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

Copyright © 2021 Stoyan Stefanov. All rights reserved.

Printed in the United States of America.

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

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jennifer Pollock

Development Editor: Angela Rufino

Production Editor: Kate Galloway

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

July 2021: Second Edition

Revision History for the Early Release

- 2020-04-23: First Release
- 2021-02-03: Second Release

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

978-1-492-05139-8

To Eva, Zlatina, and Nathalie

Chapter 1. Hello World

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at sstoo@ymail.com.

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">`)

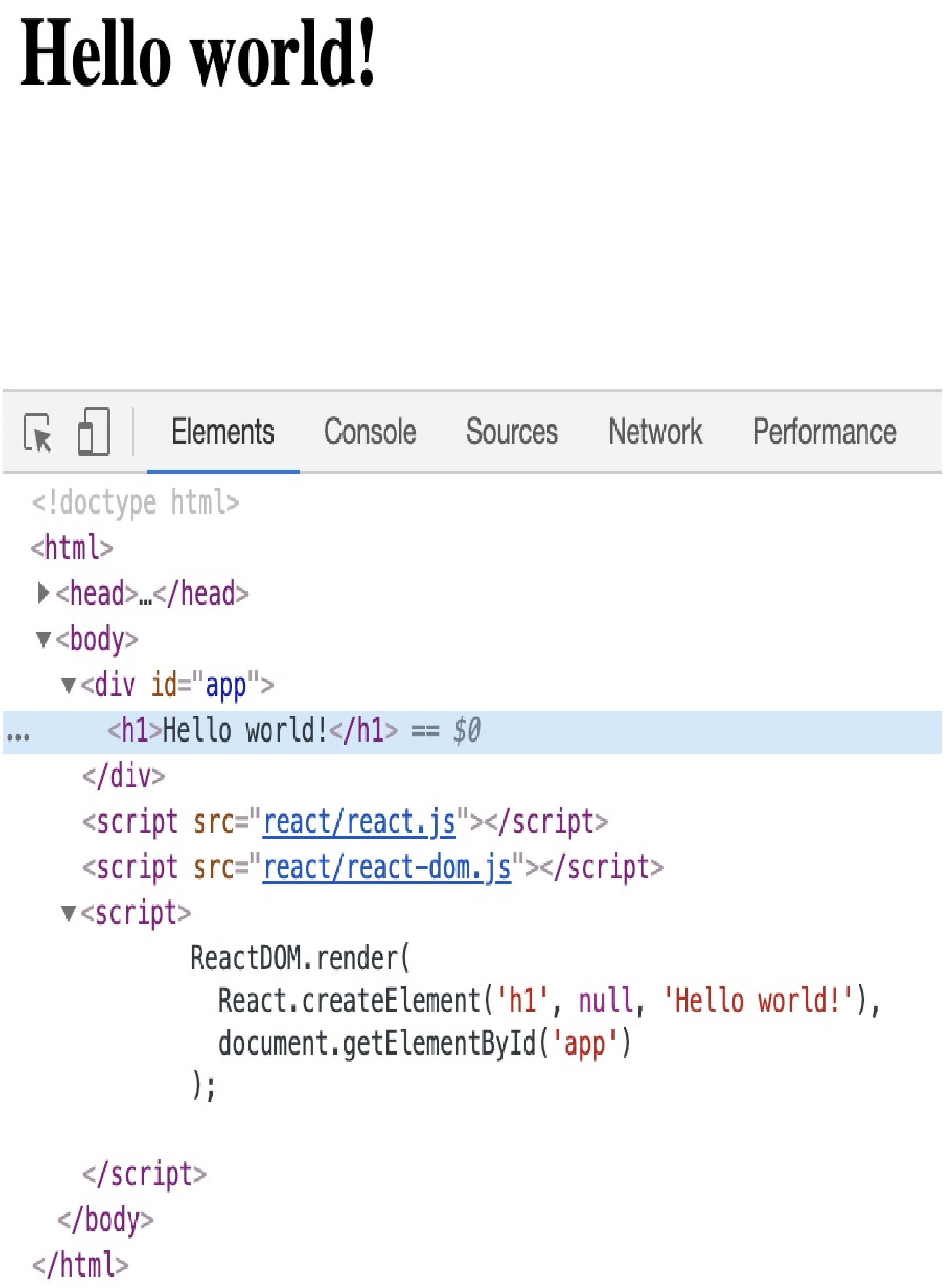
NOTE

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.

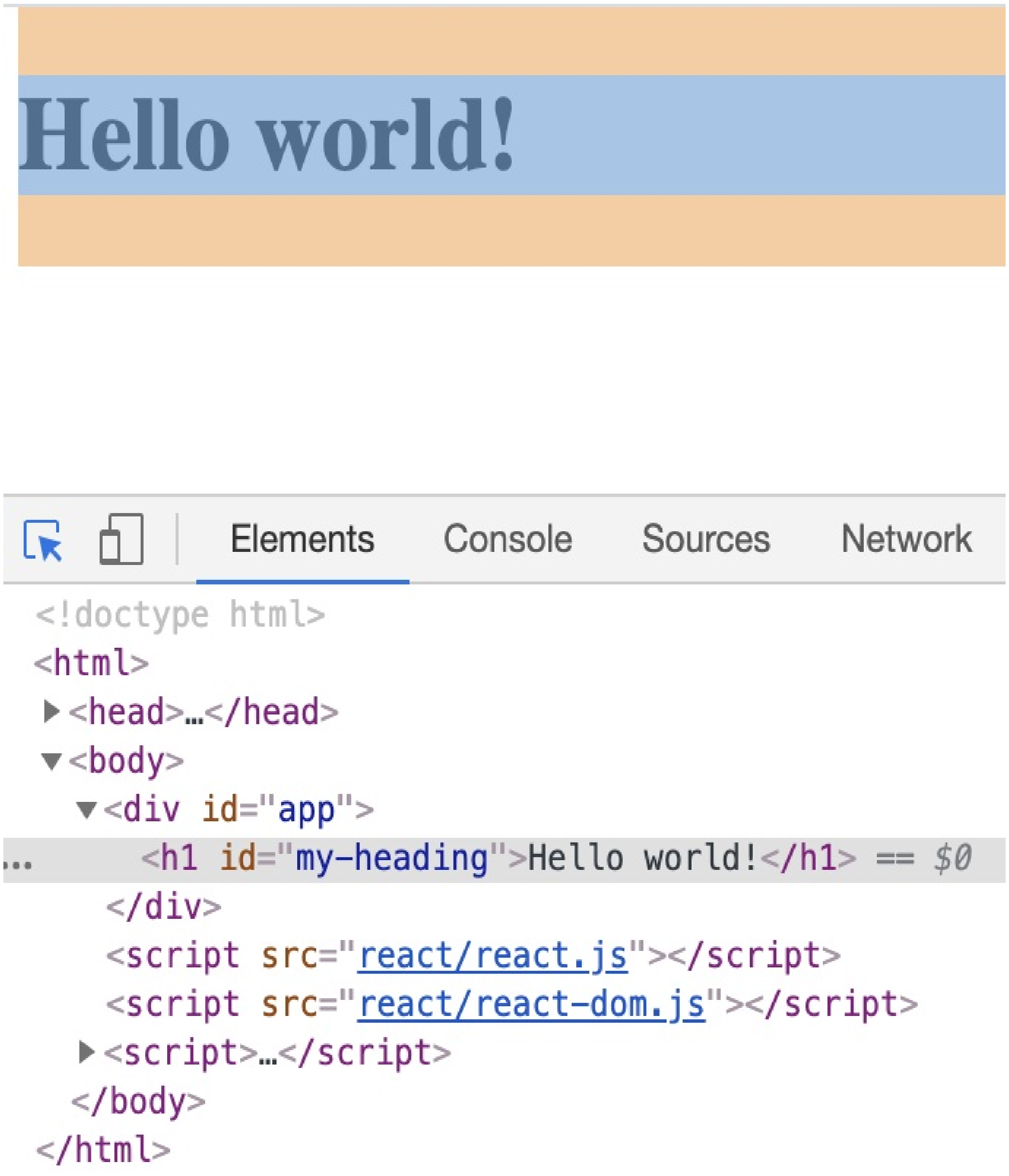
Remember the "Hello World!" app looks like this:

```
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>
```

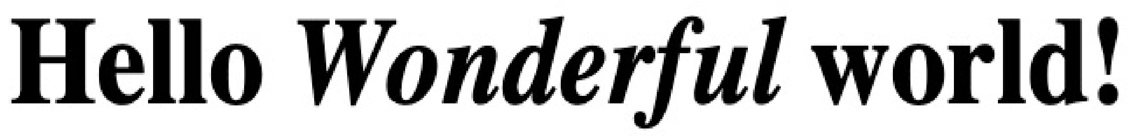
Figure 1-2. HTML generated by a `React.createElement()` call

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:

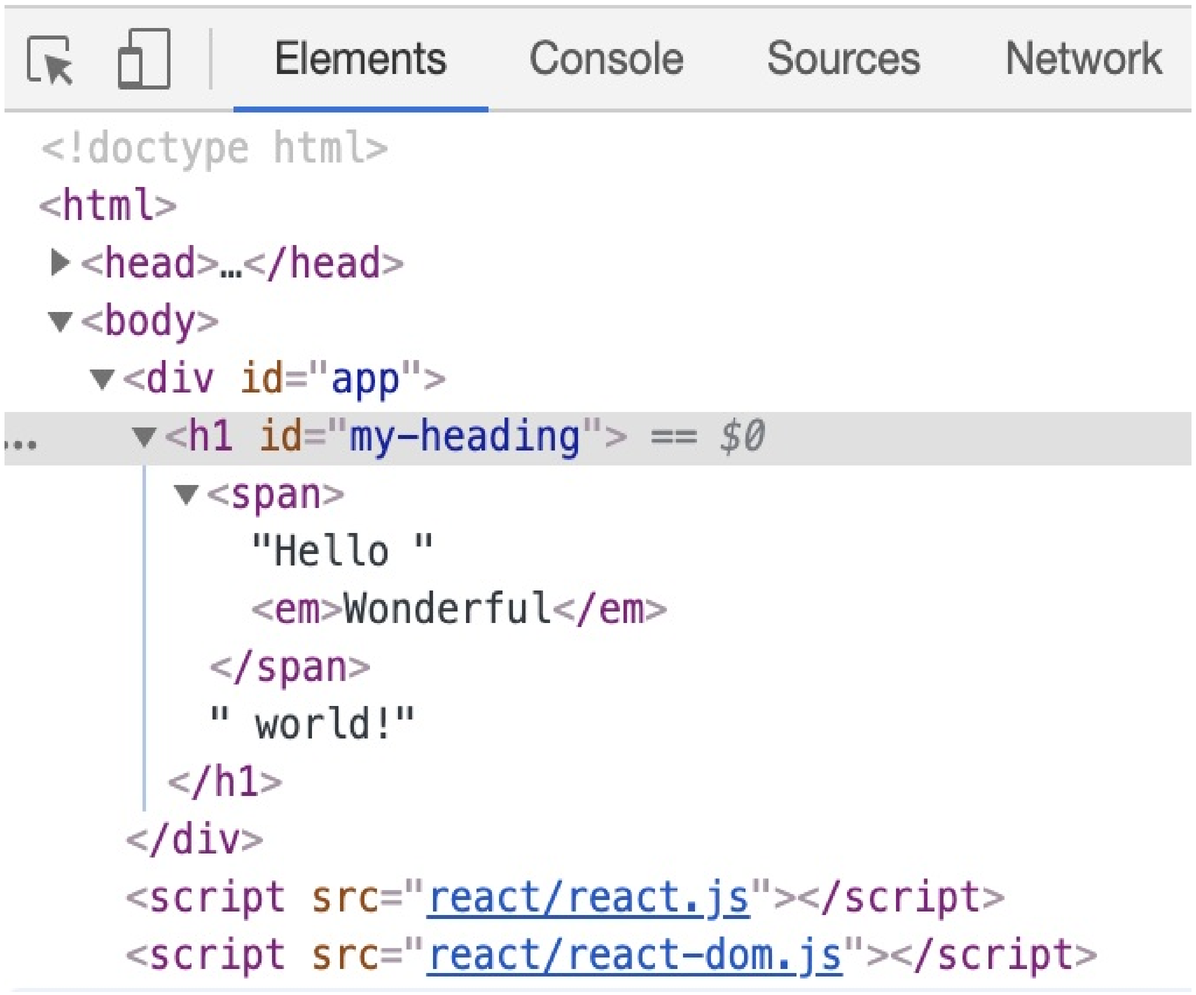
```
React.createElement(
  'h1',
  {id: 'my-heading'},
  React.createElement('span', null, 'Hello'),
  ' world!'
),
```

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!'
),
```



Hello *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

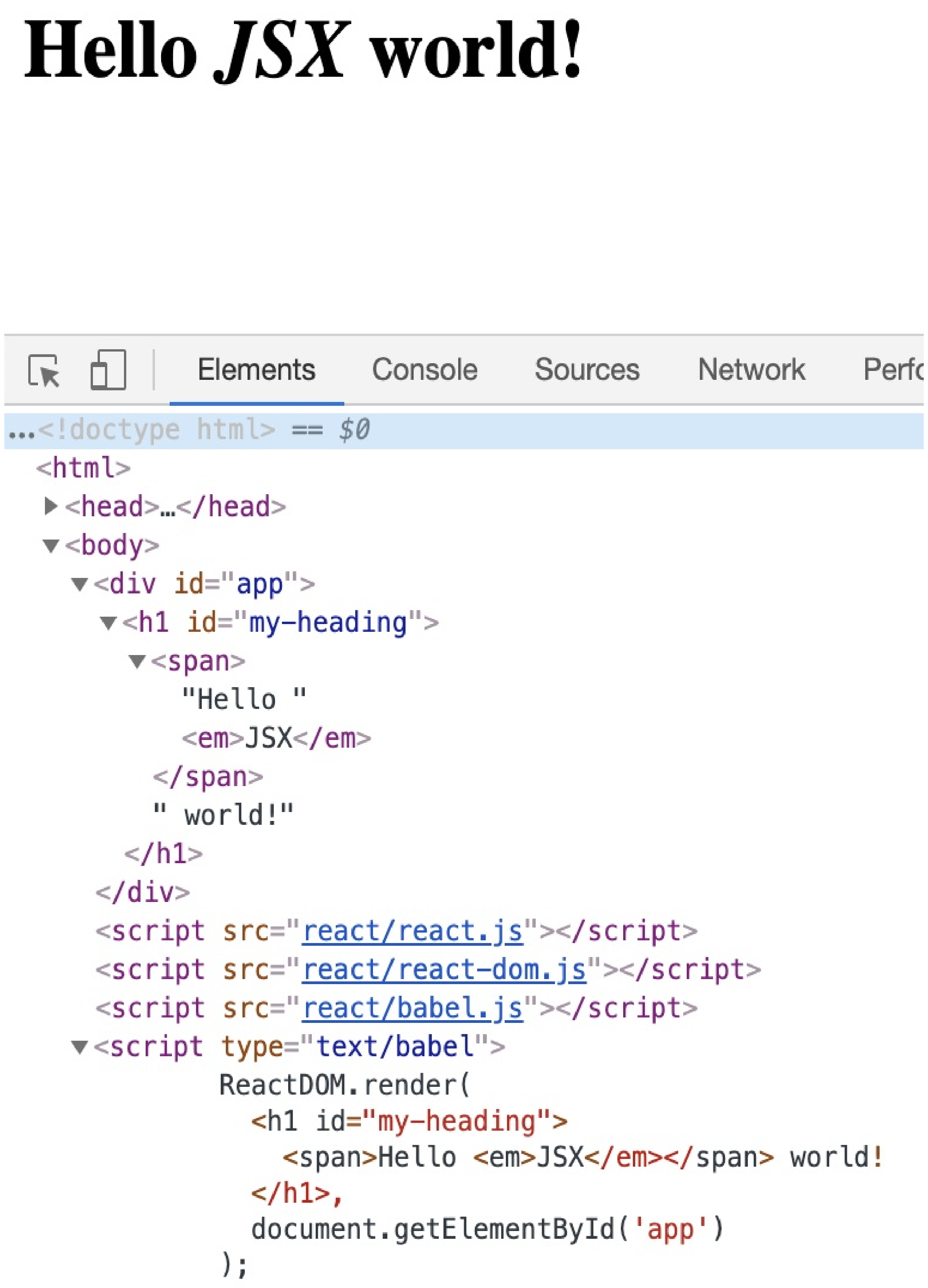
Note how `<script>` becomes `<script type="text/babel">`. This is a trick where by specifying an invalid `type`, the browser ignores the code. This gives Babel a chance to parse and transform the JSX syntax into something the browser can run.

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';
};
```

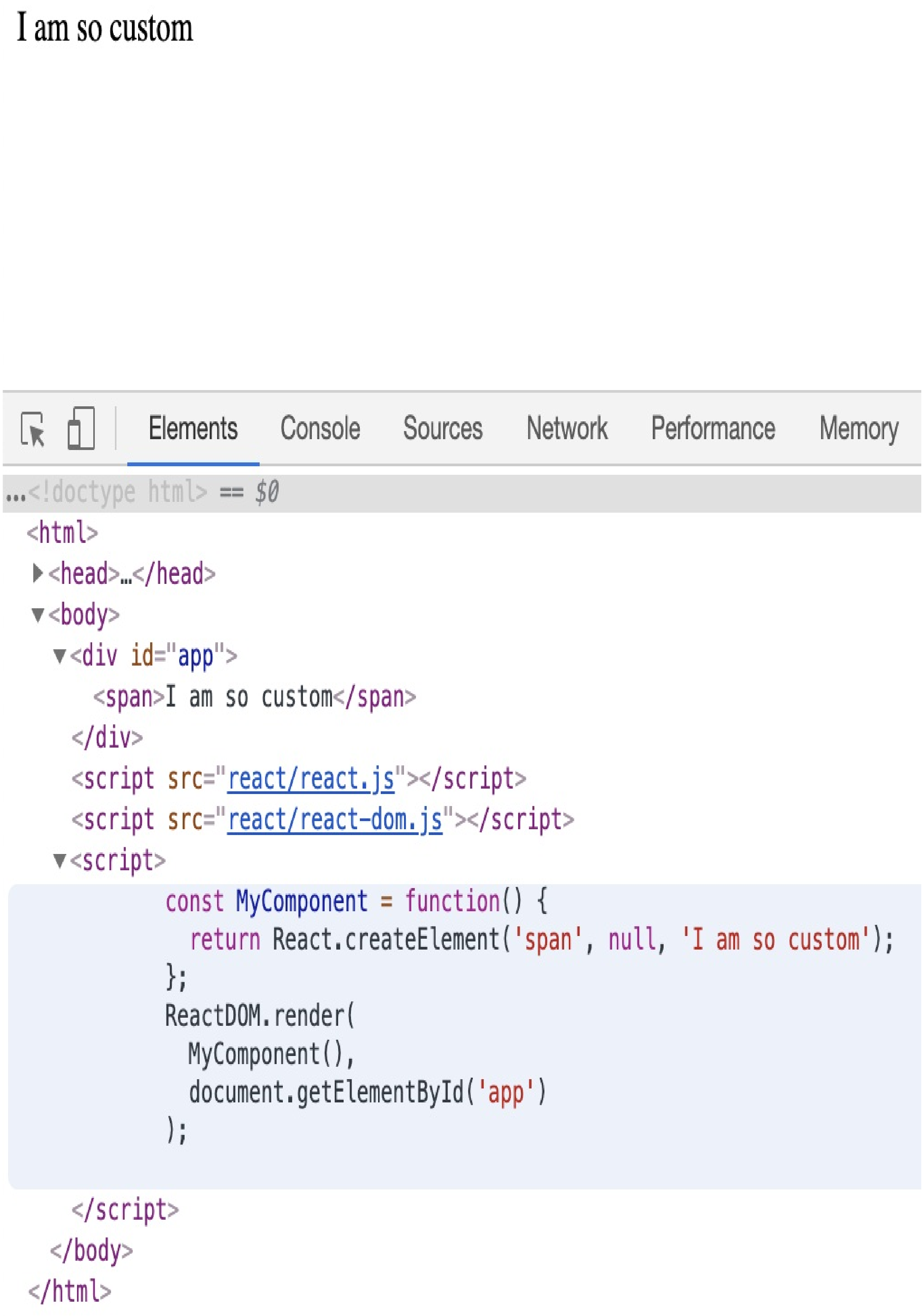
But wait, this is just a function! Yes, this is it, the custom component is just a function that returns the UI that you want. In this case, the UI is only text but often you’ll need a little bit more, most likely a composition of other components. Here’s an example of using a `span` to wrap the text:

```
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 lighter on the computer's CPU and a little less typing usually. They also feel more native to JavaScript. Actually *classes* in JavaScript were an afterthought and merely syntax sugar, classes didn't exist in early versions of the language. 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>;
  }
}
```

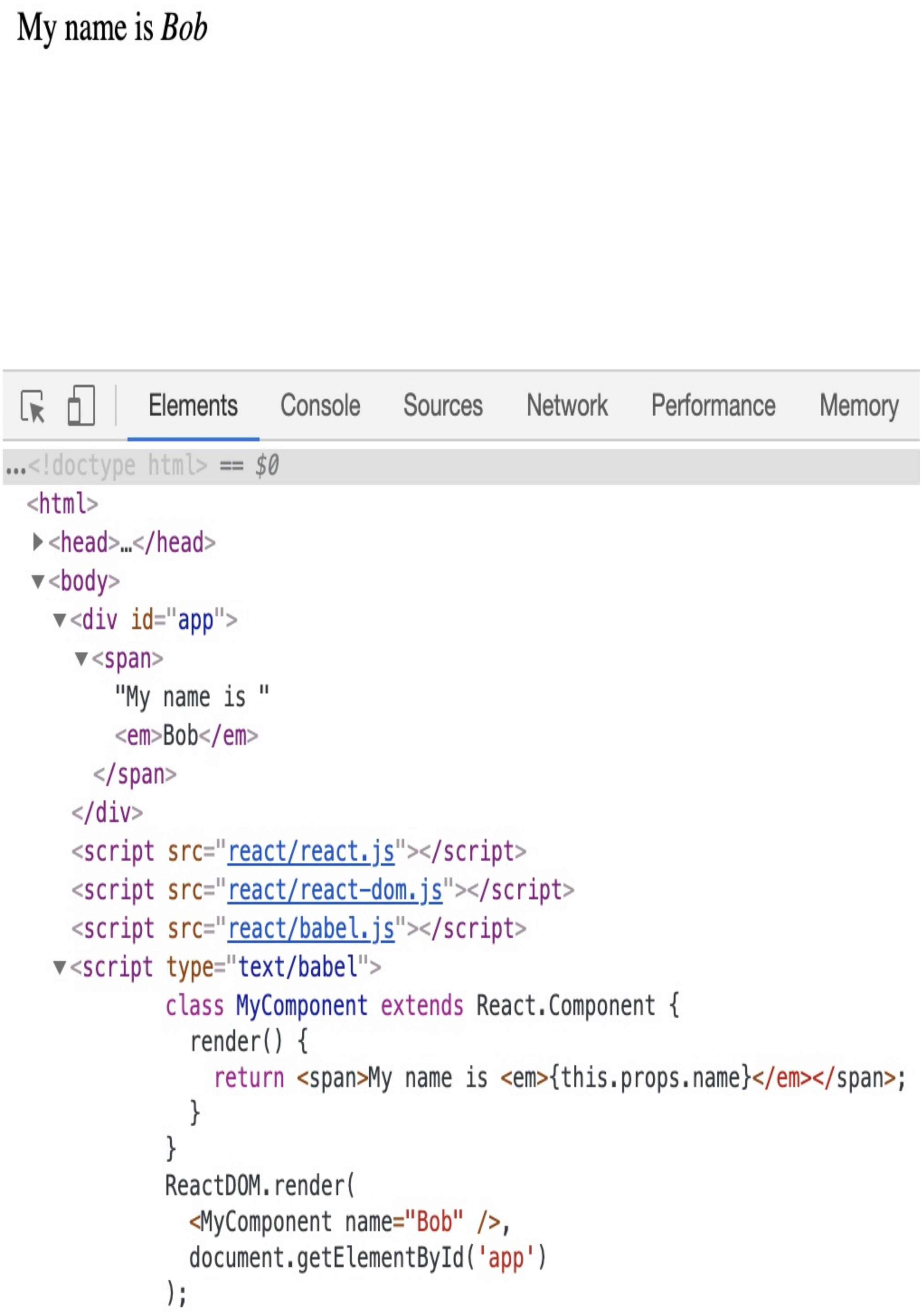
NOTE

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).

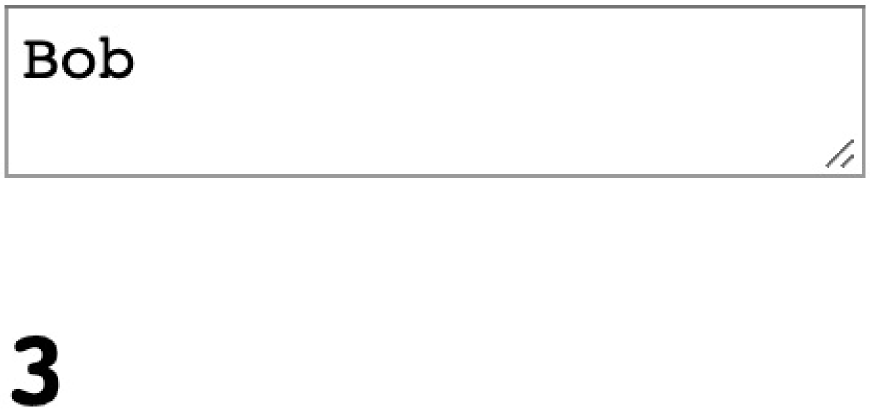


Figure 2-3. The end result of the custom textarea component

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,
  });
}
```

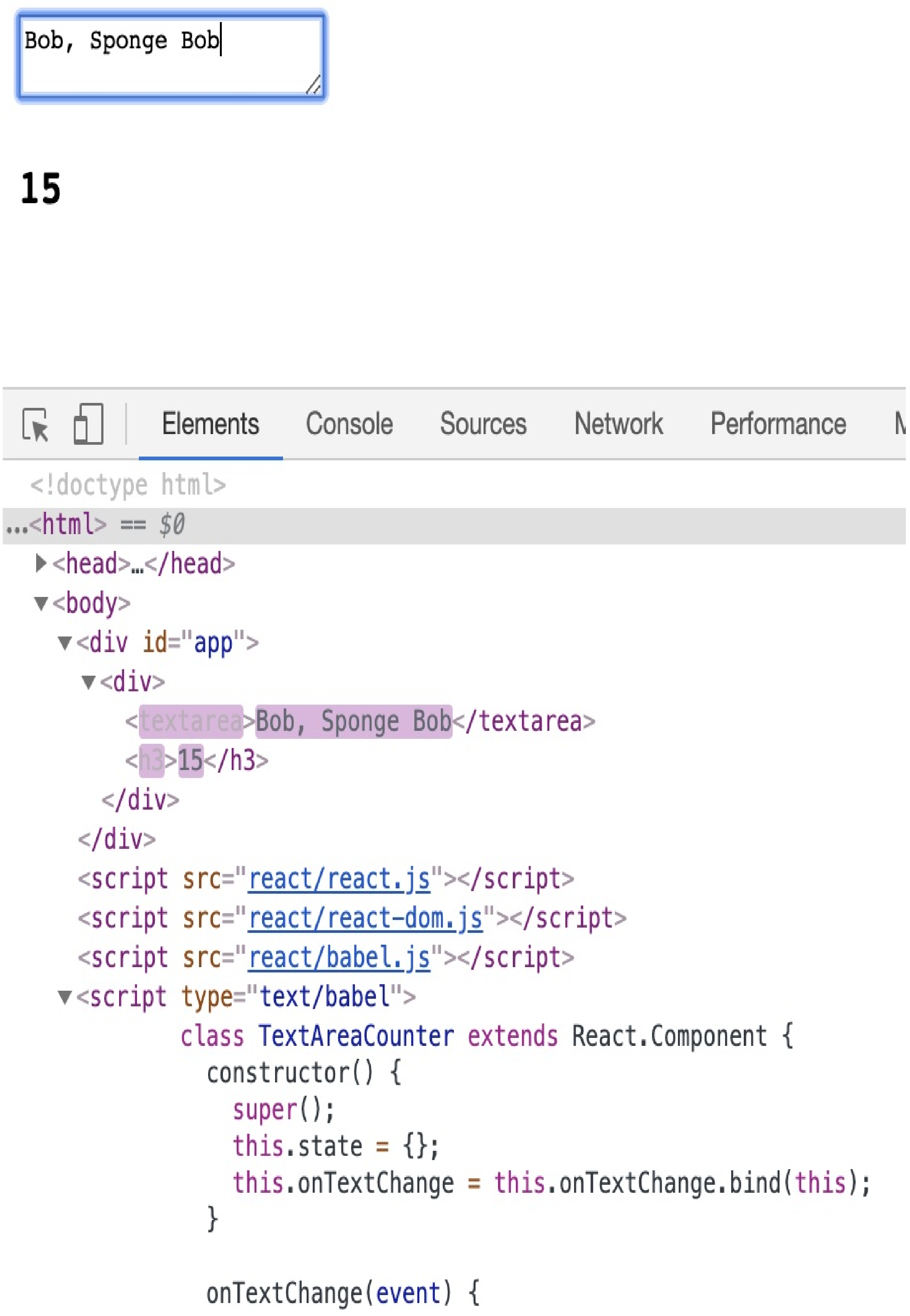
You always update the state with `this.setState()`, which takes an object and merges it with the already existing data in `this.state`. As you might guess, `onTextChange()` is an event handler that takes an `event` object and reaches into it to get the contents of the textarea input.

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

Props in Initial State: An Anti-Pattern

In the textarea example above it’s tempting to use `this.props` to set the initial `this.state`:

```
// Warning: Anti-pattern
this.state = {
  text: 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.

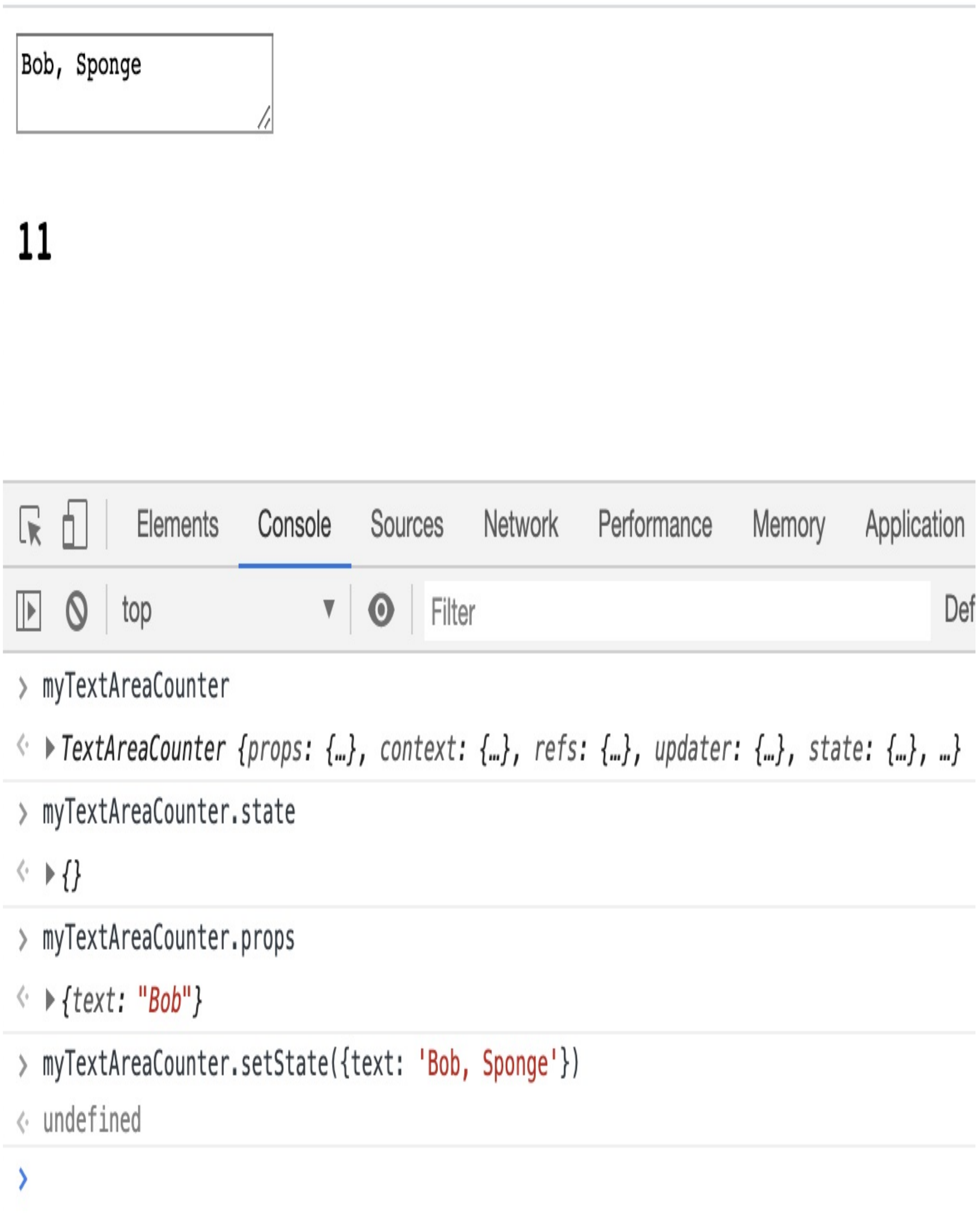
Accessing the Component from the Outside

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).


Bob, Sponge

11

```
> 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.

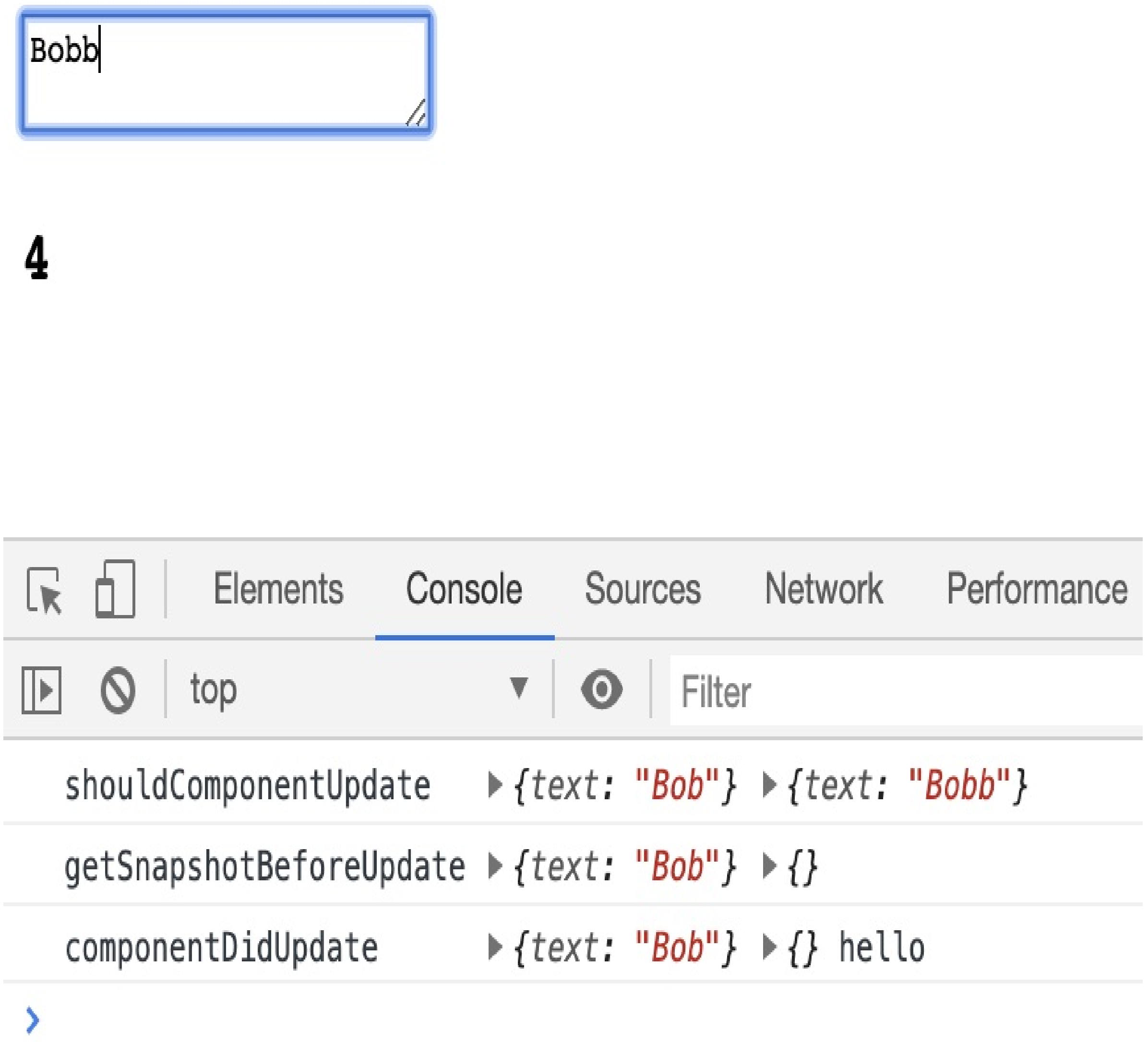
Lifecycle Example: Log It All

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.



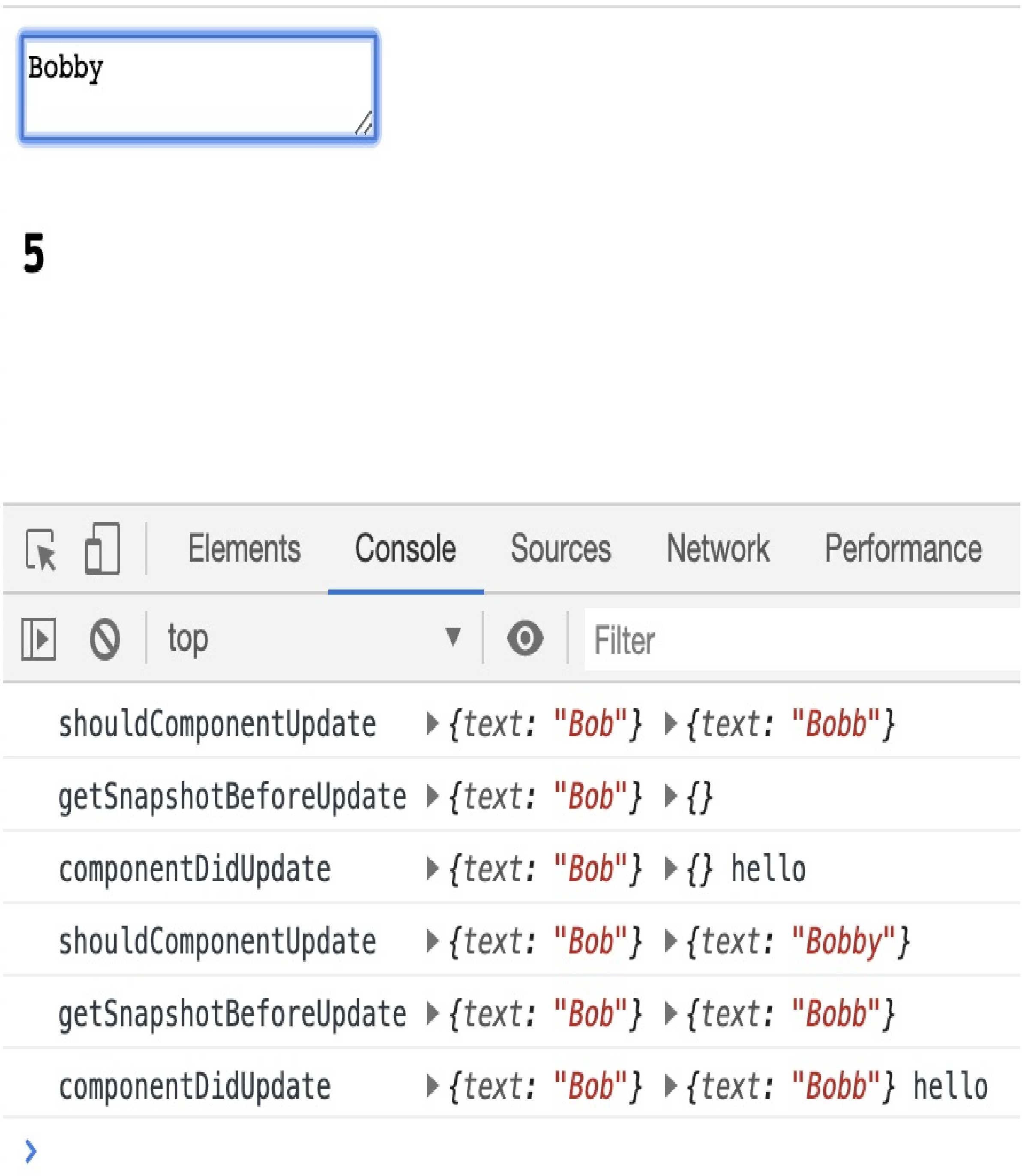
```
Bobb

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.


Bobby

5

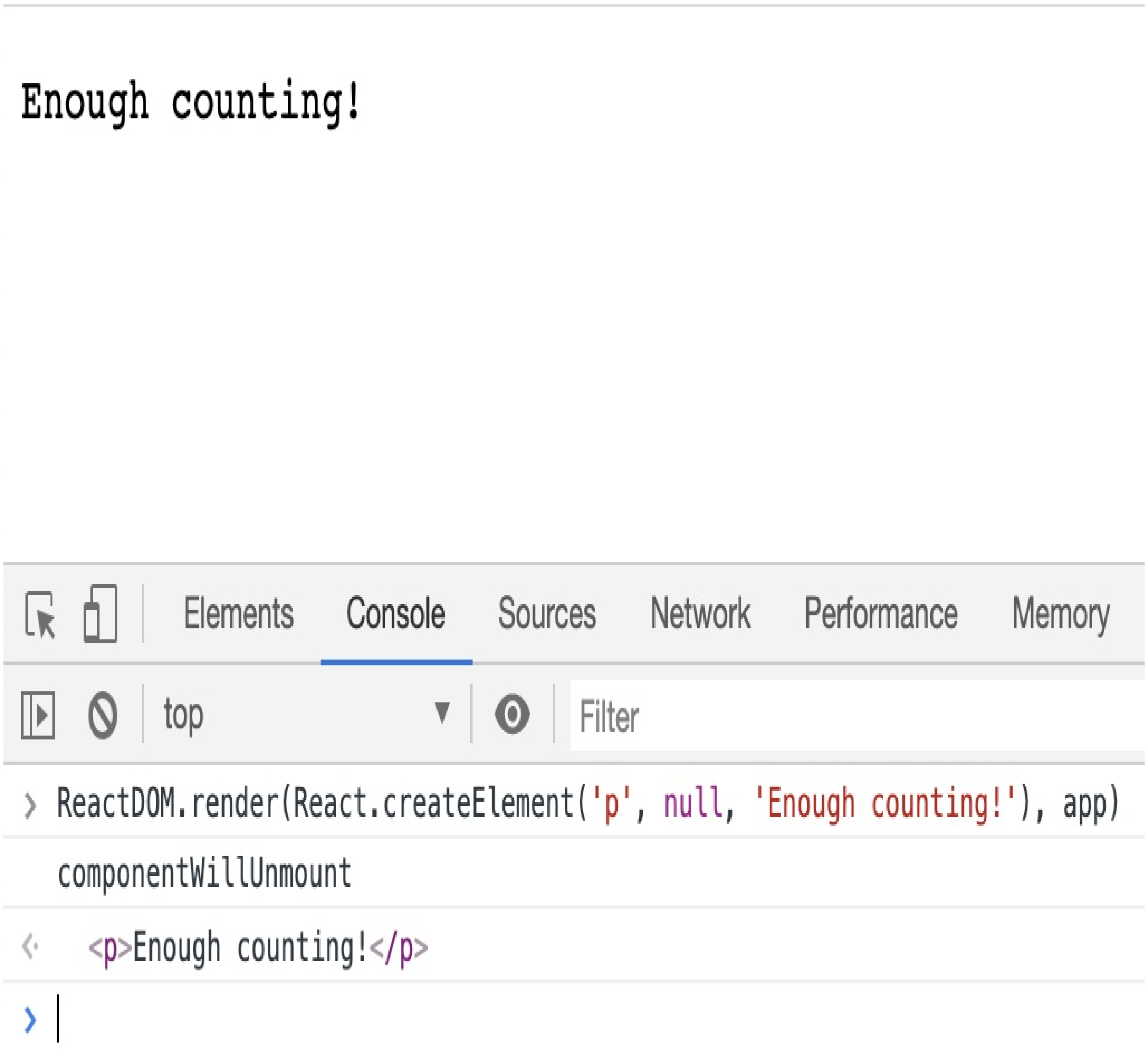
```
shouldComponentUpdate   ▶{text: "Bob"} ▶{text: "Bobb"}
getSnapshotBeforeUpdate ▶{text: "Bob"} ▶{}
componentDidUpdate      ▶{text: "Bob"} ▶{} hello
shouldComponentUpdate   ▶{text: "Bob"} ▶{text: "Bobby"}
getSnapshotBeforeUpdate ▶{text: "Bob"} ▶{text: "Bobb"}
componentDidUpdate      ▶{text: "Bob"} ▶{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 way to approach the management of state in 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.

Chapter 3. Excel: A Fancy Table Component

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at sstoo@ymail.com.

Now you know how to create custom react components, compose UI using generic DOM components as well as your own custom ones, set properties, maintain state, hook into the lifecycle of a component, and optimize performance by not rerendering when not necessary.

Let's put all of this together (and learn more about React while at it) by creating a more interesting component—a data table. Something like an early prototype of Microsoft Excel that lets you edit the contents of a data table, and also sort, search, and export the data as downloadable files.

Data First

Tables are all about the data, so the fancy table component (why not call it `Excel`?) should take an array of data and an array of headers that describe each column of data. For testing, let's grab a list of best-selling books from Wikipedia:

```
const headers = ['Book', 'Author', 'Language', 'Published', 'Sales'];
const data = [
  [
    'A Tale of Two Cities', 'Charles Dickens',
      'English', '1859', '200 million',
  ],
  [
    'Le Petit Prince (The Little Prince)', 'Antoine de Saint-Exupéry',
      'French', '1943', '150 million',
  ],
  [
    "Harry Potter and the Philosopher's Stone", 'J. K. Rowling',
      'English', '1997', '120 million',
  ],
  [
    'And Then There Were None', 'Agatha Christie',
      'English', '1939', '100 million',
  ],
  [
    'Dream of the Red Chamber', 'Cao Xueqin',
      'Chinese', '1791', '100 million',
  ],
  [
    'The Hobbit', 'J. R. R. Tolkien',
      'English', '1937', '100 million',
  ],
];
```

Table Headers Loop

The first step, just to get the new component off the ground, is to display only the headers of the table. Here’s what a bare-bones, yet a bit verbose, implementation might look like (`03.01.table-th-loop.html` in the book’s repository):

```
class Excel extends React.Component {
  render() {
    const headers = [];
    for (const idx in this.props.headers) {
      const title = this.props.headers[idx];
      headers.push(<th>{title}</th>);
    }
    return (
      <table>
        <thead>
          <tr>{headers}</tr>
        </thead>
      </table>
    );
  }
}
```

Now that you have a working component, here’s how to use it:

```
  ReactDOM.render(
    <Excel headers={headers} />,
    document.getElementById('app'),
  );
```

The result of this get-off-the-ground example is shown in Figure 3-1. There’s a little bit of CSS used, which is of no concern for the purposes of this discussion but you can find it in `03.table.css` in the book’s repo.

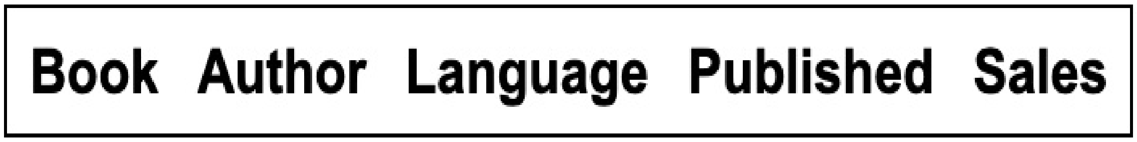


Figure 3-1. Rendering table headers

The `return` part of the component is fairly simple. It looks just like an HTML table except for the `headers` array.

```
  return (
    <table>
      <thead>
        <tr>{headers}</tr>
      </thead>
    </table>
  );
```

As you’ve seen in the previous chapter you can open curly braces in your JSX and put any JavaScript value or expression in there. If this value happens to be an array as in the case above, the JSX parser treats it as if you passed each element of the array individually, like `{headers[0]}{headers[1]}...`.

In this example the elements of the `headers` array contain more JSX content and this is perfectly fine. The loop before the `return` populates the `headers` array with JSX values which, if you were hardcoding the data, would look like so:

```
const headers = [
  <th>Book</th>,
  <th>Author</th>,
  // ...
];
```

You see how you can have JavaScript in `{}` within JSX and you can nest these `{}`-s as deep as you need. This is part of the beauty of React—you use JavaScript to create your UI and all the power of JavaScript is available to you. Loops and conditions all work as usual and you don’t need to learn another “templating” language or syntax to build the UI.

Table Headers Loop, a terse version

The example above worked fine (let’s call it v1 for Version 1) but let’s see how you can accomplish the same with less code. Let’s move the loop inside the JSX returned at the end. In essence the whole `render()` method becomes a single `return` (see `03.02.table-th-map.html`).

```
class Excel extends React.Component {
  render() {
    return (
      <table>
        <thead>
          <tr>
            {this.props.headers.map(title => <th>{title}</th>)}
          </tr>
        </thead>
      </table>
    );
  }
}
```

Here you see how the array of header content is produced by calling `map()` on the data passed via `this.props.headers`. A `map()` call takes an input array, executes a callback function on each element and creates a new array.

In the example above the callback uses the tersest *arrow functions* syntax. If this is a little too cryptic for your taste, let's call it v2 explore a few other options.

Here's v3: a more verbose `map()` loop using generous indentation and a *function expression* instead of an arrow function:

```
{
  this.props.headers.map(
    function(title) {
      return <th>{title}</th>;
    }
  )
}
```

Next, a version (v4) which is a little less verbose version going back to using an arrow functions:

```
{
  this.props.headers.map(
    (title) => {
      return <th>{title}</th>;
    }
  )
}
```

…which can be formatted with less indentation to v5:

```
{this.props.headers.map((title) => {
  return <th>{title}</th>;
})}
```

You can choose your preferred way of iterating over arrays to produce JSX context based on personal preference and complexity of the content to be rendered. Simple data is conveniently looped over inline in the JSX (v2 through v5). If the type of data is a little too much for an inline `map()` you may find it more readable to have the content generated at the top of the render function and keep the JSX simple, in a way separating data from presentation (v1 is an example). Sometimes too many inline expressions can be confusing when keeping track of all closing `)` and `}`s.

As to v2 vs. v5, they are the same except v5 has extra `()` around the callback arguments and `{}` wrapping the callback function body. While both of these are optional, they make future changes a little easier to parse in a diff/code review context or while debugging. For example adding a new line to the function body (maybe a temporary `console.log()`) in v5 is just that - adding a new line. While in v2 a new line also requires adding `{}` and reformatting and reindenting the code.

Debugging the Console Warning

If you look in the the browser console when loading the previous two examples (`03.01.table-th-loop.html` and `03.01.table-th-map.html`) you can see a warning. The warning says:

```
Warning: Each child in a list should have a unique "key" prop.
Check the render method of `Excel`.
```

What is it about and how do you fix it? As the warning message says, React wants you to provide a unique ID for the array elements so it can update them more efficiently later on. To fix the warning, you add a `key` property to each header. The values of this new property can be anything as long they are unique for each element. Here you can use the index of the array element (0, 1, 2…):

```
// before
for (const idx in this.props.headers) {
  const title = this.props.headers[idx];
  headers.push(<th>{title}</th>);
}

// after - 03.03.table-th-loop-key.html
for (const idx in this.props.headers) {
  const title = this.props.headers[idx];
  headers.push(<th key={idx}>{title}</th>);
}
```

The keys only need to be unique inside each array loop, not unique in the whole React application, so values of 0, 1 and so on are perfectly acceptable.

The same fix for the inline version (v5) takes the element index from the second argument passed to the callback function:

```
// before
<tr>
  {this.props.headers.map((title) => {
    return <th>{title}</th>;
  })}
</tr>

// after - 03.04.table-th-map-key.html
<tr>
  {this.props.headers.map((title, idx) => {
    return <th key={idx}>{title}</th>;
  })}
</tr>
```

Adding <td> Content

Now that you have a pretty table head, it’s time to add the body. The data to be rendered is a two-dimentional array (rows and columns) that looks like:

```
const data = [
  [
    'A Tale of Two Cities', 'Charles Dickens',
      'English', '1859', '200 million',
  ],
  ....
];
```

To pass the data to the `<Excel>`, let’s use a new prop called `initialData`. Why “initial” and not just “data”? As touched briefly in the previous chapter, it’s about managing expectations. The caller of your `Excel` component should be able to pass data to initialize the table. But later, as the table lives on, the data will change, because the user is able to sort, edit, and so on. In other words, the *state* of the component will change. So let’s use `this.state.data` to keep track of the changes and use `this.props.initialData` to let the caller initialize the component.

Rendering a new `Excel` component would look like so:

```
ReactDOM.render(
  <Excel headers={headers} initialData={data} />,
  document.getElementById('app'),
);
```

Next you need to add a constructor to set the initial state from the given data. The constructor receives `props` as an argument and also needs to call its parent's constructor via `super()`:

```
constructor(props) {
  super();
  this.state = {data: props.initialData};
}
```

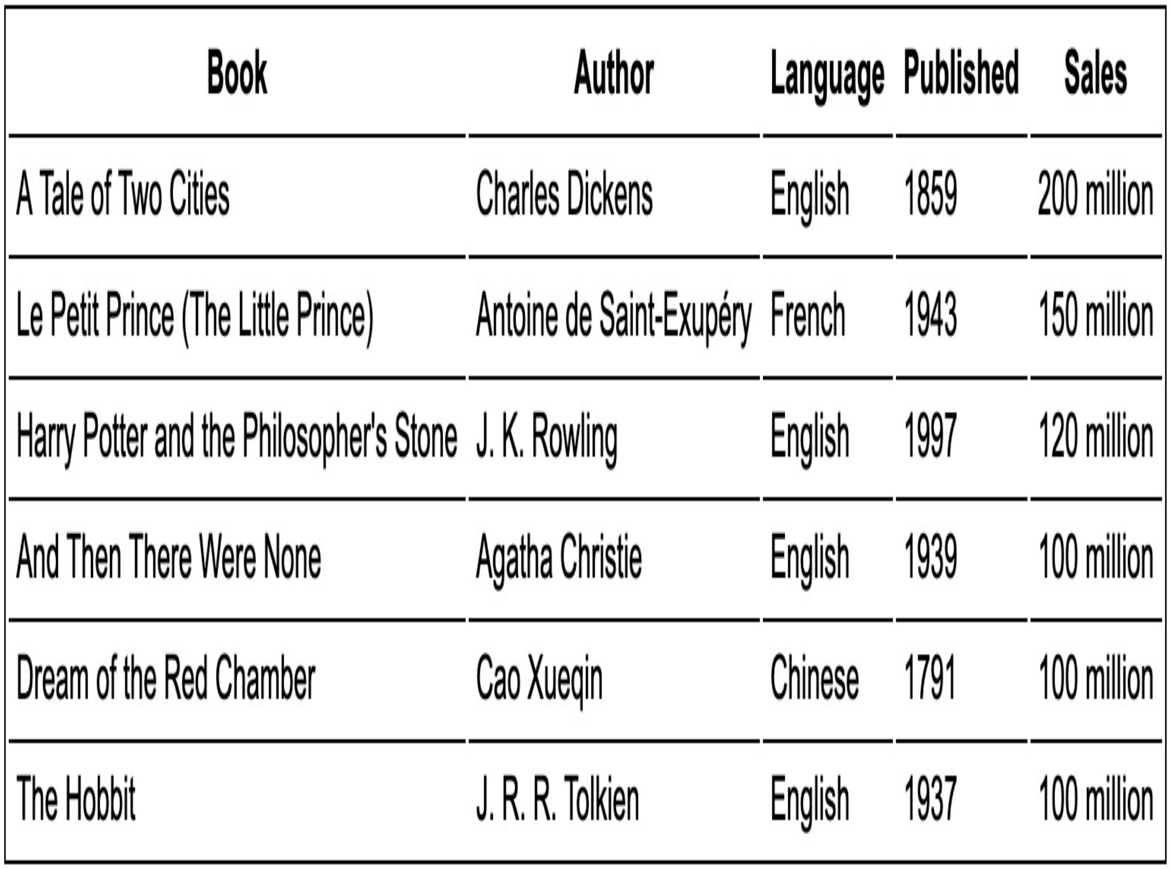
On to rendering `this.state.data`. The data is two-dimensional, so you need two loops: one that goes through rows and one that goes through the data (cells) for each row. This can be accomplished using two of the same `.map()` loops you already know how to use:

```
{this.state.data.map((row, idx) => (
  <tr key={idx}>
    {row.map((cell, idx) => (
      <td key={idx}>{cell}</td>
    ))}
  </tr>
))}
```

As you can see both loops need `key={idx}` and in this case the name `idx` was recycled for use as local variables within each loop.

A complete implementation could look like this (result shown in Figure 3-2):

```
class Excel extends React.Component {
  constructor(props) {
    super();
    this.state = {data: props.initialData};
  }
  render() {
    return (
      <table>
        <thead>
          <tr>
            {this.props.headers.map((title, idx) => (
              <th key={idx}>{title}</th>
            ))}
          </tr>
        </thead>
        <tbody>
          {this.state.data.map((row, idx) => (
            <tr key={idx}>
              {row.map((cell, idx) => (
                <td key={idx}>{cell}</td>
              ))}
            </tr>
          ))}
        </tbody>
      </table>
    );
  }
}
```



Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
And Then There Were None	Agatha Christie	English	1939	100 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1791	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million

Figure 3-2. Rendering the whole table (`03.05.table-th-td.html`)

Prop types

The ability to specify the types of variables you work with - string, number, boolean, etc. - doesn’t exist in the JavaScript language. But developers coming from other languages, and those working on larger projects with many other developers, do miss it. Two popular options exist that offer you to write

JavaScript with types - Flow and TypeScript. You can certainly use these to write React applications. But another option exist, which is limitted to only specifying the types of props your component expects: *prop types*. They were a part of React itself initially but at a point have been moved to a separate library.

Prop types allow you to be more specific as to what data `Excel` takes and as a result surface an error to the developer early on. You can setup the prop types like so (`03.06.table-th-td-prop-types.html`):

```
Excel.propTypes = {
  headers: PropTypes.arrayOf(PropTypes.string),
  initialData: PropTypes.arrayOf(PropTypes.arrayOf(PropTypes.string)),
};
```

This means that `headers` prop is expected to be an array of strings and `initialData` is expected to be an array where each element is another array of string elements.

To make this code work you need to grab the library which exposes the `PropTypes` global variable, just like you did in the beginning of Chapter 1:

```
curl -L https://unpkg.com/browse/prop-types@15.7.2/prop-types.js >
~/reactbook/react/prop-types.js
```

Then in the HTML you include the new library together with the other ones:

```
<script src="react/react.js"></script>
<script src="react/react-dom.js"></script>
<script src="react/babel.js"></script>
<script src="react/prop-types.js"></script>
<script type="text/babel">
  class Excel extends React.Component {
    /* ... */
  }
</script>
```

Now you can test how it all works by changing `headers`, for example:

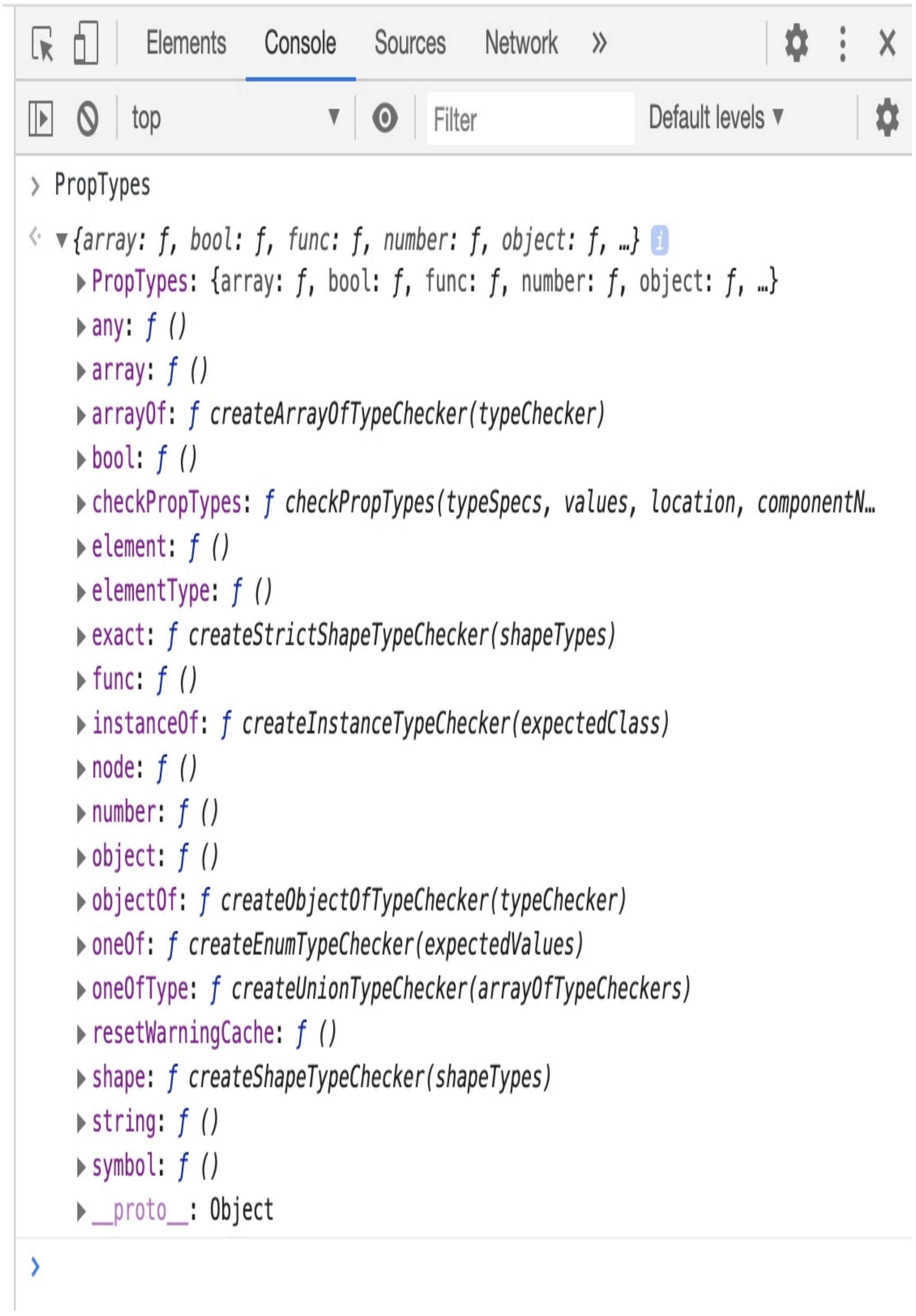
```
// before
const headers = ['Book', 'Author', 'Language', 'Published', 'Sales'];
// after
const headers = [0, 'Author', 'Language', 'Published', 'Sales'];
```

Now when you load the page (`03.06.table-th-td-prop-types.html` in the repo) you can see in the console:

```
Warning: Failed prop type: Invalid prop `headers[0]` of type `number`
supplied to `Excel`, expected `string`.
```

Now that’s strict!

To explore what other `PropTypes` exist just type `PropTypes` in the console (Figure 3-3).



```
> PropTypes
<- ▼{array: f, bool: f, func: f, number: f, object: f, …}
   ▶PropTypes: {array: f, bool: f, func: f, number: f, object: f, …}
   ▶any: f ()
   ▶array: f ()
   ▶arrayOf: f createArrayOfTypeChecker(typeChecker)
   ▶bool: f ()
   ▶checkPropTypes: f checkPropTypes(typeSpecs, values, location, componentN…
   ▶element: f ()
   ▶elementType: f ()
   ▶exact: f createStrictShapeTypeChecker(shapeTypes)
   ▶func: f ()
   ▶instanceOf: f createInstanceTypeChecker(expectedClass)
   ▶node: f ()
   ▶number: f ()
   ▶object: f ()
   ▶objectOf: f createObjectOfTypeChecker(typeChecker)
   ▶oneOf: f createEnumTypeChecker(expectedValues)
   ▶oneOfType: f createUnionTypeChecker(arrayOfTypeCheckers)
   ▶resetWarningCache: f ()
   ▶shape: f createShapeTypeChecker(shapeTypes)
   ▶string: f ()
   ▶symbol: f ()
   ▶__proto__: Object
>
```

Figure 3-3. Exploring PropTypes

Can You Improve the Component?

Allowing only string data is a bit too restrictive for a generic Excel spreadsheet. As an exercise for your own amusement, you can change this example to allow more data types (`PropTypes.any`) and then render differently depending on the type (e.g., align numbers to the right).

Sorting

How many times have you seen a table on a web page that you wished was sorted differently? Luckily, it's trivial to do this with React. Actually, this is an example where React shines, because all you need to do is sort the `data` array and all the UI updates are handled for you.

For convenience and readability, all the sorting logic is in a `sort()` method in the `Excel` class. Once you create it, two bits of plumbing are necessary. First, add a click handler to the header row:

```
<thead onClick={this.sort}>
```

And then bind `this.sort` in the constructor as you did in Chapter 2:

```
class Excel extends React.Component {
  constructor(props) {
    super();
    this.state = {data: props.initialData};
    this.sort = this.sort.bind(this);
  }
  sort(e) {
    // TODO: implement me
  }
  render() { /* ...*/}
}
```

Now let's implement the `sort()` method. You need to know which column to sort by, which can conveniently be retrieved by using the `cellIndex` DOM property of the event target (the event target is a table header `<th>`):

```
const column = e.target.cellIndex;
```

> **NOTE**
>
> You may have rarely seen `cellIndex` used in app development. It’s a property defined as early as DOM Level 1 (circa 1998) as “The index of this cell in the row” and later on made read-only in DOM Level 2.

You also need a *copy* of the data to be sorted. Otherwise, if you use the array’s `sort()` method directly, it modifies the array. Meaning that calling `this.state.data.sort()` will modify `this.state`. As you know already, `this.state` should not be modified directly, but only through `setState()`.

Various ways exist in JavaScript to make a *shallow copy* of an object or an array (arrays are obects in JavaScript), e.g. `Object.assign()` or using the spread operator `{...state}`. However there in no built-in way to do a *deep copy* of an object. A quick way to implement a soution is to encode an object to a JSON string and then decode is back to an object. Let’s use this approach for brevity, though be aware that it fails if your object/array contains `Date` objects.

```
function clone(o) {
  return JSON.parse(JSON.stringify(o));
}
```

With the handy `clone()` utility function you make a copy of the array before you start manipulating it:

```
// copy the data
const data = clone(this.state.data);
```

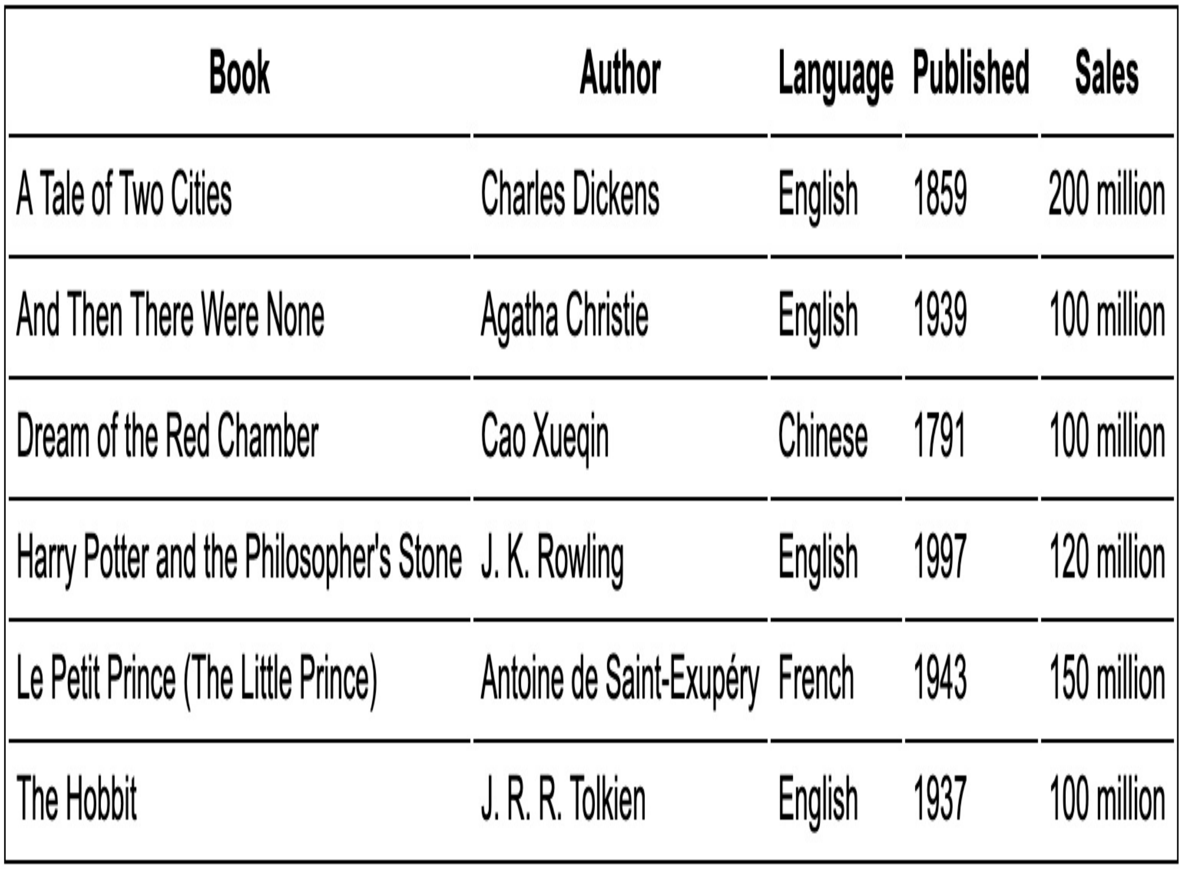
The actual sorting is done via a callback to the `sort()` method:

```
data.sort((a, b) => {
  return a[column] > b[column] ? 1 : -1;
});
```

Finally, this line sets the state with the new, sorted data:

```
this.setState({
  data,
});
```

Now, when you click a header, the contents get sorted alphabetically (Figure 3-4).



Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	200 million
And Then There Were None	Agatha Christie	English	1939	100 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1791	100 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million

Figure 3-4. Sorting by book title (`03.07.table-sort.html`)

And this is it—you don't have to touch the UI rendering at all. In the `render()` method, you've already defined once and for all how the component should look given some data. When the data changes, so does the UI; however, this is no longer your concern.

NOTE

The example used the ECMAScript *property value shorthands* feature where `this.setState({data})` is a shorter way of expressing `this.setState({data: data})` by skipping the key when it has the same name as a variable.

Can You Improve the Component?

The example above uses pretty simple sorting, just enough to be relevant to the React discussion. You can go as fancy as you need, parsing the content to see if the values are numeric, with or without a unit of measure and so on.

Sorting UI Cues

The table is nicely sorted, but it’s not clear which column it’s sorted by. Let’s update the UI to show arrows based on the column being sorted. And while at it, let’s implement descending sorting too.

To keep track of the new state, you need two new properties added to `this.state`:

`this.state.sortby`

The index of the column currently being sorted

`this.state.descending`

A boolean to determine ascending versus descending sorting

The constructor can now look like:

```
constructor(props) {
  super();
  this.state = {
    data: props.initialData,
    sortby: null,
    descending: false,
  };
  this.sort = this.sort.bind(this);
}
```

In the `sort()` function, you have to figure out which way to sort. Default is ascending (A to Z), unless the index of the new column is the same as the current sort-by column and the sorting is not already descending from a previous click on the header:

```
const column = e.target.cellIndex;
const data = clone(this.state.data);
const descending = this.state.sortby === column &&
!this.state.descending;
```

You also need a small tweak to the sorting callback:

```
data.sort((a, b) => {
  return descending
    ? (a[column] < b[column] ? 1 : -1)
    : (a[column] > b[column] ? 1 : -1);
});
```

And finally, you need to set the new state:

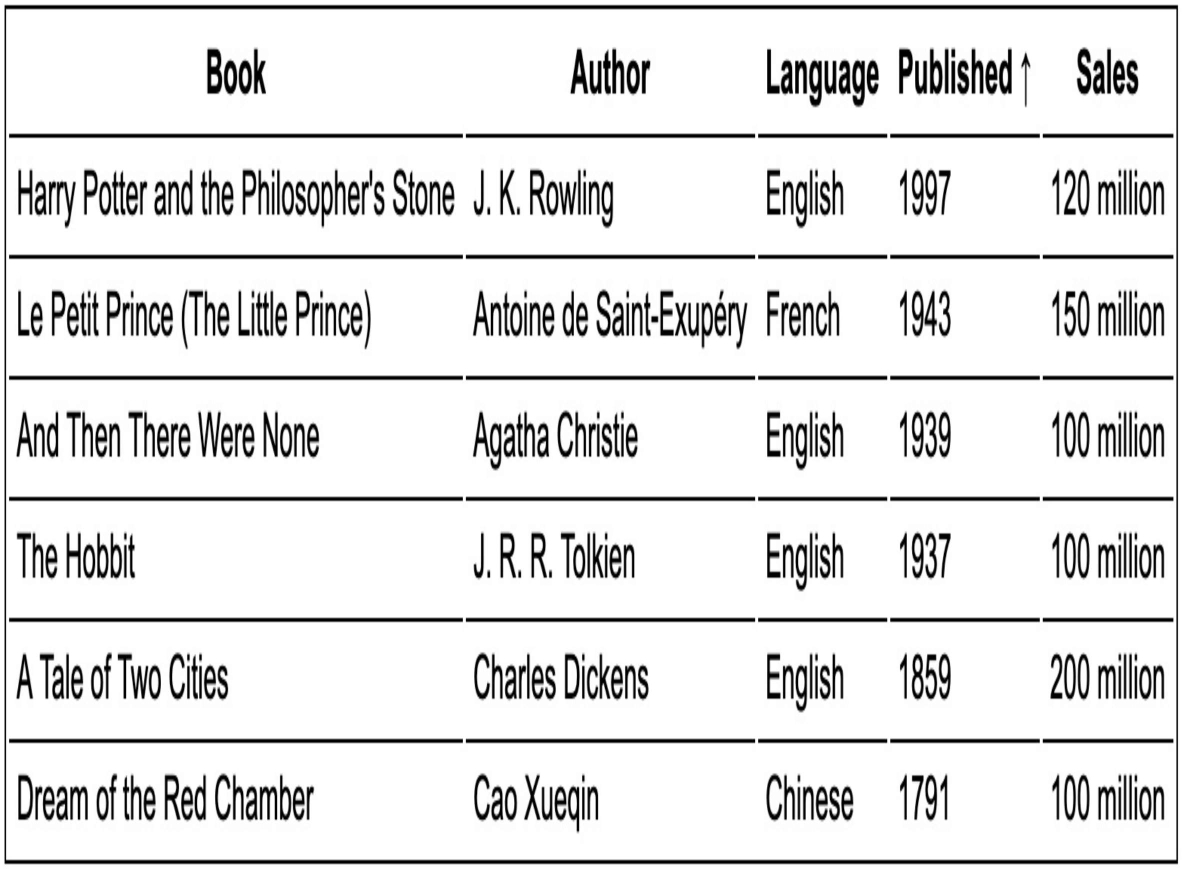
```
this.setState({
  data,
  sortby: column,
  descending,
});
```

At this point the descending ordering works. Clicking on the table headers sorts ascending first, then descending and then keeps on toggling the two.

The only thing left is to update the `render()` function to indicate sorting direction. For the currently sorted column, let's just add an arrow symbol to the title. Now the headers loop looks like:

```
{this.props.headers.map((title, idx) => {
  if (this.state.sortby === idx) {
    title += this.state.descending ? ' \u2191' : ' \u2193'
  }
  return <th key={idx}>{title}</th>
})}
```

Now the sorting is feature-complete—people can sort by any column, they can click once for ascending and once more for descending ordering, and the UI updates with the visual cue (Figure 3-5).



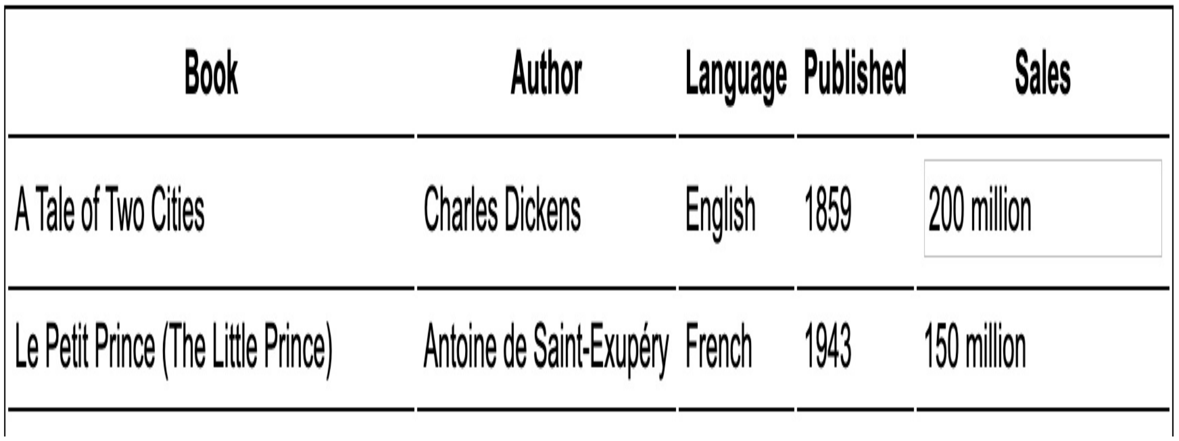
Book	Author	Language	Published ↑	Sales
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
And Then There Were None	Agatha Christie	English	1939	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million
A Tale of Two Cities	Charles Dickens	English	1859	200 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1791	100 million

Figure 3-5. Ascending/descending sorting

Editing Data

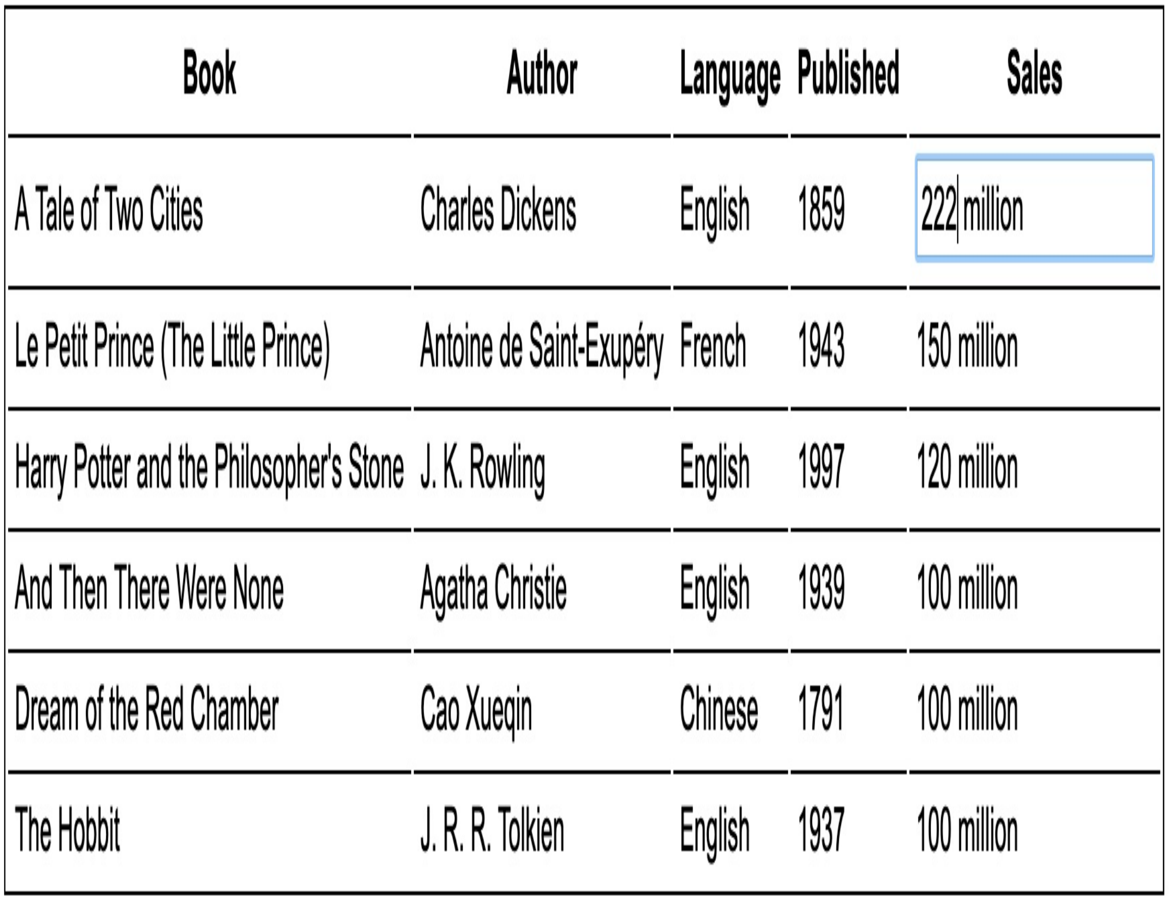
The next step for the `Excel` component is to give people the option to edit the data in the table. One solution could work like so:

1. You double-click a cell. `Excel` figures out which cell was clicked and turns its content from simple text into an input field pre-filled with the content (Figure 3-6).
2. You edit the content (Figure 3-7).
3. You hit Enter. The input field is gone, and the table is updated with the new text (Figure 3-8).



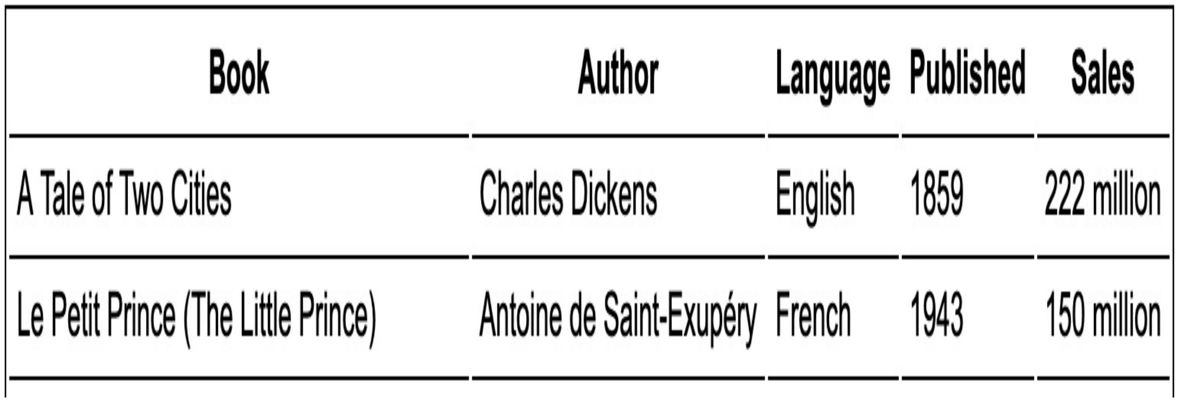
Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million

Figure 3-6. Table cell turns into an input field on double-click



Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	222 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
And Then There Were None	Agatha Christie	English	1939	100 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1791	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million

Figure 3-7. Edit the content



Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	222 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million

Figure 3-8. Content updated on pressing Enter

Editable Cell

The first thing to do is set up a simple event handler. On double-click, the component “remembers” the selected cell:

```
<tbody onDoubleClick={this.showEditor}>
```

NOTE

Note the friendlier, easier-to-read `onDoubleClick`, as opposed to W3C’s `ondblclick`.

Let’s see what `showEditor` looks like:

```
showEditor(e) {
  this.setState({
    edit: {
      row: parseInt(e.target.parentNode.dataset.row, 10),
      column: e.target.cellIndex,
    },
  });
}
```

What’s happening here?

- The function sets the `edit` property of `this.state`. This property is `null` when there’s no editing going on and then turns into an object with properties `row` and `column`, which contain the row index and the column index of the cell being edited. So if you double-click the very first cell, `this.state.edit` gets the value `{row: 0, column: 0}`.

- To figure out the column index, you use the same `e.target.cellIndex` as before, where `e.target` is the `<td>` that was double-clicked.
- There’s no `rowIndex` coming for free in the DOM, so you need to do it yourself via a `data-` attribute. Each row should have a `data-row` attribute with the row index, which you can `parseInt()` to get the index back.

Let’s take care of a few prerequisites. First, the `edit` property didn’t exist before and should also be initialized in the constructor. While dealing with the constructor, let’s bind the `showEditor()` and `save()` methods. The `save()` is the one to do the data update once the user is done editing. The updated constructor looks like this:

```
constructor(props) {
  super();
  this.state = {
    data: props.initialData,
    sortby: null,
    descending: false,
    edit: null, // {row: index, column: index}
  };
  this.sort = this.sort.bind(this);
  this.showEditor = this.showEditor.bind(this);
  this.save = this.save.bind(this);
}
```

The property `data-row` is something you need so you can keep track of row indexes. You can get the index from the array index while looping. Previously you saw how `idx` was reused as a local variable by both row and column loops. Let’s rename it and use `rowidx` and `columnidx` for clarity.

The whole `<tbody>` construction could look like:

```
<tbody onDoubleClick={this.showEditor}>
  {this.state.data.map((row, rowidx) => (
    <tr key={rowidx} data-row={rowidx}>
      {row.map((cell, columnidx) => {

        // TODO - turn `content` into an input if the `columnidx`
        // and the `rowidx` match the one being edited;
        // otherwise, just show the text content

        return <td key={columnidx}>{cell}</td>;
      })}
    </tr>
  ))}
</tbody>
```

Finally, let’s do what the `TODO` says — make an input field when required. The whole `render()` function is called again just because of the `setState()` call that sets the `edit` property. React rerenders the table, which gives you the chance to update the table cell that was double-clicked.

Input Field Cell

Let’s look at the code to replace the `TODO` comment. First, remember the edit state for brevity:

```
  const edit = this.state.edit;
```

Check if the `edit` is set and if so, whether this is the exact cell being edited:

```
  if (edit && edit.row === rowidx && edit.column === columnidx) {
    // ...
  }
```

If this is the target cell, let’s make a form and an input field with the content of the cell:

```
  cell = (
    <form onSubmit={this.save}>
      <input type="text" defaultValue={cell} />
    </form>
  );
```

As you see, it’s a form with a single input and the input is pre-filled with the text content. When the form is submitted, the submission event is trapped in the `save()` method.

Saving

The last piece of the editing puzzle is saving the content changes after the user is done typing and has submitted the form (via the Enter key):

```
save(e) {
  e.preventDefault();
  // ... do the save
}
```

After preventing the default behavior (so the page doesn't reload), you need to get a reference to the input field. The event target `e.target` is the form and its first and only child is the input:

```
const input = e.target.firstChild;
```

Clone the data, so you don't manipulate `this.state` directly:

```
const data = clone(this.state.data);
```

Update the piece of data given the new value and the column and row indices stored in the `edit` property of the `state`:

```
data[this.state.edit.row][this.state.edit.column] = input.value;
```

Finally, set the state, which causes rerendering of the UI:

```
this.setState({
  edit: null,
  data,
});
```

And with this, the table is now editable. For a complete listing see `03.09.table-editable.html`

Conclusion and Virtual DOM Diffs

At this point, the editing feature is complete. It didn't take too much code. All

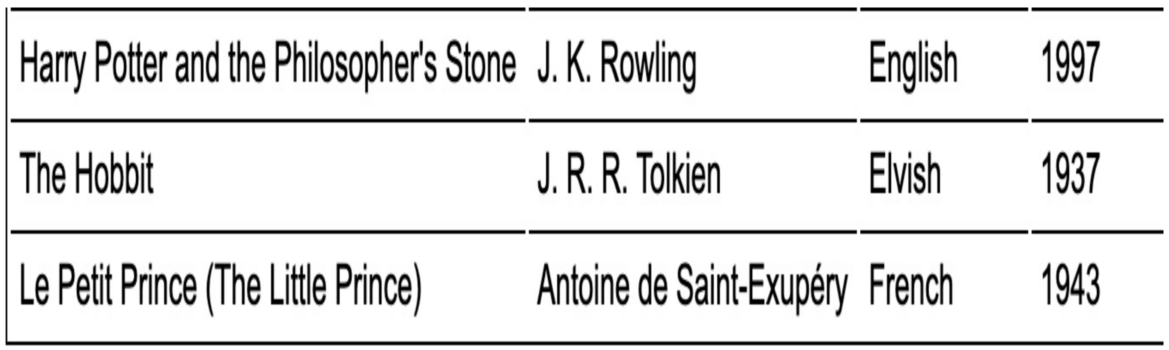
At this point, the editing feature is complete. It didn't take too much code. All you needed was to:

- Keep track of which cell to edit via `this.state.edit`
- Render an input field when displaying the table if the row and column indices match the cell the user double-clicked
- Update the data array with the new value from the input field

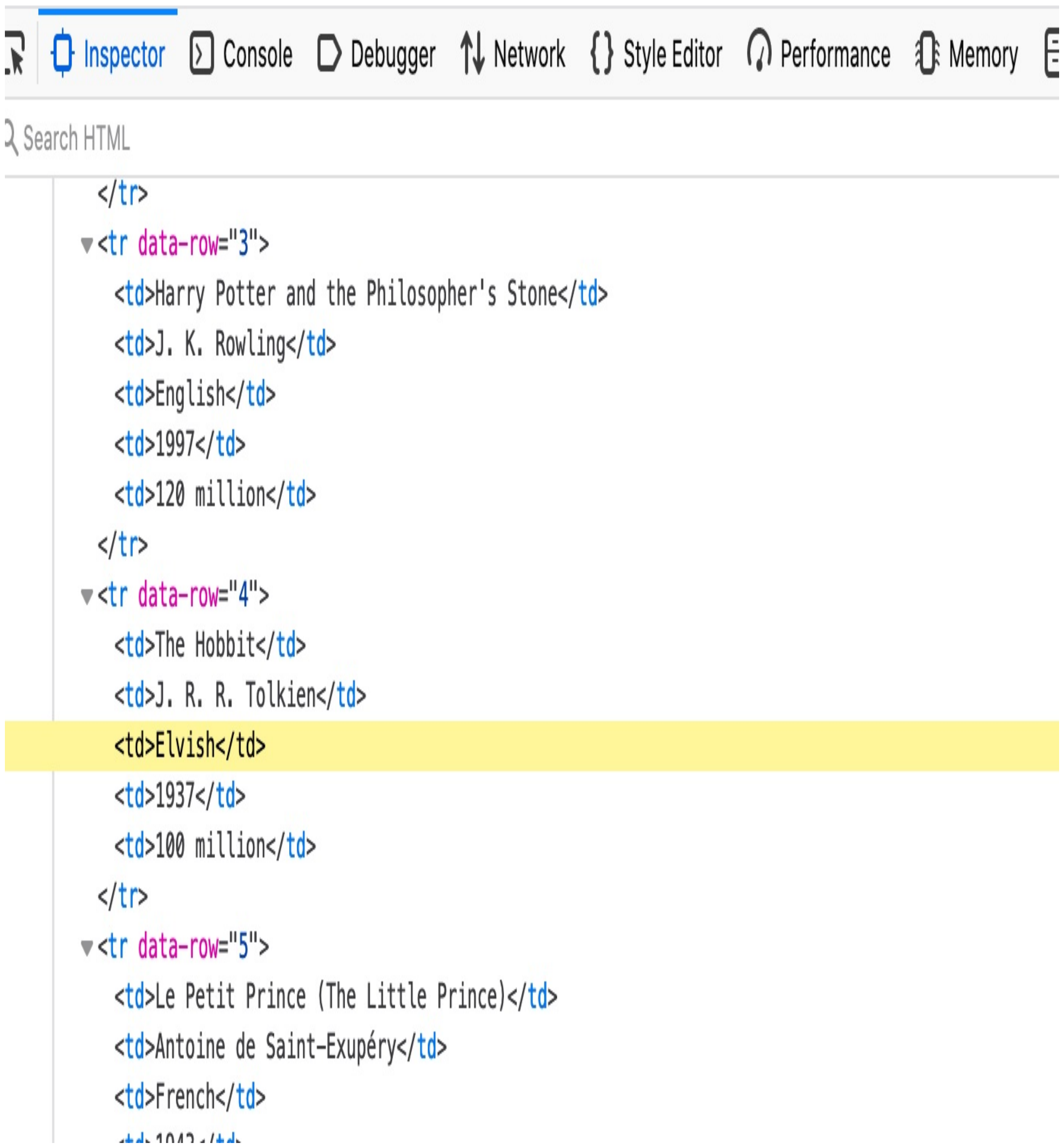
As soon as you `setState()` with the new data, React calls the component's `render()` method and the UI magically updates. It may look like it won't be particularly efficient to render the whole table for just one cell's content change. And in fact, React only updates a single cell.

If you open your browser's dev tools, you can see which parts of the DOM tree are updated as you interact with your application. In Figure 3-9, you can see the dev tools highlighting the DOM change after changing The Hobbit's language from English to Elvish.

Behind the scenes, React calls your `render()` method and creates a lightweight tree representation of the desired DOM result. This is known as a *virtual DOM tree*. When the `render()` method is called again (after a call to `setState()`, for example), React takes the virtual tree before and after and computes a diff. Based on this diff, React figures out the minimum required DOM operations (e.g., `appendChild()`, `textContent`, etc.) to carry on that change into the browser's DOM.



Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997
The Hobbit	J. R. R. Tolkien	Elvish	1937
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943



```
    </tr>
  ▼<tr data-row="3">
     <td>Harry Potter and the Philosopher's Stone</td>
     <td>J. K. Rowling</td>
     <td>English</td>
     <td>1997</td>
     <td>120 million</td>
    </tr>
  ▼<tr data-row="4">
     <td>The Hobbit</td>
     <td>J. R. R. Tolkien</td>
     <td>Elvish</td>
     <td>1937</td>
     <td>100 million</td>
    </tr>
  ▼<tr data-row="5">
     <td>Le Petit Prince (The Little Prince)</td>
     <td>Antoine de Saint-Exupéry</td>
     <td>French</td>
     <td>1943</td>
```

Figure 3-9. Highlighting DOM changes

In Figure 3-9, there is only one change required to the cell and it's not necessary to rerender the whole table. By computing the minimum set of changes and batching DOM operations, React "touches" the DOM lightly, as it's a known problem that DOM operations are slow (compared to pure JavaScript operations, function calls, etc.) and are often the bottleneck in rich web applications' rendering performance.

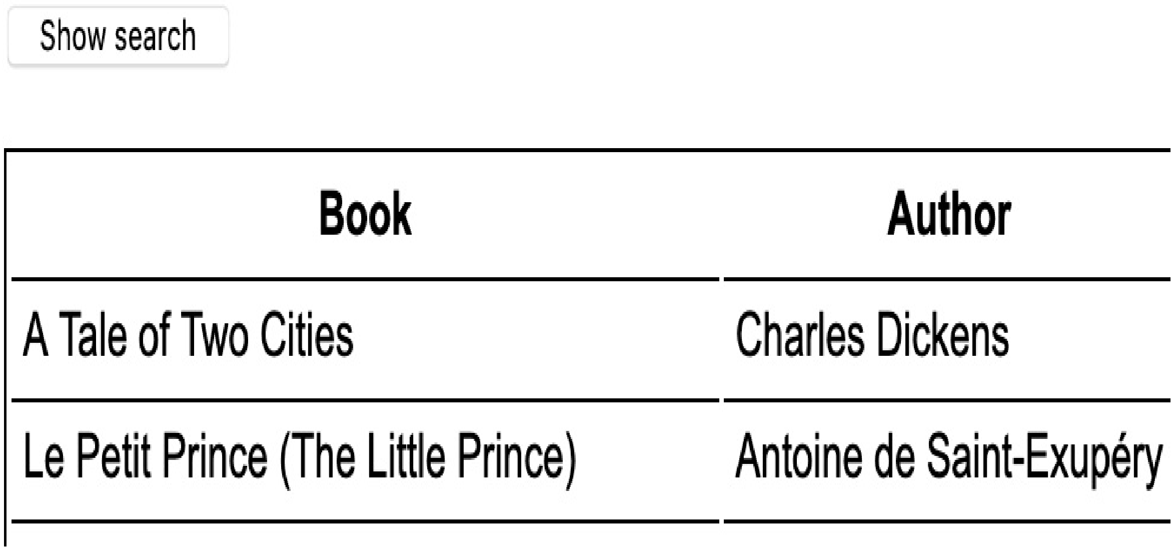
React has your back when it comes to performance and updating the UI by:

- Touching the DOM lightly
- Using event delegation for user interactions

Search

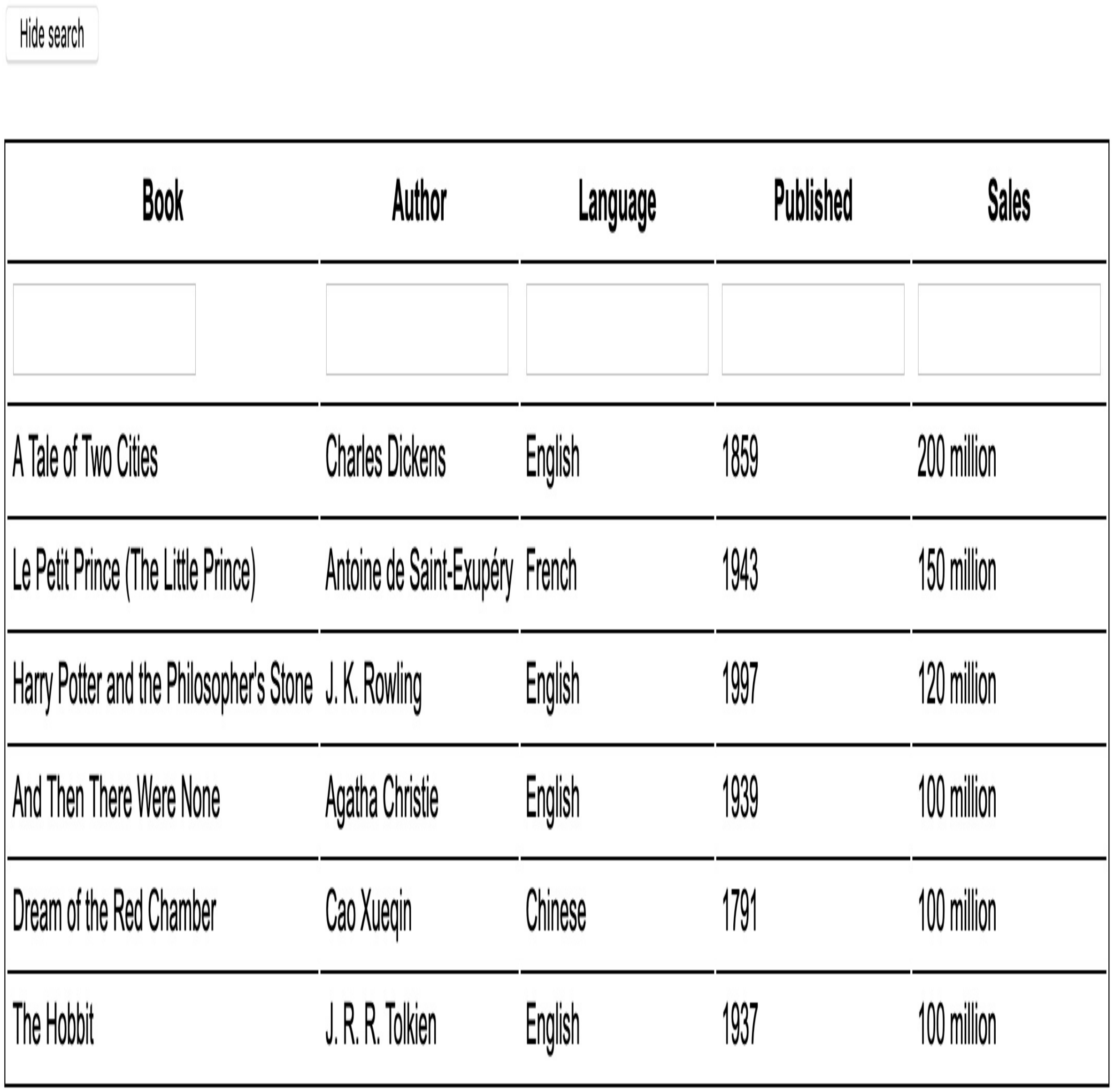
Next, let's add a search feature to the `Excel` component that allows users to filter the contents of the table. Here's the plan:

- Add a button to toggle the new feature on and off (Figure 3-10)
- If the search is on, add a row of inputs where each one searches in the corresponding column (Figure 3-11)
- As a user types in an input box, filter the array of `state.data` to only show the matching content (Figure 3-12)



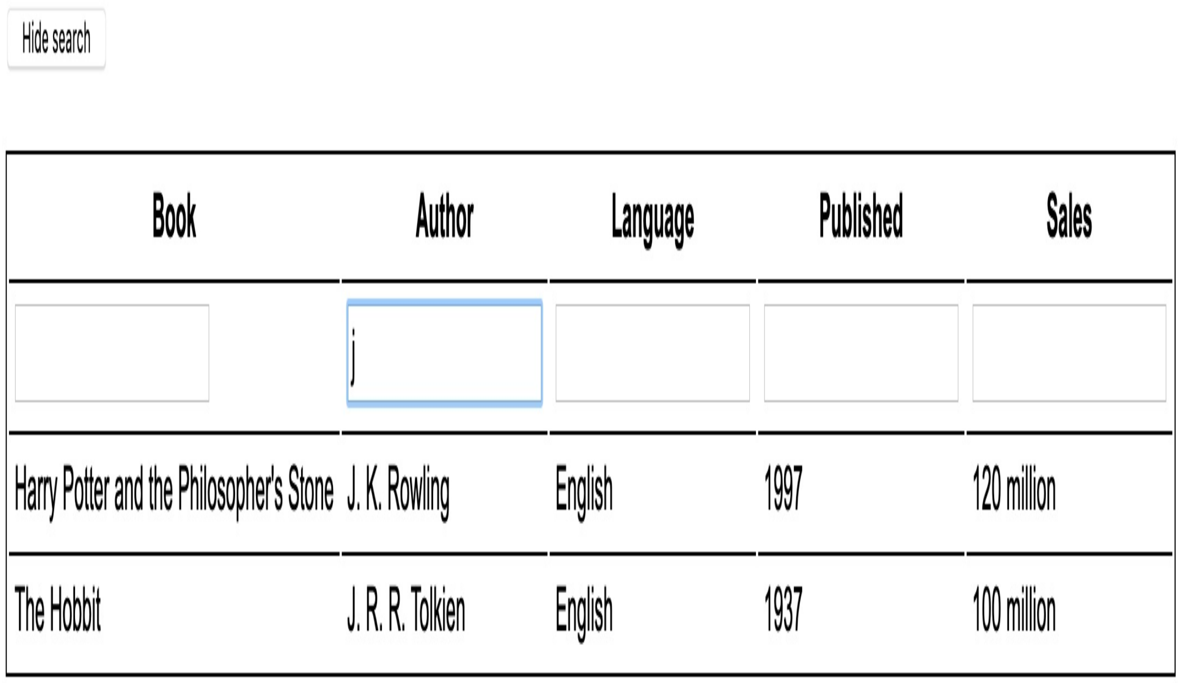
Book	Author
A Tale of Two Cities	Charles Dickens
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry

Figure 3-10. Search button



Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
And Then There Were None	Agatha Christie	English	1939	100 million
Dream of the Red Chamber	Cao Xueqin	Chinese	1791	100 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million

Figure 3-11. Row of search/filter inputs



Book	Author	Language	Published	Sales
	j			
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
The Hobbit	J. R. R. Tolkien	English	1937	100 million

Figure 3-12. Search results

State and UI

The first thing to do is update the constructor by:

- Adding a `search` property to the `this.state` object to keep track of whether the search feature is on
- Binding two new methods: `this.toggleSearch()` to turn search boxes on and off and `this.search()` to do the actual searching
- Setting up a new class property `this.preSearchData`, more about it in just a second
- Update the incoming initial data with a consecutive ID, this will help identify the rows when editing contents of filtered data

```
constructor(props) {
  super();
  const data = clone(props.initialData).map((row, idx) => {
    row.push(idx);
    return row;
  });
```

```
    this.state = {
      data,
      sortby: null,
      descending: false,
      edit: null, // {row: index, column: index}
      search: false,
    };

    this.preSearchData = null;

    this.sort = this.sort.bind(this);
    this.showEditor = this.showEditor.bind(this);
    this.save = this.save.bind(this);
    this.toggleSearch = this.toggleSearch.bind(this);
    this.search = this.search.bind(this);
  }
```

The cloning and updating of the `initialData` changes the data use din the state by adding a sort of *record ID*, this will prove useful when editing data that was already filtered. You `map()` the data adding an additional column which is an integer ID.

```
  const data = clone(props.initialData).map((row, idx) =>
    row.concat(idx),
  );
```

As a result the state `data` now looks like:

```
    [
      'A Tale of Two Cities', ..., 0
    ],
    [
      'Le Petit Prince (The Little Prince)', ..., 1
    ],
    // ...
```

This change also requires changes in the `render()` method, namely to use this record ID to identify rows, regardless if we're looking at all the data or a filtered subset of it (as a result of a search):

```
  {this.state.data.map((row, rowidx) => {
    // the last piece of data in a row is the ID
    const recordId = row[row.length - 1];
    return (
      <tr key={recordId} data-row={recordId}>
        {row.map((cell, columnidx) => {
          if (columnidx === this.props.headers.length) {
            // do not show the record ID in the table UI
            return;
          }
          const edit = this.state.edit;
          if (
            edit &&
            edit.row === recordId &&
            edit.column === columnidx
          ) {
            cell = (
              <form onSubmit={this.save}>
                <input type="text" defaultValue={cell} />
              </form>
            );
          }
          return <td key={columnidx}>{cell}</td>;
        })}
      </tr>
    );
  })}
```

Next comes updating the UI with a search button. Where before there was a `<table>` as the root, now let’s have a `<div>` with a search button and the same table.

```
  <div>
    <button className="toolbar" onClick={this.toggleSearch}>
      {this.state.search ? 'Hide search' : 'Show search'}
    </button>
    <table>
      {/* ... */}
    </table>
  </div>
```

As you see, the search button label is dynamic to reflect whether the search is on or off (`this.state.search` is `true` or `false`).

Next comes the row of search boxes. You can add it to the increasingly big chunk of JSX or have it composed upfront and added to a constant which is to be included in the main JSX. Let’s go the second route. If the search feature is not on, you don’t need to render anything, so `searchRow` is just `null`. Otherwise a new table row is created where each cell is an input element.

```
const searchRow = !this.state.search ? null : (
  <tr onChange={this.search}>
    {this.props.headers.map((_, idx) => (
      <td key={idx}>
        <input type="text" data-idx={idx} />
      </td>
    ))}
  </tr>
);
```

NOTE

Using `(_, idx)` is an illustration of a convention where an unused variable in a callback is named with an underscore _ to signal to the reader of the code that it's not used.

The row of search inputs is just another child node before the main `data` loop (the one that creates all the table rows and cells), so you include `searchRow` right there.

```
<tbody onDoubleClick={this.showEditor}>
  {searchRow}
  {this.state.data.map((row, rowidx) => (....
```

At this point, that's all for the UI updates. Let's take a look at the meat of the feature, the "business logic" if you will: the actual search.

Filtering Content

The search feature is going to be fairly simple: take the array of data, call the `Array.prototype.filter()` method on it, and return a filtered array with the elements that match the search string.

The UI still uses `this.state.data` to do the rendering, but `this.state.data` is a reduced version of itself.

You need a reference to the data before the search, so that you don't lose the data forever. This allows the user to go back to the full table or change the search string to get different matches. Let's call this reference `this.preSearchData`. Now that there's data in two places, the `save()` method will need an update, so both places are updated should the user decide to edit the data, regardless if it’s been filtered or not.

When the user clicks the “search” button, the `toggleSearch()` function is invoked. This function’s task is to turn the search feature on and off. It does its task by:

- Setting the `this.state.search` to `true` or `false` accordingly
- When enabling the search, “remembering” the current data
- When disabling the search, reverting to the remembered data.

Here’s what this function can look like:

```
toggleSearch() {
  if (this.state.search) {
    this.setState({
      data: this.preSearchData,
      search: false,
    });
    this.preSearchData = null;
  } else {
    this.preSearchData = this.state.data;
    this.setState({
      search: true,
    });
  }
}
```

The last thing to do is implement the `search()` function, which is called every time something in the search row changes, meaning the user is typing in one of the inputs. Here’s the complete implementation, followed by some more details:

```
search(e) {
  const needle = e.target.value.toLowerCase();
  if (!needle) {
    this.setState({data: this.preSearchData});
    return;
  }
  const idx = e.target.dataset.idx;
  const searchdata = this.preSearchData.filter((row) => {
    return row[idx].toString().toLowerCase().indexOf(needle) > -1;
  });
  this.setState({data: searchdata});
}
```

You get the search string from the change event’s target (which is the input box). Let’s call it “needle” as we’re looking for a needle in a haystack of data.

```
const needle = e.target.value.toLowerCase();
```

If there’s no search string (the user erased what they typed), the function takes the original, cached data and this data becomes the new state:

```
if (!needle) {
  this.setState({data: this.preSearchData});
  return;
}
```

If there is a search string, filter the original data and set the filtered results as the new state of the data:

```
const idx = e.target.dataset.idx;
const searchdata = this.preSearchData.filter((row) => {
  return row[idx].toString().toLowerCase().indexOf(needle) > -1;
});
this.setState({data: searchdata});
```

And with this, the search feature is complete. To implement the feature, all you needed to do was:

- Add search UI
- Show/hide the new UI upon request
- The actual “business logic” - a simple array `filter()` call

As always, you only worry about the state of your data and let React take care of rendering (and all the grunt DOM work associated) whenever the state of the data changes.

Update the `save()` method

Previously there was only `state.data` to be cloned and updated, but now you also have the “remembered” `preSearchData`. If the user is editing (even while searching) now the two pieces of data need an update. That’s the whole reason for adding a record ID - so you can find the real row even in a filtered state.

Updating the `preSearchData` is just like in the previous `save()` implementation - just find the row and column. Updating the state data requires one more step which is to find the record ID of the row and match it to the row currently being edited (`this.state.edit.row`).

```
  save(e) {
    e.preventDefault();
    const input = e.target.firstChild;
    const data = clone(this.state.data).map((row) => {
      if (row[row.length - 1] === this.state.edit.row) {
        row[this.state.edit.column] = input.value;
      }
      return row;
    });
    this.logSetState({
      edit: null,
      data,
    });
    if (this.preSearchData) {
      this.preSearchData[this.state.edit.row][this.state.edit.column] =
        input.value;
    }
  }
```

See `03.10.table-search.html` in the book's repo for the complete code.

Can You Improve the Search?

This was a simple working example for illustration. Can you improve the feature?

Try to implement an *additive search* in multiple boxes, meaning filter the already filtered data. If the user types "Eng" in the language row and then searches using a different search box, why not search in the search results of the previous search only? How would you implement this feature?

Instant Replay

As you know now, your components worry about their state and let React render and rerender whenever appropriate. This means that given the same data (state and properties), the application will look exactly the same, no matter what changed before or after this particular data state. This gives you a great debugging-in-the-wild opportunity.

Imagine someone encounters a bug while using your app—they can click a button to report the bug without needing to explain what happened. The bug report can just send you back a copy of `this.state` and `this.props`, and you should be able to re-create the exact application state and see the visual result.

An “undo” could be another feature based of the fact that React renders your app the same way given the same props and state. And, in fact, the “undo” implementation is somewhat trivial: you just need to go back to the previous state.

Let’s take that idea a bit further, just for fun. Let’s record each state change in the `Excel` component and then replay it. It’s fascinating to watch all your actions played back in front of you. The question of *when* the change occurred is not that important, so let’s “play” the app state changes at 1-second intervals.

To implement this feature, you need add a `logSetState()` method which first logs the new state to a `this.log` array and then calls `setState()`. And everywhere in the code you called `setState()` should now be changed to call `logSetState()`. So first search and replace all calls to `setState()` with calls to the new function.

So all calls to…

```
this.setState(...);
```

…become

```
this.logSetState(...);
```

Now let’s start with the constructor. You need to bind the two new functions: `logSetState()` and `replay()` and also declare `this.log` array and initialize it with the initial state.

```
  constructor(props) {
    // ...

    // log the initial state
    this.log = [clone(this.state)];

    // ...
    this.replay = this.replay.bind(this);
    this.logSetState = this.logSetState.bind(this);
  }
```

The `logSetState` needs to do two things: log the new state and then pass it over to `setState()`. Here's one example implementation where you make a deep copy of the state and append it to `this.log`:

```
  logSetState(newState) {
    // remember the old state in a clone
    this.log.push(clone(newState));
    // now set it
    this.setState(newState);
  }
```

Now that all state changes are logged, let's play them back. To trigger the playback, let's add a simple event listener that captures keyboard actions and invokes the `replay()` function. The place for events listeners like this is in the `componentDidMount()` lifecycle method.

```
  componentDidMount() {
    document.addEventListener('keydown', e => {
      if (e.altKey && e.shiftKey && e.keyCode === 82) {
        // ALT+SHIFT+R(eplay)
        this.replay();
      }
    });
  }
```

Finally, the `replay()` method. It uses `setInterval()` and once a second it reads the next object from the log and passes it to `setState()`:

```
  replay() {
    if (this.log.length === 1) {
      console.warn('No state changes to replay yet');
      return;
    }
    let idx = -1;
    const interval = setInterval(() => {
      if (++idx === this.log.length - 1) {
        // the end
        clearInterval(interval);
      }
      this.setState(this.log[idx]);
    }, 1000);
  }
```

And with this, the new feature is complete (`03.11.table-replay.html` in the repo). Play around with the component, sort, edit… Then press ALT+SHIFT+R (OPTION+SHIFT+R on Mac) and see the past unfolding before you.

Cleaning up event handlers

The replay feature needs just a bit of cleanup. When this component is the only thing hapenning on the page, that's not necessary, but in a real application components get added and removed from the DOM more frequently. When removing from the DOM a "good citizen" component should take care of cleaning up after itself. In the example above there are two items that need cleaning up: the `keydown` event listener and the replay interval callback.

If you don't clean up the keydown event listener function, it will linger on in memory after the component is gone. And because it's using `this`, the whole `Excel` instance needs to be retained in memory. This is in effect a memory leak. Too many of those and the user may run out of memory and your application may crash the browser tab. As to the interval, well, the callback function will continue executing after the component is gone and cause another memry leak. The callback will also attempt to call `setState()` on a non-existing component which React handles gracefully and gives you a warning.

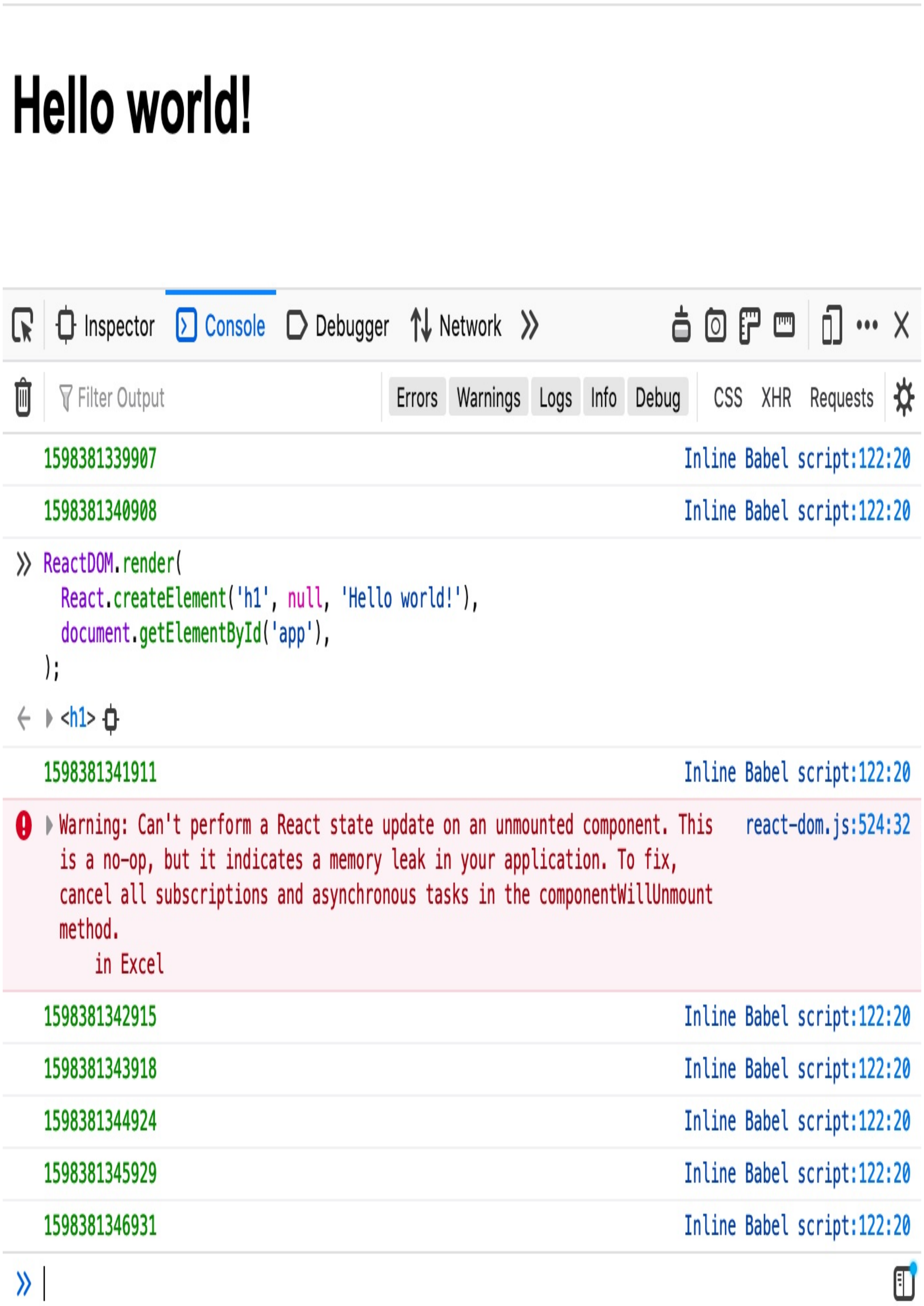
You can test the latter behavior by removing the component from the DOM while the replay is still going. To remove the component from the DOM you can just replace it, e.g. run the "Hello world" from Chapter 1 in the console:

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

You can also log a timestamp to the console in the interval callback to see that it keeps on being executed.

```
  const interval = setInterval(() => {
    // ...
    console.log(Date.now());
    // ...
  }, 1000);
```

Now when you replace the component during replay, you see an error from React and the timestamps of the interval callback still being logged as evidence that the callback is still running (Figure 3-13).


Hello world!

```
1598381339907                                              Inline Babel script:122:20
1598381340908                                              Inline Babel script:122:20
» ReactDOM.render(
    React.createElement('h1', null, 'Hello world!'),
    document.getElementById('app'),
  );
← ▸ <h1>
1598381341911                                              Inline Babel script:122:20
❗ ▸ Warning: Can't perform a React state update on an unmounted component. This   react-dom.js:524:32
    is a no-op, but it indicates a memory leak in your application. To fix,
    cancel all subscriptions and asynchronous tasks in the componentWillUnmount
    method.
        in Excel
1598381342915                                              Inline Babel script:122:20
1598381343918                                              Inline Babel script:122:20
1598381344924                                              Inline Babel script:122:20
1598381345929                                              Inline Babel script:122:20
1598381346931                                              Inline Babel script:122:20
»
```

Figure 3-13. Memory leak in action

Similarly, you can test the event listener memory leak by pressing ALT+SHIFT+R after the component has been removed from the DOM. The

Cleaning solution

Taking care of these memory leaks is fairly straightforward. You need to keep references to the handlers and intervals/timeouts you want to clean up. Then clean them up in `componentWillUnmount()`.

For the event handler, have it as a class method, as opposed to an inline function:

```
keydownHandler(e) {
  if (e.altKey && e.shiftKey && e.keyCode === 82) {
    // ALT+SHIFT+R(eplay)
    this.replay();
  }
}
```

Then `componentDidMount()` becomes the simpler:

```
componentDidMount() {
  document.addEventListener('keydown', this.keydownHandler);
}
```

For the interval replay ID, have it as a class property as opposed to a local variable:

```
this.replayID = setInterval(() => {
  if (++idx === this.log.length - 1) {
    // the end
    clearInterval(this.replayID);
  }
  this.setState(this.log[idx]);
}, 1000);
```

You need to, of course, bind the new method and add the new property in the constructor:

```
constructor(props) {
  // ...
```

```
    this.replayID = null;

    // ...
    this.keydownHandler = this.keydownHandler.bind(this);
  }
```

And, finally, the cleanup in the `componentWillUnmount()`:

```
  componentWillUnmount() {
    document.removeEventListener('keydown', this.keydownHandler);
    clearInterval(this.replayID);
  }
```

Now all the leaks are plugged (`03.12.table-replay-clean.html`).

Can You Improve the Replay?

How about implementing an Undo/Redo feature? Say when the person uses the ALT+Z keyboard combination, you go back one step in the state log and on ALT+SHIFT+Z you go forward.

An Alternative Implementation?

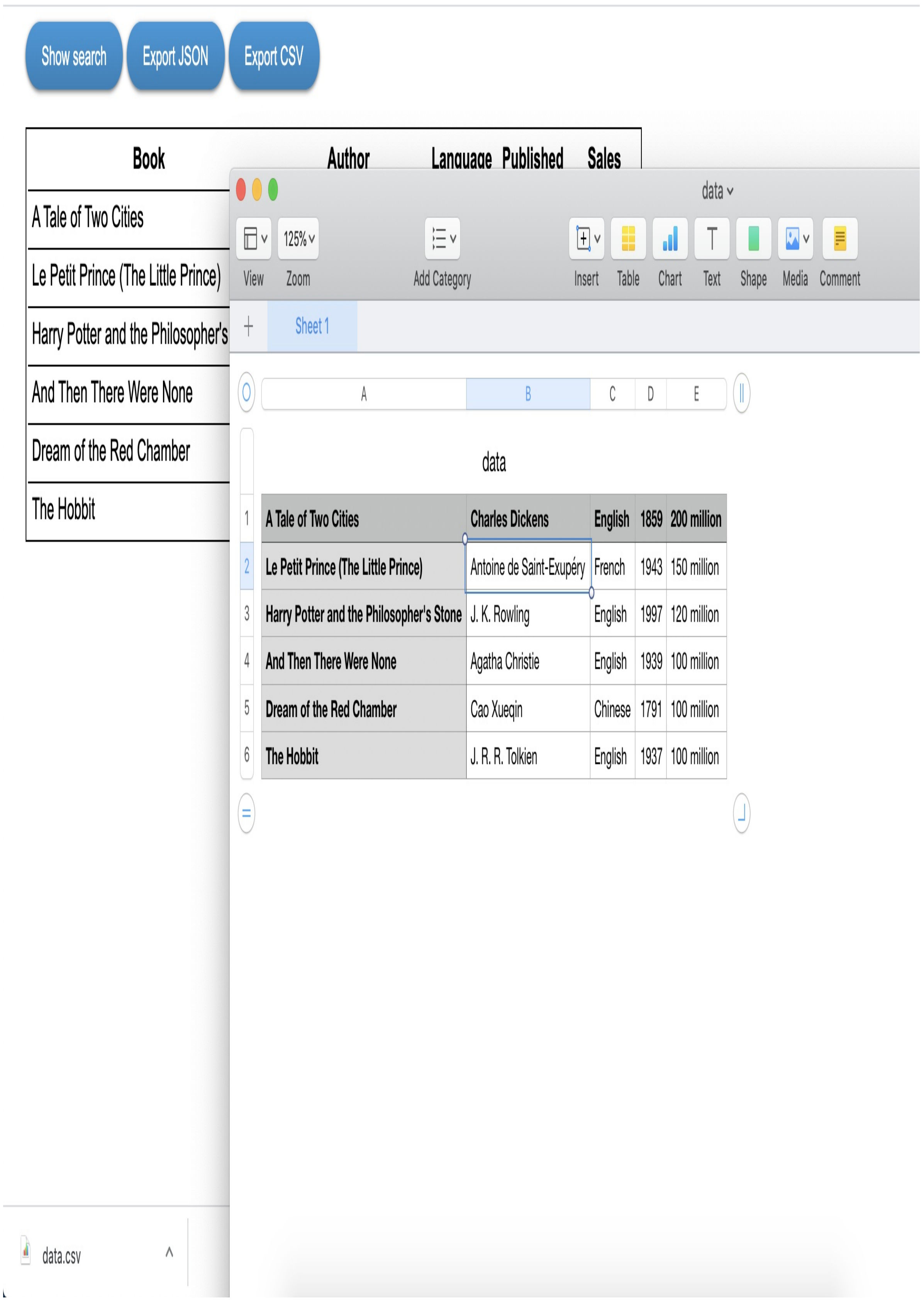
Is there another way to implement replay/undo type of functionality without changing all your `setState()` calls? Maybe use an appropriate lifecycle method (Chapter 2)?

Download the Table Data

After all the sorting, editing, and searching, the user is finally happy with the state of the data in the table. It would be nice if they could download the data, the result of all their labor, to use at a later time.

Luckily, there's nothing easier in React. All you need to do is grab the current `this.state.data` and give it back—for example in JSON or CSV format.

Figure 3-14 shows the end result when a user clicks "Export CSV," downloads the file called *data.csv* (see the bottom left of the browser window), and opens this file in Numbers (on a Mac, or Microsoft Excel on a PC or Mac).



Book	Author	Language	Published	Sales
A Tale of Two Cities				
Le Petit Prince (The Little Prince)				
Harry Potter and the Philosopher's				
And Then There Were None				
Dream of the Red Chamber				
The Hobbit				

data

	A	B	C	D	E
1	**A Tale of Two Cities**	**Charles Dickens**	**English**	**1859**	**200 million**
2	**Le Petit Prince (The Little Prince)**	Antoine de Saint-Exupéry	French	1943	150 million
3	**Harry Potter and the Philosopher's Stone**	J. K. Rowling	English	1997	120 million
4	**And Then There Were None**	Agatha Christie	English	1939	100 million
5	**Dream of the Red Chamber**	Cao Xueqin	Chinese	1791	100 million
6	**The Hobbit**	J. R. R. Tolkien	English	1937	100 million

Figure 3-14. Export table data to Numbers via CSV

The first thing to do is add new options to the toolbar (where the Search button is). Let’s use some HTML magic that forces `<a>` links to trigger file downloads, so the new “buttons” have to be links disguised as buttons with some CSS:

```
<div className="toolbar">
  <button onClick={this.toggleSearch}>
    {this.state.search ? 'Hide search' : 'Show search'}
  </button>
  <a href="data.json" onClick={this.downloadJSON}>
    Export JSON
  </a>
  <a href="data.csv" onClick={this.downloadCSV}>
    Export CSV
  </a>
</div>
```

As you see, you need `downloadJSON` and `downloadCSV()` methods. These have some repeating logic, so they can be done by a single `download()` function bound with the `format` (meaning file type) argument. The `download()` method’s signature could be like:

```
download(format, ev) {
  // TODO: implement me
}
```

In the constructor you can bind this method twice, like so:

```
this.downloadJSON = this.download.bind(this, 'json');
this.downloadCSV = this.download.bind(this, 'csv');
```

All the React work is done, now for the `download()` function. While exporting to JSON is trivial, CSV (comma-separated values) needs a little bit more work. In essence, it’s just a loop over all rows and all cells in a row, producing a long string. Once this is done, the function initiates the downloads via the `download` attribute and the `href` blob created by `window.URL`:

```
download(format, ev) {
  const data = clone(this.state.data).map(row => {
    row.pop(); // drop the last column, the recordId
    return row;
  });
  const contents =
    format === 'json'
      ? JSON.stringify(data, null, '  ')
      : data.reduce((result, row) => {
          return (
            result +
            row.reduce((rowcontent, cellcontent, idx) => {
              const cell = cellcontent.replace(/"/g, '""');
              const delimiter = idx < row.length - 1 ? ',' : '';
              return `${rowcontent}"${cellcontent}"${delimiter}`;
            }, '') +
            '\n'
          );
        }, '');

  const URL = window.URL || window.webkitURL;
  const blob = new Blob([contents], {type: 'text/' + format});
  ev.target.href = URL.createObjectURL(blob);
  ev.target.download = 'data.' + format;
}
```

The complete code is in `03.13.table-download.html` in the repo.

Fetching data

All through the chapter the `Excel` component had access to the `data` in the same file. But what if the data lives elsewhere, on a server, and needs to be fetched. There are various solutions to this and you’ll see more further in the book, but let try one of the simplest—fetching the data in `componentDidMount()`.

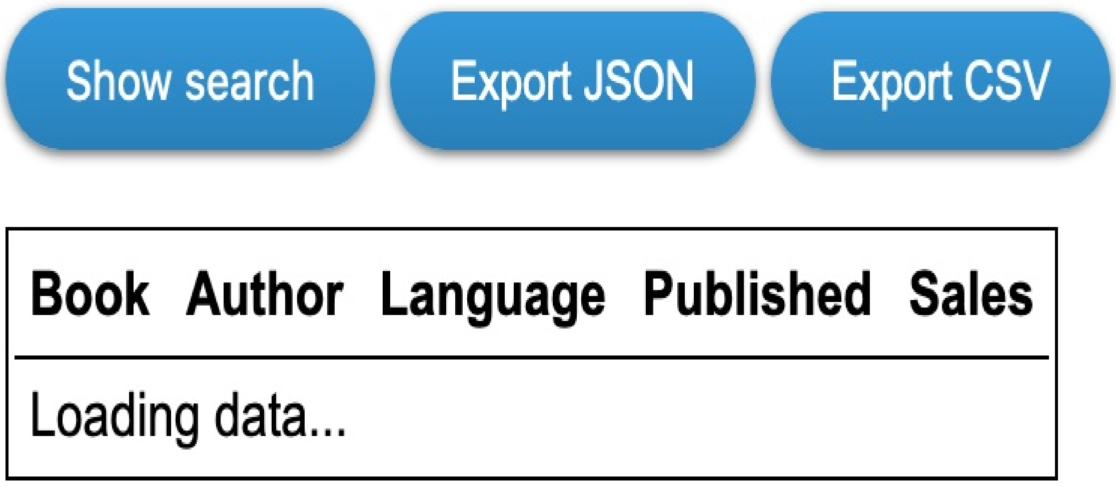
Let’s say the `Excel` component is created with an empty `initialData` property:

```
ReactDOM.render(
  <Excel headers={headers} initialData={[]} />,
  document.getElementById('app'),
);
```

The component can gracefully render an intermendiate state to let the user know that data is coming. In the `render()` method you can have a condition and render a different table body if data is not there:

```
{this.state.data.length === 0 ? (
  <tbody>
    <tr>
      <td colSpan={this.props.headers.length}>
        Loading data...
      </td>
    </tr>
  </tbody>
) : (
  <tbody onDoubleClick={this.showEditor}>
    {/* ... same as before ...*/}
  </tbody>
)}
```

While waiting for the data the user sees a loading indicator (Figure 3-15), in this case a simple text, but you can have an animation if you like.



Book	Author	Language	Published	Sales
Loading data...				

Figure 3-15. Waiting for the data to be fetched

Now let’s fetch the data. Using the HTML Fetch API, you make a request to a server and once the response arrives, you set the state with the new data. You also need to take care of adding the record ID which was previously the job of the contstructor. The updated `componentDidMount()` can look like so:

```
componentDidMount() {
  document.addEventListener('keydown', this.keydownHandler);
  fetch('https://www.phpied.com/files/reactbook/table-data.json')
    .then((response) => response.json())
```

```
      .then((initialData) => {
        const data = clone(initialData).map((row, idx) => {
          row.push(idx);
          return row;
        });
        this.setState({data});
      });
  }
```

The complete code is in `03.14.table-fetch.html` in the repo.

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.