O'REILLY®

Second Edition

Early Release

RAW & UNEDITED

React Up & Running

Building Web Applications

Stoyan Stefanov

React: Up & Running

Building Web Applications

SECOND EDITION

Stoyan Stefanov

React: Up & Running

by Stoyan Stefanov

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Jennifer Pollock

Development Editor: Angela Rufino

Production Editor: Kate Galloway

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

July 2021: Second Edition

Revision History for the Early Release

- 2020-04-23: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492051466 for release details.

978-1-492-05139-8

To Eva, Zlatina, and Nathalie

Preface

It’s yet another wonderful warm California night. The faint ocean breeze only helping you feel 100% “aaah!” The place: Los Angeles; the time: 2000-something. I was just getting ready to FTP my new little web app called CSSsprites.com to my server and release it to the world. I contemplated a problem on the last few evenings I spent working on the app: why on earth did it take 20% effort to wrap up the “meat” of the app and then 80% to wrestle with the user interface? How many other tools could I have made if I didn’t have to `getElementById()` all the time and worry about the state of the app? (Is the user done uploading? What, an error? Is this dialog still on?) Why is UI development so time consuming? And what’s up with all the different browsers? Slowly, the “aaah” was turning into “aarrggh!”

Fast forward to March 2015 at Facebook’s F8 conference. The team I’m part of is ready to announce a complete rewrite of two web apps: our third-party comments offering and a moderation tool to go with it. Compared to my little CSSsprites.com app, these were fully fledged web apps with tons more features, way more power, and insane amounts of traffic. Yet, the development was a joy. Teammates new to the app (and some even new to JavaScript and CSS) were able to come and contribute a feature here and an improvement there, picking up speed quickly and effortlessly. As one member of the team said, “Ah-ha, now I see what all the love is all about!”

What happened along the way? React.

React is a library for building UIs—it helps you define the UI once and for all. Then, when the state of the app changes, the UI is rebuilt to *react* to the change and you don't need to do anything extra. After all, you've defined the UI already. Defined? More like *declared*. You use small manageable *components* to build a large powerful app. No more spending half of your function's body hunting for DOM nodes; all you do is maintain the `state` of your app (with a regular old JavaScript object) and the rest just follows.

Learning React is a sweet deal—you learn one library and use it to create all of the following:

- Web apps
- Native iOS and Android apps
- Canvas apps
- TV apps
- Native desktop apps

You can create native apps with native performance and native controls (*real* native controls, not native-looking copies) using the same ideas of building components and UIs. It's not about "write once, run everywhere" (our industry keeps failing at this), it's about "learn once, use everywhere."

To cut a long story short: learn React, take 80% of your time back, and focus on the stuff that matters (like the real reason your app exists).

About This Book

This book focuses on learning React from a web development point of view. For the first three chapters, you start with nothing but a blank HTML file and keep building up from there. This allows you to focus on learning React and not any of the new syntax or auxiliary tools.

Chapter 4 introduces JSX, which is a separate and optional technology that is usually used in conjunction with React.

From there you learn about what it takes to develop a real-life app and the additional tools that can help you along the way. Examples include JavaScript packaging tools (Browserify), unit testing (Jest), linting (ESLint), types (Flow), organizing data flow in the app (Flux), and immutable data (Immutable.js). All of the discussions about auxiliary technologies are kept to a minimum so that the focus is still on React; you'll become familiar with these tools and be able to make an informed decision about which to use.

Good luck on your journey toward learning React—may it be a smooth and fruitful one!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/stoyan/reactbook*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*React: Up & Running* by Stoyan Stefanov (O'Reilly). Copyright 2016 Stoyan Stefanov, 978-1-491-93182-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/react_up_running*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I’d like to thank to everyone who read different drafts of this book and sent feedback and corrections: Andreea Manole, Iliyan Peychev, Kostadin Ilov, Mark Duppenthaler, Stephan Alber, Asen Bozhilov.

Thanks to all the folks at Facebook who work on (or with) React and answer my questions day in and day out. Also to the extended React community that keeps producing great tools, libraries, articles, and usage patterns.

Many thanks to Jordan Walke.

Thanks to everyone at O’Reilly who made this book possible: Meg Foley, Kim Cofer, Nicole Shelby, and many others.

Thanks to Yavor Vatchkov who designed the UI of the example app developed in this book (try it at *whinepad.com*).

Chapter 1. Hello World

Let's get started on the journey to mastering application development using React. In this chapter, you will learn how to set up React and write your first "Hello World" web app.

Setup

First things first: you need to get a copy of the React library. There are various ways to go about it. Let's go with the simplest one that doesn't require any special tools and can get you learning and hacking away in no time.

Create a folder for all the code in the book in a location where you'll be able to find it.

For example:

```
mkdir ~/reactbook
```

Create a `react` folder to keep the React library code separate.

```
mkdir ~/reactbook/react
```

Next, you need to add two files: one is React itself, the other is the ReactDOM add-on. You can grab the latest 16.* versions of the two from the unpkg.com host, like so:

```
curl -L https://unpkg.com/react@16/umd/react.development.js >
~/reactbook/react/react.js
curl -L https://unpkg.com/react-dom@16/umd/react-dom.development.js >
~/reactbook/react/react-dom.js
```

Note that React doesn't impose any directory structure; you're free to move to a different directory or rename *react.js* however you see fit.

You don't have to download the libraries, you can use them directly from unpkg.com but having them locally makes it possible to learn anywhere and without an internet connection.

NOTE

The `@16` in the URLs above gets you a copy of the latest React 16, which is current at the time of writing this book. Omit `@16` to get the latest available React version. Alternatively, you can explicitly specify the version you require, for example `@16.13.0`.

Hello React World

Let's start with a simple page in your working directory (*~/reactbook/01.01.hello.html*):

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello React</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="app">
      <!-- my app renders here -->
    </div>
    <script src="react/react.js"></script>
    <script src="react/react-dom.js"></script>
    <script>
      // my app's code
    </script>
  </body>
</html>
```

NOTE

You can find all the code from this book in the accompanying repository.

Only two notable things are happening in this file:

- You include the React library and its DOM add-on (via `<script src>` tags)
- You define where your application should be placed on the page (`<div id="app">`)

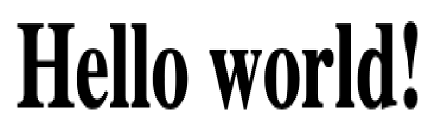
NOTE

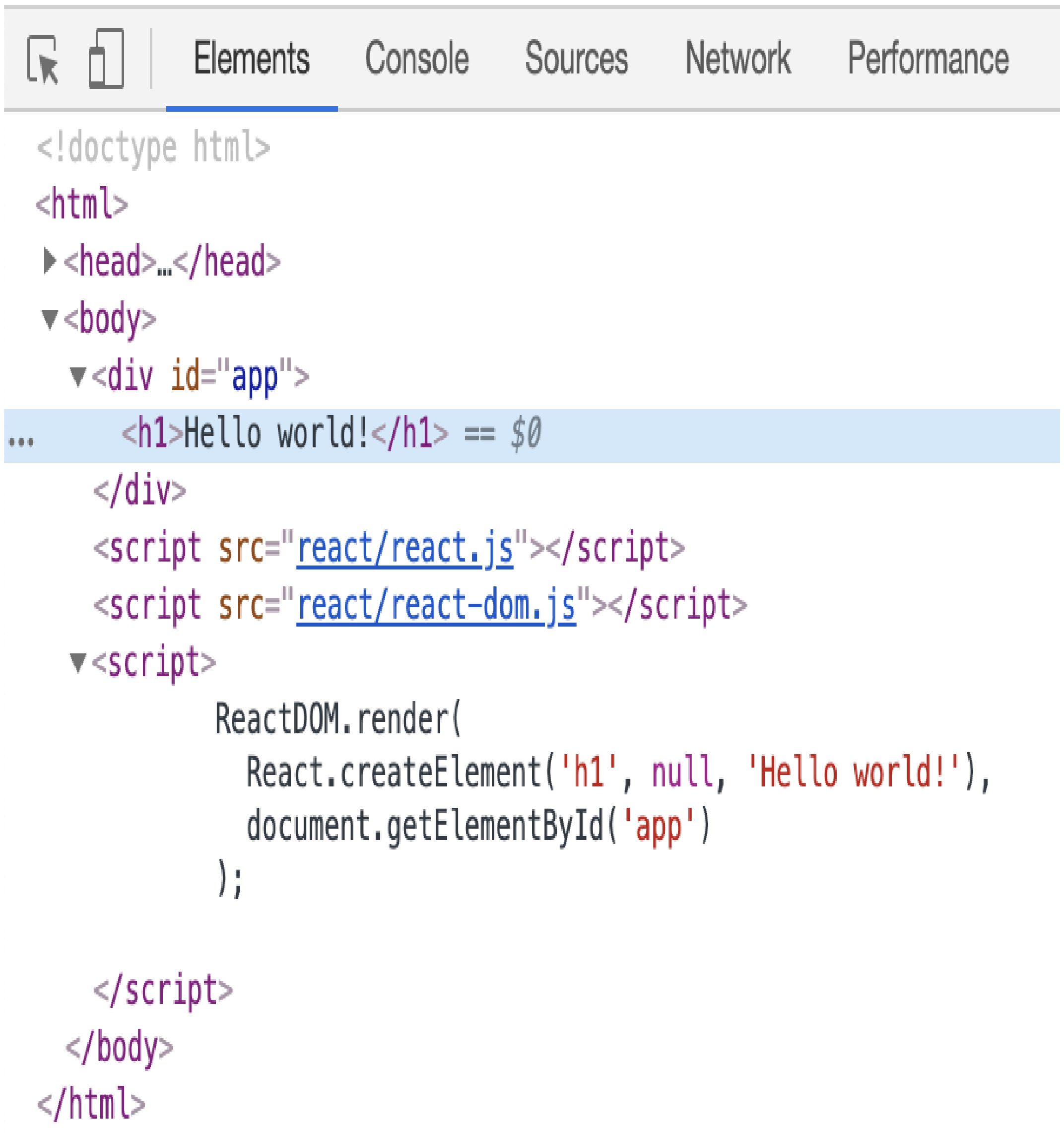
You can always mix regular HTML content as well as other JavaScript libraries with a React app. You can also have several React apps on the same page. All you need is a place in the DOM where you can point React to and say “do your magic right here.”

Now let’s add the code that says “hello” - update *01.01.hello.html* and replace `// my app's code` with:

```
ReactDOM.render(
  React.createElement('h1', null, 'Hello world!'),
  document.getElementById('app')
);
```

Load *01.01.hello.html* in your browser and you’ll see your new app in action (Figure 1-1).





```
<!doctype html>
<html>
 ▶<head>…</head>
 ▼<body>
   ▼<div id="app">
...     <h1>Hello world!</h1> == $0
    </div>
    <script src="react/react.js"></script>
    <script src="react/react-dom.js"></script>
   ▼<script>
            ReactDOM.render(
              React.createElement('h1', null, 'Hello world!'),
              document.getElementById('app')
            );

    </script>
  </body>
</html>
```

Figure 1-1. Hello World in action

Congratulations, you’ve just built your first React application!

Figure 1-1 also shows the *generated* code in Chrome Developer Tools where you can see that the contents of the `<div id="app">` placeholder was replaced with the contents generated by your React app.

What Just Happened?

There are a few things of interest in the code that made your first app work.

First, you see the use of the `React` object. All of the APIs available to you are accessible via this object. The API is intentionally minimal, so there are not a lot of method names to remember.

You can also see the `ReactDOM` object. It has only a handful of methods, `render()` being the most useful. `ReactDOM` is responsible for rendering the app *in the browser*. You can, in fact, create React apps and render them in different environments outside the browser —for example in canvas, or natively in Android or iOS.

Next, there is the concept of *components*. You build your UI using components and you combine these components in any way you see fit. In your applications, you’ll end up creating your custom components, but to get you off the ground, React provides wrappers around HTML DOM elements. You use the wrappers via the `React.createElement` function. In this first example, you can see the use of the `h1` element. It corresponds to the `<h1>` in HTML and is available to you using a call to `React.createElement('h1')`.

Finally, you see the good old `document.getElementById('app')` DOM access. You use this to tell React where the application should be located on the page. This is the bridge crossing over from the DOM manipulation as you know it to React-land.

Once you cross the bridge from DOM to React, you don't have to worry about DOM manipulation anymore, because React does the translation from components to the underlying platform (browser DOM, canvas, native app). In fact, not worrying about the DOM is one of the great things about React. You worry about composing the components and their data—the meat of the application—and let React take care of updating the DOM most efficiently. No more hunting for DOM nodes, `firstChild`, `appendChild()` and so on.

NOTE

You *don't have to* worry about DOM, but that doesn't mean you cannot. React gives you "escape latches" if you want to go back to DOM-land for any reason you may need.

Now that you know what each line does, let's take a look at the big picture. What happened is this: you rendered one React component in a DOM location of your choice. You always render one top-level component and it can have as many children (and grandchildren, etc.) components as you need. Even in this simple example, the `h1` component has a child—the "Hello World!" text.

React.createElement()

As you know now, you can use a number of HTML elements as React components via the `React.createElement()` method. Let’s take a close look at this API.

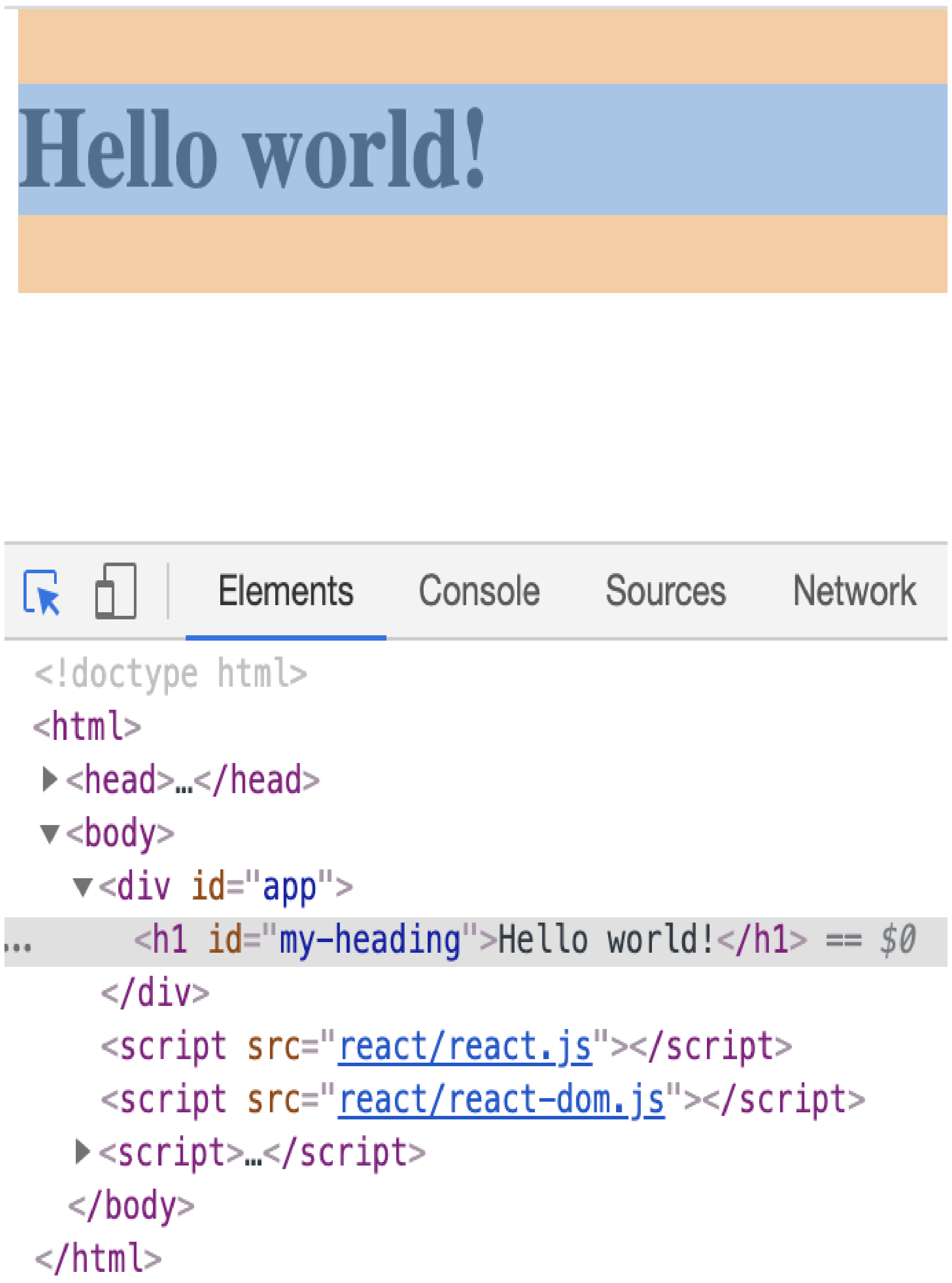
Remember the “Hello World!” app looks like this:

```
ReactDOM.render(
  React.createElement('h1', null, 'Hello world!'),
  document.getElementById('app')
);
```

The first parameter to `createElement` is the type of element to be created. The second (which is `null` in this case) is an object that specifies any properties (think DOM attributes) that you want to pass to your element. For example, you can do:

```
React.createElement(
  'h1',
  {
    id: 'my-heading',
  },
  'Hello world!'
),
```

The HTML generated by this example is shown in Figure 1-2.



```
<!doctype html>
<html>
 ▶<head>…</head>
 ▼<body>
   ▼<div id="app">
      <h1 id="my-heading">Hello world!</h1> == $0
    </div>
    <script src="react/react.js"></script>
    <script src="react/react-dom.js"></script>
   ▶<script>…</script>
  </body>
</html>
```

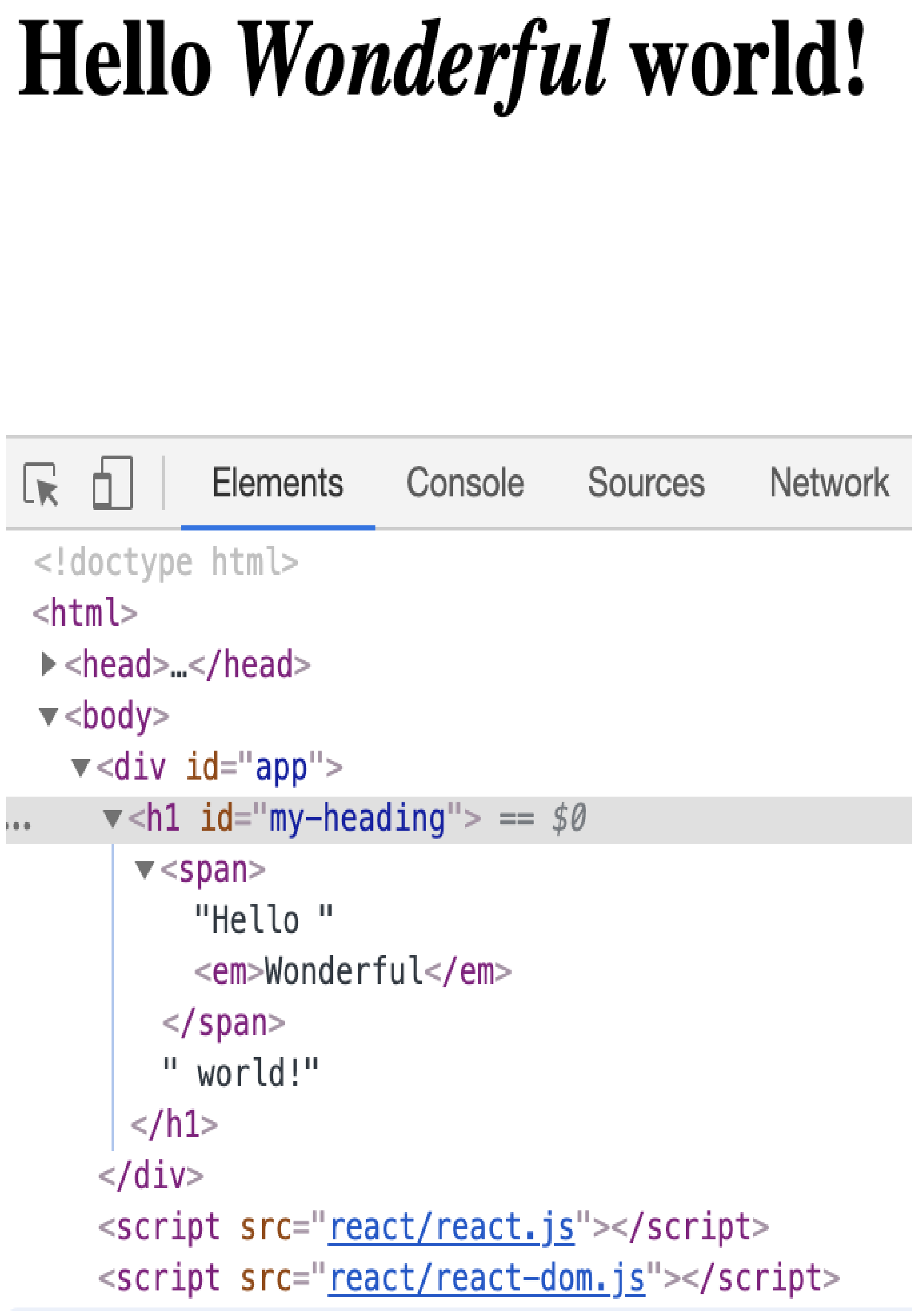
Figure 1-2. HTML generated by a `React.createElement()` call

The third parameter (`"Hello World!"` in this example) defines a child of the component. The simplest case is just a text child (a `Text` node in DOM-speak) as you see in the preceding code. But you can have as many nested children as you like and you pass them as additional parameters. For example:

```
React.createElement(
  'h1',
  {id: 'my-heading'},
  React.createElement('span', null, 'Hello'),
  ' world!'
),
```

Another example, this time with nested components (result shown in Figure 1-3) is as follows:

```
React.createElement(
  'h1',
  {id: 'my-heading'},
  React.createElement(
    'span',
    null,
    'Hello ',
    React.createElement('em', null, 'Wonderful'),
  ),
  ' world!'
),
```



```
<!doctype html>
<html>
 ▶<head>…</head>
 ▼<body>
   ▼<div id="app">
...  ▼<h1 id="my-heading"> == $0
       ▼<span>
          "Hello "
          <em>Wonderful</em>
        </span>
        " world!"
      </h1>
    </div>
    <script src="react/react.js"></script>
    <script src="react/react-dom.js"></script>
```

Figure 1-3. HTML generated by nesting `React.createElement()` calls

You can see in Figure 1-3 that the DOM generated by React has the `<em>` element as a child of the `<span>` which is in turn a child of the `<h1>` element (and a sibling of the “world” text node).

JSX

When you start nesting components, you quickly end up with a lot of function calls and parentheses to keep track of. To make things easier, you can use the *JSX syntax*. JSX is a little controversial: people often find it repulsive at first sight (ugh, XML in my JavaScript!), but indispensable after.

Here’s the previous snippet but this time using JSX syntax:

```
ReactDOM.render(
  <h1 id="my-heading">
    <span>Hello <em>Wonderful</em></span> world!
  </h1>,
  document.getElementById('app')
);
```

This is much more readable. This syntax looks very much like HTML and you already know HTML. However it’s not valid JavaScript that browsers can understand. You need to *transpile* this code to make it work in the browser. Again, for learning purposes, you can do this without special tools. You need the Babel library which translates cutting-edge JavaScript (and JSX) to old school JavaScript that works in ancient browsers.

Setup Babel

Just like with React, get a local copy of Babel:

```
curl -L https://unpkg.com/babel-standalone/babel.min.js >
~/reactbook/react/babel.js
```

Then you need to update your learning template to include Babel. Create a file *01.04.hellojsx.html* like so:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello React+JSX</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="app">
      <!-- my app renders here -->
    </div>
    <script src="react/react.js"></script>
    <script src="react/react-dom.js"></script>
    <script src="react/babel.js"></script>
    <script type="text/babel">
      // my app's code
    </script>
  </body>
</html>
```

NOTE

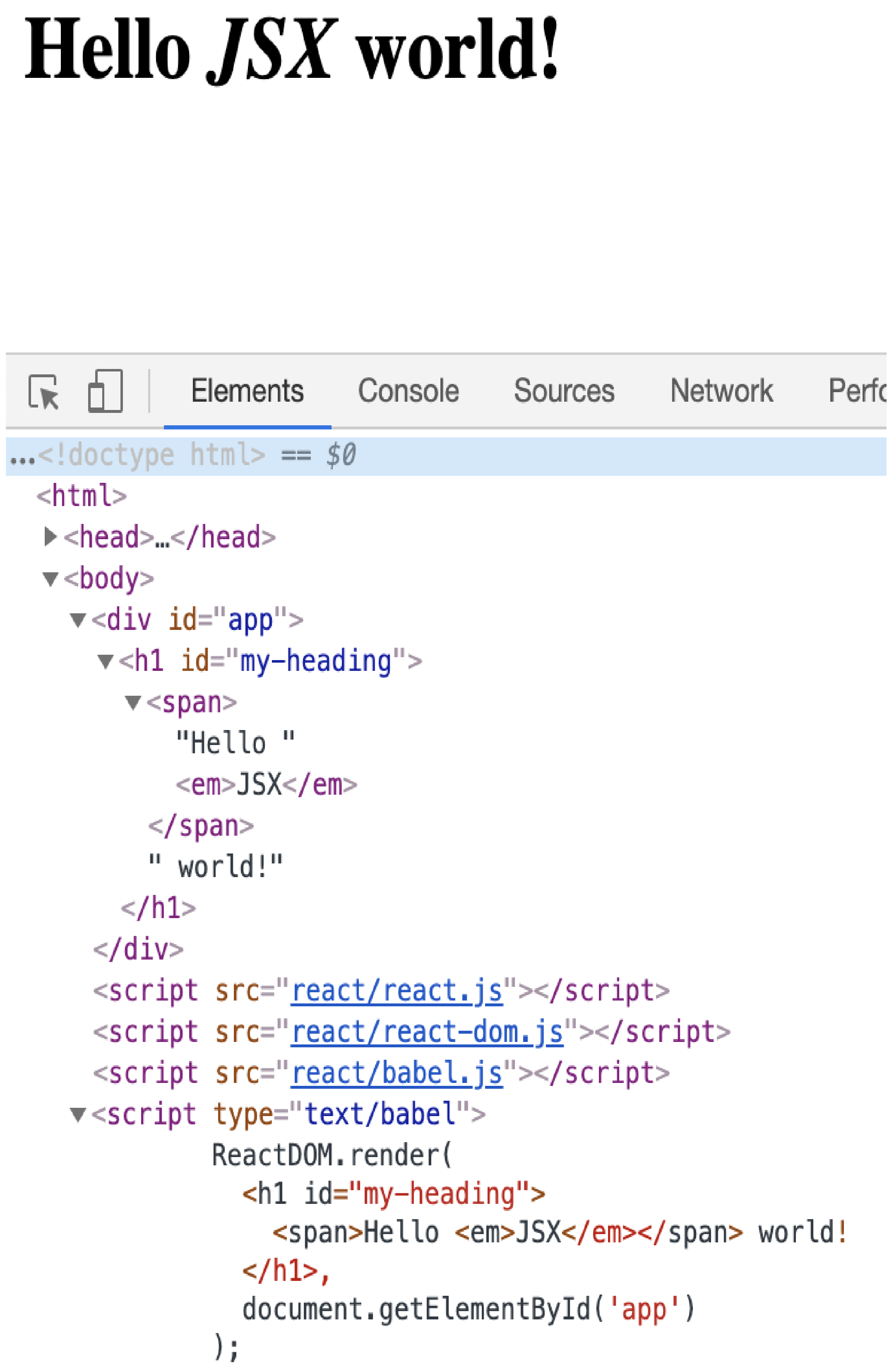
Note how `<script>` becomes `<script type="text/babel">`. This is a trick where by specifying an invalid `type`, the browser ignores the code. This gives Babel a chance to parse and transform the JSX syntax into something the browser can run.

Hello JSX world

With this bit of setup out of the way, let's try JSX. Replace the `// my app's code` part in the HTML above with:

```
ReactDOM.render(
  <h1 id="my-heading">
    <span>Hello <em>JSX</em></span> world!
  </h1>,
  document.getElementById('app')
);
```

The result of running this in the browser is shown on Figure 1-4.



Hello *JSX* world!

```
…<!doctype html> == $0
  <html>
   ▶<head>…</head>
   ▼<body>
     ▼<div id="app">
       ▼<h1 id="my-heading">
         ▼<span>
            "Hello "
            <em>JSX</em>
          </span>
          " world!"
        </h1>
      </div>
      <script src="react/react.js"></script>
      <script src="react/react-dom.js"></script>
      <script src="react/babel.js"></script>
     ▼<script type="text/babel">
              ReactDOM.render(
                <h1 id="my-heading">
                  <span>Hello <em>JSX</em></span> world!
                </h1>,
                document.getElementById('app')
              );
```

Figure 1-4. Hello JSX world

What just happened?

It’s great that you got the JSX and Babel to work, but maybe a few more words won’t hurt, especially if you’re new to Babel and the process of transpilation. If you’re already familiar, feel free to skip this part where we familiarize a bit with the terms *JSX*, *Babel*, and *transpilation*.

JSX is a separate technology from React and is completely optional. As you see, the first examples in this chapter didn’t even use JSX. You can opt into never coming anywhere near JSX at all. But it’s very likely that once you try it, you won’t go back to function calls.

NOTE

It’s not quite clear what the acronym JSX stands for, but it’s most likely JavaScriptXML or JavaScript Syntax eXtension. The official home of the open-source project is *http://facebook.github.io/jsx/*.

The process of *transpilation* is a process of taking source code and rewriting it to accomplish the same results but using syntax that’s understood by older browsers. It’s different than using *polyfills*. An example of a polyfill is adding a method to `Array.prototype` such as `map()`, which was introduced in ECMAScript5, and making it work in browsers that only support ECMAScript3. A polyfill is a solution in pure JavaScript-land. It’s a good solution when adding new methods to existing objects or implementing new objects (such as `JSON`). But it's not sufficient when new syntax is introduced into the language. Any new syntax in the eyes of browser that does not support it is just invalid and throws a parse error. There's no way to polyfill it. New syntax, therefore, requires a compilation (transpilation) step so it's transformed *before* it's served to the browser.

Transpiling JavaScript is getting more and more common as programmers want to use the latest JavaScript (ECMAScript) features without waiting for browsers to implement them. If you already have a build process set up (that does e.g., minification or any other code transformation), you can simply add the JSX step to it. Assuming you *don't* have a build process, you'll see later in the book the necessary steps of setting one up.

For now, let's leave the JSX transpilation on the client-side (in the browser) and move on with learning React. Just be aware that this is only for education and experimentation purposes. Client-side transforms are not meant for live production sites as they are slower and more resource intensive that serving already transpiled code.

Next: Custom Components

At this point, you're done with the bare-bones "Hello World" app. Now you know how to:

- Set up the React library for experimentation and learning (it's really just a question of a few `<script>` tags)

- Render a React component in a DOM location of your choice (e.g., `ReactDOM.render(reactWhat, domWhere)`)
- Use built-in components, which are wrappers around regular DOM elements (e.g., `React.createElement(element, attributes, content, children)`)

The real power of React, though, comes when you start using custom components to build (and update!) the user interface (UI) of your app. Let’s learn how to do just that in the next chapter.

Chapter 2. The Life of a Component

Now that you know how to use the ready-made DOM components, it's time to learn how to make some of your own.

There are two ways to define a custom component, both accomplishing the same result but using different syntax:

- Using a function (components created this way are referred to as *functional components*)
- Using a class that extends `React.Component` (commonly referred to as *class components*)

A Custom Functional Component

Here's an example of a functional component:

```
const MyComponent = function() {
  return 'I am so custom';
};
```

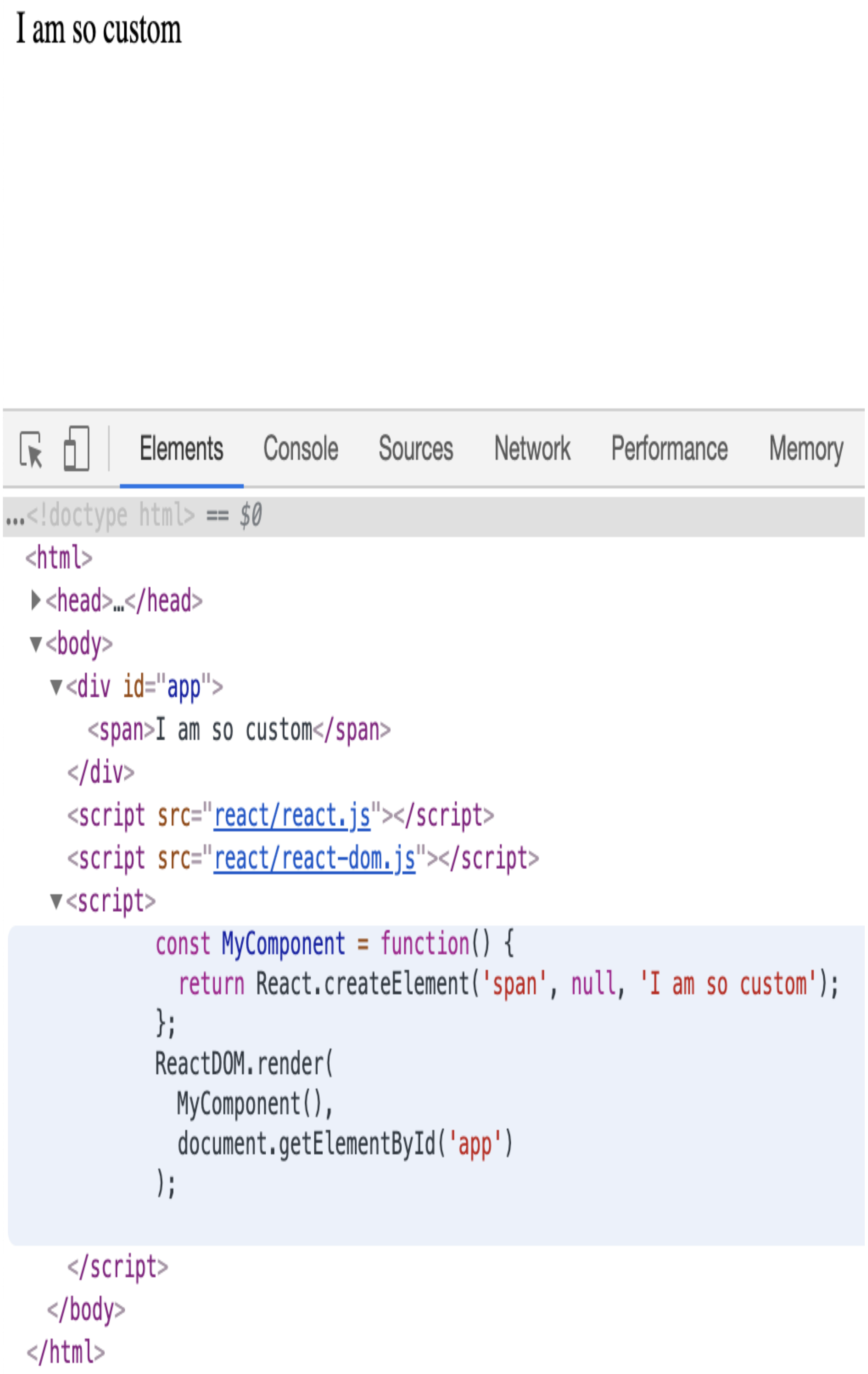
But wait, this is just a function! Yes, this is it, the custom component is just a function that returns the UI that you want. In this case, the UI is only text but often you'll need a little bit more, most likely a composition of other components. Here's an example of using a `span` to wrap the text:

```
const MyComponent = function() {
  return React.createElement('span', null, 'I am so custom');
};
```

Using your new shiny component in an application is similar to using the DOM components from Chapter 1, except you *call* the function that defines the component:

```
ReactDOM.render(
  MyComponent(),
  document.getElementById('app')
);
```

The result of rendering your custom component is shown in Figure 2-1.



```
I am so custom
```

```
...<!doctype html> == $0
<html>
 ▶<head>…</head>
 ▼<body>
   ▼<div id="app">
      <span>I am so custom</span>
    </div>
    <script src="react/react.js"></script>
    <script src="react/react-dom.js"></script>
   ▼<script>
            const MyComponent = function() {
              return React.createElement('span', null, 'I am so custom');
            };
            ReactDOM.render(
              MyComponent(),
              document.getElementById('app')
            );

    </script>
  </body>
</html>
```

Figure 2-1. Your first custom component (`02.01.custom-functional.html` in the book's repository)

A JSX Version

The same example using JSX would look a little easier to read. Defining the component looks like this:

```
const MyComponent = function() {
  return <span>I am so custom</span>;
};
```

Using the component the JSX way looks like the following, regardless of how the component itself was defined (with JSX or not).

```
ReactDOM.render(
  <MyComponent />,
  document.getElementById('app')
);
```

> **NOTE**
>
> Notice that in the self-closing tag `<MyComponent />` the slash is not optional. That applies to HTML elements used in JSX too. `<br>` and `<img>` are not going to work, you need to close them like `<br/>` and `<img/>`.

A Custom Class Component

The second way to create a component is to define a class that extends `React.Component` and implements a `render()` function:

```
class MyComponent extends React.Component {
  render() {
    return React.createElement('span', null, 'I am so custom');
    // or with JSX:
    // return <span>I am so custom</span>;
  }
}
```

Rendering the component on the page:

```
ReactDOM.render(
  React.createElement(MyComponent),
  document.getElementById('app')
);
```

If you use JSX, you don’t need to know how the component was defined (using a class or a function), in both cases using the component is the same:

```
ReactDOM.render(
  <MyComponent />,
  document.getElementById('app')
);
```

Which Syntax to Use?

You may be wondering: with all these options (JSX vs. pure JavaScript, a class component vs. a functional one), which one to use? JSX is the most common. And, unless you dislike the XML syntax in your JavaScript, the path of least resistance and of less typing is to go with JSX. This book uses JSX from now on, unless to illustrate a concept. Why then even talk about a no-JSX way? Well, you should know that there *is* another way and also that JSX is not some special voodoo but rather a thin syntax layer that transforms

XML into plain JavaScript function calls such as `React.createElement()` before sending the code to the browser.

What about *class* vs *functional* components? This is a question of preference. If you’re comfortable with object-oriented programming (OOP) and you like how classes are laid out, then by all means, go for it. Functional components are a little less typing usually, they feel more native to JavaScript (classes in JavaScript were an afterthought and merely syntax sugar) and a little lighter on the computer’s CPU. Historically, functional components were not able to accomplish everything that classes could. Until the invention of *hooks*, which we’ll get to in due time. This book teaches you both ways and doesn’t decide for you. OK, maybe there’s a slight preference towards functional components.

Properties

Rendering *hard-coded* UI in your custom components is perfectly fine and has its uses. But the components can also take *properties* and render or behave differently, depending on the values of the properties. Think about the `<a>` element in HTML and how it acts differently based on the value of the `href` attribute. The idea of properties in React is similar (and so is the JSX syntax).

In class components all properties are available via the `this.props` object. Let’s see an example:

```
class MyComponent extends React.Component {
  render() {
    return <span>My name is <em>{this.props.name}</em></span>;
  }
}
```

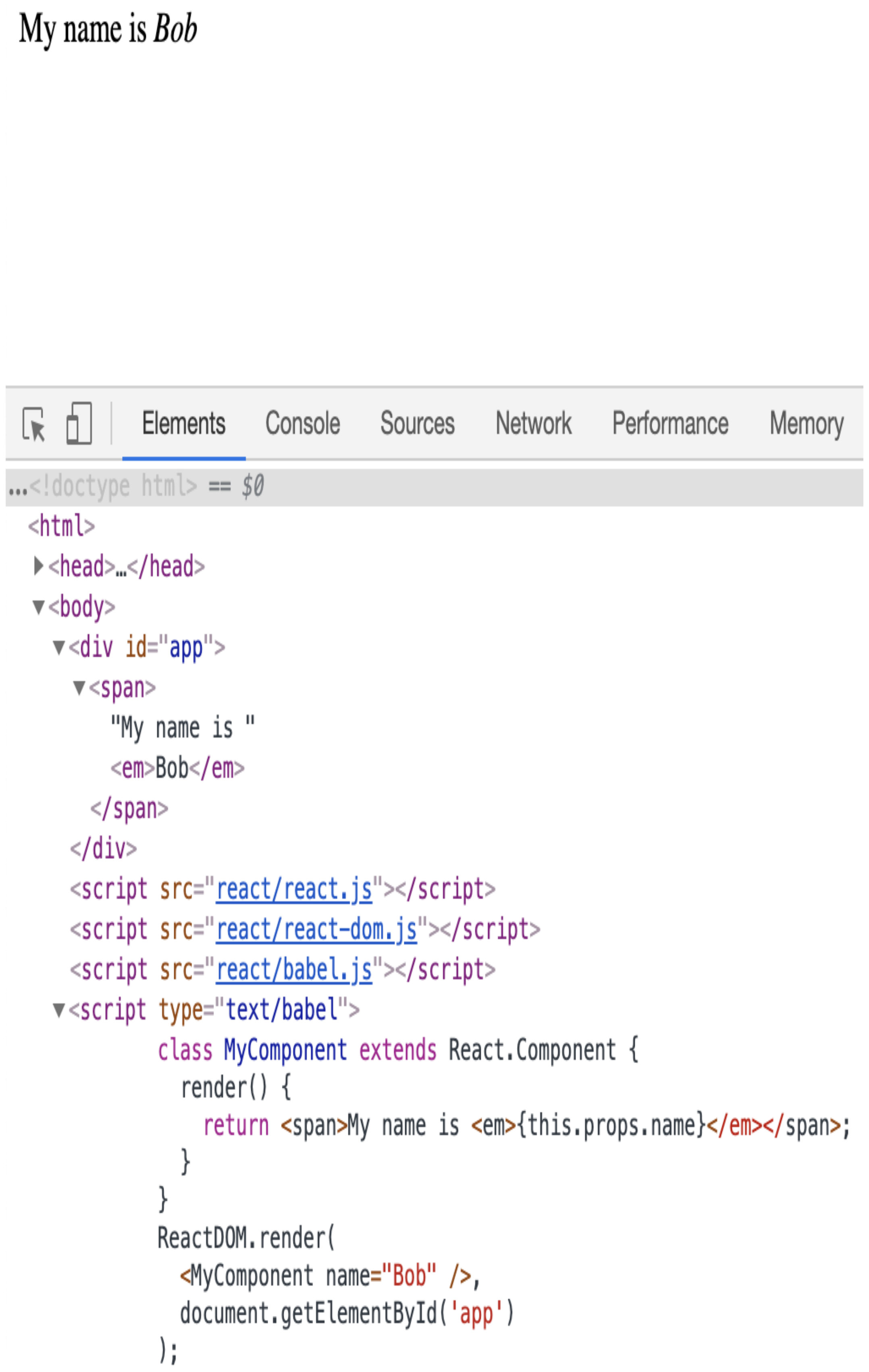
NOTE

As demonstrated in this example, you can open curly braces and sprinkle JavaScript values (and expressions too) within your JSX. You’ll learn more about this behavior as you progress with the book.

Passing a value for the `name` property when rendering the component looks like this:

```
ReactDOM.render(
  <MyComponent name="Bob" />,
  document.getElementById('app')
);
```

The result is shown in Figure 2-2.



My name is *Bob*

```
...<!doctype html> == $0
  <html>
   ▶<head>…</head>
   ▼<body>
     ▼<div id="app">
       ▼<span>
          "My name is "
          <em>Bob</em>
        </span>
      </div>
      <script src="react/react.js"></script>
      <script src="react/react-dom.js"></script>
      <script src="react/babel.js"></script>
     ▼<script type="text/babel">
              class MyComponent extends React.Component {
                render() {
                  return <span>My name is <em>{this.props.name}</em></span>;
                }
              }
              ReactDOM.render(
                <MyComponent name="Bob" />,
                document.getElementById('app')
              );
```

Figure 2-2. Using component properties (`02.05.this.props.html`)

It's important to remember that `this.props` is read-only. It's meant to carry on configuration from parent components to children, it's not a general-purpose storage of values. If you feel tempted to set a property of `this.props`, just use additional local variables or properties of your component's class instead (meaning use `this.thing` as opposed to `this.props.thing`).

Properties in Functional Components

In functional components, there's no `this` (in JavaScript's *strict* mode) or it refers to the global object (in non-strict, dare we say *sloppy*, mode). So instead of `this.props`, you get a `props` object passed to your function as the first argument.

```
const MyComponent = function(props) {
  return <span>My name is <em>{props.name}</em></span>;
};
```

A common pattern is to use JavaScript's *destructuring assignment* and assign the property values to local variables. In other words the example above becomes:

```
// 02.07.props.destructuring.html
const MyComponent = function({name}) {
  return <span>My name is <em>{name}</em></span>;
};
```

You can have as many properties as you want. If, for example, you need two properties `name` and `job` you can use them like:

```
// 02.08.props.destruct.multi.html
const MyComponent = function({name, job}) {
  return <span>My name is <em>{name}</em>, the {job}</span>;
};
ReactDOM.render(
  <MyComponent name="Bob" job="engineer"/>,
  document.getElementById('app')
);
```

Default Properties

Your component may offer a number of properties, but sometimes a few of the properties may have default values that work well for the most common cases. You can specify default property values using `defaultProps` property for both functional and class components.

Functional:

```
const MyComponent = function({name, job}) {
  return <span>My name is <em>{name}</em>, the {job}</span>;
};
MyComponent.defaultProps = {
  job: 'engineer',
};
ReactDOM.render(
  <MyComponent name="Bob" />,
  document.getElementById('app')
);
```

Class components:

```
class MyComponent extends React.Component {
  render() {
    return (
      <span>My name is <em>{this.props.name}</em>,
      the {this.props.job}</span>
    );
  }
}
MyComponent.defaultProps = {
  job: 'engineer',
};
ReactDOM.render(
  <MyComponent name="Bob" />,
  document.getElementById('app')
);
```

In both cases, the result is the output: “My name is *Bob*, the engineer”

TIP

Notice how the `render()` method’s `return` statement wraps the returned value in parentheses. This is just because of JavaScript’s *automatic semi-colon insertion* (ASI) mechanism. A `return` statement followed by a new line is the same as `return;` which is the same as `return undefined;` which is the definitely not what you want. Wrapping the returned expression in parentheses allows for better code formatting while retaining the correctness.

State

The examples so far were pretty static (or “stateless”). The goal was just to give you an idea of the building blocks of composing your UI. But where React really shines (and where old-school browser DOM manipulation and maintenance gets complicated) is when the data in your application changes. React has the concept of *state*, which is any data that components want to use to render themselves. When state changes, React rebuilds the UI without you having to do anything. After you build your UI initially in your `render()` method (or in the rendering function in case of a functional component) all you care about is updating the data. You don’t need to worry about UI changes at all. After all, your render method/function has already provided the blueprint of what the component should look like.

NOTE

“Stateless” is not a bad word, not at all. Stateless components are much easier to manage and think about. In fact, whenever you can, prefer to go stateless. But applications are complicated and you do need state. So let’s proceed.

Similarly to how you access properties via `this.props`, you *read* the state via the object `this.state`. To *update* the state, you use `this.setState()`. When `this.setState()` is called, React calls the render method of your component (and all of its children) and updates the UI.

The updates to the UI after calling `this.setState()` are done using a queuing mechanism that efficiently batches changes. Updating `this.state` directly can have unexpected behavior and you shouldn’t do it. Just like with `this.props`, consider the `this.state` object read-only, not only because it’s semantically a bad idea, but because it can act in ways you don’t expect. Similarly, don’t ever call `this.render()` yourself—instead, leave it to React to batch changes, figure out the least amount of work, and call `render()` when and if appropriate.

A Textarea Component

Let’s build a new component—a textarea that keeps count of the number of characters typed in (Figure 2-3).

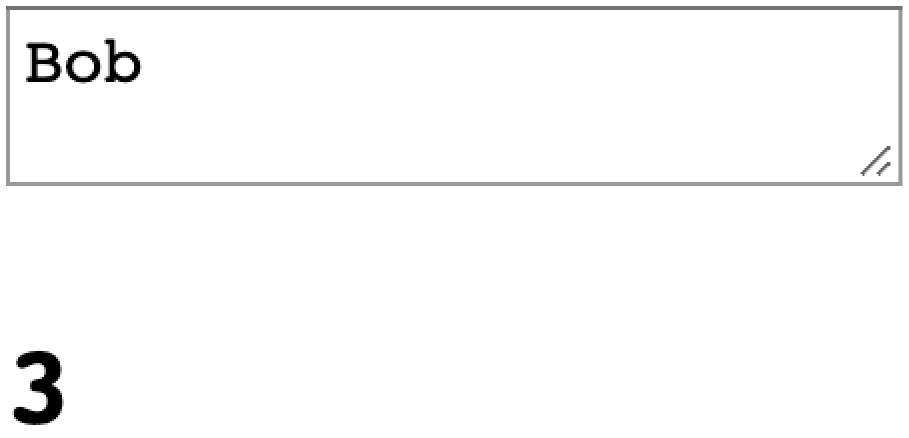


Figure 2-3. The end result of the custom textarea component

You (as well as other future consumers of this amazingly reusable component) can use the new component like so:

```
ReactDOM.render(
  <TextAreaCounter text="Bob" />,
  document.getElementById('app')
);
```

Now, let’s implement the component. Start first by creating a “stateless” version that doesn’t handle updates; this is not too different from all the previous examples:

```
class TextAreaCounter extends React.Component {
  render() {
    const text = this.props.text;
    return (
      <div>
        <textarea defaultValue={text}/>
        <h3>{text.length}</h3>
      </div>
    );
  }
}
TextAreaCounter.defaultProps = {
  text: 'Count me as I type',
};
```

NOTE

You may have noticed that the `<textarea>` in the preceding snippet takes a `defaultValue` property, as opposed to a text child node, as you're accustomed to in regular HTML. This is because there are some slight differences between React and old-school HTML when it comes to form elements. These are discussed further in the book, but rest assured, there are not too many of them. Additionally , you'll find that these differences make sense and make your life as a developer easier.

As you can see, the `TextAreaCounter` component takes an optional `text` string property and renders a textarea with the given value, as well as an `<h3>` element that displays the string's `length`. If the `text` property is not supplied, the default "Count me as I type" value is used.

Make it Stateful

The next step is to turn this *stateless* component into a *stateful* one. In other words, let's have the component maintain some data (state) and use this data to render itself initially and later on update itself (re-render) when data changes.

First, you need to set the initial state in the class constructor using `this.state`. Bear in mind that the constructor is the only place where it's ok to set the state directly without calling `this.setState()`.

Initializing `this.state` is required, if you don't do it, consecutive access to `this.state` in the `render()` method will fail.

In this case it's not necessary to initialize `this.state.text` with a value as you can fallback to the property `this.prop.text` (try `02.12.this.state.html` in the book's repo):

```
class TextAreaCounter extends React.Component {
  constructor() {
    super();
    this.state = {};
  }
  render() {
    const text = 'text' in this.state ? this.state.text :
this.props.text;
    return (
      <div>
        <textarea defaultValue={text} />
        <h3>{text.length}</h3>
      </div>
    );
  }
}
```

NOTE

Calling `super()` in the constructor is required before you can use `this`.

The data this component maintains is the contents of the textarea, so the state has only one property called `text`, which is accessible via `this.state.text`. Next you need a way to update the state. You can use a helper method for this purpose:

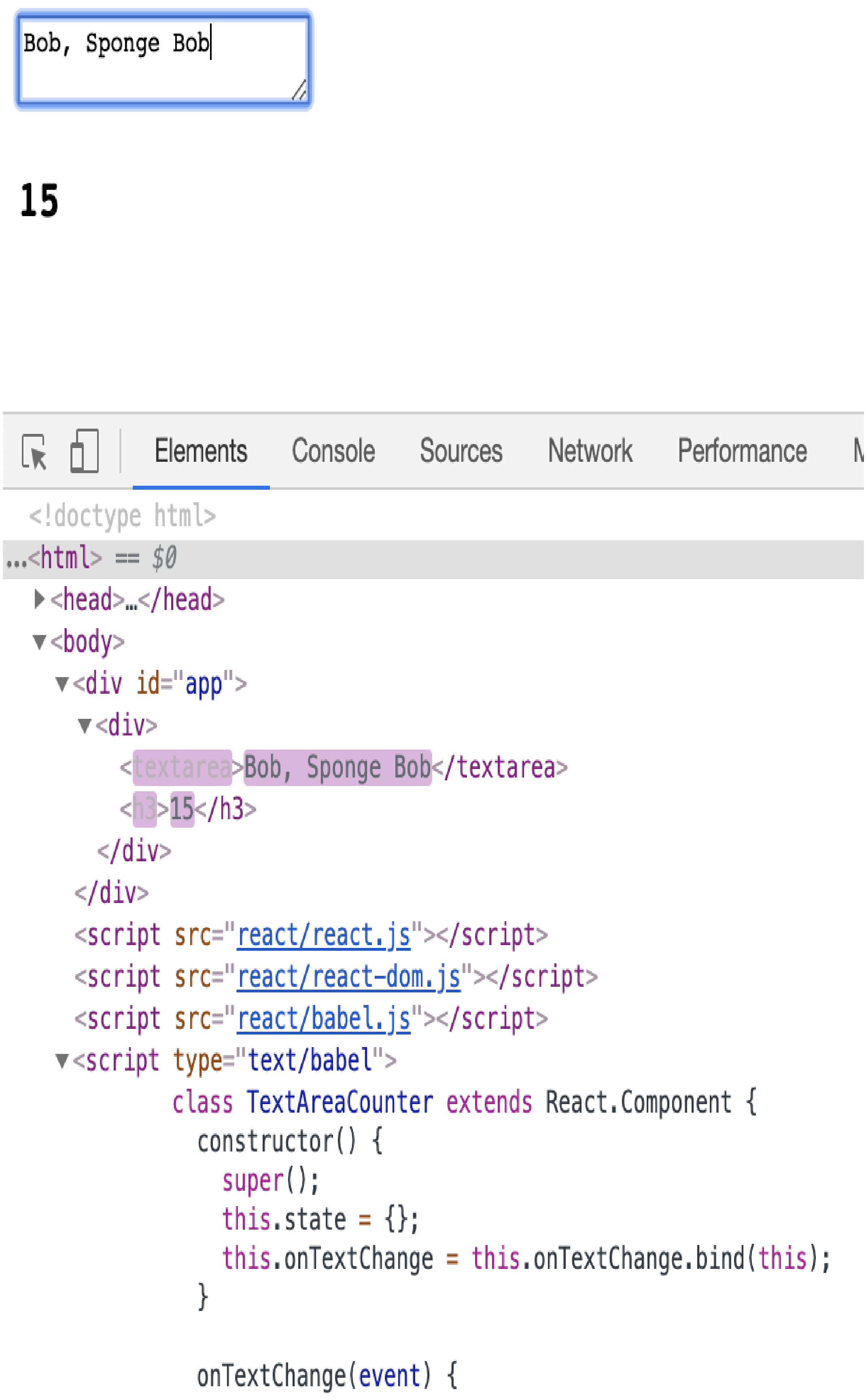
```
onTextChange(event) {
  this.setState({
    text: event.target.value,
  });
}
```

You always update the state with `this.setState()`, which takes an object and merges it with the already existing data in `this.state`. As you might guess, `onTextChange()` is an event handler that takes an `event` object and reaches into it to get the contents of the textarea input.

The last thing left to do is update the `render()` method to set up the event handler:

```
render() {
  const text = 'text' in this.state ? this.state.text : this.props.text;
  return (
    <div>
      <textarea
        value={text}
        onChange={event => this.onTextChange(event)}
      />
      <h3>{text.length}</h3>
    </div>
  );
}
```

Now whenever the user types into the textarea, the value of the counter updates to reflect the contents (Figure 2-4).



```
Bob, Sponge Bob
```

15

```
<!doctype html>
…<html> == $0
  ▶<head>…</head>
  ▼<body>
    ▼<div id="app">
      ▼<div>
          <textarea>Bob, Sponge Bob</textarea>
          <h3>15</h3>
        </div>
      </div>
      <script src="react/react.js"></script>
      <script src="react/react-dom.js"></script>
      <script src="react/babel.js"></script>
    ▼<script type="text/babel">
              class TextAreaCounter extends React.Component {
                constructor() {
                  super();
                  this.state = {};
                  this.onTextChange = this.onTextChange.bind(this);
                }

                onTextChange(event) {
```

Figure 2-4. Typing in the textarea (`02.12.this.state.html`)

Note that `<teaxarea defaultValue...>` in now `<textarea value...>`. This is because of the way inputs work in HTML where their state is maintained by the browser. But React can do better. In this example implementing `onChange` means that the textarea is now *controlled* by React. More on *controlled components* is coming further in the book.

A Note on DOM Events

To avoid any confusion, a few clarifications are in order regarding the line:

```
onChange={event => this.onTextChange(event)}
```

React uses its own *synthetic* events system for performance, as well as convenience and sanity reasons. To help understand why, you need to consider how things are done in the pure DOM world.

Event Handling in the Olden Days

It's very convenient to use *inline* event handlers to do things like this:

```
<button onclick="doStuff">
```

While convenient and easy to read (the event listener is right there with the UI code), it's inefficient to have too many event listeners scattered like this. It's also hard to have more than one listener on the same button, especially if said button is in somebody else's "component" or library and you don't want to go in there and "fix" or fork their code. That’s why in the DOM world it’s common to use `element.addEventListener` to set up listeners (which now leads to having code in two places or more) and *event delegation* (to address the performance issues). Event delegation means you listen to events at some parent node, say a `<div>` that contains many buttons, and you set up one listener for all the buttons, instead of one listener per button. Hence you *delegate* the event handling to a parent authority.

With event delegation you do something like:

```
<div id="parent">
  <button id="ok">OK</button>
  <button id="cancel">Cancel</button>
</div>

<script>
document.getElementById('parent').addEventListener('click',
function(event) {
  const button = event.target;

  // do different things based on which button was clicked
  switch (button.id) {
    case 'ok':
      console.log('OK!');
      break;
    case 'cancel':
      console.log('Cancel');
      break;
    default:
      new Error('Unexpected button ID');
  };
});
</script>
```

This works and performs fine, but there are drawbacks:

- Declaring the listener is further away from the UI component, which makes code harder to find and debug
- Using delegation and always `switch`-ing creates unnecessary boilerplate code even before you get to do the actual work (responding to a button click in this case)
- Browser inconsistencies (omitted here) actually require this code to be longer

Unfortunately, when it comes to taking this code live in front of real users, you need a few more additions if you want to support old browsers:

- You need `attachEvent` in addition to `addEventListener`
- You need `const event = event || window.event;` at the top of the listener
- You need `const button = event.target || event.srcElement;`

All of these are necessary and annoying enough that you end up using an event library of some sort. But why add another library (and study more APIs) when React comes bundled with a solution to the event handling nightmares?

Event Handling in React

React uses *synthetic events* to wrap and normalize the browser events, which means no more browser inconsistencies. You can always rely on the fact that `event.target` is available to you in all browsers. That’s why in the `TextAreaCounter` snippet you only need `event.target.value` and it just works. It also means the API to cancel events is the same in all browsers; in other words, `event.stopPropagation()` and `event.preventDefault()` work even in old versions of Internet Explorer.

The syntax makes it easy to keep the UI and the event listeners together. It looks like old-school inline event handlers, but behind the scenes it’s not. Actually, React uses event delegation for performance reasons.

React uses camelCase syntax for the event handlers, so you use `onClick` instead of `onclick`.

If you need the original browser event for whatever reason, it’s available to you as `event.nativeEvent`, but it’s unlikely that you’ll ever need to go there.

And one more thing: the `onChange` event (as used in the textarea example) behaves as you’d expect: it fires when the user types, as opposed to after they’ve finished typing and have navigated away from the field, which is the behavior in plain DOM.

Event-Handling Syntax

The example above used an arrow function to call the helper `onTextChange` event:

```
onChange={event => this.onTextChange(event)}
```

This is because the shorter `onChange={this.onTextChange}` wouldn’t have worked.

Another option is to bind the method, like so:

```
onChange={this.onTextChange.bind(this)}
```

And yet another option, and a common pattern, is to bind all the event handling methods in the constructor:

```
constructor() {
  super();
  this.state = {};
  this.onTextChange = this.onTextChange.bind(this);
}
// ....
<textarea
  value={text}
  onChange={this.onTextChange}
/>
```

It’s a bit of necessary boilerplate, but this way the event handler is bound only once, as opposed to every time the `render()` method is called, which helps reduce the memory footprint of your app.

Props Versus State

Now you know that you have access to `this.props` and `this.state` when it comes to displaying your component in your `render()` method. You may be wondering when you should use versus the other.

Properties are a mechanism for the outside world (users of the component) to configure your component. State is your internal data maintenance. So if you consider an analogy with object-oriented programming, `this.props` is like a collection of all the *arguments* *passed to a class constructor*, while `this.state` is a bag of your *private properties*.

In general, prefer to split your application in a way that you have fewer *stateful* components and more *stateless* ones.

Props in Initial State: An Anti-Pattern

In the textarea example above it was tempting to use `this.props` inside of the constructor to set `this.state`:

```
this.state = {
  text: this.props.text,
};
```

This is considered an anti-pattern. Ideally, you use any combination of `this.state` and `this.props` as you see fit to build your UI in your `render()` method. But sometimes you want to take a value passed to your component and use it to construct the initial state. There's nothing wrong with this, except that the callers of your component may expect the property (`text` in the preceding example) to always have the latest value and the code above would violate this expectation. To set expectation straight, a simple naming change is sufficient—for example, calling the property something like `defaultText` or `initialValue` instead of just `text`:

NOTE

Chapter 4 illustrates how React solves this for its implementation of inputs and textareas where people may have expectations coming from their prior HTML knowledge.

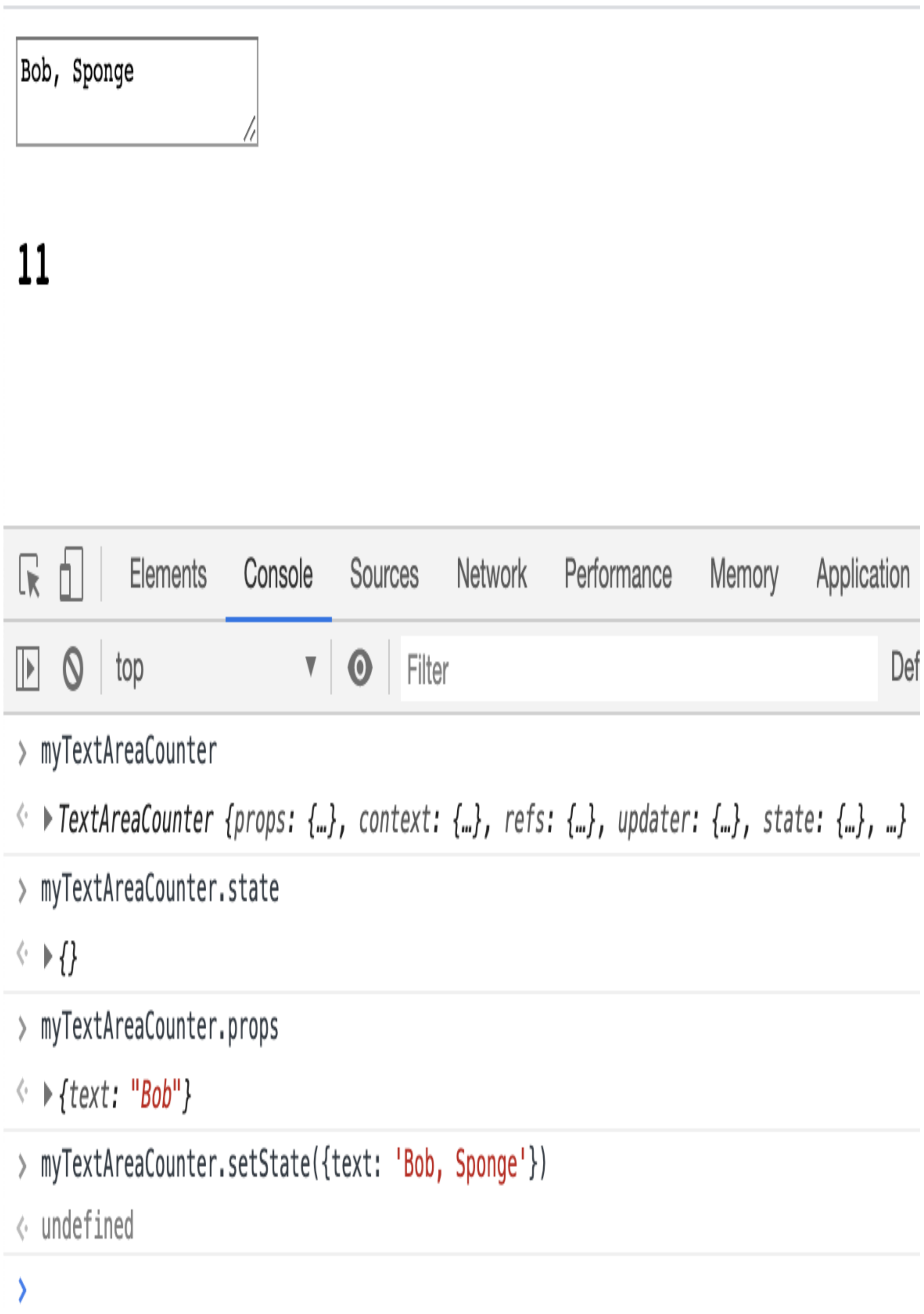
Accessing the Component from the Outside

You don’t always have the luxury of starting a brand-new React app from scratch. Sometimes you need to hook into an existing application or a website and migrate to React one piece at a time. Luckily, React was designed to work with any pre-existing codebase you might have. After all, the original creators of React couldn’t stop the world and rewrite an entire huge application (Facebook.com) completely from scratch, especially in the early days when React was young.

One way to have your React app communicate with the outside world is to get a reference to a component you render with `ReactDOM.render()` and use it from outside of the component:

```
const myTextAreaCounter = ReactDOM.render(
  <TextAreaCounter text="Bob" />,
  document.getElementById('app')
);
```

Now you can use `myTextAreaCounter` to access the same methods and properties you normally access with `this` when inside the component. You can even play with the component using your JavaScript console (Figure 2-5).



```
> myTextAreaCounter
< ▸TextAreaCounter {props: {…}, context: {…}, refs: {…}, updater: {…}, state: {…}, …}
> myTextAreaCounter.state
< ▸{}
> myTextAreaCounter.props
< ▸{text: "Bob"}
> myTextAreaCounter.setState({text: 'Bob, Sponge'})
< undefined
>
```

Figure 2-5. Accessing the rendered component by keeping a reference

In this example, `myTextAreaCounter.state` checks the current state (empty initially), `myTextAreaCounter.props` checks the properties and this line sets a new state:

```
myTextAreaCounter.setState({text: "Hello outside world!"});
```

This line gets a reference to the main parent DOM node that React created:

```
const reactAppNode = ReactDOM.findDOMNode(myTextAreaCounter);
```

This is the first child of the `<div id="app">`, which is where you told React to do its magic.

NOTE

You have access to the entire component API from outside of your component. But you should use your new superpowers sparingly, if at all. It may be tempting to fiddle with the state of components you don’t own and “fix” them, but you’d be violating expectations and cause bugs down the road because the component doesn’t anticipate such intrusions.

Lifecycle Methods

React offers several so-called *lifecycle* methods. You can use the lifecycle methods to listen to changes in your component as far as the DOM manipulation is concerned. The life of a component goes through three steps:

- Mounting - the component is added to the DOM initially

- Updating - the component is updated as a result of calling `setState()`
- Unmounting - the component is removed from the DOM

React does part of its work before updating the DOM, this is also called *rendering phase*. Then it updates the DOM and this phase is called a *commit phase*. With this background let's consider some lifecycle methods:

- After the initial mounting and after the commit to the DOM, the method `componentDidMount()` of your component is called, if it exists. This is the place to do any initialization work that requires the DOM. Any initialization work that *does not* require the DOM should be in the constructor. And most of your initialization shouldn't require the DOM. But in this method you can, for example, measure the height of the component that was just rendered, add any event listeners (e.g. `addEventListener('resize')`), or fetch data from the server.
- Right before the component is removed from the DOM, the method `componentWillUnmount()` is called. This is the place to do any cleanup work you may need. Any event handlers, or anything else that may leak memory, should be cleaned up here. After this, the component is gone forever.
- Before the component is updated, e.g. as a result of `setState()`, you can use `getSnapshotBeforeUpdate()`. This method receives the previous properties and state as arguments. And it can return a "snapshot" value, which is any value you want to pass over to the next lifecycle method, which is…
- `componentDidUpdate(previousProps, previousState, snapshot)`. This is called whenever the component was

updated. Since at this point `this.props` and `this.state` have updated values, you get a copy of the previous ones. You can use this information to compare the old and the new state and potentially make more network requests if necessary.

- And then there’s `shouldComponentUpdate(newProps, newState)` which is an opportunity for an optimization. You’re given the state-to-be which you can compare with the current state and decide not to update the component, so its `render()` method is not called.

Of these, `componentDidMount()` and `componentDidUpdate()` are the most common ones.

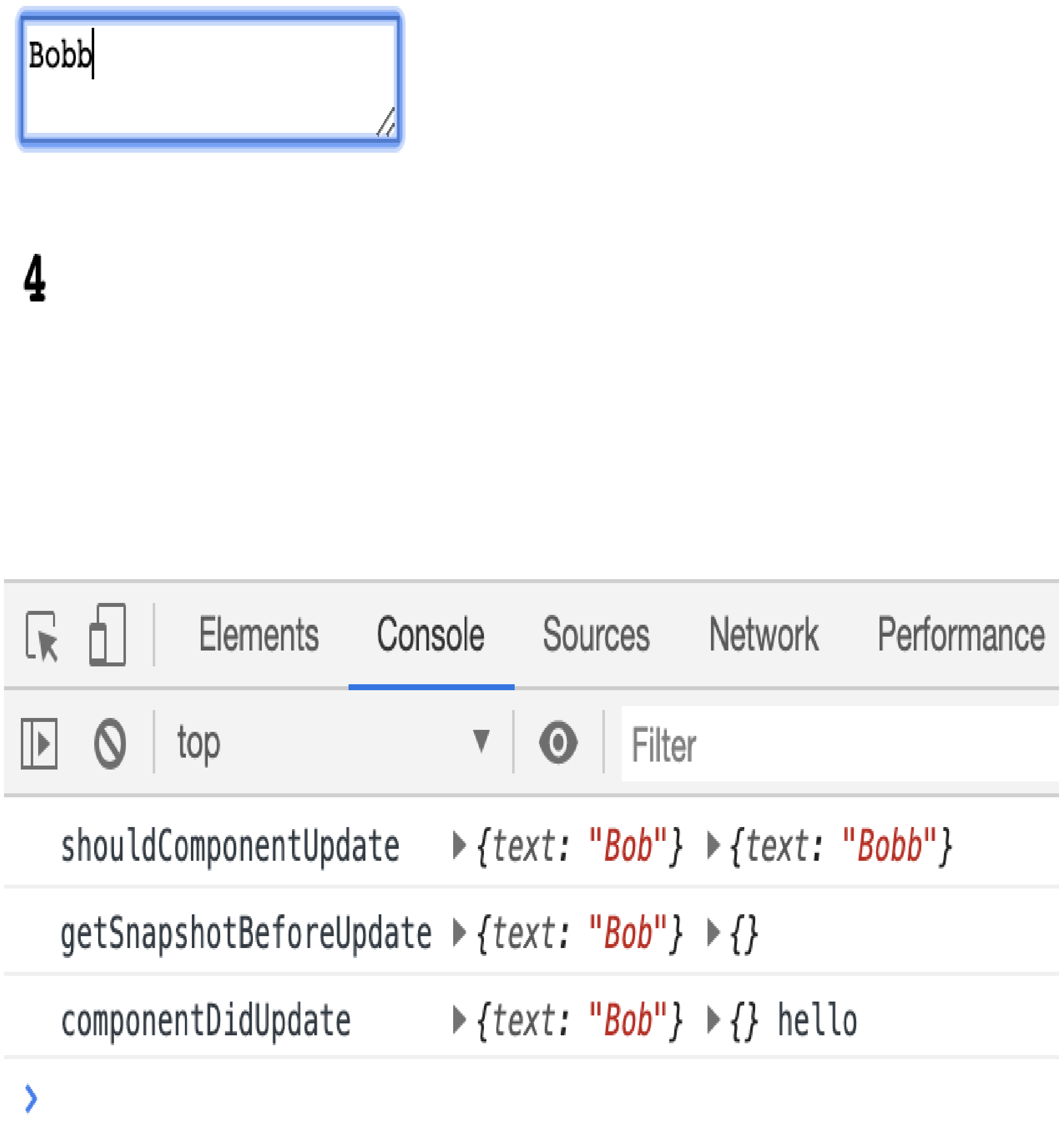
Lifecycle Example: Log It All

To better understand the life of a component, let’s add some logging in the `TextAreaCounter` component. Simply implement all of the lifecycle methods to log to the console when they are invoked, together with any arguments:

```
componentDidMount() {
  console.log('componentDidMount');
}
componentWillUnmount() {
  console.log('componentWillUnmount');
}
componentDidUpdate(prevProps, prevState, snapshot) {
  console.log('componentDidUpdate     ', prevProps, prevState,
snapshot);
}
getSnapshotBeforeUpdate(prevProps, prevState) {
  console.log('getSnapshotBeforeUpdate', prevProps, prevState);
  return 'hello';
}
shouldComponentUpdate(newProps, newState) {
  console.log('shouldComponentUpdate  ', newProps, newState);
  return true;
}
```

After loading the page, the only message in the console is “componentDidMount”.

Next, what happens when you type “b” to make the text “Bobb”? (See Figure 2-6.) `shouldComponentUpdate()` is called with the new props (same as the old) and the new state. Since this method returns `true`, React proceeds with calling `getSnapshotBeforeUpdate()` passing the old props and state. This is your chance to do something with them and with the old DOM and pass any resulting information as a snapshot to the next method. For example this is an opportunity to do some element measurements or a scroll position and snapshot them to see if they change after the update. Finally, `componentDidUpdate()` is called with the old info (you have the new one in `this.state` and `this.props`) and any snapshot defined by the previous method.



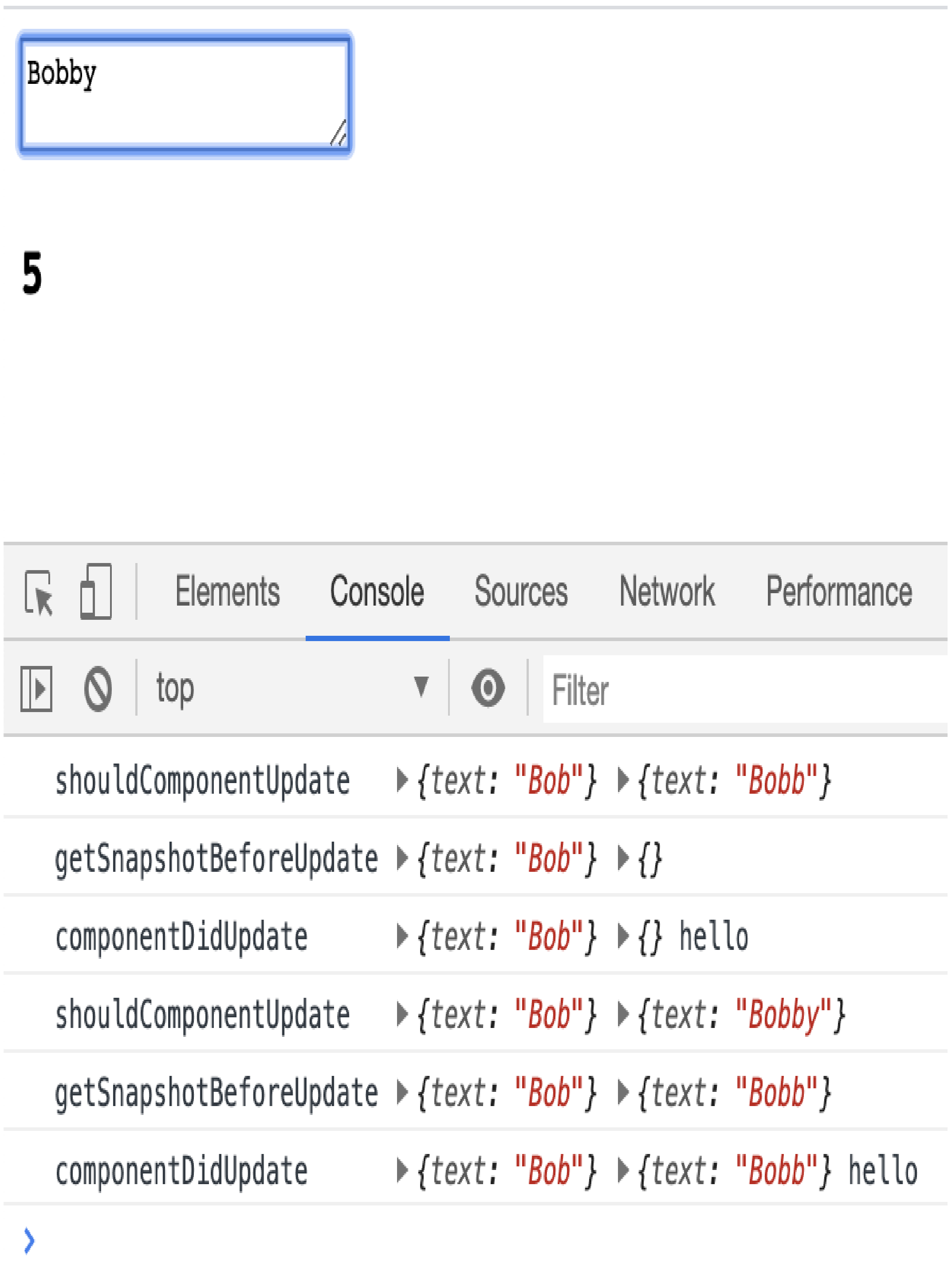
Bobb

4

```
shouldComponentUpdate   ▸{text: "Bob"} ▸{text: "Bobb"}
getSnapshotBeforeUpdate ▸{text: "Bob"} ▸{}
componentDidUpdate      ▸{text: "Bob"} ▸{} hello
```

Figure 2-6. Updating the component

Let’s update the textarea one more time, this time typing “y”. The result is shown on Figure 2-7.



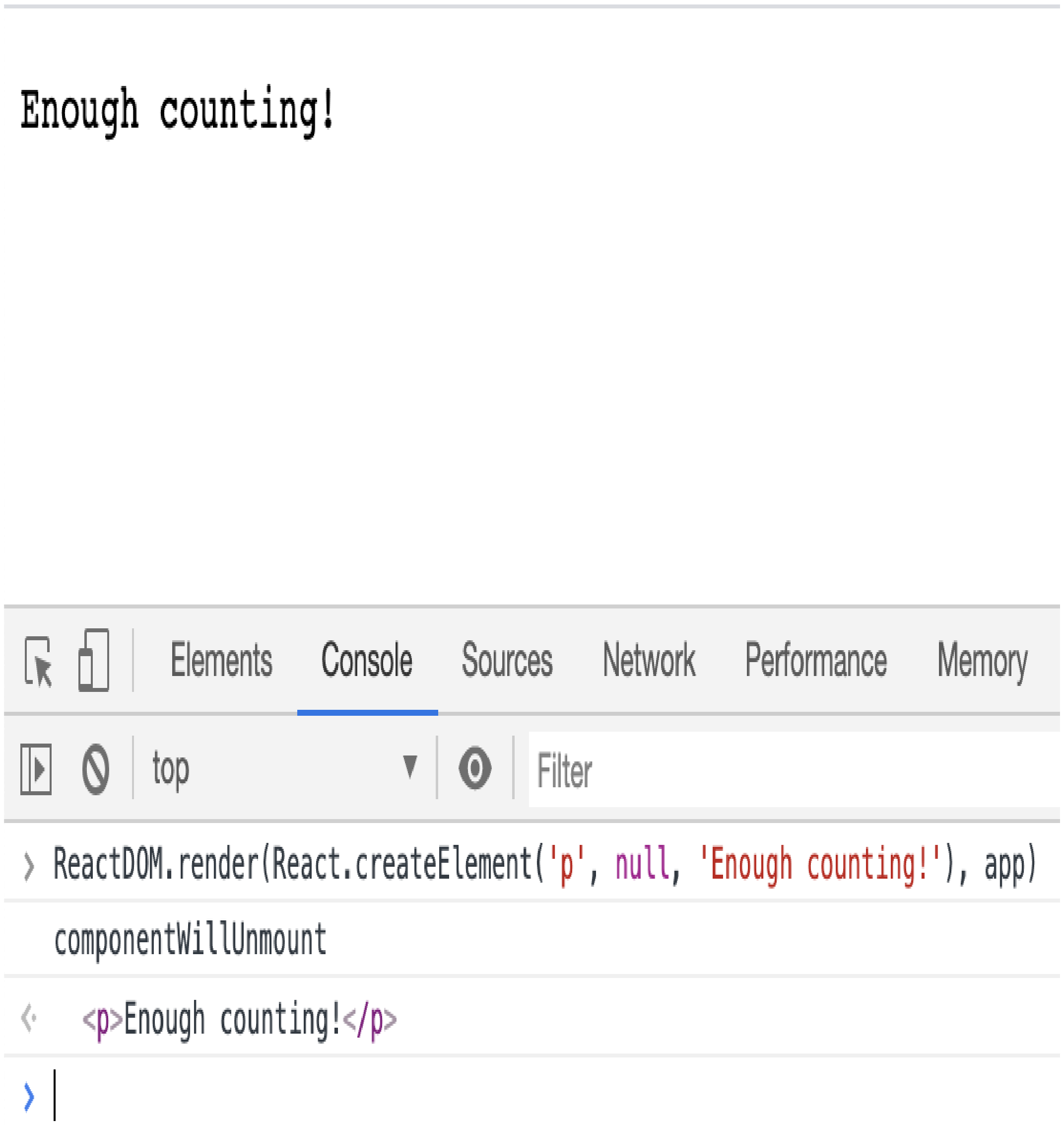
```
shouldComponentUpdate   ▸{text: "Bob"} ▸{text: "Bobb"}
getSnapshotBeforeUpdate ▸{text: "Bob"} ▸{}
componentDidUpdate      ▸{text: "Bob"} ▸{} hello
shouldComponentUpdate   ▸{text: "Bob"} ▸{text: "Bobby"}
getSnapshotBeforeUpdate ▸{text: "Bob"} ▸{text: "Bobb"}
componentDidUpdate      ▸{text: "Bob"} ▸{text: "Bobb"} hello
```

Figure 2-7. One more update to the component

Finally, to demonstrate `componentWillUnmount()` in action (using the example `02.14.lifecycle.html` from this book's GitHub repo) you can type in the console:

```
ReactDOM.render(React.createElement('p', null, 'Enough counting!'),
app);
```

This replaces the whole textarea component with a new `<p>` component. Then you can see the log message "componentWillUnmount" in the console (Figure 2-8).


Enough counting!

```
> ReactDOM.render(React.createElement('p', null, 'Enough counting!'), app)
  componentWillUnmount
<· <p>Enough counting!</p>
>
```

Figure 2-8. Removing the component from the DOM

Paranoid State Protection

Say you want to restrict the number of characters to be typed in the textarea. You should do this in the event handler `onTextChange()`, which is called as the user types. But what if someone (a younger, more naive you?) calls `setState()` from the outside of the component? (Which, as mentioned earlier, is a bad idea.) Can you still protect the consistency and well-being of your component? Sure. You can do the validation in `componentDidUpdate()` and if the number of characters is greater than allowed, revert the state back to what it was. Something like:

```
componentDidUpdate(prevProps, prevState) {
  if (this.state.text.length > 3) {
    this.setState({
      text: prevState.text || this.props.text,
    });
  }
}
```

The condition `prevState.text || this.props.text` is in place for the very first update when there’s no previous state.

This may seem overly paranoid, but it’s still possible to do. Another way to accomplish the same protection is by leveraging `shouldComponentUpdate()`:

```
shouldComponentUpdate(_, newState) {
  return newState.text.length > 3 ? false : true;
}
```

See `02.15.paranoid.html` in the book’s repo to play with these concepts.

Lifecycle Example: Using a Child Component

You know you can mix and nest React components as you see fit. So far you've only seen `ReactDOM` components (as opposed to custom ones) in the `render()` methods. Let's take a look at another simple custom component to be used as a child.

Let's isolate the counter part into its own component. After all, divide and conquer is what it's all about!

First, let's isolate the lifestyle logging into a separate class and have the two components inherit it. Inheritance is almost never warranted when it comes to React because for UI work composition is preferable and for non-UI work a regular JavaScript module would do. But this is just for education and for demonstration that it is possible. And also to avoid copy-pasting the logging methods.

This is the parent:

```
class LifecycleLoggerComponent extends React.Component {
  static getName() {}
  componentDidMount() {
    console.log(this.constructor.getName() + '::componentDidMount');
  }
  componentWillUnmount() {
    console.log(this.constructor.getName() + '::componentWillUnmount');
  }
  componentDidUpdate(prevProps, prevState, snapshot) {
    console.log(this.constructor.getName() + '::componentDidUpdate');
  }
}
```

The new `Counter` component simply shows the count. It doesn't maintain state, but displays the `count` property given by the parent.

```
class Counter extends LifecycleLoggerComponent {
  static getName() {
    return 'Counter';
  }
  render() {
    return <h3>{this.props.count}</h3>;
  }
}
Counter.defaultProps = {
  count: 0,
};
```

The textarea component sets up a static `getName()` method:

```
class TextAreaCounter extends LifecycleLoggerComponent {
  static getName() {
    return 'TextAreaCounter';
  }
  // ....
}
```

And finally, the textarea's `render()` gets to use `<Counter/>` and use it conditionally; if the count is 0, nothing is displayed.

```
render() {
  const text = 'text' in this.state ? this.state.text : this.props.text;
  return (
    <div>
      <textarea
        value={text}
        onChange={this.onTextChange}
      />
      {text.length > 0
        ? <Counter count={text.length} />
        : null
      }
    </div>
  );
}
```

NOTE

Notice the conditional statement in JSX. You wrap the expression in `{}` and conditionally render either `<Counter/>` or nothing (`null`). And just for demonstration: another option is to move the condition outside the `return`. Assigning the result of a JSX expression to a variable is perfectly fine.

```
render() {
  const text = 'text' in this.state
    ? this.state.text
    : this.props.text;
  let counter = null;
  if (text.length > 0) {
    counter = <Counter count={text.length} />;
  }
  return (
    <div>
      <textarea
        value={text}
        onChange={this.onTextChange}
      />
      {counter}
    </div>
  );
}
```

Now you can observe the lifecycle methods being logged for both components. Open `02.16.child.html` from the book's repo in your browser to see what happens when you load the page and then change the contents of the textarea.

During initial load, the child component is mounted and updated before the parent. You see in the console log:

```
Counter::componentDidMount
TextAreaCounter::componentDidMount
```

After deleting two characters you see how the child is updated, then the parent:

```
Counter::componentDidUpdate
TextAreaCounter::componentDidUpdate
Counter::componentDidUpdate
TextAreaCounter::componentDidUpdate
```

After deleting the last character, the child component is completely removed from the DOM:

```
Counter::componentWillUnmount
TextAreaCounter::componentDidUpdate
```

Finally, typing a character brings back the counter component to the DOM:

```
Counter::componentDidMount
TextAreaCounter::componentDidUpdate
```

Performance Win: Prevent Component Updates

You already know about `shouldComponentUpdate()` and saw it in action. It’s especially important when building performance-critical parts of your app. It’s invoked before `componentWillUpdate()` and gives you a chance to cancel the update if you decide it's not necessary.

There is a class of components that only use `this.props` and `this.state` in their `render()` methods and no additional function calls. These components are called "pure" components. They can implement `shouldComponentUpdate()` and compare the state and the properties before and after an update and if there aren't any changes, return `false` and save some processing power. Additionally, there can be pure static components that use neither `props` nor `state`. These can straight out return `false`.

React offers a way to make it easier to use the common (and generic) case of checking all props and state in `shouldComponentUpdate()`. Instead of repeating this work you can have your components inherit `React.PureComponent` instead of `React.Component`. This way you don't need to implement `shouldComponentUpdate()`, it's done for you. Let's take advantage and tweak the previous example.

Since both components inherit the logger, all you need is:

```
class LifecycleLoggerComponent extends React.PureComponent {
  // ... no other changes
}
```

Now both components are *pure*. Let's also add a log message in the `render()` methods:

```
render() {
  console.log(this.constructor.getName() + '::render');
  // ... no other changes
}
```

Now loading the page (`02.17.pure.html` from the repo) prints out:

```
TextAreaCounter::render
Counter::render
Counter::componentDidMount
TextAreaCounter::componentDidMount
```

Changing “Bob” to “Bobb” gives us the expected result of rendering and updating.

```
TextAreaCounter::render
Counter::render
Counter::componentDidUpdate
TextAreaCounter::componentDidUpdate
```

Now if you *paste* the string “LOLz” replacing “Bobb” (or any string with 4 characters), you see:

```
TextAreaCounter::render
TextAreaCounter::componentDidUpdate
```

As you see there’s no reason to re-render `<Counter>`, because its `props` have not changed. The new string has the same number of characters.

Whatever Happened to Functional Components?

You may have noticed that functional components dropped out of this chapter by the time `this.state` got involved. They come back later in the book, when you’ll also learn the concept of *hooks*. Since there’s no `this` in functions, there needs to be another to accomplish the same state management of a component. The good news is that once you understand the concepts of state and props, the functional component differences are just syntax.

As much “fun” as it was to spend all this time on a textarea, let’s move on to something more interesting, before introducing any new concepts. In the next chapter, you’ll see where React’s benefits come into play - namely focusing on your *data* and have the UI updates take care of themselves.

Index

About the Author

Stoyan Stefanov is a Facebook engineer. Previously at Yahoo, he was the creator of the smush.it online image-optimization tool and architect of the YSlow 2.0. performance tool. Stoyan is the author of *JavaScript Patterns* (O’Reilly, 2010) and *Object-Oriented JavaScript* (Packt Publishing, 2008), a contributor to *Even Faster Web Sites* and *High-Performance JavaScript*, a blogger, and a frequent speaker at conferences, including Velocity, JSConf, Fronteers, and many others.

Colophon

The animal on the cover of *React: Up & Running* is an ‘i’iwi (pronounced *ee-EE-vee*) bird, which is also known as a scarlet Hawaiian honeycreeper. The author’s daughter chose this animal after doing school report on it. The ‘i’iwi is the third most common native land bird in the Hawaiian Islands, though many species in its family, Fringillidae, are endangered or extinct. This small, brilliantly colored bird is a recognizable symbol of Hawai’i, with the largest colonies living on the islands of Hawai’i, Maui, and Kaua’i.

Adult ‘i’iwis are mostly scarlet, with black wings and tails and a long, curved bill. The bright red color easily contrasts with the surrounding green foliage, making the ‘i’iwi very easy to spot in the wild. Though its feathers were used extensively to decorate the cloaks and helmets of Hawaiian nobility, it avoided extinction because it was considered less sacred than its relative, the Hawaiian mamo.

The ‘i’iwi’s diet consists mostly of nectar from flowers and the 'ōhi‘a lehua tree, though it will occasionally eat small insects. It is also an altitudinal migrator; it follows the progress of flowers as they bloom at increasing altitudes throughout the year. This means that they are able to migrate between islands, though they are rare on O’ahu and Moloka’i due to habitat destruction, and have been extinct from Lāna‘i since 1929.

There are several efforts to preserve the current ‘i’iwi population; the birds are very susceptible to fowlpox and avian influenza, and are suffering from the effects of deforestation and invasive plant species.

Wild pigs create wallows that harbor mosquitos, so blocking off forest areas has helped to control mosquito-borne diseases, and there are projects underway that attempt to restore forests and remove nonnative plant species, giving the flowers that ‘i’iwis prefer the chance to thrive.

Many of the animals on O’Reilly covers are endangered; all of them are important to the world.

The cover image is from Wood’s *Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.