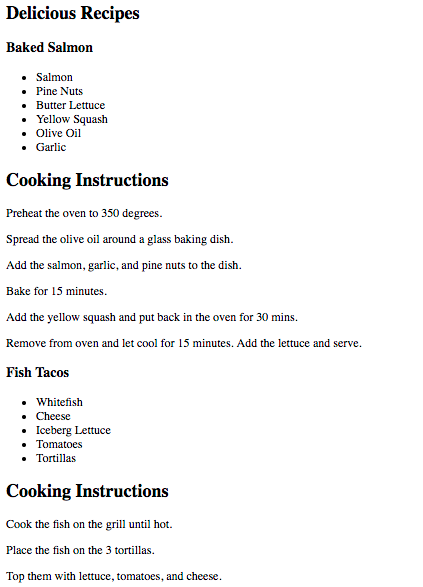
<Menu recipes={data}

title="Delicious Recipes" />,

document.getElementById("react-container")

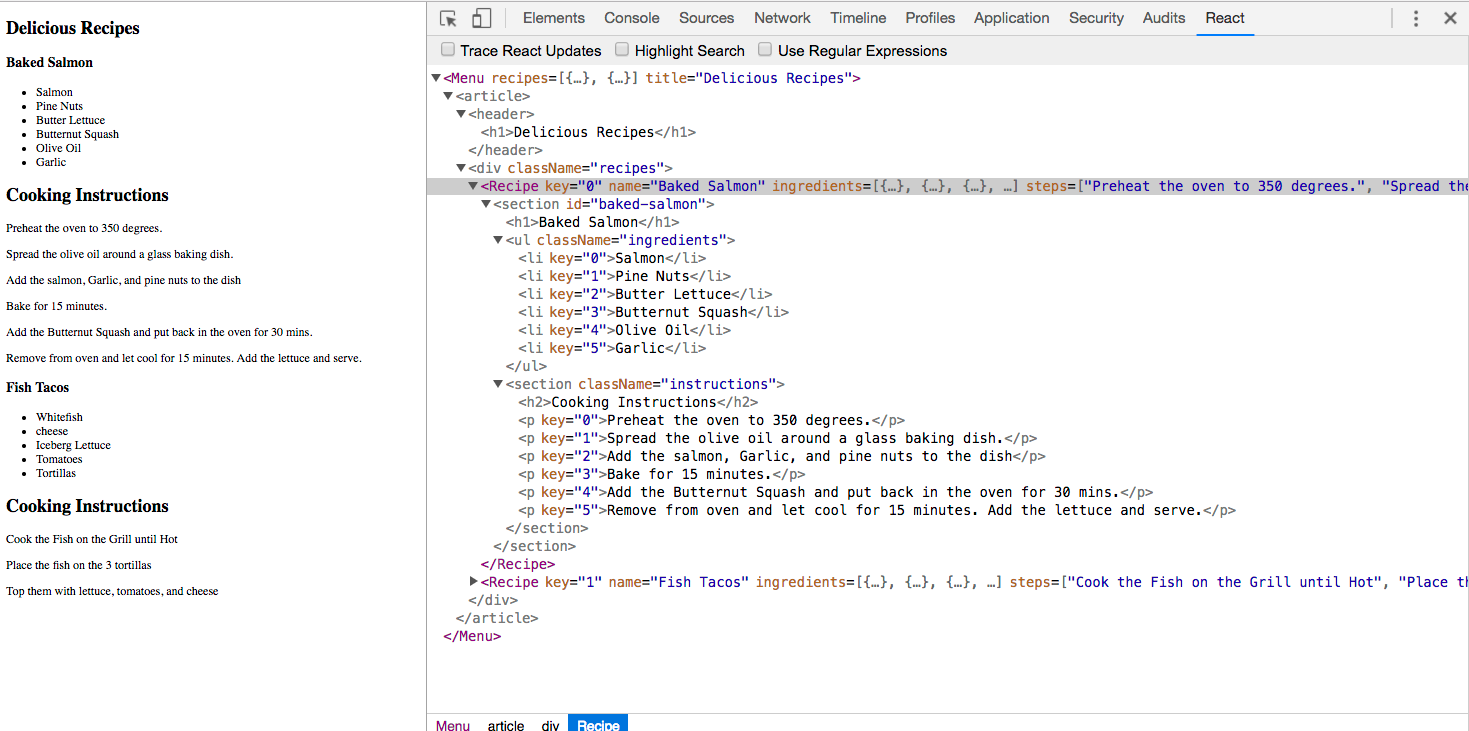
)

When we run this code in the browser, React will construct a UI using our instructions with the recipe data as shown in [Figure 5-2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#fig0502).



*Figure 5-2. Delicious Recipes output*

If you are using Google Chrome and you have the React Developer Tools Extension installed, you can take a look at the present state of the virtual DOM. To do this, open the developer tools and select the React tab ([Figure 5-3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#fig0503)).



*Figure 5-3. Resulting virtual DOM in React Developer Tools*

Here we can see our Menu and its child elements. The data array contains two objects for recipes, and we have two Recipe elements. Each Recipe element has properties for the recipe name, ingredients, and steps.

The virtual DOM is constructed based on the application’s state data being passed to the Menu component as a property. If we change the recipes array and rerender our Menu component, React will change this DOM as efficiently as possible.

**BABEL PRESETS**

Babel 6 breaks possible transformations up into modules called *presets*. It requires engineers to explicitly define which transformations should be run by specifying which presets to use. The goal was to make everything more modular to allow developers to decide which syntax should be converted. The plugins fall into a few categories, and all are opt-in based on the needs of the application. The presets you’re most likely to use are:

babel-preset-es2015

Compiles ES2015, or ES6, to ES5.

babel-preset-es2016

Compiles what is in ES2016 to ES2015

babel-preset-es2017

Compiles what is in ES2017 to ES2017

babel-preset-env

Compiles everything from ES2015, ES2016, ES2017. A catch-all for the previous three presets

babel-preset-react

Compiles JSX to React.createElement calls.

When a new feature is proposed for inclusion in the ECMAScript spec, it goes through stages of acceptance from stage 0, Strawman (newly proposed and very experimental), to stage 4, Finished (accepted as part of the standard). Babel provides presets for each of these stages, so you can choose which stage you want to allow in your application:

* babel-preset-stage-0: Strawman
* babel-preset-stage-1: Proposal
* babel-preset-stage-2: Draft
* babel-preset-stage-3: Candidate

**Intro to Webpack**

Once we start working in production with React, there are a lot of questions to consider: How do we want to deal with JSX and ES6+ transformation? How can we manage our dependencies? How can we optimize our images and CSS?

Many different tools have emerged to answer these questions, including Browserify, Gulp, and Grunt. Due to its features and widespread adoption by large companies, *webpack* has also emerged as one of the leading tools for bundling CommonJS modules (see [Chapter 2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch02.html#emerging_javascript) for more on CommonJS).

Webpack is billed as a module bundler. A module bundler takes all of our different files (JavaScript, LESS, CSS, JSX, ES6, and so on) and turns them into a single file. The two main benefits of modular bundling are *modularity*and *network performance*.

Modularity will allow you to break down your source code into parts, or modules, that are easier to work with, especially in a team environment.

Network performance is gained by only needing to load one dependency in the browser, the bundle. Each script tag makes an HTTP request, and there is a latency penalty for each HTTP request. Bundling all of the dependencies into a single file allows you to load everything with one HTTP request, thereby avoiding additional latency.

Aside from transpiling, webpack also can handle:

Code splitting

Splits up your code into different chunks that can be loaded when you need them. Sometimes these are called *rollups* or *layers*; the aim is to break up code as needed for different pages or devices.

Minification

Removes whitespace, line breaks, lengthy variable names, and unnecessary code to reduce the file size.

Feature flagging

Sends code to one or more—but not all—environments when testing out features.

Hot Module Replacement (HMR)

Watches for changes in source code. Changes only the updated modules immediately.

**Webpack Loaders**

A *loader* is a function that handles the transformations that we want to put our code through during the build process. If our application uses ES6, JSX, CoffeeScript, and other languages that can’t be read natively by the browser, we’ll specify the necessary loaders in the *webpack.config.js* file to do the work of converting the code into syntax that can be read natively by the browser.

Webpack has a huge number of loaders that fall into a few categories. The most common use case for loaders is transpiling from one dialect to another. For example, ES6 and React code is transpiled by including the babel-loader. We specify the types of files that Babel should be run on, then webpack takes care of the rest.

Another popular category of loaders is for styling. The css-loader looks for files with the *.scss* extension and compiles them to CSS. The css-loader can be used to include CSS modules in your bundle. All CSS is bundled as JavaScript and automatically added when the bundled JavaScript file is included. There’s no need to use link elements to include stylesheets.

Check out the full [list of loaders](https://webpack.js.org/concepts/loaders/) if you’d like to see all of the different options.

**Recipes App with a Webpack Build**

The Recipes app that we built earlier in this chapter has some limitations that webpack will help us alleviate. Using a tool like webpack to statically build your client JavaScript makes it possible for teams to work together on large-scale web applications. We can also gain the following benefits by incorporating the webpack module bundler:

Modularity

Using the CommonJS module pattern in order to export modules that will later be imported or required by another part of the application makes our source code more approachable. It allows development teams to easily work together by allowing them to create and work with separate files that will be statically combined into a single file before sending to production.

Composing

With modules, we can build small, simple, reusable React components that we can compose efficiently into applications. Smaller components are easier to comprehend, test, and reuse. They are also easier to replace down the line when enhancing your applications.

Speed

Packaging all of the application’s modules and dependencies into a single client bundle will reduce the load time of your application because there is latency associated with each HTTP request. Packaging everything together in a single file means that the client will only need to make a single request. Minifying the code in the bundle will improve load time as well.

Consistency

Since webpack will transpile JSX into React and ES6 or even ES7 into universal JavaScript, we can start using tomorrow’s JavaScript syntax today. Babel supports a wide range of ESNext syntax, which means we do not have to worry about whether the browser supports our code. It allows developers to consistently use cutting-edge JavaScript syntax.

**BREAKING COMPONENTS INTO MODULES**

Approaching the Recipes app with the ability to use webpack and Babel allows us to break our code down into modules that use ES6 syntax. Let’s take a look at our stateless functional component for recipes ([Example 5-13](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0513)).

*Example 5-13. Current Recipe component*

**const** Recipe = ({ name, ingredients, steps }) =>

<section id="baked-salmon">

<h1>{name}</h1>

<ul className="ingredients">

{ingredients.map((ingredient, i) =>

<li key={i}>{ingredient.name}</li>

)}

</ul>

<section className="instructions">

<h2>Cooking Instructions</h2>

{steps.map((step, i) =>

<p key={i}>{step}</p>

)}

</section>

</section>

This component is doing quite a bit. We are displaying the name of the recipe, constructing an unordered list of ingredients, and displaying the instructions, with each step getting its own paragraph element.

A more functional approach to the Recipe component would be to break it up into smaller, more focused stateless functional components and compose them together. We can start by pulling the instructions out into their own stateless functional component and creating a module in a separate file that we can use for any set of instructions ([Example 5-14](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0514)).

*Example 5-14. Instructions component*

**const** Instructions = ({ title, steps }) =>

<section className="instructions">

<h2>{title}</h2>

{steps.map((s, i) =>

<p key={i}>{s}</p>

)}

</section>

**export** **default** Instructions

Here we have created a new component called Instructions. We will pass the title of the instructions and the steps to this component. This way we can reuse this component for “Cooking Instructions,” “Baking Instructions,” “Prep Instructions”, or a “Pre-cook Checklist”—anything that has steps.

Now think about the ingredients. In the Recipe component, we are only displaying the ingredient names, but each ingredient in the data for the recipe has an amount and measurement as well. We could create a stateless functional component to represent a single ingredient ([Example 5-15](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0515)).

*Example 5-15. Ingredient component*

**const** Ingredient = ({ amount, measurement, name }) =>

<li>

<span className="amount">{amount}</span>

<span className="measurement">{measurement}</span>

<span className="name">{name}</span>

</li>

**export** **default** Ingredient

Here we assume each ingredient has an amount, a measurement, and a name. We destructure those values from our props object and display them each in independent classed span elements.

Using the Ingredient component, we can construct an IngredientsList component that can be used any time we need to display a list of ingredients ([Example 5-16](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0516)).

*Example 5-16. IngredientsList using Ingredient component*

**import** Ingredient from './Ingredient'

**const** IngredientsList = ({ list }) =>

<ul className="ingredients">

{list.map((ingredient, i) =>

<Ingredient key={i} {...ingredient} />

)}

</ul>

**export** **default** IngredientsList

In this file, we first import the Ingredient component because we are going to use it for each ingredient. The ingredients are passed to this component as an array in a property called list. Each ingredient in the list array will be mapped to the Ingredient component. The JSX spread operator is used to pass all of the data to the Ingredient component as props.

Using spread operator:

<Ingredient {...ingredient} />

is another way of expressing:

<Ingredient amount={ingredient.amount}

measurement={ingredient.measurement}

name={ingredient.name} />

So, given an ingredient with these fields:

**let** ingredient = {

amount: 1,

measurement: 'cup',

name: 'sugar'

}

we get:

<Ingredient amount={1}

measurement="cup"

name="sugar" />

Now that we have components for ingredients and instructions, we can compose recipes using these components ([Example 5-17](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0517)).

*Example 5-17. Refactored Recipe component*

**import** IngredientsList from './IngredientsList'

**import** Instructions from './Instructions'

**const** Recipe = ({ name, ingredients, steps}) =>

<section id={name.toLowerCase().replace(/ /g, '-')}>

<h1>{name}</h1>

<IngredientsList list={ingredients} />

<Instructions title="Cooking Instructions"

steps={steps} />

</section>

**export** **default** Recipe

First we import the components that we are going to use, IngredientsList and Instructions. Now we can use them to create the Recipe component. Instead of a bunch of complicated code building out the entire recipe in one place, we have expressed our recipe more declaratively by composing smaller components. Not only is the code nice and simple, but it also reads well. This shows us that a recipe should display the name of the recipe, a list of ingredients, and some cooking instructions. We’ve abstracted away what it means to display ingredients and instructions into smaller, simple components.

In a modular approach with CommonJS, the Menu component would look pretty similar. The key difference is that it would live in its own file, import the modules that it needs to use, and export itself ([Example 5-18](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0518)).

*Example 5-18. Completed Menu component*

**import** Recipe from './Recipe'

**const** Menu = ({ recipes }) =>

<article>

<header>

<h1>Delicious Recipes</h1>

</header>

<div className="recipes">

{ recipes.map((recipe, i) =>

<Recipe key={i} {...recipe} />)

}

</div>

</article>

**export** **default** Menu

We still need to use ReactDOM to render the Menu component. We will still have a *index.js* file, but it will look much different ([Example 5-19](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0519)).

*Example 5-19. Completed index.js file*

**import** React from 'react'

**import** { render } from 'react-dom'

**import** Menu from './components/Menu'

**import** data from './data/recipes'

window.React = React

render(

<Menu recipes={data} />,

document.getElementById("react-container")

)

The first four statements import the necessary modules for our app to work. Instead of loading react and react-dom via the script tag, we import them so webpack can add them to our bundle. We also need the Menu component, and a sample data array which has been moved to a separate module. It still contains two recipes: Baked Salmon and Fish Tacos.

All of our imported variables are local to the *index.js* file. Setting window.React to React exposes the React library globally in the browser. This way all calls to React.createElement are assured to work.

When we render the Menu component, we pass the array of recipe data to this component as a property. This single ReactDOM.render call will mount and render our Menu component.

Now that we have pulled our code apart into separate modules and files, let’s create a static build process with webpack that will put everything back together into a single file.

**INSTALLING WEBPACK DEPENDENCIES**

In order to create a static build process with webpack, we’ll need to install a few things. Everything that we need can be installed with npm. First, we might as well install webpack globally so we can use the webpack command anywhere:

sudo npm install -g webpack

Webpack is also going to work with Babel to transpile our code from JSX and ES6 to JavaScript that runs in the browser. We are going to use a few loaders along with a few presets to accomplish this task:

npm install babel-core babel-loader babel-preset-env babel-preset-react

babel-preset-stage-0 --save-dev

Our application uses React and ReactDOM. We’ve been loading these dependencies with the script tag. Now we are going to let webpack add them to our bundle. We’ll need to install the dependencies for React and ReactDOM locally:

npm install react react-dom --save

This adds the necessary scripts for react and react-dom to the *./node\_modules* folder. Now we have everything needed to set up a static build process with webpack.

**WEBPACK CONFIGURATION**

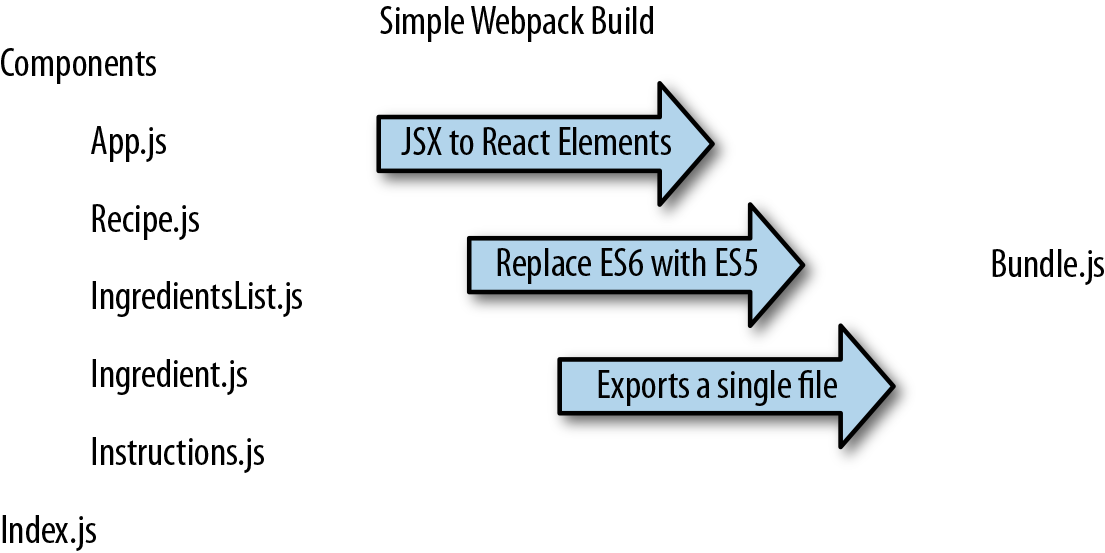
For this modular Recipes app to work, we are going to need to tell webpack how to bundle our source code into a single file. We can do this with configuration files, and the default webpack configuration file is always *webpack.config.js*.

The starting file for our Recipes app is *index.js*. It imports React, ReactDOM, and the *Menu.js* file. This is what we want to run in the browser first. Wherever webpack finds an import statement, it will find the associated module in the filesystem and include it in the bundle. *Index.js* imports *Menu.js*, *Menu.js* imports *Recipe.js*, *Recipe.js* imports *Instructions.js* and *IngredientsList.js*, and *IngredientsList.js* imports *Ingredient.js*. Webpack will follow this import tree and include all of these necessary modules in our bundle.

**ES6 IMPORT STATEMENTS**

We are using ES6 import statements, which are not presently supported by most browsers or by Node.js. The reason ES6 import statements work is because Babel will convert them into require('module/path'); statements in our final code. The require function is how CommonJS modules are typically loaded.

As webpack builds our bundle, we need to tell webpack to transpile JSX to pure React elements. We also need to convert any ES6 syntax to ES5 syntax. Our build process will initially have three steps ([Figure 5-4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#fig0504)).



*Figure 5-4. Recipe app build process*

The *webpack.config.js* file is just another module that exports a JavaScript literal object that describes the actions that webpack should take. This file ([Example 5-20](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0520)) should be saved to the root folder of the project, right next to the *index.js* file.

*Example 5-20. webpack.config.js*

module.exports = {

entry: "./src/index.js",

output: {

path: "dist/assets",

filename: "bundle.js"

},

module: {

rules: [

{

test: /\.js$/,

exclude: /(node\_modules)/,

loader: ['babel-loader'],

query: {

presets: ['env', 'stage-0', 'react']

}

}

]

}

}

First, we tell webpack that our client entry file is *./src/index.js*. It will automatically build the dependency tree based upon import statements starting in that file. Next, we specify that we want to output a bundled JavaScript file to *./dist/assets/bundle.js*. This is where webpack will place the final packaged JavaScript.

The next set of instructions for webpack consists of a list of loaders to run on specified modules. The rules field is an array because there are many types of loaders that you can incorporate with webpack. In this example, we are only incorporating babel.

Each loader is a JavaScript object. The test field is a regular expression that matches the file path of each module that the loader should operate on. In this case, we are running the babel-loader on all imported JavaScript files except those found in the *node\_modules* folder. When the babel-loader runs, it will use presets for ES2015 (ES6) and React to transpile any ES6 or JSX syntax into JavaScript that will run in most browsers.

Webpack is run statically. Typically bundles are created before the app is deployed to the server. Since you have installed webpack globally, you can run it from the command line:

$ **webpack**

Time: 1727ms

Asset Size Chunks Chunk Names

bundle.js 693 kB 0 [emitted] main

+ 169 hidden modules

Webpack will either succeed and create a bundle, or fail and show you an error. Most errors have to do with broken import references. When debugging webpack errors, look closely at the filenames and file paths used in import statements.

**LOADING THE BUNDLE**

We have a bundle, so now what? We exported the bundle to the *dist* folder. This folder contains the files that we want to run on the web server. The *dist* folder is where the *index.html* file ([Example 5-21](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0521)) should be placed. This file needs to include a target div element where the React Menu component will be mounted. It also requires a single script tag that will load our bundled JavaScript.

*Example 5-21. index.html*

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** charset="utf-8"**>**

**<title>**React Recipes App**</title>**

**</head>**

**<body>**

**<div** id="react-container"**></div>**

**<script** src="assets/bundle.js"**></script>**

**</body>**

**</html>**

This is the home page for your app. It will load everything it needs from one file, one HTTP request: *bundle.js*. You will need to deploy these files to your web server or build a web server application that will serve these files with something like Node.js or Ruby on Rails.

**SOURCE MAPPING**

Bundling our code into a single file can cause some setbacks when it comes time to debug the application in the browser. We can eliminate this problem by providing a *source map*. A source map is a file that maps a bundle to the original source files. With webpack, all we have to do is add a couple of lines to our *webpack.config.js* file ([Example 5-22](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0522)).

*Example 5-22. webpack.config.js with source mapping*

module.exports = {

entry: "./src/index.js",

output: {

path: "dist/assets",

filename: "bundle.js",

sourceMapFilename: 'bundle.map'

},

devtool: '#source-map',

module: {

rules: [

{

test: /\.js$/,

exclude: /(node\_modules)/,

loader: ['babel-loader'],

query: {

presets: ['env', 'stage-0', 'react']

}

}

]

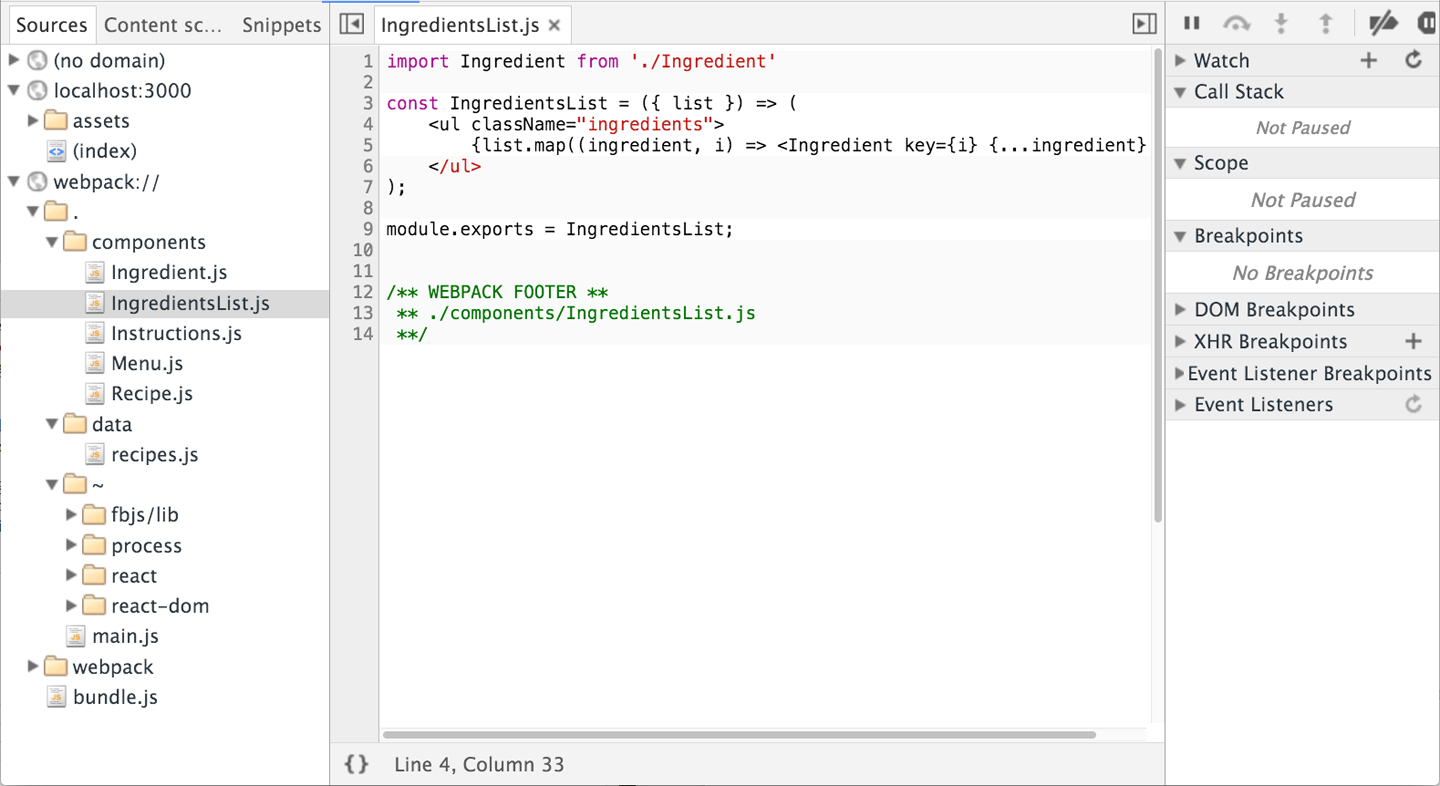
}

}

Setting the devtool property to '#source-map' tells webpack that you want to use source mapping. A sourceMapFilename is required. It is always a good idea to name your source map file after the target dependency. Webpack will associate the bundle with the source map during the export.

The next time you run webpack, you will see that two output files are generated and added to the *assets* folder: the original *bundle.js* and *bundle.map*.

The source map is going to let us debug using our original source files. In the Sources tab of your browser’s developer tools, you should find a folder named *webpack://*. Inside of this folder, you will see all of the source files in your bundle ([Figure 5-5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#fig0505)).



*Figure 5-5. Sources panel of Chrome Developer Tools*

You can debug from these files using the browser’s step-through debugger. Clicking on any line number adds a breakpoint. Refreshing the browser will pause JavaScript processing when any breakpoints are reached in your source file. You can inspect scoped variables in the Scope panel or add variables to Watch in the watch panel.

**OPTIMIZING THE BUNDLE**

The output bundle file is still simply a text file, so reducing the amount of text in this file will reduce the file size and cause it to load faster over HTTP. Some things that can be done to reduce the file size include removing all whitespace, reducing variable names to a single character, and removing any lines of code that the interpreter will never reach. Reducing the size of your JavaScript file with these tricks is referred to as *minifying* or *uglifying*your code.

Webpack has a built-in plugin that you can use to uglify the bundle. In order to use it, you will need to install webpack locally:

npm install webpack --save-dev

We can add extra steps to the build process using webpack plugins. In this example, we are going to add a step to our build process to uglify our output bundle, which will significantly reduce the file size ([Example 5-23](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#example0523)).

*Example 5-23. webpack.config.js with Uglify plugin*

**var** webpack = require("webpack");

module.exports = {

entry: "./src/index.js",

output: {

path: "dist/assets",

filename: "bundle.js",

sourceMapFilename: 'bundle.map'

},

devtool: '#source-map',

module: {

rules: [

{

test: /\.js$/,

exclude: /(node\_modules)/,

loader: ['babel-loader'],

query: {

presets: ['env', 'stage-0', 'react']

}

}

]

},

plugins: [

**new** webpack.optimize.UglifyJsPlugin({

sourceMap: **true**,

warnings: **false**,

mangle: **true**

})

]

}

To use the Uglify plugin, we need to require webpack, which is why we needed to install webpack locally.

UglifyJsPlugin is a function that gets instructions from its arguments. Once we uglify our code, it will become unrecognizable. We are going to need a source map, which is why sourceMap is set to true. Setting warnings to false will remove any console warnings from the exported bundle. Mangling our code means that we are going to reduce long variable names like recipes or ingredients to a single letter.

The next time you run webpack, you will see that the size of your bundled output file has been significantly reduced, and it’s no longer recognizable. Including a source map will still allow you to debug from your original source even though your bundle has been minified.

**BUNDLING CSS**

Another nice feature of webpack is that it can bundle CSS into the same file as the bundled JavaScript. This allows your users to download a single file that contains all of the CSS and JavaScript necessary for your app.

CSS can be included into the bundle with import statements. These statements tell webpack to bundle up associated CSS files with a JavaScript module:

**import** Recipe from './Recipe'

**import** '../../stylesheets/Menu.css'

**const** Menu = ({ recipes }) =>

<article>

<header>

<h1>Delicious Recipes</h1>

</header>

<div className="recipes">

{ recipes.map((recipe, i) =>

<Recipe key={i} {...recipe} />)

}

</div>

</article>

**export** **default** Menu

In order to implement CSS bundling in your webpack configuration, you will need to install some loaders:

npm install style-loader css-loader postcss-loader --save-dev

Finally, you have to incorporate this loader into your webpack configuration:

rules: [

{

test: /\.js$/,

exclude: /(node\_modules)/,

loader: ['babel-loader'],

query: {

presets: ['env', 'stage-0', 'react']

}

},

**{**

**test: /\.css$/,**

**use: ['style-loader','css-loader', {**

**loader: 'postcss-loader',**

**options: {**

**plugins: () => [require('autoprefixer')]**

**}}]**

**}**

]

Bundling up CSS files with webpack will cause your site to load faster by reducing the number of requests that your browser needs to make for resources.

**CREATE-REACT-APP**

As the Facebook team mentions in their blog, “the React ecosystem has commonly become associated with an overwhelming explosion of tools.”[1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#idm140391599443984) In response to this, the React team launched [create-react-app](http://bit.ly/2mtQwNC) a command-line tool that autogenerates a React project. create-react-app was inspired by the [Ember CLI project](https://ember-cli.com/), and it lets developers get started with React projects quickly without the manual configuration of webpack, Babel, ESLint, and associated tools.

To get started with create-react-app, install the package globally:

npm install -g create-react-app

Then, use the command and the name of the folder where you’d like the app to be created:

create-react-app my-react-project

This will create a React project in that directory with just three dependencies: React, ReactDOM, and react-scripts. react-scripts was also created by Facebook and is where the real magic happens. It installs Babel, ESLint, webpack, and more, so that you don’t have to configure them manually. Within the generated project folder, you’ll also find a *src* folder containing an *App.js* file. Here, you can edit the root component and import other component files.

From within the *my-react-project* folder, you can run npm start. If you prefer, you can also run yarn start.

You can run tests with npm test or yarn test. This runs all of the test files in the project in an interactive mode.

This will start your application on port 3000. You can also run the npm run build command. Using yarn, run yarn build.

This will create a production-ready bundle that has been transpiled and minified.

create-react-app is a great tool for beginners and experienced React developers alike. As the tool evolves, more functionality will likely be added, so you can keep an eye on the changes on GitHub.

[1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html#idm140391599443984-marker) Dan Abramov, [“Create Apps with No Configuration”](http://bit.ly/2ndUXzR), React Blog, July 22, 2016.

* [**Copy**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html)
* [**Add Highlight**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html)
* [**Add Note**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch05.html)

**Chapter 6. Props, State, and the Component Tree**

In the last chapter, we talked about how to create components. We primarily focused on how to build a user interface by composing React components. This chapter is filled with techniques that you can use to better manage data and reduce time spent debugging applications.

Data handling within component trees is one of the key advantages of working with React. There are techniques that you can use when working with data in React components that will make your life much easier in the long run. Our applications will be easier to reason about and scale if we can manage data from a single location and construct the UI based on that data.

**Property Validation**

JavaScript is a loosely typed language, which means that the data type of a variable’s value can change. For example, you can initially set a JavaScript variable as a string, then change its value to an array later, and JavaScript will not complain. Managing our variable types inefficiently can lead to a lot of time spent debugging applications.

React components provide a way to specify and validate property types. Using this feature will greatly reduce the amount of time spent debugging applications. Supplying incorrect property types triggers warnings that can help us find bugs that may have otherwise slipped through the cracks.

React has built-in [automatic property validation](http://bit.ly/2okjSzJ) for the variable types, as shown in [Table 6-1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#table0601).

| **Type** | **Validator** |
| --- | --- |
| Arrays | React.PropTypes.array |
| Boolean | React.PropTypes.bool |
| Functions | React.PropTypes.func |
| Numbers | React.PropTypes.number |
| Objects | React.PropTypes.object |
| Strings | React.PropTypes.string |
| *Table 6-1. React property validation* | |

In this section, we will create a Summary component for our recipes. The Summary component will display the title of the recipe along with counts for both ingredients and steps (see [Figure 6-1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#fig0601)).



*Figure 6-1. Summary component output for Baked Salmon*

In order to display this data, we must supply the Summary component with three properties: a title, an array of ingredients, and an array of steps. We want to validate these properties to make sure the first is a string and the others are arrays, and supply defaults for when they are unavailable. How to implement property validation depends upon how components are created. Stateless functional components and ES6 classes have different ways of implementing property validation.

First, let’s look at why we should use property validation and how to implement it in components created with React.createClass.

**Validating Props with createClass**

We need to understand why it is important to validate component property types. Consider the following implementation for the Summary component:

**const** Summary = createClass({

displayName: "Summary",

render() {

**const** {ingredients, steps, title} = **this**.props

**return** (

<div className="summary">

<h1>{title}</h1>

<p>

<span>{ingredients.length} Ingredients</span> |

<span>{steps.length} Steps</span>

</p>

</div>

)

}

})

The Summary component destructures ingredients, steps, and title from the properties object and then constructs a UI to display that data. Since we expect both ingredients and steps to be arrays, we use Array.length to count the array’s items.

What if we rendered this Summary component accidentally using strings?

render(

<Summary title="Peanut Butter and Jelly"

ingredients="peanut butter, jelly, bread"

steps="spread peanut butter and jelly between bread" />,

document.getElementById('react-container')

)

JavaScript will not complain, but finding the length will count the number of characters in each string ([Figure 6-2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#fig0602)).



*Figure 6-2. Summary component output for Peanut Butter and Jelly*

The output of this code is odd. No matter how fancy your peanut butter and jelly might be, it’s doubtful that you are going to have 27 ingredients and 44 steps. Instead of seeing the correct number of steps and ingredients, we are seeing the length in characters of each string. A bug like this is easy to miss. If we validated the property types when we created the Summary component, React could catch this bug for us:

**const** Summary = createClass({

displayName: "Summary",

**propTypes: {**

**ingredients: PropTypes.array,**

**steps: PropTypes.array,**

**title: PropTypes.string**

**},**

render() {

**const** {ingredients, steps, title} = **this**.props

**return** (

<div className="summary">

<h1>{title}</h1>

<p>

<span>{ingredients.length} Ingredients | </span>

<span>{steps.length} Steps</span>

</p>

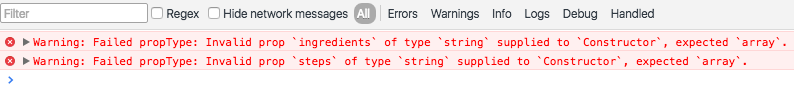
</div>

)

}

})

Using React’s built-in property type validation, we can make sure that both ingredients and steps are arrays. Additionally, we can make sure that the title value is a string. Now when we pass incorrect property types, we will see an error ([Figure 6-3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#fig0603)).



*Figure 6-3. Property type validation warning*

What would happen if we rendered the Summary component without sending it any properties?

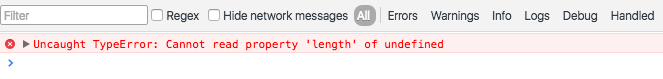
render(

<Summary />,

document.getElementById('react-container')

)

Rendering the Summary component without any properties causes a JavaScript error that takes down the web app ([Figure 6-4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#fig0604)).



*Figure 6-4. Error generated from missing array*

This error occurs because the type of the ingredients property is undefined, and undefined is not an object that has a length property like an array or a string. React has a way to specify required properties. When those properties are not supplied, React will trigger a warning in the console:

**const** Summary = createClass({

displayName: "Summary",

propTypes: {

**ingredients: PropTypes.array.isRequired,**

**steps: PropTypes.array.isRequired,**

title: PropTypes.string

},

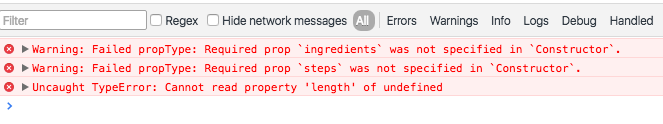
render() {

...

}

})

Now when we render the Summary component without any properties, React directs our attention to the problem with a console warning just before the error occurs. This makes it easier to figure out what went wrong ([Figure 6-5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#fig0605)).



*Figure 6-5. React warnings for missing properties*

The Summary component expects an array for ingredients and an array for steps, but it only uses the length property of each array. This component is designed to display counts (numbers) for each of those values. It may make more sense to refactor our code to expect numbers instead, since the component doesn’t actually need arrays:

**import** { createClass, PropTypes } from 'react'

**export** **const** Summary = createClass({

displayName: "Summary",

propTypes: {

**ingredients: PropTypes.number.isRequired,**

**steps: PropTypes.number.isRequired,**

title: PropTypes.string

},

render() {

**const** {ingredients, steps, title} = **this**.props

**return** (

<div className="summary">

<h1>{title}</h1>

<p>

**<span>{ingredients} Ingredients</span> |**

**<span>{steps} Steps</span>**

</p>

</div>

)

}

})

Using numbers for this component is a more flexible approach. Now the Summary component simply displays the UI; it sends the burden of actually counting ingredients or steps further up the component tree to a parent or ancestor.

**Default Props**

Another way to improve the quality of components is to assign default values for properties.[1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#idm140391598582032) The validation behavior is similar to what you might expect: the default values you establish will be used if other values are not provided.

Let’s say we want the Summary component to work even when the properties are not supplied:

**import** { render } from 'react-dom'

render(<Summary />, document.getElementById('react-container'))

With createClass, we can add a method called getDefaultProps that returns default values for properties that are not assigned:

**const** Summary = createClass({

displayName: "Summary",

propTypes: {

ingredients: PropTypes.number,

steps: PropTypes.number,

title: PropTypes.string

},

**getDefaultProps() {**

**return {**

**ingredients: 0,**

**steps: 0,**

**title: "[recipe]"**

**}**

**},**

render() {

**const** {ingredients, steps, title} = **this**.props

**return** (

<div className="summary">

<h1>{title}</h1>

<p>

<span>{ingredients} Ingredients | </span>

<span>{steps} Steps</span>

</p>

</div>

)

}

}

Now when we try to render this component without properties, we will see some default data instead, as in [Figure 6-6](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#fig0606).



*Figure 6-6. Summary component output with default properties*

Using default properties can extend the flexibility of your component and prevent errors from occurring when your users do not explicitly require every property.

**Custom Property Validation**

React’s built-in validators are great for making sure that your variables are required and typed correctly. But there are instances that require more robust validation. For example, you may want to make sure that a number is within a specific range or that a value contains a specific string. React provides a way to build your own custom validation for such cases.

Custom validation in React is implemented with a function. This function should either return an error when a specific validation requirement is not met or null when the property is valid.

With basic property type validation, we can only validate a property based on one condition. The good news is that the custom validator will allow us to test the property in many different ways. In this custom function, we’ll first check that the property’s value is a string. Then we’ll limit its length to 20 characters ([Example 6-2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#example0602)).

*Example 6-2. Custom prop validation*

propTypes: {

ingredients: PropTypes.number,

steps: PropTypes.number,

**title: (props, propName) =>**

**(typeof props[propName] !== 'string') ?**

**new Error("A title must be a string") :**

**(props[propName].length > 20) ?**

**new Error(`title is over 20 characters`) :**

**null**

}

All property type validators are functions. To implement our custom validator, we will set the value of the titleproperty, under the propTypes object, to a callback function. When rendering the component, React will inject the props object and the name of the current property into the function as arguments. We can use those arguments to check the specific value for a specific property.

In this case, we first check the title to make sure it is a string. If the title is not a string, the validator returns a new error with the message: “A title must be a string.” If the title is a string, then we check its value to make sure it is not longer than 20 characters. If the title is under 20 characters, the validator function returns null. If the title is over 20 characters, then the validator function returns an error. React will capture the returned error and display it in the console as a warning.

Custom validators allow you to implement specific validation criteria. A custom validator can perform multiple validations and only return errors when specific criteria are not met. Custom validators are a great way to prevent errors when using and reusing your components.

**ES6 Classes and Stateless Functional Components**

In the previous sections, we discovered that property validation and default property values can be added to our component classes using React.createClass. This type checking also works for ES6 classes and stateless functional components, but the syntax is slightly different.

When working with ES6 classes, propTypes and defaultProps declarations are defined on the class instance, outside of the class body. Once a class is defined, we can set the propTypes and defaultProps object literals ([Example 6-3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#example0603)).

*Example 6-3. ES6 class*

**class** Summary **extends** React.Component {

render() {

**const** {ingredients, steps, title} = **this**.props

**return** (

<div className="summary">

<h1>{title}</h1>

<p>

<span>{ingredients} Ingredients | </span>

<span>{steps} Steps</span>

</p>

</div>

)

}

}

Summary.propTypes = {

ingredients: PropTypes.number,

steps: PropTypes.number,

title: (props, propName) =>

(**typeof** props[propName] !== 'string') ?

**new** Error("A title must be a string") :

(props[propName].length > 20) ?

**new** Error(`title is over 20 characters`) :

**null**

}

Summary.defaultProps = {

ingredients: 0,

steps: 0,

title: "[recipe]"

}

The propTypes and defaultProps object literals can also be added to stateless functional components ([Example 6-4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#example0604)).

*Example 6-4. Stateless functional component*

**const** Summary = ({ ingredients, steps, title }) => {

**return** <div>

<h1>{title}</h1>

<p>{ingredients} Ingredients | {steps} Steps</p>

</div>

}

Summary.propTypes = {

ingredients: React.PropTypes.number.isRequired,

steps: React.PropTypes.number.isRequired

}

Summary.defaultProps = {

ingredients: 1,

steps: 1

}

With a stateless functional component, you also have the option of setting default properties directly in the function arguments. We can set default values for ingredients, steps, and title when we destructure the properties object in the function arguments as follows:

**const** Summary = **({ ingredients=0, steps=0, title='[recipe]' })** => {

**return** <div>

<h1>{title}</h1>

<p>{ingredients} Ingredients | {steps} Steps</p>

</div>

}

**CLASS STATIC PROPERTIES**

In the previous section, we looked at how defaultProps and propTypes are defined outside of the class. An alternative to this is emerging in one of the latest proposals to the ECMAScript spec: *Class Fields & Static Properties*.

Class static properties allow us to encapsulate propTypes and defaultProps inside of the class declaration. Property initializers also provide encapsulation and cleaner syntax:

**class** Summary **extends** React.Component {

**static propTypes = {**

**ingredients: PropTypes.number,**

**steps: PropTypes.number,**

**title: (props, propName) =>**

**(typeof props[propName] !== 'string') ?**

**new Error("A title must be a string") :**

**(props[propName].length > 20) ?**

**new Error(`title is over 20 characters`) :**

**null**

**}**

**static defaultProps = {**

**ingredients: 0,**

**steps: 0,**

**title: "[recipe]"**

**}**

render() {

**const** {ingredients, steps, title} = **this**.props

**return** (

<div className="summary">

<h1>{title}</h1>

<p>

<span>{ingredients} Ingredients | </span>

<span>{steps} Steps</span>

</p>

</div>

)

}

}

Property validation, custom property validation, and the ability to set default property values should be implemented in every component. This makes the component easier to reuse because any problems with component properties will show up as console warnings.

**Refs**

References, or *refs*, are a feature that allow React components to interact with child elements. The most common use case for refs is to interact with UI elements that collect input from the user. Consider an HTML formelement. These elements are initially rendered, but the users can interact with them. When they do, the component should respond appropriately.

For the rest of this chapter, we are going to be working with an application that allows users to save and manage specific hexadecimal color values. This application, the color organizer, allows users to add colors to a list. Once a color is in the list, it can be rated or removed by the user.

We will need a form to collect information about new colors from the user. The user can supply the color’s title and hex value in the corresponding fields. The AddColorForm component renders the HTML with a text input and a color input for collecting hex values from the color wheel ([Example 6-5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#example0606)).

*Example 6-5. AddColorForm*

**import** { Component } from 'react'

**class** AddColorForm **extends** Component {

render() {

**return** (

<form onSubmit={e=>e.preventDefault()}>

<input type="text"

placeholder="color title..." required/>

<input type="color" required/>

<button>ADD</button>

</form>

)

}

}

The AddColorForm component renders an HTML form that contains three elements: a text input for the title, a color input for the color’s hex value, and a button to submit the form. When the form is submitted, a handler function is invoked where the default form event is ignored. This prevents the form from trying to send a GET request once submitted.

Once we have the form rendered, we need to provide a way to interact with it. Specifically, when the form is first submitted, we need to collect the new color information and reset the form’s fields so that the user can add more colors. Using refs, we can refer to the title and color elements and interact with them ([Example 6-6](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#example0607)).

*Example 6-6. AddColorForm with submit method*

**import** { Component } from 'react'

**class** AddColorForm **extends** Component {

**constructor(props) {**

**super(props)**

**this.submit = this.submit.bind(this)**

**}**

**submit(e) {**

**const { \_title, \_color } = this.refs**

**e.preventDefault();**

**alert(`New Color: ${\_title.value} ${\_color.value}`)**

**\_title.value = '';**

**\_color.value = '#000000';**

**\_title.focus();**

**}**

render() {

**return** (

<form onSubmit={**this.submit**}>

<input **ref="\_title"**

type="text"

placeholder="color title..." required/>

<input **ref="\_color"**

type="color" required/>

<button>ADD</button>

</form>

)

}

}

We needed to add a constructor to this ES6 component class because we moved submit to its own function. With ES6 component classes, we must bind the scope of the component to any methods that need to access that scope with this.

Next, in the render method, we’ve set the form’s onSubmit handler by pointing it to the component’s submitmethod. We’ve also added ref fields to the components that we want to reference. A ref is an identifier that React uses to reference DOM elements. Creating \_title and \_color ref attributes for each input means that we can access those elements with this.refs\_title or this.refs\_color.

When the user adds a new title, selects a new color, and submits the form, the component’s submit method will be invoked to handle the event. After we prevent the form’s default submit behavior, we send the user an alert that echoes back the data collected via refs. After the user dismisses the alert, refs are used again to reset the form values and focus on the title field.

**BINDING THE ‘THIS’ SCOPE**

When using React.createClass to create your components, there is no need to bind the thisscope to your component methods. React.createClass automatically binds the this scope for you.

**Inverse Data Flow**

It’s nice to have a form that echoes back input data in an alert, but there is really no way to make money with such a product. What we need to do is collect data from the user and send it somewhere else to be handled. This means that any data collected may eventually make its way back to the server, which we will cover in [Chapter 12](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch12.html#react_and_the_server). First, we need to collect the data from the form component and pass it on.

A common solution for collecting data from a React component is *inverse data flow*.[2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#idm140391597166672) It is similar to, and sometimes described as, *two-way data binding*. It involves sending a callback function to the component as a property that the component can use to pass data back as arguments. It’s called inverse data flow because we send the component a function as a property, and the component sends data back as function arguments.

Let’s say we want to use the color form, but when a user submits a new color we want to collect that information and log it to the console.

We can create a function called logColor that receives the title and color as arguments. The values of those arguments can be logged to the console. When we use the AddColorForm, we simply add a function property for onNewColor and set it to our logColor function. When the user adds a new color, logColor is invoked, and we’ve sent a function as a property:

**const** logColor = (title, color) =>

console.log(`New Color: ${title} | ${value}`)

<AddColorForm onNewColor={logColor} />

To ensure that data is flowing properly, we will invoke onNewColor from props with the appropriate data:

submit() {

**const** {\_title, \_color} = **this**.refs

**this.props.onNewColor(\_title.value, \_color.value)**

\_title.value = ''

\_color.value = '#000000'

\_title.focus()

}

In our component, this means that we’ll replace the alert call with a call to this.props.onNewColor and pass the new title and color values that we have obtained through refs.

The role of the AddColorForm component is to collect data and pass it on. It is not concerned with what happens to that data. We can now use this form to collect color data from users and pass it on to some other component or method to handle the collected data:

<AddColorForm onNewColor={(title, color) => {

console.log(`TODO: add new ${title} and ${color} to the list`)

console.log(`TODO: render UI with new Color`)

}} />

When we are ready, we can collect the information from this component and add the new color to our list of colors.

**OPTIONAL FUNCTION PROPERTIES**

In order to make two-way data binding optional, you must first check to see if the function property exists before trying to invoke it. In the last example, not supplying an onNewColor function property would lead to a JavaScript error because the component will try to invoke an undefined value.

This can be avoided by first checking for the existence of the function property:

**if** (**this**.props.onNewColor) {

**this**.props.onNewColor(\_title.value, \_color.value)

}

A better solution is to define the function property in the component’s propTypes and defaultProps:

AddColorForm.propTypes = {

onNewColor: PropTypes.func

}

**AddColorForm.defaultProps = {**

**onNewColor: f=>f**

**}**

Now when the property supplied is some type other than function, React will complain. If the onNewColor property is not supplied, it will default to this dummy function, f=>f. This is simply a placeholder function that returns the first argument sent to it. Although this placeholder function doesn’t do anything, it can be invoked by JavaScript without causing errors.

**Refs in Stateless Functional Components**

Refs can also be used in stateless functional components. These components do not have this, so it’s not possible to use this.refs. Instead of using string attributes, we will set the refs using a function. The function will pass us the input instance as an argument. We can capture that instance and save it to a local variable.

Let’s refactor AddColorForm as a stateless functional component:

**const** AddColorForm = ({**onNewColor=f=>f**}) => {

**let** \_title, \_color

**const** submit = e => {

e.preventDefault()

**onNewColor(\_title.value, \_color.value)**

\_title.value = ''

\_color.value = '#000000'

\_title.focus()

}

**return** (

<form onSubmit={submit}>

<input **ref={input => \_title = input}**

type="text"

placeholder="color title..." required/>

<input **ref={input => \_color = input}**

type="color" required/>

<button>ADD</button>

</form>

)

}

In this stateless functional component, refs are set with a callback function instead of a string. The callback function passes the element’s instance as an argument. This instance can be captured and saved into a local variable like \_title or \_color. Once we’ve saved the refs to local variables, they are easily accessed when the form is submitted.

**React State Management**

Thus far we’ve only used properties to handle data in React components. Properties are immutable. Once rendered, a component’s properties do not change. In order for our UI to change, we would need some other mechanism that can rerender the component tree with new properties. React state is a built-in option for managing data that will change within a component. When application state changes, the UI is rerendered to reflect those changes.

Users interact with applications. They navigate, search, filter, select, add, update, and delete. When a user interacts with an application, the state of that application changes, and those changes are reflected back to the user in the UI. Screens and menus appear and disappear. Visible content changes. Indicators light up or are turned off. In React, the UI is a reflection of application state.

State can be expressed in React components with a single JavaScript object. When the state of a component changes, the component renders a new UI that reflects those changes. What can be more functional than that? Given some data, a React component will represent that data as the UI. Given a change to that data, React will update the UI as efficiently as possible to reflect that change.

Let’s take a look at how we can incorporate state within our React components.

**Introducing Component State**

State represents data that we may wish to change within a component. To demonstrate this, we will take a look at a StarRating component ([Figure 6-7](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#fig0607)).



*Figure 6-7. The StarRating component*

The StarRating component requires two critical pieces of data: the total number of stars to display, and the rating, or the number of stars to highlight.

We’ll need a clickable Star component that has a selected property. A stateless functional component can be used for each star:

**const** Star = ({ selected=**false**, onClick=f=>f }) =>

<div className={(selected) ? "star selected" : "star"}

onClick={onClick}>

</div>

Star.propTypes = {

selected: PropTypes.bool,

onClick: PropTypes.func

}

Every Star element will consist of a div that includes the class 'star'. If the star is selected, it will additionally add the class 'selected'. This component also has an optional onClick property. When a user clicks on any star div, the onClick property will be invoked. This will tell the parent component, the StarRating, that a Star has been clicked.

The Star is a stateless functional component. It says it right in the name: you cannot use state in a stateless functional component. Stateless functional components are meant to be the children of more complex, stateful components. It’s a good idea to try to keep as many of your components as possible stateless.

**THE STAR IS IN THE CSS**

Our StarRating component uses CSS to construct and display a star. Specifically, using a *clip path*, we can clip the area of our div to look like a star. The clip path is collection of points that make up a polygon:

**.star** {

**cursor**: pointer;

**height**: 25px;

**width**: 25px;

**margin**: 2px;

**float**: left;

**background-color**: grey;

**clip-path**: polygon(

50% 0%,

63% 38%,

100% 38%,

69% 59%,

82% 100%,

50% 75%,

18% 100%,

31% 59%,

0% 38%,

37% 38%

);

}

**.star.selected** {

**background-color**: red;

}

A regular star has a background color of grey, but a selected star will have a background color of red.

Now that we have a Star, we can use it to create a StarRating. StarRating will obtain the total number of stars to display from the component’s properties. The rating, the value that the user can change, will be stored in the state.

First, let’s look at how to incorporate state into a component defined with createClass:

**const** StarRating = createClass({

displayName: 'StarRating',

propTypes: {

totalStars: PropTypes.number

},

getDefaultProps() {

**return** {

totalStars: 5

}

},

**getInitialState() {**

**return {**

**starsSelected: 0**

**}**

**},**

**change(starsSelected) {**

**this.setState({starsSelected})**

**},**

render() {

**const** {totalStars} = **this**.props

**const {starsSelected} = this.state**

**return** (

<div className="star-rating">

{[...Array(totalStars)].map((n, i) =>

<Star key={i}

**selected={i<starsSelected}**

**onClick={() => this.change(i+1)}**

/>

)}

<p>**{starsSelected}** **of** {totalStars} stars</p>

</div>

)

}

})

When using createClass, state can be initialized by adding getInitialState to the component configuration and returning a JavaScript object that initially sets the state variable, starsSelected to 0.

When the component renders, totalStars is obtained from the component’s properties and used to render a specific number of Star elements. Specifically, the spread operator is used with the Array constructor to initialize a new array of a specific length that is mapped to Star elements.

The state variable starsSelected is destructured from this.state when the component renders. It is used to display the rating as text in a paragraph element. It is also used to calculate the number of selected stars to display. Each Star element obtains its selected property by comparing its index to the number of stars that are selected. If three stars are selected, the first three Star elements will set their selected property to true and any remaining stars will have a selected property of false.

Finally, when a user clicks a single star, the index of that specific Star element is incremented and sent to the change function. This value is incremented because it is assumed that the first star will have a rating of 1 even though it has an index of 0.

Initializing state in an ES6 component class is slightly different than using createClass. In these classes, state can be initialized in the constructor:

**class** StarRating **extends** Component {

constructor(props) {

**super**(props)

**this.state = {**

**starsSelected: 0**

**}**

**this.change = this.change.bind(this)**

}

**change(starsSelected) {**

**this.setState({starsSelected})**

**}**

render() {

**const** {totalStars} = **this**.props

**const** {starsSelected} = **this**.state

**return** (

<div className="star-rating">

{[...Array(totalStars)].map((n, i) =>

<Star key={i}

selected={i<starsSelected}

**onClick={() => this.change(i+1)}**

/>

)}

<p>{starsSelected} **of** {totalStars} stars</p>

</div>

)

}

}

StarRating.propTypes = {

totalStars: PropTypes.number

}

StarRating.defaultProps = {

totalStars: 5

}

When an ES6 component is mounted, its constructor is invoked with the properties injected as the first argument. Those properties are, in turn, sent to the superclass by invoking super. In this case, the superclass is React.Component. Invoking super initializes the component instance, and React.Component decorates that instance with functionality that includes state management. After invoking super , we can initialize our component’s state variables.

Once the state is initialized, it operates as it does in createClass components. State can only be changed by calling this.setState, which updates specific parts of the state object. After every setState call, the render function is called, updating the state with the new UI.

**Initializing State from Properties**

We can initialize our state values using incoming properties. There are only a few necessary cases for this pattern. The most common case for this is when we create a reusable component that we would like to use across applications in different component trees.

When using createClass, a good way to initialize state variables based on incoming properties is to add a method called componentWillMount. This method is invoked once when the component mounts, and you can call this.setState() from this method. It also has access to this.props, so you can use values from this.props to help you initialize state:

**const** StarRating = createClass({

displayName: 'StarRating',

propTypes: {

totalStars: PropTypes.number

},

getDefaultProps() {

**return** {

totalStars: 5

}

},

getInitialState() {

**return** {

starsSelected: 0

}

},

**componentWillMount() {**

**const { starsSelected } = this.props**

**if (starsSelected) {**

**this.setState({starsSelected})**

**}**

**},**

change(starsSelected) {

**this**.setState({starsSelected})

},

render() {

**const** {totalStars} = **this**.props

**const** {starsSelected} = **this**.state

**return** (

<div className="star-rating">

{[...Array(totalStars)].map((n, i) =>

<Star key={i}

selected={i<starsSelected}

onClick={() => **this**.change(i+1)}

/>

)}

<p>{starsSelected} **of** {totalStars} stars</p>

</div>

)

}

})

render(

<StarRating totalStars={7} starsSelected={3} />,

document.getElementById('react-container')

)

componentWillMount is a part of the component lifecycle. It can be used to help you initialize state based on property values in components created with createClass or ES6 class components. We will dive deeper into the component lifecycle in the next chapter.

There is an easier way to initialize state within an ES6 class component. The constructor receives properties as an argument, so you can simply use the props argument passed to the constructor:

constructor(**props**) {

**super**(props)

**this.state = {**

**starsSelected: props.starsSelected || 0**

**}**

**this**.change = **this**.change.bind(**this**)

}

For the most part, you’ll want to avoid setting state variables from properties. Only use these patterns when they are absolutely required. You should find this goal easy to accomplish because when working with React components, you want to limit the number of components that have state.[3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#idm140391620621312)

**UPDATING COMPONENT PROPERTIES**

When initializing state variables from component properties, you may need to reinitialize component state when a parent component changes those properties. The componentWillRecievePropslifecycle method can be used to solve this issue. [Chapter 7](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#enhancing_components) goes into greater detail on this issue and the available methods of the component lifecycle.

**State Within the Component Tree**

All of your React components can have their own state, but should they? The joy of using React does not come from chasing down state variables all over your application. The joy of using React comes from building scalable applications that are easy to understand. The most important thing that you can do to make your application easy to understand is limit the number of components that use state as much as possible.

In many React applications, it is possible to group all state data in the root component. State data can be passed down the component tree via properties, and data can be passed back up the tree to the root via two-way function binding. The result is that all of the state for your entire application exists in one place. This is often referred to as having a “single source of truth.”[4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#idm140391620747152)

Next, we will look at how to architect presentation layers where all of the state is stored in one place, the root component.

**Color Organizer App Overview**

The color organizer allows users to add, name, rate, and remove colors in their customized lists. The entire state of the color organizer can be represented with a single array:

{

colors: [

{

"id": "0175d1f0-a8c6-41bf-8d02-df5734d829a4",

"title": "ocean at dusk",

"color": "#00c4e2",

"rating": 5

},

{

"id": "83c7ba2f-7392-4d7d-9e23-35adbe186046",

"title": "lawn",

"color": "#26ac56",

"rating": 3

},

{

"id": "a11e3995-b0bd-4d58-8c48-5e49ae7f7f23",

"title": "bright red",

"color": "#ff0000",

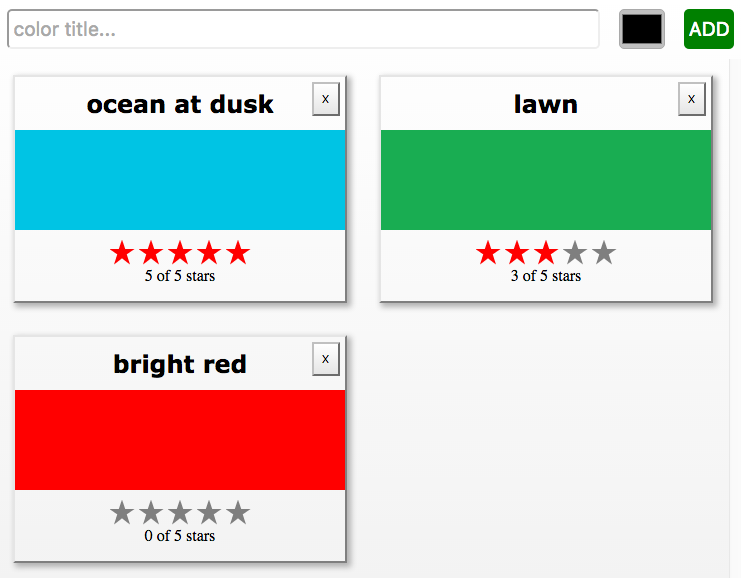
"rating": 0

}

]

}

The array tells us that we need to display three colors: ocean at dusk, lawn, and bright red ([Figure 6-8](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#fig0608)). It gives us the colors’ hex values and the current rating for each color in the display. It also provides a way to uniquely identify each color.



*Figure 6-8. Color organizer with three colors in state*

This state data will drive our application. It will be used to construct the UI every time this object changes. When users add or remove colors, they will be added to or removed from this array. When users rate colors, their ratings will change in the array.

**Passing Properties Down the Component Tree**

Earlier in this chapter, we created a StarRating component that saved the rating in the state. In the color organizer, the rating is stored in each color object. It makes more sense to treat the StarRating as a *presentational component*[5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#idm140391620331248) and declare it as a stateless functional component. Presentational components are only concerned with how things look in the application. They only render DOM elements or other presentational components. All data is sent to these components via properties and passed out of these components via callback functions.

In order to make the StarRating component purely presentational, we need to remove state. Presentational components only use props. Since we are removing state from this component, when a user changes the rating, that data will be passed out of this component via a callback function:

**const** StarRating = ({**starsSelected=0**, totalStars=5, **onRate=f=>f**}) =>

<div className="star-rating">

{[...Array(totalStars)].map((n, i) =>

<Star key={i}

**selected={i<starsSelected}**

onClick={() => **onRate(i+1)**}/>

)}

<p>**{starsSelected}** **of** {totalStars} stars</p>

</div>

First, starsSelected is no longer a state variable; it is a property. Second, an onRate callback property has been added to this component. Instead of calling setState when the user changes the rating, this component now invokes onRate and sends the rating as an argument.

**STATE IN REUSABLE COMPONENTS**

You may need to create stateful UI components for distribution and reuse across many different applications. It is not absolutely required that you remove every last state variable from components that are only used for presentation. It is a good rule to follow, but sometimes it may make sense to keep state in a presentation component.

Restricting state to a single location, the root component, means that all of the data must be passed down to child components as properties ([Figure 6-9](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#fig0609)).

In the color organizer, state consists of an array of colors that is declared in the App component. Those colors are passed down to the ColorList component as a property:

**class** App **extends** Component {

constructor(props) {

**super**(props)

**this.state = {**

**colors: []**

**}**

}

render() {

**const { colors } = this.state**

**return** (

<div className="app">

<AddColorForm />

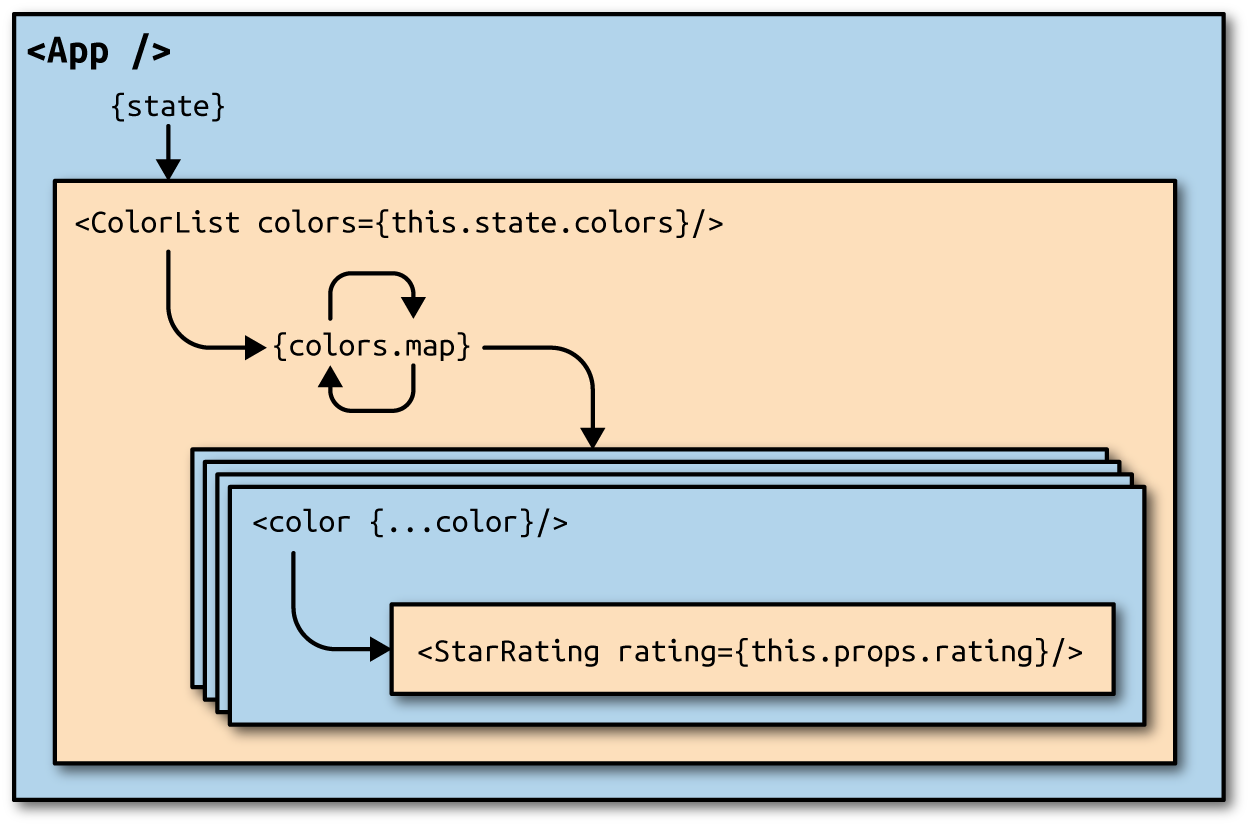
**<ColorList colors={colors} />**

</div>

)

}

}



*Figure 6-9. State is passed from the App component to child components as properties*

Initially the colors array is empty, so the ColorList component will display a message instead of each color. When there are colors in the array, data for each individual color is passed to the Color component as properties:

**const** ColorList = (**{ colors=[] }**) =>

<div className="color-list">

{(colors.length === 0) ?

<p>No Colors Listed. (Add a Color)</p> :

**colors.map(color =>**

**<Color key={color.id} {...color} />**

**)**

}

</div>

Now the Color component can display the color’s title and hex value and pass the color’s rating down to the StarRating component as a property:

**const** Color = (**{ title, color, rating=0 }**) =>

<section className="color">

<h1>**{title}**</h1>

<div className="color"

style=**{{ backgroundColor: color }}**>

</div>

<div>

<StarRating **starsSelected={rating}** />

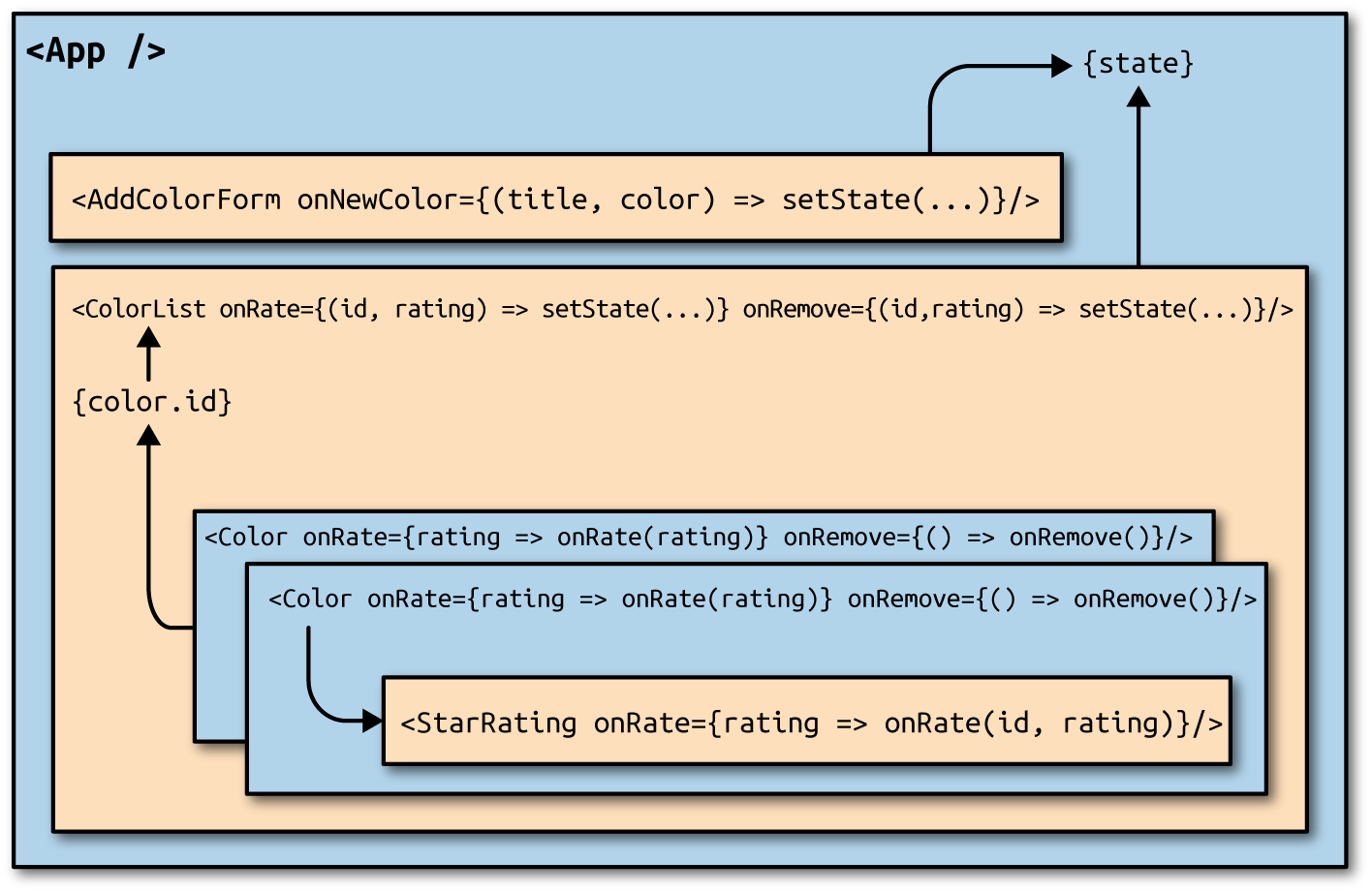
</div>

</section>

The number of starsSelected in the star rating comes from each color’s rating. All of the state data for every color has been passed down the tree to child components as properties. When there is a change to the data in the root component, React will change the UI as efficiently as possible to reflect the new state.

**Passing Data Back Up the Component Tree**

State in the color organizer can only be updated by calling setState from the App component. If users initiate any changes from the UI, their input will need to be passed back up the component tree to the App component in order to update the state ([Figure 6-10](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#fig0610)). This can be accomplished through the use of callback function properties.



*Figure 6-10. Passing data back up to the root component when there are UI events*

In order to add new colors, we need a way to uniquely identify each color. This identifier will be used to locate colors within the state array. We can use the uuid library to create absolutely unique IDs:

npm install uuid --save

All new colors will be added to the color organizer from the AddColorForm component that we constructed in [“Refs”](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#refs_ch6). That component has an optional callback function property called onNewColor. When the user adds a new color and submits the form, the onNewColor callback function is invoked with the new title and color hex value obtained from the user:

**import** { Component } from 'react'

**import { v4 } from 'uuid'**

**import** AddColorForm from './AddColorForm'

**import** ColorList from './ColorList'

**export** **class** App **extends** Component {

constructor(props) {

**super**(props)

**this**.state = {

colors: []

}

**this.addColor = this.addColor.bind(this)**

}

**addColor(title, color) {**

**const colors = [**

**...this.state.colors,**

**{**

**id: v4(),**

**title,**

**color,**

**rating: 0**

**}**

**]**

**this.setState({colors})**

**}**

render() {

**const { addColor } = this**

**const** { colors } = **this**.state

**return** (

<div className="app">

**<AddColorForm onNewColor={addColor} />**

<ColorList colors={colors} />

</div>

)

}

}

All new colors can be added from the addColor method in the App component. This function is bound to the component in the constructor, which means that it has access to this.state and this.setState.

New colors are added by concatenating the current colors array with a new color object. The ID for the new color object is set using uuid’s v4 function. This creates a unique identifier for each color. The title and color are passed to the addColor method from the AddColorForm component. Finally, the initial value for each color’s rating will be 0.

When the user adds a color with the AddColorForm component, the addColor method updates the state with a new list of colors. Once the state has been updated, the App component rerenders the component tree with the new list of colors. The render method is invoked after every setState call. The new data is passed down the tree as properties and is used to construct the UI.

If the user wishes to rate or remove a color, we need to collect information about that color. Each color will have a remove button: if the user clicks the remove button, we’ll know they wish to remove that color. Also, if the user changes the color’s rating with the StarRating component, we want to change the rating of that color:

**const** Color = ({title,color,rating=0,**onRemove=f=>f,onRate=f=>f**}) =>

<section className="color">

<h1>{title}</h1>

<button **onClick={onRemove}**>X</button>

<div className="color"

style={{ backgroundColor: color }}>

</div>

<div>

<StarRating starsSelected={rating} **onRate={onRate}** />

</div>

</section>

The information that will change in this app is stored in the list of colors. Therefore, onRemove and onRatecallback properties will have to be added to each color to pass those events back up the tree. The Colorcomponent will also have onRate and onRemove callback function properties. When colors are rated or removed, the ColorList component will need to notify its parent, the App component, that the color should be rated or removed:

**const** ColorList = ({ colors=[], **onRate=f=>f, onRemove=f=>f** }) =>

<div className="color-list">

{(colors.length === 0) ?

<p>No Colors Listed. (Add a Color)</p> :

colors.map(color =>

<Color key={color.id}

{...color}

**onRate={(rating) => onRate(color.id, rating)}**

**onRemove={() => onRemove(color.id)}** />

)

}

</div>

The ColorList component will invoke onRate if any colors are rated and onRemove if any colors are removed. This component manages the collection of colors by mapping them to individual Color components. When individual colors are rated or removed the ColorList identifies which color was rated or removed and passes that info to its parent via callback function properties.

ColorList’s parent is App. In the App component, rateColor and removeColor methods can be added and bound to the component instance in the constructor. Any time a color needs to be rated or removed, these methods will update the state. They are added to the ColorList component as callback function properties:

**class** App **extends** Component {

constructor(props) {

**super**(props)

**this**.state = {

colors: []

}

**this**.addColor = **this**.addColor.bind(**this**)

**this.rateColor = this.rateColor.bind(this)**

**this.removeColor = this.removeColor.bind(this)**

}

addColor(title, color) {

**const** colors = [

...**this**.state.colors,

{

id: v4(),

title,

color,

rating: 0

}

]

**this**.setState({colors})

}

**rateColor(id, rating) {**

**const colors = this.state.colors.map(color =>**

**(color.id !== id) ?**

**color :**

**{**

**...color,**

**rating**

**}**

**)**

**this.setState({colors})**

**}**

**removeColor(id) {**

**const colors = this.state.colors.filter(**

**color => color.id !== id**

**)**

**this.setState({colors})**

**}**

render() {

**const { addColor, rateColor, removeColor } = this**

**const** { colors } = **this**.state

**return** (

<div className="app">

<AddColorForm onNewColor={addColor} />

<ColorList colors={colors}

**onRate={rateColor}**

**onRemove={removeColor}** />

</div>

)

}

}

Both rateColor and removeColor expect the ID of the color to rate or remove. The ID is captured in the ColorList component and passed as an argument to rateColor or removeColor. The rateColor method finds the color to rate and changes its rating in the state. The removeColor method uses Array.filter to create a new state array without the removed color.

Once setState is called, the UI is rerendered with the new state data. All data that changes in this app is managed from a single component, App. This approach makes it much easier to understand what data the application uses to create state and how that data will change.

React components are quite robust. They provide us with a clean way to manage and validate properties, communicate with child elements, and manage state data from within a component. These features make it possible to construct beautifully scalable presentation layers.

We have mentioned many times that state is for data that changes. You can also use state to cache data in your application. For instance, if you had a list of records that the user could search, the records list could be stored in state until they are searched.

Reducing state to root components is often recommended. You will encounter this approach in many React applications. Once your application reaches a certain size, two-way data binding and explicitly passing properties can become quite a nuisance. The Flux design pattern and Flux libraries like Redux can be used to manage state and reduce boilerplate in these situations.

React is a relatively small library, and thus far we’ve reviewed much of its functionality. The major features of React components that we have yet to discuss include the component lifecycle and higher-order components, which we will cover in the next chapter.

[1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#idm140391598582032-marker) React Docs, [“Default Prop Values”](http://bit.ly/2oYLr4r)

[2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#idm140391597166672-marker) Pete Hunt, [“Thinking in React”](http://bit.ly/2nvMwgl).

[3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#idm140391620621312-marker) React Docs, [“Lifting State Up”](http://bit.ly/2o6ob0z).

[4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#idm140391620747152-marker) Paul Hudson, [“State and the Single Source of Truth”](http://bit.ly/2ne6BdY), Chapter 12 of *Hacking with React*.

[5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html#idm140391620331248-marker) Dan Abramov, [“Presentational and Container Components”](http://bit.ly/2ndQ9u0), Medium, March 23, 2015.

* [**Copy**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html)
* [**Add Highlight**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html)
* [**Add Note**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch06.html)

**Chapter 7. Enhancing Components**

So far we’ve learned how to mount and compose components to create application presentation layers with React. It is possible to build quite a few applications using only the React component’s render method. However, the world of JavaScript is complex. There is asynchronicity everywhere. We have latency to deal with when we load our data. We have delays to work with when we create animations. It is highly likely that you have preferred JavaScript libraries to help you navigate the complexity of real-world JavaScript.

Before we can enhance our applications with third-party JavaScript libraries or backend data requests, we must first understand how to work with the *component lifecycle*: a series of methods that can be invoked every time we mount or update a component.

We will start this chapter by exploring the component lifecycle. After we introduce the lifecycle, we will review how we can use it to load data, incorporate third-party JavaScript, and even improve our component’s performance. Next, we will explore how to reuse functionality across our applications with *higher-order components*. We will wrap up this chapter by looking at alternative application architectures that manage state entirely outside of React.

**Component Lifecycles**

The component lifecycle consists of methods that are invoked in series when a component is mounted or updated. These methods are invoked either before or after the component renders the UI. In fact, the render method itself is a part of the component lifecycle. There are two primary lifecycles: the mounting lifecycle and the updating lifecycle.

**Mounting Lifecycle**

The *mounting lifecycle* consists of methods that are invoked when a component is mounted or unmounted. In other words, these methods allow you to initially set up state, make API calls, start and stop timers, manipulate the rendered DOM, initialize third-party libraries, and more. These methods allow you to incorporate JavaScript to assist in the initialization and destruction of a component.

The mounting lifecycle is slightly different depending upon whether you use ES6 class syntax or React.createClass to create components. When you use createClass, getDefaultProps is invoked first to obtain the component’s properties. Next, getInitialState is invoked to initialize the state.

ES6 classes do not have these methods. Instead, default props are obtained and sent to the constructor as an argument. The constructor is where the state is initialized. Both ES6 class constructors and getInitialStatehave access to the properties and, if required, can use them to help define the initial state.

[Table 7-1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#table0701) lists the methods of the component mounting lifecycle.

| **ES6 class** | **React.createClass()** |
| --- | --- |
|  | getDefaultProps() |
| constructor(props) | getInitialState() |
| componentWillMount() | componentWillMount() |
| render() | render() |
| componentDidMount() | componentDidMount() |
| componentWillUnmount() | componentWillUnmount() |
| *Table 7-1. The component mounting lifecycle* | |

**CLASS CONSTRUCTORS**

Technically, the constructor is not a lifecycle method. We include it because it is used for component initialization (this is where the state is initialized). Also, the constructor is always the first function invoked when a component is mounted.

Once the properties are obtained and state is initialized, the componentWillMount method is invoked. This method is invoked before the DOM is rendered and can be used to initialize third-party libraries, start animations, request data, or perform any additional setup that may be required before a component is rendered. It is possible to invoke setState from this method to change the component state just before the component is initially rendered.

Let’s use the componentWillMount method to initialize a request for some members. When we get a successful response, we will update the state. Remember the getFakeMembers promise that we created in [Chapter 2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch02.html#emerging_javascript)? We will use that to load a random list of members from *randomuser.me*:

**const** getFakeMembers = count => **new** Promise((resolves, rejects) => {

**const** api = `https://api.randomuser.me/?nat=US&results=${count}`

**const** request = **new** XMLHttpRequest()

request.open('GET', api)

request.onload = () => (request.status == 200) ?

resolves(JSON.parse(request.response).results) :

reject(Error(request.statusText))

request.onerror = err => rejects(err)

request.send()

})

We will use this promise in the componentWillMount method in a MemberList component. This component will use a Member component to display each user’s picture, name, email address, and location:

**const** Member = ({ email, picture, name, location }) =>

<div className="member">

<img src={picture.thumbnail} alt="" />

<h1>{name.first} {name.last}</h1>

<p><a href={"mailto:" + email}>{email}</a></p>

<p>{location.city}, {location.state}</p>

</div>

**class** MemberList **extends** Component {

constructor() {

**super**()

**this**.state = {

members: [],

loading: **false**,

error: **null**

}

}

**componentWillMount() {**

**this.setState({loading: true})**

**getFakeMembers(this.props.count).then(**

**members => {**

**this.setState({members, loading: false})**

**},**

**error => {**

**this.setState({error, loading: false})**

**}**

**)**

**}**

componentWillUpdate() {

console.log('updating lifecycle')

}

render() {

**const** { members, loading, error } = **this**.state

**return** (

<div className="member-list">

{(loading) ?

<span>Loading Members</span> :

(members.length) ?

members.map((user, i) =>

<Member key={i} {...user} />

) :

<span>0 members loaded...</span>

}

{(error) ? <p>Error Loading Members: error</p> : ""}

</div>

)

}

}

Initially, when the component is mounted, MemberList has an empty array for members and loading is false. In the componentWillMount method, the state is changed to reflect the fact that a request was made to load some users. Next, while waiting for the request to complete, the component is rendered. Because loadingis now true, a message will be displayed alerting the user to the latency. When the promise passes or fails, the loading state is returned to false and either the members have been loaded or an error has been returned. Calling setState at this point will rerender our UI with either some members or an error.

**USING SETSTATE IN COMPONENTWILLMOUNT**

Calling setState before the component has rendered will not kick off the updating lifecycle. Calling setState after the component has been rendered will kick off the updating lifecycle. If you call setState inside an asynchronous callback defined within the componentWillMount method, it will be invoked after the component has rendered and will trigger the updating lifecycle.

The other methods of the component mounting lifecycle include componentDidMount and componentWillUnmount. componentDidMount is invoked just after the component has rendered, and componentWillUnmount is invoked just before the component is unmounted.

componentDidMount is another good place to make API requests. This method is invoked after the component has rendered, so any setState calls from this method will kick off the updating lifecycle and rerender the component.

componentDidMount is also a good place to initialize any third-party JavaScript that requires a DOM. For instance, you may want to incorporate a drag-and-drop library or a library that handles touch events. Typically, these libraries require a DOM before they can be initialized.

Another good use for componentDidMount is to start background processes like intervals or timers. Any processes started in componentDidMount or componentWillMount can be cleaned up in componentWillUnmount. You don’t want to leave background processes running when they are not needed.

Components are unmounted when their parents remove them or they have been unmounted with the unmountComponentAtNode function found in react-dom. This method is used to unmount the root component. When a root component is unmounted, its children are unmounted first.

Let’s take a look at a clock example. When the Clock component has mounted, a timer will be started. When the user clicks on the close button, the clock will be unmounted with unmountComponentAtNode and the timer stopped:

**import** React from 'react'

**import** { render, **unmountComponentAtNode** } from 'react-dom'

**import** { getClockTime } from './lib'

**const** { Component } = React

**const** target = document.getElementById('react-container')

**class** Clock **extends** Component {

constructor() {

**super**()

**this**.state = getClockTime()

}

**componentDidMount() {**

**console.log("Starting Clock")**

**this.ticking = setInterval(() =>**

**this.setState(getClockTime())**

**, 1000)**

**}**

**componentWillUnmount() {**

**clearInterval(this.ticking)**

**console.log("Stopping Clock")**

**}**

render() {

**const** { hours, minutes, seconds, timeOfDay } = **this**.state

**return** (

<div className="clock">

<span>{hours}</span>

<span>:</span>

<span>{minutes}</span>

<span>:</span>

<span>{seconds}</span>

<span>{timeOfDay}</span>

<button onClick={**this**.props.onClose}>x</button>

</div>

)

}

}

render(

<Clock onClose={() => **unmountComponentAtNode(target)** }/>,

target

)

In [Chapter 3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch03.html#functional_programming), we created a serializeTime function that abstracts civilian time with leading zeros from the data object. Every time serializeTime is invoked, the current time is returned in an object that contains hours, minutes, seconds, and the a.m. or p.m. indicator. Initially, we call serializeTime to get the initial state for our clock.

After the component has mounted, we start an interval called ticking. It invokes setState with a new time every second. The UI for the clock changes its value to the updated time every second.

When the close button is clicked, the Clock component is unmounted. Just before the clock is removed from the DOM, the ticking interval is cleared so that it no longer runs in the background.

**Updating Lifecycle**

The *updating lifecycle* is a series of methods that are invoked when a component’s state changes or when new properties are received from the parent. This lifecycle can be used to incorporate JavaScript before the component updates or to interact with the DOM after the update. Additionally, it can be used to improve the performance of an application because it gives you the ability to cancel unnecessary updates.

The updating lifecycle kicks off every time setState is called. Calling setState within the updating lifecycle will cause an infinite recursive loop that results in a stack overflow error. Therefore, setState can only be called in componentWillReceiveProps, which allows the component to update state when its properties are updated.

The updating lifecycle methods include:

componentWillReceiveProps(nextProps)

Only invoked if new properties have been passed to the component. This is the only method where setState can be called.

shouldComponentUpdate(nextProps, nextState)

The update lifecycle’s gatekeeper—a predicate that can call off the update. This method can be used to improve performance by only allowing necessary updates.

componentWillUpdate(nextProps, nextState)

Invoked just before the component updates. Similar to componentWillMount, only it is invoked before each update occurs.

componentDidUpdate(prevProps, prevState)

Invoked just after the update takes place, after the call to render. Similar to componentDidMount, but it is invoked after each update.

Let’s modify the color organizer application that we created in the last chapter. Specifically, we’ll add some updating lifecycle functions to the Color component that will allow us to see how the updating lifecycle works. Let’s assume that we already have four colors in the state array: Ocean Blue, Tomato, Lawn, and Party Pink. First, we will use the componentWillMount method to initialize color objects with a style, and set all four Color elements to have grey backgrounds:

**import** { Star, StarRating } from '../components'

**export** **class** Color **extends** Component {

**componentWillMount() {**

**this.style = { backgroundColor: "#CCC" }**

**}**

render() {

**const** { title, rating, color, onRate } = **this**.props

**return**

<section className="color" **style={this.style}**>

<h1 ref="title">{title}</h1>

<div className="color"

style={{ backgroundColor: color }}>

</div>

<StarRating starsSelected={rating}

onRate={onRate} />

</section>

}

}

Color.propTypes = {

title: PropTypes.string,

rating: PropTypes.number,

color: PropTypes.string,

onRate: PropTypes.func

}

Color.defaultProps = {

title: **undefined**,

rating: 0,

color: "#000000",

onRate: f=>f

}

When the color list originally mounts, each color background will be grey ([Figure 7-1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#fig0701)).



*Figure 7-1. Mounted colors with grey background*

We can add componentWillUpdate to the Color component in order to remove the grey background from each color just before the color updates:

componentWillMount() {

**this**.style = { backgroundColor: "#CCC" }

}

**componentWillUpdate() {**

**this.style = null**

**}**

Adding these lifecycle functions allows us to see when a component has mounted and when that component is updating. Initially, mounted components will have a grey background. Once each color is updated, the background will return to white.

If you run this code and rate any color, you will notice that all four colors update even though you have only changed the rating of a single color (see [Figure 7-2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#fig0702)).



*Figure 7-2. Rating blue triggers update, and all four components update*

Here, changing the rating of Ocean Blue from three to four stars caused all four colors to update because when the parent, the ColorList, updates state, it rerenders each Color component. Components that are rerendered are not re-mounted; if they are already there, an update occurs instead. When a component is updated, all of its children are also updated. When a single color is rated, all four colors are updated, all four StarRatingcomponents are updated, and all five stars on each component are updated.

We can improve the performance of our application by preventing colors from being updated when their property values have not changed. Adding the lifecycle function shouldComponentUpdate prevents unnecessary updates from occurring. This method returns either true or false (true when the component should update and false when updating should be skipped):

componentWillMount() {

**this**.style = { backgroundColor: "#CCC" }

}

**shouldComponentUpdate(nextProps) {**

**const { rating } = this.props**

**return rating !== nextProps.rating**

**}**

componentWillUpdate() {

**this**.style = **null**

}

The shouldComponentUpdate method can compare the new properties with the old ones. The new properties are passed to this method as an argument, the old properties are still the current props, and the component has not updated. If the rating is the same in the current properties and the new ones, there is no need for the color to update. If the color does not update, none of its children will update either. When the rating does not change, the entire component tree under each Color will not update.

This can be demonstrated by running this code and updating any of the colors. The componentWillUpdatemethod is only called if the component is going to update. It comes after shouldComponentUpdate in the lifecycle. The backgrounds will stay grey until the Color components are updated by changing their ratings ([Figure 7-3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#fig0703)).

If the shouldComponentUpdate method returns true, the rest of the updating lifecycle will get to it. The rest of the lifecycle functions also receive the new props and new state as arguments. (The componentDidUpdatemethod receives the previous props and the previous state because once this method is reached, the update already has occurred and the props have been changed.)

Let’s log a message after the component updates. In the componentDidUpdate function, we’ll compare the current properties to the old ones to see if the rating got better or worse:

componentWillMount() {

**this**.style = { backgroundColor: "#CCC" }

}

shouldComponentUpdate(nextProps) {

**const** { rating } = **this**.props

**return** rating !== nextProps.rating

}

componentWillUpdate() {

**this**.style = **null**

}

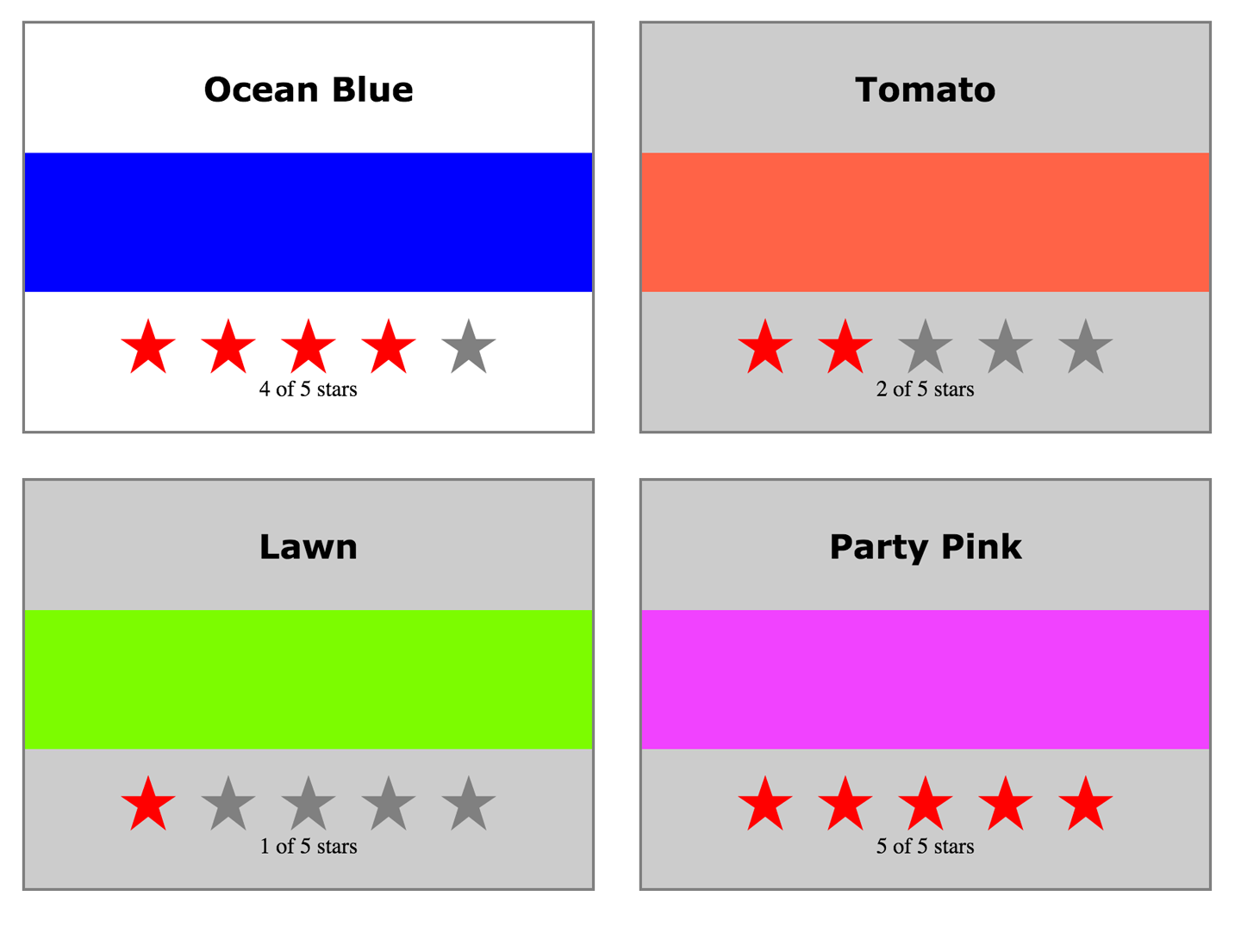
**componentDidUpdate(prevProps) {**

**const { title, rating } = this.props**

**const status = (rating > prevProps.rating) ? 'better' : 'worse'**

**console.log(`${title} is getting ${status}`)**

**}**



*Figure 7-3. One update at a time with shouldComponentUpdate*

The updating lifecycle methods componentWillUpdate and componentDidUpdate are great places to interact with DOM elements before or after updates. In this next sample, the updating process will be paused with an alert in componentWillUpdate:

componentWillMount() {

**this**.style = { backgroundColor: "#CCC" }

}

shouldComponentUpdate(nextProps) {

**return** **this**.props.rating !== nextProps.rating

}

componentWillUpdate(**nextProps**) {

**const** { title, rating } = **this**.props

**this**.style = **null**

**this.refs.title.style.backgroundColor = "red"**

**this.refs.title.style.color = "white"**

**alert(`${title}: rating ${rating} -> ${nextProps.rating}`)**

}

componentDidUpdate(prevProps) {

**const** { title, rating } = **this**.props

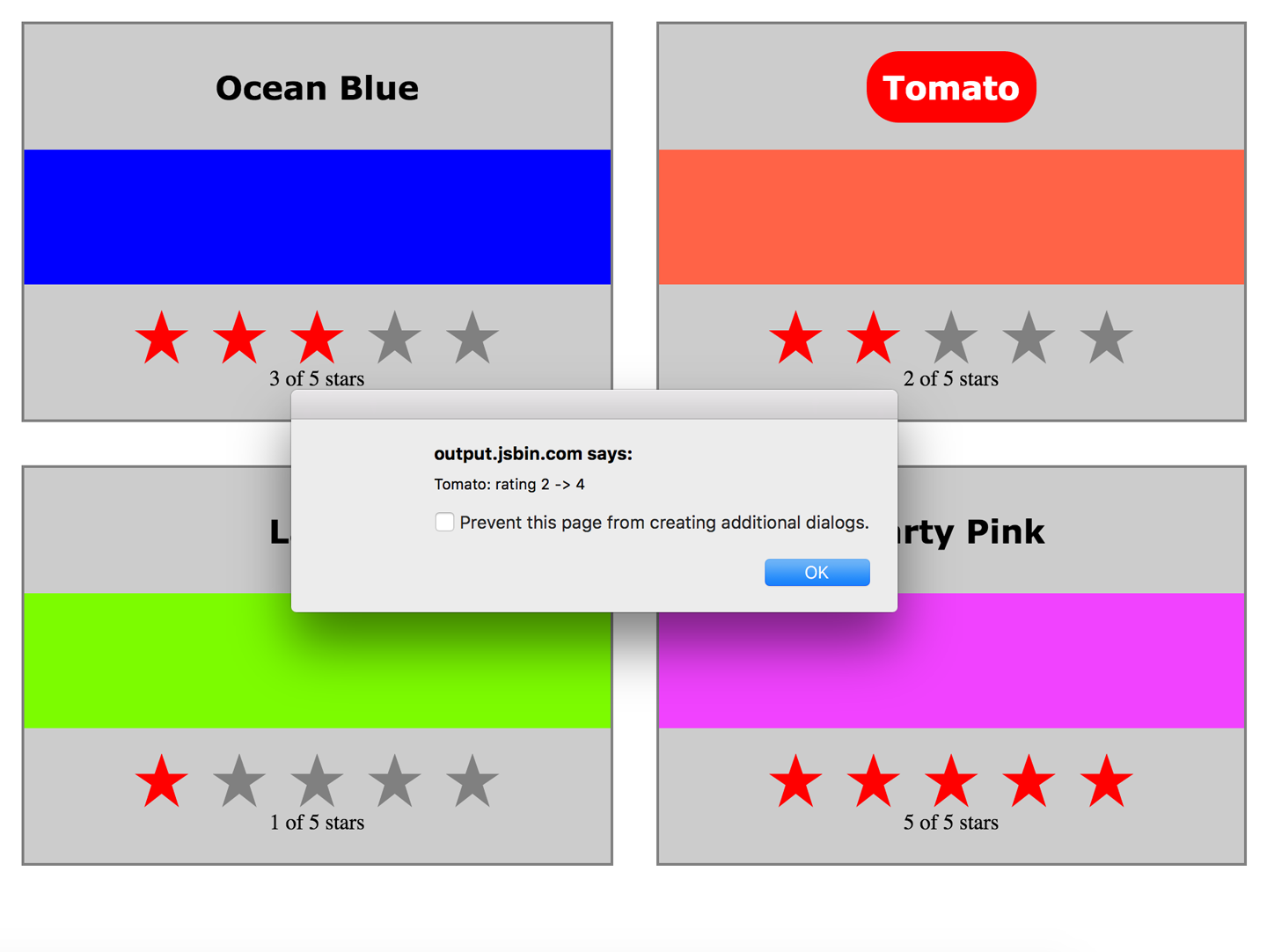
**const** status = (rating > prevProps.rating) ? 'better' : 'worse'

**this.refs.title.style.backgroundColor = ""**

**this.refs.title.style.color = "black"**

}

If change the rating of Tomato from two to four stars, the updating process will be paused by an alert ([Figure 7-4](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#fig0704)). The current DOM element for the color’s title is given a different background and text color.



*Figure 7-4. Updating paused with alert*

As soon as we clear the alert, the component updates and componentDidUpdate is invoked, clearing the title’s background color ([Figure 7-5](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#fig0705)).



*Figure 7-5. componentDidUpdate removes the title highlight*

Sometimes our components hold state that is originally set based upon properties. We can set the initial state of our component classes in the constructor or the componentWillMount lifecycle method. When those properties change, we will need to update the state using the componentWillReceiveProps method.

In [Example 7-1](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#example0702), we have a parent component that holds state, HiddenMessages. This component holds three messages in state and shows only one message at a time. When HiddenMessages mounts, an interval is added to cycle through the messages, only displaying one at a time.

*Example 7-1. HiddenMessages component*

**class** HiddenMessages **extends** Component {

constructor(props) {

**super**(props)

**this**.state = {

messages: [

"The crow crows after midnight",

"Bring a watch and dark clothes to the spot",

"Jericho Jericho Go"

],

showing: -1

}

}

componentWillMount() {

**this.interval = setInterval(() => {**

**let { showing, messages } = this.state**

**showing = (++showing >= messages.length) ?**

**-1 :**

**showing**

**this.setState({showing})**

**}, 1000)**

}

componentWillUnmount() {

clearInterval(**this**.interval)

}

render() {

**const** { messages, showing } = **this**.state

**return** (

<div className="hidden-messages">

{messages.map((message, i) =>

**<HiddenMessage key={i}**

**hide={(i!==showing)}>**

**{message}**

**</HiddenMessage>**

)}

</div>

)

}

}

The HiddenMessages component cycles through each of the messages in the state array and shows one at a time. The logic for this is set up in componentWillMount. When the component mounts, an interval is added that updates the index for the message that should be showing. The component renders all of the messages using the HiddenMessage component and only sets the hide property on one of them to true on each cycle. The rest of the properties are set to false, and the hidden message changes every second.

Take a look at the HiddenMessage component, the one used for each message ([Example 7-2](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#example0703)). When this component is originally mounted, the hide property is used to determine its state. However, when the parent updates this component’s properties, nothing happens. This component will not know about it.

*Example 7-2. HiddenMessage component*

**class** HiddenMessage **extends** Component {

constructor(props) {

**super**(props)

**this**.state = {

hidden: **(props.hide) ? props.hide : true**

}

}

render() {

**const** { children } = **this**.props

**const** { hidden } = **this**.state

**return** (

<p>

**{(hidden) ?**

**children.replace(/[a-zA-Z0-9]/g, "x") :**

**children**

**}**

</p>

)

}

}

The problem occurs when the parent component changes the hide property. That change does not automatically cause the state of HiddenMessage to change.

The componentWillReceiveProps lifecycle method was created for these scenarios. It will be invoked when the properties have been changed by the parent, and those changed properties can be used to modify the state internally:

**class** HiddenMessage **extends** Component {

constructor(props) {

**super**(props)

**this**.state = {

hidden: (props.hide) ? props.hide : **true**

}

}

**componentWillReceiveProps(nextProps) {**

**this.setState({hidden: nextProps.hide})**

**}**

render() {

**const** { children } = **this**.props

**const** { hidden } = **this**.state

**return** (

<p>

{(hidden) ?

children.replace(/[a-zA-Z0-9]/g, "x") :

children

}

</p>

)

}

}

When the parent component, HiddenMessages, changes the property for hide, componentWillReceiveProps allows us to update the state.

**SETTING STATE FROM PROPS**

The previous code sample has been reduced to demonstrate the use of componentWillReceiveProps. If this is all we are doing with HiddenMessage, then we should use a stateless functional component instead. The only reason we would ever add state to a child component is when we want that component to change things about itself internally.

For example, using componentWillReceiveProps to modify state would be warranted if the component required a setState call:

**hide() {**

**const hidden = true**

**this.setState({hidden})**

**}**

**show() {**

**const hidden = false**

**this.setState({hidden})**

**}**

**return**

<p **onMouseEnter={this.show}**

**onMouseLeave={this.hide}**>

{(hidden) ?

children.replace(/[a-zA-Z0-9]/g, "x") :

children

}

</p>

In this case, it would be appropriate to store state in the HiddenMessage component. If the component is not going to change itself, keep it stateless and manage the state from the parent only.

The component lifecycle methods give us much more control over how a component should be rendered or updated. They provide hooks for us to add functionality before or after both mounting and updating have occurred. Next, we will explore how these lifecycle methods can be used to incorporate third-party JavaScript libraries. First, however, we’ll take a brief look at the “React.Children” API.

**React.Children**

React.Children provides a way of working with the children of a particular component. It allows you to count, map, loopover, or convert props.children to an array. It also allows you to verify that you are displaying a single child with React.Children.only:

**import** { **Children**, PropTypes } from 'react'

**import** { render } from 'react-dom'

**const** Display = ({ ifTruthy=**true**, children }) =>

(ifTruthy) ?

**Children.only(children)** :

**null**

**const** age = 22

render(

<Display ifTruthy={age >= 21}>

**<h1>You can enter</h1>**

</Display>,

document.getElementById('react-container')

)

In this example, the Display component will display only a single child, the h1 element. If the Displaycomponent contained multiple children, React would throw an error: “onlyChild must be passed a children with exactly one child.”

We can also use React.Children to convert the children property to an array. This next sample extends the Display component to additionally handle else cases:

**const** { **Children**, PropTypes } = React

**const** { render } = ReactDOM

**const** findChild = (children, child) =>

**Children.toArray(children)**

**.filter(c => c.type === child )[0]**

**const** WhenTruthy = ({children}) =>

**Children.only(children)**

**const** WhenFalsy = ({children}) =>

**Children.only(children)**

**const** Display = ({ ifTruthy=**true**, children }) =>

(ifTruthy) ?

findChild(children, WhenTruthy) :

findChild(children, WhenFalsy)

**const** age = 19

render(

<Display ifTruthy={age >= 21}>

<WhenTruthy>

**<h1>You can Enter</h1>**

</WhenTruthy>

<WhenFalsy>

**<h1>Beat it Kid</h1>**

</WhenFalsy>

</Display>,

document.getElementById('react-container')

)

The Display component will display a single child when a condition is true or another when the condition is false. To accomplish this, we create WhenTruthy and WhenFalsy components and use them as children in the Display component. The findChild function uses React.Children to convert the children into an array. We can filter that array to locate and return an individual child by component type.

**JavaScript Library Integration**

Frameworks such as Angular and jQuery come with their own tools for accessing data, rendering the UI, modeling state, handling routing, and more. React, on the other hand, is simply a library for creating views, so we may need to work with other JavaScript libraries. If we understand how the lifecycle functions operate, we can make React play nice with just about any JavaScript library.

**REACT WITH JQUERY**

Using jQuery with React is generally frowned upon by the community. It is possible to integrate jQuery and React, and the integration could be a good choice for learning React or migrating legacy code to React. However, applications perform much better if we incorporate smaller libraries with React, as opposed to large frameworks. Additionally, using jQuery to manipulate the DOM directly bypasses the virtual DOM, which can lead to strange errors.

In this section, we’ll incorporate a couple of different JavaScript libraries into React components. Specifically, we’ll look at ways to make API calls and visualize data with the support of other JavaScript libraries.

**Making Requests with Fetch**

Fetch is a polyfill created by the WHATWG group that allows us to easily make API calls using promises. In this section we will introduce isomorphic-fetch, a version of Fetch that works nicely with React. Let’s install isomorphic-fetch:

npm install isomorphic-fetch --save

The component lifecycle functions provide us a place to integrate JavaScript. In this case, they are where we will make an API call. Components that make API calls have to handle *latency*, the delay that the user experiences while waiting for a response. We can address these issues in our state by including variables that tell the component whether a request is pending or not.

In the following example, the CountryList component creates an ordered list of country names. Once mounted, the component makes an API call and changes the state to reflect that it is loading data. The loading state remains true until there is a response from this API call:

**import** { Component } from 'react'

**import** { render } from 'react-dom'

**import** fetch from 'isomorphic-fetch'

**class** CountryList **extends** Component {

constructor(props) {

**super**(props)

**this.state = {**

**countryNames: [],**

**loading: false**

**}**

}

componentDidMount() {

**this.setState({loading: true})**

**fetch('https://restcountries.eu/rest/v1/all')**

**.then(response => response.json())**

**.then(json => json.map(country => country.name))**

**.then(countryNames =>**

**this.setState({countryNames, loading: false})**

**)**

}

render() {

**const** { countryNames, loading } = **this**.state

**return** (loading) ?

<div>Loading Country Names...</div> :

(!countryNames.length) ?

<div>No country Names</div> :

<ul>

{countryNames.map(

(x,i) => <li key={i}>{x}</li>

)}

</ul>

}

}

render(

<CountryList />,

document.getElementById('react-container')

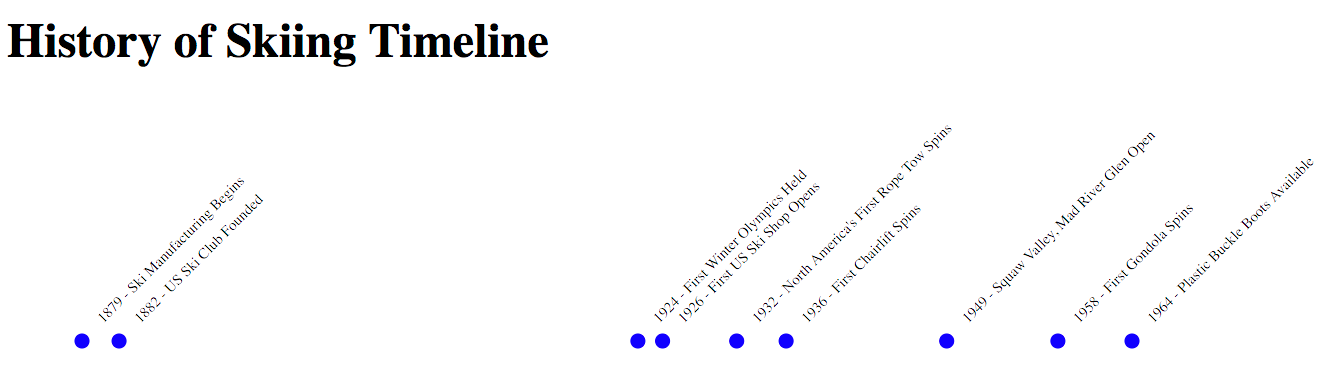
)

When the component mounts, just before the fetch call, we set the loading state to true. This tells our component, and ultimately our users, that we are in the process of retrieving the data. When we get a response from our fetch call, we obtain the JSON object and map it to an array of country names. Finally, the country names are added to the state and the DOM updated.

**Incorporating a D3 Timeline**

Data Driven Documents (D3) is a JavaScript framework that can be used to construct data visualizations for the browser. D3 provides a rich set of tools that allow us to scale and interpolate data. Additionally, D3 is functional. You compose D3 applications by chaining function calls together to produce a DOM visualization from an array of data.

A timeline is an example of a data visualization. A timeline takes event dates as data and represents that information visually with graphics. Historic events that occurred earlier are represented to the left of those events that occurred later. The space between each event on a timeline in pixels represents the time that has elapsed between the events ([Figure 7-6](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#fig0706)).



*Figure 7-6. Timeline data visualization*

This timeline visualizes almost 100 years’ worth of events in just 500 pixels. The process of converting year values to their corresponding pixel values is called *interpolation*. D3 provides all of the tools necessary for interpolating data ranges from one measurement to another.

Let’s take a look at how to incorporate D3 with React to build this timeline. First, we’ll need to install D3:

npm install d3 --save

D3 takes data, typically arrays of objects, and develops visualizations based upon that data. Take a look at the array of historic ski dates. This is the data for our timeline:

**const** historicDatesForSkiing = [

{

year: 1879,

event: "Ski Manufacturing Begins"

},

{

year: 1882,

event: "US Ski Club Founded"

},

{

year: 1924,

event: "First Winter Olympics Held"

},

{

year: 1926,

event: "First US Ski Shop Opens"

},

{

year: 1932,

event: "North America's First Rope Tow Spins"

},

{

year: 1936,

event: "First Chairlift Spins"

},

{

year: 1949,

event: "Squaw Valley, Mad River Glen Open"

},

{

year: 1958,

event: "First Gondola Spins"

},

{

year: 1964,

event: "Plastic Buckle Boots Available"

}

]

The easiest way to incorporate D3 into a React component is to let React render the UI, then have D3 create and add the visualization. In the following example, D3 is incorporated into a React component. Once the component renders, D3 builds the visualization and adds it to the DOM:

**import** d3 from 'd3'

**import** { Component } from 'react'

**import** { render } from 'react-dom'

**class** Timeline **extends** Component {

constructor({data=[]}) {

**const times = d3.extent(data.map(d => d.year))**

**const range = [50, 450]**

**super**({data})

**this**.state = {data, times, range}

}

componentDidMount() {

**let** group

**const** { data, times, range } = **this**.state

**const** { target } = **this**.refs

**const scale = d3.time.scale().domain(times).range(range)**

**d3.select(target)**

**.append('svg')**

**.attr('height', 200)**

**.attr('width', 500)**

**group = d3.select(target.children[0])**

**.selectAll('g')**

**.data(data)**

**.enter()**

**.append('g')**

**.attr(**

**'transform',**

**(d, i) => 'translate(' + scale(d.year) + ', 0)'**

**)**

**group.append('circle')**

**.attr('cy', 160)**

**.attr('r', 5)**

**.style('fill', 'blue')**

**group.append('text')**

**.text(d => d.year + " - " + d.event)**

**.style('font-size', 10)**

**.attr('y', 115)**

**.attr('x', -95)**

**.attr('transform', 'rotate(-45)')**

}

render() {

**return** (

<div className="timeline">

<h1>{**this**.props.name} Timeline</h1>

<div ref="target"></div>

</div>

)

}

}

render(

<Timeline name="History of Skiing"

data={historicDatesForSkiing} />,

document.getElementById('react-container')

)

In this example, some of the D3 setup occurs in the constructor, but most of the heavy lifting is done by D3 in the componentDidMount function. Once the DOM is rendered, D3 builds the visualization using Scalable Vector Graphics (SVG). This approach will work and is a good way to quickly incorporate existing D3 visualizations into React components.

We can, however, take this integration one step further by letting React manage the DOM and D3 do the math. Take a look at these three lines of code:

**const** times = d3.extent(data.map(d => d.year))

**const** range = [50, 450]

**const** scale = d3.time.scale().domain(times).range(range)

Both times and range are set up in the constructor and added to the component state. times represents our domain. It contains the values for the earliest year and the latest year. It is calculated by using D3’s extentfunction to find the minimum and maximum values in an array of numeric values. range represents the range in pixels for the timeline. The first date, 1879, will be placed at 0 px on the x-scale and the last date, 1964, will be placed at 450 px on the x-scale.

The next line creates the scale, which is a function that can be used to interpolate the pixel value for any year on our time scale. The scale is created by sending the domain and the range to the D3 time.scale function. The scale function is used in the visualization to get the *x* position for every date that falls between 1879 and 1964.

Instead of creating the scale in componentDidMount, we can add it to the component in the constructor after we have the domain and range. Now the scale can be accessed anywhere in the component using this.scale(year):

constructor({data=[]}) {

**const** times = d3.extent(data.map(d => d.year))

**const** range = [50, 450]

**super**({data})

**this**.scale = d3.time.scale().domain(times).range(range)

**this**.state = {data, times, range}

}

Within componentDidMount, D3 first creates an SVG element and adds it to the target ref:

d3.select(target)

.append('svg')

.attr('height', 200)

.attr('width', 500)

Constructing a UI is a task for React. Instead of using D3 for this task, let’s create a Canvas component that returns an SVG element:

const Canvas = ({children}) =>

<svg height="200" width="500">

{children}

</svg>

Next, D3 selects the svg element, the first child under the target, and adds a group element for every data point in our timeline array. After it is added, the group element is positioned by transforming the x-axis value using the scale function:

group = d3.select(target.children[0])

.selectAll('g')

.data(data)

.enter()

.append('g')

.attr(

'transform',

(d, i) => 'translate(' + scale(d.year) + ', 0)'

)

The group element is a DOM element, so we can let React handle this task too. Here is a TimelineDotcomponent that can be used to set up group elements and position them along the x-axis:

**const** TimelineDot = ({position}) =>

<g transform={`translate(${position},0)`}></g>

Next, D3 adds a circle element and some “style” to the group. The text element gets its value by concatenating the event year with the event title. It then positions and rotates that text around the blue circle:

group.append('circle')

.attr('cy', 160)

.attr('r', 5)

.style('fill', 'blue')

group.append('text')

.text(d => d.year + " - " + d.event)

.style('font-size', 10)

.attr('y', 115)

.attr('x', -95)

.attr('transform', 'rotate(-45)')

All we need to do is modify our TimelineDot component to include a circle element and a text element that retrieves the text from the properties:

**const** TimelineDot = ({position, **txt**}) =>

<g transform={`translate(${position},0)`}>

**<circle cy={160}**

**r={5}**

**style={{fill: 'blue'}} />**

**<text y={115}**

**x={-95}**

**transform="rotate(-45)"**

**style={{fontSize: '10px'}}>{txt}</text>**

</g>

React is now responsible for managing the UI using the virtual DOM. The role of D3 has been reduced, but it still provides some essential functionality that React does not. It helps create the domain and range and constructs a scale function that we can use to interpolate pixel values from years. This is what our refactored Timelinecomponent might look like:

**class** Timeline **extends** Component {

constructor({data=[]}) {

**const** times = d3.extent(data.map(d => d.year))

**const** range = [50, 450]

**super**({data})

**this**.scale = d3.time.scale().domain(times).range(range)

**this**.state = {data, times, range}

}

render() {

**const** { data } = **this**.state

**const** { scale } = **this**

**return** (

<div className="timeline">

<h1>{**this**.props.name} Timeline</h1>

<Canvas>

{data.map((d, i) =>

<TimelineDot position={scale(d.year)}

txt={`${d.year} - ${d.event}`}

/>

)}

</Canvas>

</div>

)

}

}

We can integrate just about any JavaScript library with React. The lifecycle functions are the place where other JavaScript can pick up where React leaves off. However, we should avoid adding libraries that manage the UI: that’s React’s job.

**Higher-Order Components**

A *higher-order component*, or HOC, is a simply a function that takes a React component as an argument and returns another React component. Typically, HOCs wrap the incoming component with a class that maintains state or has functionality. Higher-order components are the best way to reuse functionality across React components.

**MIXINS NOT SUPPORTED**

Until React v0.13, the best way to incorporate functionality in a React component was to use a *mixin*. Mixins can be added directly to components created with createClass as a configuration property. You can still use mixins with React.createClass, but they are not supported in ES6 classes or stateless functional components. They will also not be supported by future versions of React.

An HOC allows us to wrap a component with another component. The parent component can hold state or contain functionality that can be passed down to the composed component as properties. The composed component does not need to know anything about the implementation of an HOC other than the names of the properties and methods that it makes available.

Take a look at this PeopleList component. It loads random users from an API and renders a list of member names. While the users are loading, a loading message is displayed. Once they have loaded, they are displayed on the DOM:

**import** { Component } from 'react'

**import** { render } from 'react-dom'

**import** fetch from 'isomorphic-fetch'

**class** PeopleList **extends** Component {

constructor(props) {

**super**(props)

**this**.state = {

data: [],

loaded: **false**,

loading: **false**

}

}

componentWillMount() {

**this**.setState({loading:**true**})

fetch('https://randomuser.me/api/?results=10')

.then(response => response.json())

.then(obj => obj.results)

.then(data => **this**.setState({

loaded: **true**,

loading: **false**,

data

}))

}

render() {

**const** { data, loading, loaded } = **this**.state

**return** (loading) ?

<div>Loading...</div> :

<ol className="people-list">

{data.map((person, i) => {

**const** {first, last} = person.name

**return** <li key={i}>{first} {last}</li>

})}

</ol>

}

}

render(

<PeopleList />,

document.getElementById('react-container')

)

PeopleList incorporates a getJSON call from jQuery to load people from a JSON API. When the component is rendered, it displays a loading message or renders a list of names based upon whether or not the loading state is true.

If we harness this loading functionality, we can reuse it across components. We could create a higher-order component, the DataComponent, that can be used to create React components that load data. To use the DataComponent, we strip the PeopleList of state and create a stateless functional component that receives data via props:

**import** { render } from 'react-dom'

**const** PeopleList = ({data}) =>

<ol className="people-list">

{data.results.map((person, i) => {

**const** {first, last} = person.name

**return** <li key={i}>{first} {last}</li>

})}

</ol>

**const RandomMeUsers = DataComponent(**

**PeopleList,**

**"https://randomuser.me/api/"**

**)**

render(

<RandomMeUsers count={10} />,

document.getElementById('react-container')

)

Now we are able to create a RandomMeUsers component that always loads and displays users from the same source, *randomuser.me*. All we have to do is provide the count of how many users we wish to load. The data handling has been moved into the HOC, and the UI is handled by the PeopleList component. The HOC provides the state for loading and the mechanism to load data and change its own state. While data is loading, the HOC displays a loading message. Once the data has loaded, the HOC handles mounting the PeopleList and passing it people as the data property:

**const** DataComponent = (ComposedComponent, url) =>

**class** DataComponent **extends** Component {

constructor(props) {

**super**(props)

**this**.state = {

data: [],

loading: **false**,

loaded: **false**

}

}

componentWillMount() {

**this**.setState({loading:**true**})

fetch(url)

.then(response => response.json())

.then(data => **this**.setState({

loaded: **true**,

loading: **false**,

data

}))

}

render() {

**return** (

<div className="data-component">

{(**this**.state.loading) ?

<div>Loading...</div> :

<ComposedComponent {...**this**.state} />}

</div>

)

}

}

Notice that DataComponent is actually a function. All higher-order components are functions. ComposedComponent is the component that we will wrap. The returned class, DataComponent, stores and manages the state. When that state changes and the data has loaded, the ComposedComponent is rendered and that data is passed to it as a property.

This HOC can be used to create any type of data component. Let’s take a look at how DataComponent can be reused to create a CountryDropDown that is populated with a country name for every country in the world delivered from the *restcountries.eu* API:

**import** { render } from 'react-dom'

**const** CountryNames = ({data, selected=""}) =>

<select className="people-list" defaultValue={selected}>

{data.map(({name}, i) =>

<option key={i} value={name}>{name}</option>

)}

</select>

**const** CountryDropDown =

DataComponent(

CountryNames,

"https://restcountries.eu/rest/v1/all"

)

render(

<CountryDropDownselected="United States"/>,

document.getElementById('react-container')

)

The CountryNames component obtains the country names via props. DataComponent handles loading and passing information about each country.

Notice that the CountryNames component also has a selected property. This property should cause the component to select “United States” by default. However, at present, it is not working. We did not pass the properties to the composed component from our HOC.

Let’s modify our HOC to pass any properties that it receives down to the composed component:

render() {

**return** (

<div className="data-component">

{(**this**.state.loading) ?

<div>Loading...</div> :

**<ComposedComponent {...this.state}**

**{...this.props} />**

}

</div>

)

}

Now the HOC passes state and props down to the composed component. If we run this code now, we will see that the CountryDropDown preselects “United States”.

Let’s take a look at another HOC. We developed a HiddenMessage component earlier in this chapter. The ability to show or hide content is something that can be reused. In this next example, we have an ExpandableHOC that functions similarly to the HiddenMessage component. You can show or hide content based upon the Boolean property collapsed. This HOC also provides a mechanism for toggling the collapsed property ([Example 7-3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#example0705)).

*Example 7-3. ./components/hoc/Expandable.js*

**import** { Component } from 'react'

**const** Expandable = ComposedComponent =>

**class** Expandable **extends** Component {

constructor(props) {

**super**(props)

**const** collapsed =

(props.hidden && props.hidden === **true**) ?

**true** :

**false**

**this**.state = {collapsed}

**this**.expandCollapse = **this**.expandCollapse.bind(**this**)

}

expandCollapse() {

**let** collapsed = !**this**.state.collapsed

**this**.setState({collapsed})

}

render() {

**return** <ComposedComponent

expandCollapse={**this**.expandCollapse}

{...**this**.state}

{...**this**.props} />

}

}

The Expandable HOC takes a ComposedComponent and wraps it with state and functionality that allows it to show or hide content. Initially, the collapsed state is set using incoming properties, or it defaults to false. The collapsed state is passed down to the ComposedComponent as a property.

This component also has a method for toggling the collapsed state called expandCollapse. This method is also passed down to the ComposedComponent. Once invoked, it will change the collapsed state and update the ComposedComponent with the new state.

If the properties of the DataComponent are changed by a parent, the component will update the collapsed state and pass the new state down to the ComposedComponent as a property.

Finally, all state and props are passed down to the ComposedComponent. Now we can use this HOC to create several new components. First, let’s use it to create the HiddenMessage component that we defined earlier in this chapter:

**const** ShowHideMessage = ({children, collapsed, expandCollapse}) =>

<p onClick={expandCollapse}>

{(collapsed) ?

children.replace(/[a-zA-Z0-9]/g, "x") :

children}

</p>

**const** HiddenMessage = Expandable(ShowHideMessage)

Here we create a HiddenMessage component that will replace every letter or number in a string with an “x” when the collapsed property is true. When the collapsed property is false, the message will be shown. Try this HiddenMessage component out in the HiddenMessages component that we defined earlier in this chapter.

Let’s use this same HOC to create a button that shows and hides hidden content in a div. In the following example, the MenuButton can be used to create PopUpButton, a component that toggles content display:

**class** MenuButton **extends** Component {

componentWillReceiveProps(nextProps) {

**const** collapsed =

(nextProps.collapsed && nextProps.collapsed === **true**) ?

**true** :

**false**

**this**.setState({collapsed})

}

render() {

**const** {children, collapsed, txt, expandCollapse} = **this**.props

**return** (

<div className="pop-button">

<button onClick={expandCollapse}>{txt}</button>

{(!collapsed) ?

<div className="pop-up">

{children}

</div> :

""

}

</div>

)

}

}

**const** PopUpButton = Expandable(MenuButton)

render(

<PopUpButton hidden={**true**} txt="toggle popup">

<h1>Hidden Content</h1>

<p>This content will start off hidden</p>

</PopUpButton>,

document.getElementById('react-container')

)

The PopUpButton is created with the MenuButton component. It will pass the collapsed state along with the function to change that state to the MenuButton as properties. When users click on the button, it will invoke expandCollapse and toggle the collapsed state. When the state is collapsed, we only see a button. When it is expanded we see a button and a div with the hidden content.

Higher-order components are a great way to reuse functionality and abstract away the details of how component state or lifecycle are managed. They will allow you to produce more stateless functional components that are solely responsible for the UI.

**Managing State Outside of React**

State management in React is great. We could build a lot of applications using React’s built-in state management system. However, when our applications get larger, state becomes a little bit harder for us to wrap our heads around. Keeping state in one place at the root of your component tree will help make this task easier, but even then, your application may grow to a point where it makes the most sense to isolate state data in its own layer, independent of the UI.

One of the benefits of managing state outside of React is that it will reduce the need for many, if any, class components. If you are not using state, it is easier to keep most of your components stateless. You should only need to create a class when you need lifecycle functions, and even then you can isolate class functionality to HOCs and keep components that only contain UI stateless. Stateless functional components are easier to understand and easier to test. They are pure functions, so they fit into strictly functional applications quite nicely.

Managing state outside of React could mean a lot of different things. You can use React with Backbone Models, or with any other MVC library that models state. You can create your own system for managing state. You can manage state using global variables or localStorage and plain JavaScript. Managing state outside of React simply means not using React state or setState in your applications.

**Rendering a Clock**

Back in [Chapter 3](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch03.html#functional_programming), we created a ticking clock that followed the rules of functional programming. The entire application consists of functions and higher-order functions that are composed into larger functions that compose a startTicking function that starts the clock and displays the time in the console:

**const** startTicking = () =>

setInterval(

compose(

clear,

getCurrentTime,

abstractClockTime,

convertToCivilianTime,

doubleDigits,

formatClock("hh:mm:ss tt"),

display(log)

),

oneSecond()

)

startTicking()

But instead of displaying the clock in the console, what if we displayed it in the browser? We could build a React component to display the clock time in a div:

**const** AlarmClockDisplay = ({hours, minutes, seconds, ampm}) =>

<div className="clock">

<span>{hours}</span>

<span>:</span>

<span>{minutes}</span>

<span>:</span>

<span>{seconds}</span>

<span>{ampm}</span>

</div>

This component takes in properties for hours, minutes, seconds, and time of day. It then creates a DOM where those properties can be displayed.

We could replace the log method with a render method and send our component to be used to render the civilian time, with leading zeros added to values less than 10:

**const** startTicking = () =>

setInterval(

compose(

getCurrentTime,

abstractClockTime,

convertToCivilianTime,

doubleDigits,

**render(AlarmClockDisplay)**

),

oneSecond()

)

startTicking()

The render method will need to be a higher-order function. It will need to take the AlarmClockDisplay as a property initially when the startTicking method is composed and hang on to it. Eventually, it will need to use that component to render the display with the formatted time every second:

**const** render = Component => civilianTime =>

ReactDOM.render(

<Component {...civilianTime} />,

document.getElementById('react-container')

)

The higher-order function for render invokes ReactDOM.render every second and updates the DOM. This approach takes advantage of React’s speedy DOM rendering, but does not require a component class with state.

The state of this application is managed outside of React. React allowed us to keep our functional architecture in place by providing our own higher-order function that renders a component with ReactDOM.render. Managing state outside of React is not a requirement, it is simply another option. React is a library, and it is up to you to decide how best to use it in your applications.

Next, we will introduce Flux, a design pattern that was created as an alternative to state management in React.

**Flux**

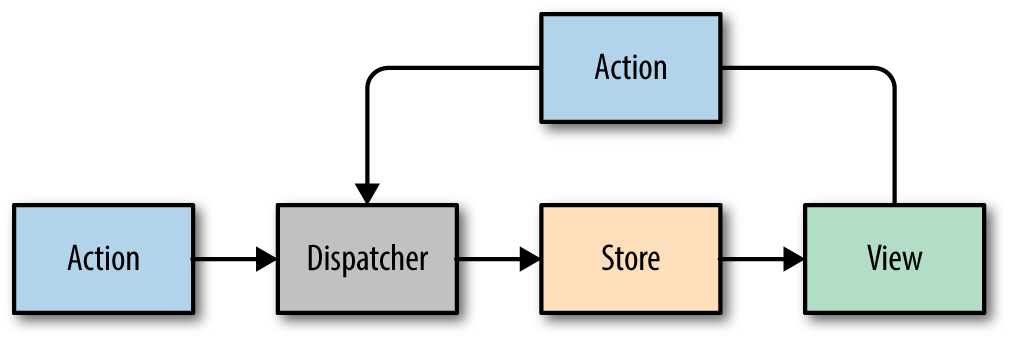
Flux is a design pattern developed at Facebook that was designed to keep data flowing in one direction. Before Flux was introduced, web development architecture was dominated by variations of the MVC design pattern. Flux is an alternative to MVC, an entirely different design pattern that complements the functional approach.

What does React or Flux have to do with functional JavaScript? For starters, a stateless functional component is a pure function. It takes in instructions as props and returns UI elements. A React class uses state or props as input and also will produce UI elements. React components are composed into a single component. Immutable data provides the component with input and output as UI elements are returned:

**const** Countdown = ({count}) => <h1>{count}</h1>

Flux provides us with a way to architect web applications that complements how React works. Specifically, Flux provides a way to provide the data that React will use to create the UI.

In Flux, application state data is managed outside of React components in *stores*. Stores hold and change the data, and are the only thing that can update a view in Flux. If a user were to interact with a web page—say, click a button or submit a form—then an *action* would be created to represent the user’s request. An action provides the instructions and data required to make a change. Actions are dispatched using a central control component called the *dispatcher*. The dispatcher is designed to queue up our actions and dispatch them to the appropriate store. Once a store receives an action, it will use it as instructions to modify state and update the view. Data flows in one direction: action to a dispatcher to the store and finally to the view ([Figure 7-7](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#fig0707)).



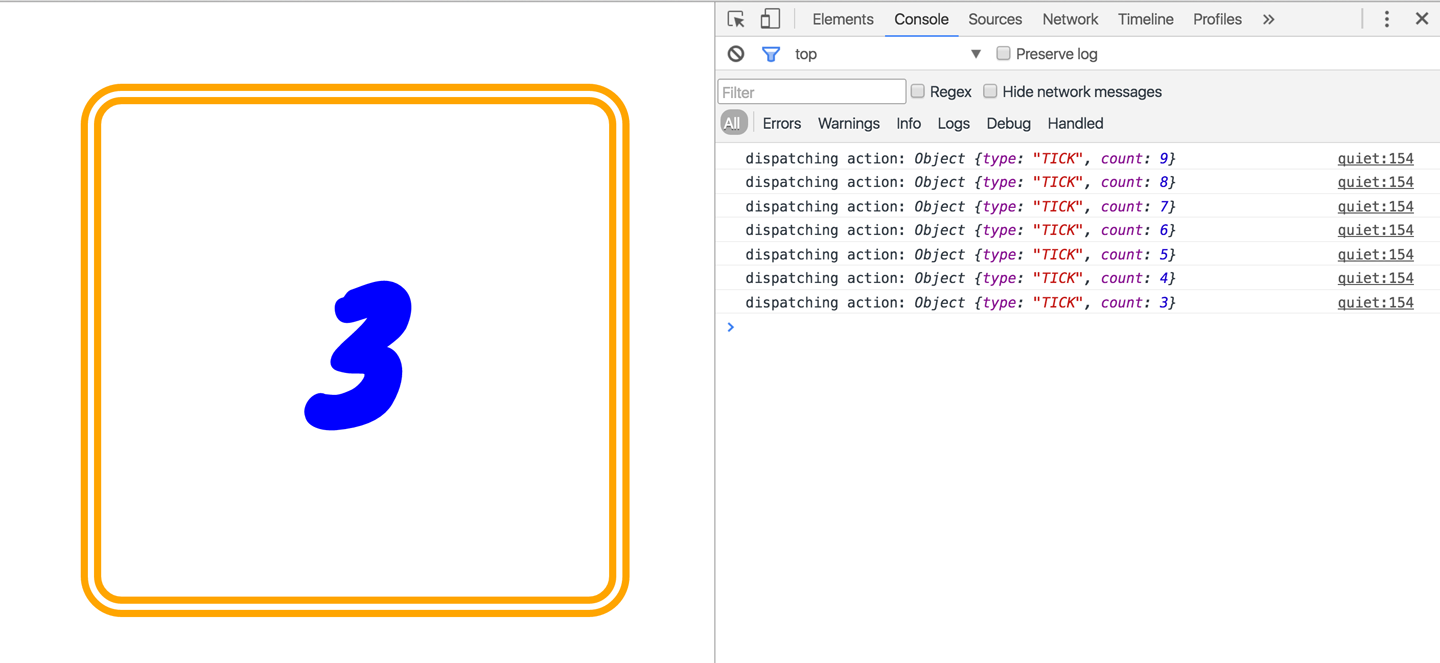
*Figure 7-7. Facebook’s Flux Design Pattern*

Actions and state data are immutable in Flux. Actions can be dispatched from a view, or they can come from other sources, typically a web server.

Every change requires an action. Every action provides the instructions to make the change. Actions also serve as receipts that tell us what has changed, what data was used to make the change, and where the action originated. This pattern causes no side effects. The only thing that can make a change is a store. Stores update the data, views render those updates in the UI, and actions tell us how and why the changes have occurred.

Restricting the data flow of your application to this design pattern will make your application much easier to fix and scale. Take a look at the application in [Figure 7-8](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html#fig0708). We can see that every dispatched action has been logged to the console. These actions tell us about how we got to the current UI that is displaying a giant number 3.

As we can see, the last state change that occurred was a TICK. It changed the count to 3 from the count before that, which looks to have been a 4. The actions tell us how the state has been changing. We can trace the actions back to the source to see that first change was to a 9, so presumably this app is counting down from 10.



*Figure 7-8. Countdown app with Flux*

Let’s take a look at how this countdown is constructed using the Flux design pattern. We will introduce each part of the design pattern and discuss its contribution to the unidirectional data flow that makes up this countdown.

**Views**

Let’s begin by looking at the view, a React stateless component. Flux will manage our application state, so unless you need a lifecycle function you will not need class components.

The countdown view takes in the count to display as a property. It also receives a couple of functions, tick and reset:

**const** Countdown = ({count, tick, reset}) => {

**if** (count) {

setTimeout(() => tick(), 1000)

}

**return** (count) ?

<h1>{count}</h1> :

<div onClick={() => reset(10)}>

<span>CELEBRATE!!!</span>

<span>(click to start over)</span>

</div>

}

When this view renders it will display the count, unless the count is 0, in which case it will display a message instructing the user to “CELEBRATE!!!” If the count is not 0, then the view sets a timeout, waits for a second, and invokes a TICK.

When the count is 0, this view will not invoke any other action creators until a user clicks the main div and triggers a reset. This resets the count to 10 and starts the whole countdown process over again.

**STATE IN COMPONENTS**

Using Flux does not mean that you cannot have state in any of your view components. It means that application state is not managed in your view components. For example, Flux can manage the dates and times that make up timelines. It would not be off-limits to use a timeline component that has internal state to visualize your application’s timelines.

State should be used sparingly—only when needed, from reusable components that internally manage their own state. The rest of the application does not need to be “aware” of a child component’s state.

**Actions and Action Creators**

Actions provide the instructions and data that the store will use to modify the state. *Action creators* are functions that can be used to abstract away the nitty-gritty details required to build an action. Actions themselves are objects that at minimum contain a type field. The action type is typically an uppercase string that describes the action. Additionally, actions may package any data required by the store. For example:

**const** countdownActions = dispatcher =>

({

tick(currentCount) {

dispatcher.handleAction(**{ type: 'TICK' }**)

},

reset(count) {

dispatcher.handleAction(**{**

**type: 'RESET',**

**count**

**}**)

}

})

When countdown action creators are loaded, the dispatcher is sent as an argument. Every time a TICK or a RESET is invoked, the dispatcher’s handleAction method is invoked, which “dispatches” the action object.

**Dispatcher**

There is only ever one dispatcher, and it represents the air traffic control part of this design pattern. The dispatcher takes the action, packages it with some information about where the action was generated, and sends it on to the appropriate store or stores that will handle the action.

Although Flux is not a framework, Facebook does open source a Dispatcher class that you can use. How dispatchers are implemented is typically standard, so it is better to use Facebook’s dispatcher rather than coding your own:

**import** Dispatcher from 'flux'

**class** CountdownDispatcher **extends** Dispatcher {

handleAction(action) {

console.log('dispatching action:', action)

**this**.dispatch({

source: 'VIEW\_ACTION',

action

})

}

}

When handleAction is invoked with an action, it is dispatched along with some data about where the action originated. When a store is created, it is registered with the dispatcher and starts listening for actions. When an action is dispatched it is handled in the order that it was received and sent to the appropriate stores.

**Stores**

Stores are objects that hold the application’s logic and state data. Stores are similar to models in the MVC pattern, but stores are not restricted to managing data in a single object. It is possible to build Flux applications that consist of a single store that manages many different data types.

Current state data can be obtained from a store via properties. Everything a store needs to change state data is provided in the action. A store will handle actions by type and change their data accordingly. Once data is changed, the store will emit an event and notify any views that have subscribed to the store that their data has changed. Let’s take a look at an example:

**import** { EventEmitter } from 'events'

**class** CountdownStore **extends** EventEmitter {

constructor(count=5, dispatcher) {

**super**()

**this**.\_count = count

**this**.dispatcherIndex = dispatcher.register(

**this**.dispatch.bind(**this**)

)

}

get count() {

**return** **this**.\_count

}

dispatch(payload) {

**const** { type, count } = payload.action

**switch**(type) {

**case** "TICK":

**this**.\_count = **this**.\_count - 1

**this**.emit("TICK", **this**.\_count)

**return** **true**

**case** "RESET":

**this**.\_count = count

**this**.emit("RESET", **this**.\_count)

**return** **true**

}

}

}

This store holds the countdown application’s state, the count. The count can be accessed through a read-only property. When actions are dispatched, the store uses them to change the count. A TICK action decrements the count. A RESET action resets the count entirely with data that is included with the action.

Once the state has changed, the store emits an event to any views that may be listening.

**Putting It All Together**

Now that you understand how data flows through each part of a Flux application, let’s take a look at how all these parts get connected:

**const** appDispatcher = **new** CountdownDispatcher()

**const** actions = countdownActions(appDispatcher)

**const** store = **new** CountdownStore(10, appDispatcher)

**const** render = count => ReactDOM.render(

<Countdown count={count} {...actions} />,

document.getElementById('react-container')

)

store.on("TICK", () => render(store.count))

store.on("RESET", () => render(store.count))

render(store.count)

First, we create the appDispatcher. Next, we use the appDispatcher to generate our action creators. Finally, the appDispatcher is registered with our store, and the store sets the initial count to 10.

The render method is used to render the view with a count that it receives as an argument. It also passes the action creators to the view as properties.

Finally, some listeners are added to the store, which completes the circle. When the store emits a TICK or a RESET, it yields a new count, which is immediately rendered in the view. After that, the initial view is rendered with the store’s count. Every time the view emits a TICK or RESET, the action is sent through this circle and eventually comes back to the view as data that is ready to be rendered.

**Flux Implementations**

There are different approaches to the implementation of Flux. A few libraries have been open-sourced based upon specific implementations of this design pattern. Here are a few approaches to Flux worth mentioning:

[Flux](https://facebook.github.io/flux/)

Facebook’s Flux is the design pattern that we just covered. The Flux library includes an implementation of a dispatcher.

[Reflux](https://github.com/reflux/refluxjs)

A simplified approach to unidirectional data flow that focuses on actions, stores, and views.

[Flummox](http://acdlite.github.io/flummox)

A Flux implementation that allows you to build Flux modules through extending JavaScript classes.

[Fluxible](http://fluxible.io/)

A Flux framework created by Yahoo for working with isomorphic Flux applications. Isomorphic applications will be discussed in [Chapter 12](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch12.html#react_and_the_server).

[Redux](http://redux.js.org/)

A Flux-like library that achieves modularity through functions instead of objects.

[MobX](https://mobx.js.org/getting-started.html)

A state management library that uses observables to respond to changes in state.

All of these implementations have stores, actions, and a dispatch mechanism, and favor React components as the view layer. They are all variations of the Flux design pattern, which at its core is all about unidirectional data flow.

Redux has quickly become one of the more popular Flux frameworks. The next chapter covers how to use Redux to construct functional data architectures for your client applications.

* [**Copy**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html)
* [**Add Highlight**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html)
* [**Add Note**](https://www.safaribooksonline.com/library/view/learning-react-1st/9781491954614/ch07.html)