React for Real

Front-End Code, Untangled

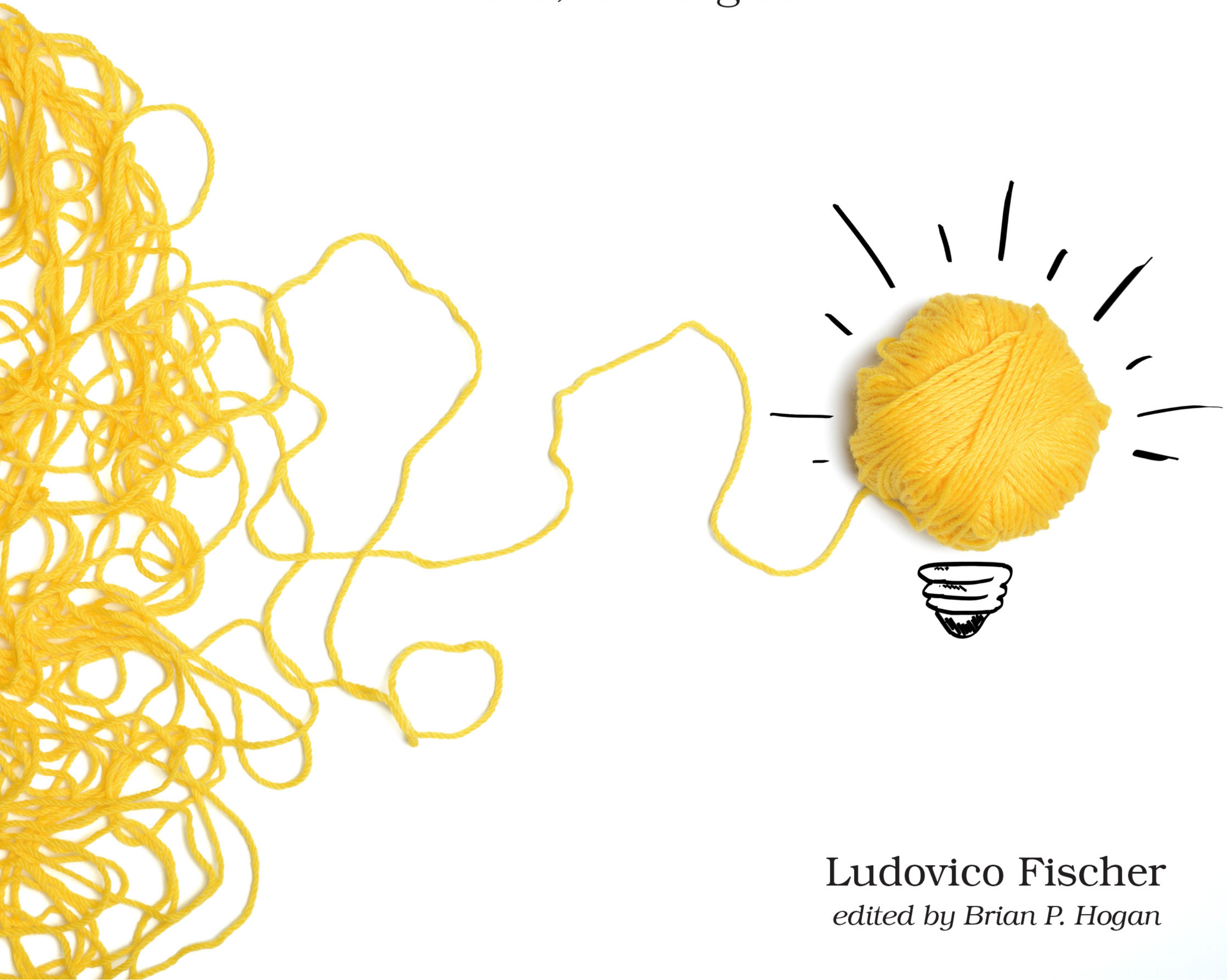
The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

React for Real

Front-End Code, Untangled



Ludovico Fischer

edited by Brian P. Hogan

React for Real

Front-End Code, Untangled

Ludovico Fischer

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Development Editor: Brian P. Hogan
Copy Editor: Nicole Abramowtiz
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-263-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2017

Design Presentational Components

Once we've decided on the general layout of the application, we can start working on the individual presentational components that make up the interface.

Styling with CSS is an important part of defining the appearance and functionality of a UI. CSS techniques such as BEM[4] strive to organize CSS classes into components, so that adding a component class to an element endows it with a bunch of properties. React makes these techniques less useful. With React, your whole interface is made up of React components. React components are much smaller than traditional templates, so each component encapsulates a potentially reusable part of the interface. By reusing React components, you automatically reuse the styles you define for each component, so CSS components are less useful. We'll use a CSS library called Tachyons,[5] which organizes individual CSS properties along regular increments to create a consistent appearance.

Let's create a new directory for the word counter project. As before, we'll use a file named index.html in the top directory to provide the HTML skeleton. Create index.html and add the CDN link to Tachyons in index.html:

wordcounter-single/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1"/>
➤   <link rel="stylesheet"
➤     href="https://unpkg.com/tachyons@4.8.0/css/tachyons.min.css"/>
  </head>
  <body>
    <div id="app"></div>
  </body>
</html>
```

From this point on, we'll store all our JavaScript sources in a directory named src. That way, we'll distinguish them more easily when we add additional configuration files to our project in the next chapters.

Create a new directory named src. Create a new file named index.js in the src directory and link it in index.html. Ensure there's a <div> with an id=*app*.

wordcounter-single/index.html

```
<body>
```

4. http://getbem.com
5. http://tachyons.io

```
    <div id="app"></div>
➤   <script src="https://unpkg.com/react@15/dist/react.js"></script>
➤   <script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>
➤   <script
➤           src="https://unpkg.com/babel-standalone@6/babel.min.js"
➤    ></script>
➤   <script type="text/babel" src="src/index.js"></script>
  </body>
```

In `index.js`, define a function `Counter()` that outputs the value of its `count` prop in a `<p>` element:

wordcounter-single/src/index.js

```
function Counter({ count }) {
  return (
    <p className="mb2">
      Word count: {count}
    </p>
  );
}
```

Counter() expects an object with a count key. With destructuring, assign the value of the count parameter property to count in the Counter() body. Return a JSX expression with the count variable between braces to output a React element displaying its value. React uses className instead of class for legacy reasons and to avoid clashes with the JavaScript class keyword.

Joe asks:

Any more JSX gotchas?

In addition to class and className, another difference between JSX and HTML is that you must use htmlFor instead of for to point a label to the element it labels. Probably the most important point to keep in mind when writing JSX is that React interprets elements that start with a capital letter as based on custom components, and elements that start with lowercase as plain HTML elements.

Next, create the progress bar component. In index.js, create a ProgressBar() function that takes a completion prop, and returns a progress bar with its value= attribute set to the completion prop:

wordcounter-single/src/index.js

```
function ProgressBar({ completion }) {
  const percentage = completion * 100;
  return (
    <div className="mv2 flex flex-column">
      <label htmlFor="progress" className="mv2">
        Progress
      </label>
      <progress value={completion} id="progress" className="bn">
        {percentage}%
      </progress>
    </div>
  );
}
```

<progress> is a standard HTML element. The value= attribute determines how much it fills. 0.5 means 50%, 0.6 means 60%, and so on, so completion should contain a number between 0 and 1. To make the progress bar accessible, we include a label for the <progress> element.

Next, let's create the text editor itself. In index.js, create an Editor component that defines a plain <textarea>. Editor takes the text as prop:

wordcounter-single/src/index.js

```
function Editor({
  text,
}) {
  return (
    <div className="flex flex-column mv2">
      <label htmlFor="editor" className="mv2">
        Enter your text:
      </label>
      <textarea
        value={text}
        id="editor"
      />
    </div>
  );
}
```

In HTML, the <textarea> child element determines the <textarea> content, but the <textarea> React element uses the value prop. Set the value prop of the <textarea> element to the value of the text prop.

You have almost all the ingredients in place. Now we'll combine Editor, ProgressBar, and Counter into a new component called WordCounter.