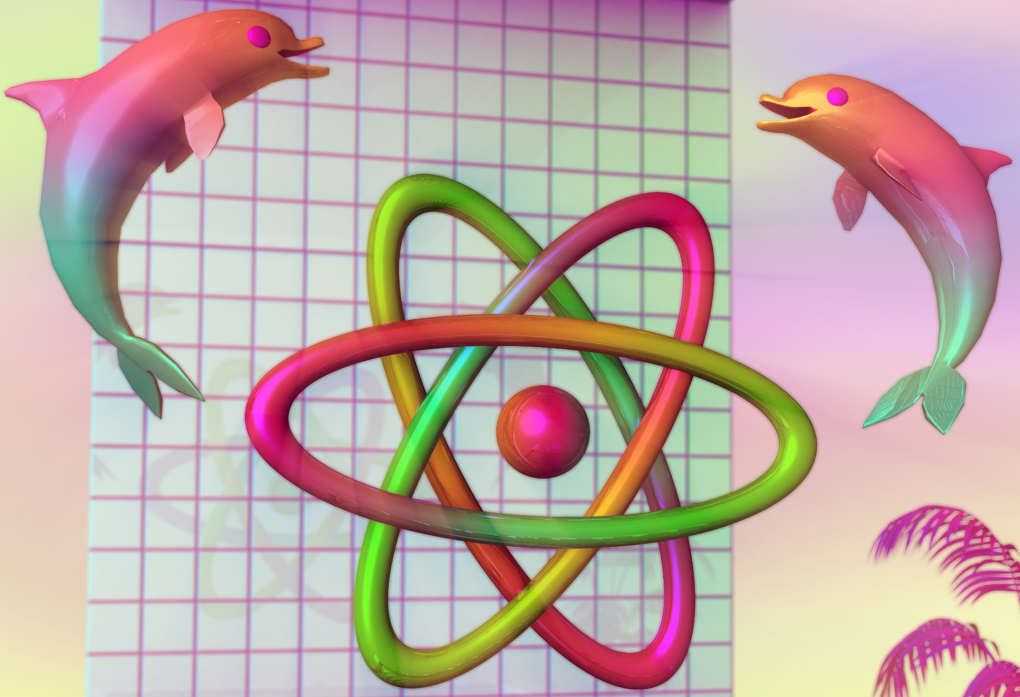
FULLSTACK REACT

The Complete Guide to ReactJS and Friends



ANTHONY ACCOMAZZO ARI LERNER
CLAY ALLSOPP DAVID GUTTMAN
TYLER MCGINNIS NATE MURRAY

FULLSTACK.io

Fullstack React

The Complete Guide to ReactJS and Friends

Written by Anthony Accomazzo, Nate Murray, Ari Lerner, Clay Allsopp, David Guttman, and Tyler McGinnis

Technical Advisor: Sophia Shoemaker
Contributors: Seulgi Kim

Published by \newline (formerly Fullstack.io).

Contents

Book Revision

Revision 40 - Supports React 16.12.0 (2020-01-13)

Updated January 13th, 2020

Join our Official Community Discord

There is an *official* community chat room for the book: join us here[1]

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: us@fullstack.io[2].

For further help dealing with issues, refer to "How to Get the Most Out of This Book."

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow @fullstackio[3]

We'd love to hear from you!

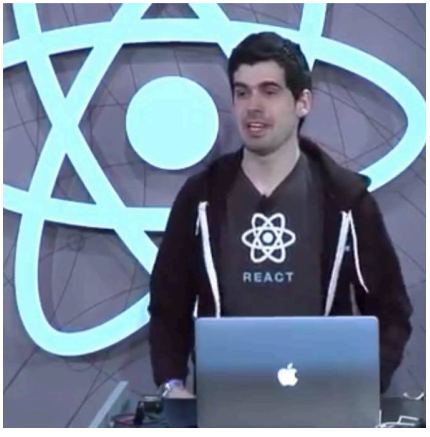
Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[4].

[1] https://newline.co/discord/react
[2] mailto:us@fullstack.io?Subject=Fullstack%20React%20book%20feedback
[3] https://twitter.com/fullstackio
[4] mailto:us@fullstack.io?Subject=react%20testimonial

Foreword

Web development is often seen as a crazy world where the way you develop software is by throwing hacks on top of hacks. I believe that React breaks from this pattern and instead has been designed from first principle which gives you a solid foundation to build on.

A major source of bugs for front-end applications was around synchronizing the data model with the DOM. It is very hard to make sure that whenever data changes, everything in the UI is updated with it.



Christopher Chedeau - Front-end Engineer at Facebook

React's first innovation was to introduce a pure-JavaScript representation of the DOM and implement diffing in userland and then use events which send simple commands: create, update, delete.

With React, by conceptually **re-rendering everything whenever anything changes**, not only do you have code that is **safe by default**, it is also **less work** as you only need to write the creation path: updates are taken care of for you.

Browsers have, for a long time, been incompatible in various ways due to the large API surface area of what they have to support to make the DOM work. Not only does React provide a great way to solve browser differences, but it enables use cases that were never before possible for a front-end library, such as server-side rendering and the ability to implement rendering targets like native iOS, Android, and even hardware components.

But the most important thing about React and the main reason why you should read this book: not only will you use it to **make great applications for your users**, it will

also **make you a better developer**. Libraries come and go all the time and React is likely going to be no exception. What makes it special is that it teaches you **concepts that can be reused throughout your entire career**.

You will become better at JavaScript because React doesn't come with a templating system. Instead, React pushes you to use the **full power of JavaScript** to build your user interface.

You are going to practice using parts of *functional programming* with `map` and `filter` and also encouraged to use the **latest features of JavaScript** (including ES6). By not abstracting away data management, React will force you to think about how to architect your app and encourage you to consider concepts like immutability.

I'm very proud that the community built around React is not afraid of "rethinking best practices." The community challenges the status quo in many areas. My advice to you is to read this **excellent** book to learn and understand the fundamentals of React. Learning new concepts may feel strange but "give it 5 minutes" and practice them until you feel comfortable.

Then, **try to break the rules**. There is no one best way to build software and React is no exception. React actually embraces this fact by providing you with escape hatches when you want to do things outside of the React-way.

Come up with crazy ideas and who knows, maybe you are going to invent the successor to React!

– Christopher Chedeau @vjeux[5] Front-end Engineer at Facebook

[5] https://twitter.com/Vjeux

How to Get the Most Out of This Book

Overview

This book aims to be the single most useful resource on learning React. By the time you're done reading this book, you (and your team) will have everything you need to build reliable, powerful React apps.

React core is lean and powerful. After the first few chapters, you'll have a solid understanding of React's fundamentals and will be able to build a wide array of rich, interactive web apps with the framework.

But beyond React's core, there are many tools in its ecosystem that you might find helpful for building production apps. Things like client-side routing between pages, managing complex state, and heavy API interaction at scale.

This book consists of two parts.

In Part I, we cover all the fundamentals with a progressive, example-driven approach. You'll create your **first apps**, learn **how to write components**, start **handling user interaction**, manage rich **forms**, and even **interact with servers**.

We bookend the first part by exploring the inner workings of **Create React App** (Facebook's tool for running React apps), writing automated **unit tests**, and building a multi-page app that uses **client-side routing**.

Part II of this book moves into more **advanced concepts** that you'll see used in large, production applications. These concepts explore strategies for *data architecture, transport, and management*:

Redux is a state management paradigm based on Facebook's Flux architecture. Redux provides a structure for large state trees and allows you to decouple user interaction in your app from state changes.

GraphQL is a powerful, typed, REST API alternative where the client describes the data it needs. We also cover how to **write your own GraphQL servers** for your own data.

Relay is the glue between GraphQL and React. Relay is a data-fetching library that makes it easy to write flexible, performant apps without a lot of data-fetching code.

Finally, in the last chapter, we'll talk about how to write native, cross-platform mobile apps using **React Native**.

There are a few guidelines we want to give you **in order to get the most out of this book**.

First, know that you do not need to read this book linearly from cover-to-cover. **However**, we've ordered the contents of the book in a way we feel fits the order you should learn the concepts. We encourage you to learn all the concepts in Part I of the book first before diving into concepts in Part II.

Second, keep in mind this package is more than just a book - it's a course complete with example code for every chapter. Below, we'll tell you:

- how to approach **the code examples** and
- **how to get help** if something goes wrong

Running Code Examples

This book comes with a library of runnable code examples. The code is available to download from the same place where you purchased this book. If you purchased this book on Amazon, you should have received an email with instructions.

If you have any trouble finding or downloading the code examples, email us at **react@fullstack.io**.

We use the program npm[6] to run **every example** in this book. You can boot most apps with the following two commands:

npm install npm start

[6] https://www.npmjs.com/

If you're unfamiliar with `npm`, we cover how to get it installed in the "Setting Up" section in the first chapter.

After running `npm start`, you will see some output on your screen that will tell you what URL to open to view your app.

Some apps require a few more commands to setup. **If you're ever unclear on how to run a particular sample app, checkout the `README.md` in that project's directory**. Every sample project contains a `README.md` that will give you the instructions you need to run each app.

Project setups

The first two projects begin with a simple React setup that allows us to quickly write React applications.

From there, with a couple exceptions, every project in this book was built using Create React App[7].

Create React App is based on Webpack, a tool which helps process and bundle our various JavaScript, CSS, HTML, and image files. We explore Create React App in-depth in the chapter "Using Webpack with Create React App." But, **Create React App is not a requirement** for using React. It's simply a wrapper around Webpack (and some other tooling) that makes it easy to get started.

Code Blocks and Context

Nearly every code block in this book is pulled from a **runnable code example** which you can find in the sample code. For example, here is a code block pulled from the first chapter:

[7] https://github.com/facebookincubator/create-react-app

voting_app/public/js/app-2.js

```
class ProductList extends React.Component {
  render() {
    return (
      <div className='ui unstackable items'>
        <Product />
      </div>
    );
  }
}
```

Notice that the header of this code block states the path to the file which contains this code: `voting_app/public/js/app-2.js`.

If you ever feel like you're missing the context for a code example, open up the full code file using your favorite text editor. **This book is written with the expectation that you'll also be looking at the example code alongside the manuscript**.

For example, we often need to `import` libraries to get our code to run. In the early chapters of the book we show these `import` statements, because it's not clear where the libraries are coming from otherwise. However, the later chapters of the book are more advanced and they focus on *key concepts* instead of repeating boilerplate code that was covered earlier in the book. **If at any point you're not clear on the context, open up the code example on disk**.

Code Block Numbering

In this book, we sometimes build up a larger example in steps. If you see a file being loaded that has a numeric suffix, that generally means we're building up to something bigger.

For instance, above the code block has the filename: `app-2.js`. When you see the `-N.js` syntax that means we're building up to a final version of the file. You can jump into that file and see the state of all the code at that particular stage.

Getting Help

While we've made every effort to be clear, precise, and accurate you may find that when you're writing your code you run into a problem.

Generally, there are three types of problems:

- A "bug" in the book (e.g. how we describe something is wrong)
- A "bug" in our code
- A "bug" in your code

If you find an inaccuracy in how we describe something, or you feel a concept isn't clear, ask us in Discord! We want to make sure that the book is both accurate and clear.

If you suspect a problem with the example code, make sure that your version of the book's code package is up to date. We release code updates periodically.

If you're using the latest code download and you think you've found a bug in our *code* we definitely want to hear about it.

If you're having trouble getting your own app working (and it isn't *our* example code), this case is a bit harder for us to handle.

Your first line of defense, when getting help with your custom app, should be our official community Discord chat room[8]. We (the authors) are there from time-to-time, but there are hundreds of other readers there who may be able to help you faster than we can.

If you're still stuck, we'd still love to hear from you, and here some tips for getting a clear, timely response.

Emailing Us

If you're emailing us asking for technical help, here's what we'd like to know:

[8] https://newline.co/discord/react

- What revision of the book are you referring to?
- What operating system are you on? (e.g. Mac OS X 10.8, Windows 95)
- Which chapter and which example project are you on?
- What were you trying to accomplish?
- What have you tried[9] already?
- What output did you expect?
- What actually happened? (Including relevant log output.)

The **absolute best way to get technical support** is to send us a short, self-contained example of the problem. Our preferred way to receive this would be for you to send us a Plunkr link by using this URL[10].

That URL contains a runnable, boilerplate React app. If you can copy and paste your code into that project, reproduce your error, and send it to us **you'll greatly increase the likelihood of a prompt, helpful response**.

When you've written down these things, first try us in Discord here[11]. If that doesn't help, then you may email us at **us@fullstack.io**. We look forward to chatting with you!

Get excited

Writing web apps with React is *fun*. And by using this book, **you're going to learn how to build real React apps** fast. (Much faster than spending hours parsing outdated blog posts.)

If you've written client-side JavaScript before, you'll find React refreshingly intuitive. If this is your first serious foray into the front-end, you'll be blown away at how quickly you can create something worth sharing.

So hold on tight - you're about to become a React expert and have a lot of fun along the way. Let's dig in!

- Nate (@eigenjoy[12]) & Anthony

[9] http://mattgemmell.com/what-have-you-tried/
[10] http://bit.ly/fsr-plunker
[11] https://newline.co/discord/react
[12] https://twitter.com/eigenjoy

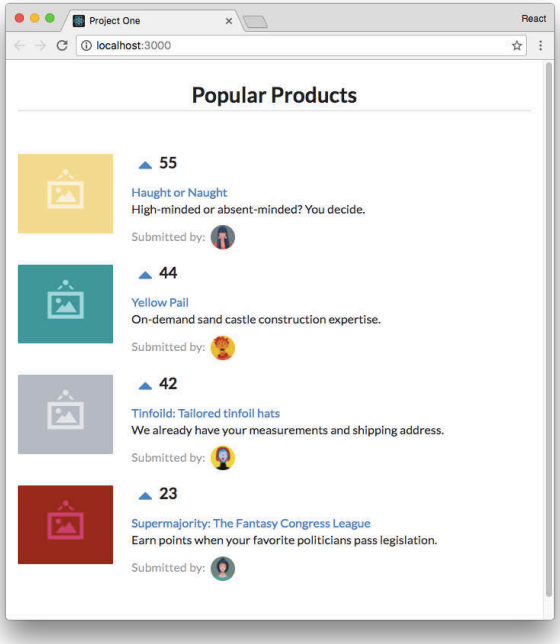
Part I

Your first React Web Application

Building Product Hunt

In this chapter, you're going to get a crash course on React by building a simple voting application inspired by Product Hunt[13]. You'll become familiar with how React approaches front-end development and all the fundamentals necessary to build an interactive React app from start to finish. Thanks to React's core simplicity, by the end of the chapter you'll already be well on your way to writing a variety of fast, dynamic interfaces.

[13] http://producthunt.com



Completed version of the app

We'll focus on getting our React app up and running fast. We take a deeper look at concepts covered in this section throughout the book.

Setting up your development environment

Code editor

As you'll be writing code throughout this book, you'll need to make sure you have a code editor you're comfortable working with. If you don't already have a preferred editor, we recommend installing VS Code[14].

[14] https://code.visualstudio.com/

Node.js and npm

For all the projects in this book, we'll need to make sure we have a working Node.js[15] development environment along with npm.

There are a couple different ways you can install Node.js so please refer to the Node.js website for detailed information: https://nodejs.org/download/[16]

If you're on a Mac, your best bet is to install Node.js directly from the Node.js website instead of through another package manager (like Homebrew). Installing Node.js via Homebrew is known to cause some issues.

The Node Package Manager (npm for short) is installed as a part of Node.js. To check if npm is available as a part of our development environment, we can open a terminal window and type:

```
1 $ npm -v
```

If a version number is not printed out and you receive an error, make sure to download a Node.js installer that includes npm.

Install Git

The app in this chapter requires Git to install some third-party libraries.

If you don't have Git installed, see these instructions[17] for installing Git for your platform.

After installing Git, we recommend restarting your computer.

Browser

Last, we highly recommend using the Google Chrome Web Browser[18] to develop React apps. We'll use the Chrome developer toolkit throughout this book. To follow along with our development and debugging we recommend downloading it now.

[15] http://nodejs.org
[16] https://nodejs.org/download/
[17] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
[18] https://www.google.com/chrome/

Special instruction for Windows users

All the code in this book has been tested on Windows 10 with PowerShell.

Ensure IIS is installed

If you're on a Windows machine and have yet to do any web development on it, you may need to install IIS (Internet Information Services) in order to run web servers locally.

See this tutorial[19] for installing IIS.

JavaScript ES6/ES7

JavaScript is the language of the web. It runs on many different browsers, like Google Chrome, Firefox, Safari, Microsoft Edge, and Internet Explorer. Different browsers have different JavaScript interpreters which execute JavaScript code.

Its widespread adoption as the internet's client-side scripting language led to the formation of a standards body which manages its specification. The specification is called **ECMAScript** or ES.

The 5th edition of the specification is called ES5. You can think of ES5 as a "version" of the JavaScript programming language. Finalized in 2009, ES5 was adopted by all major browsers within a few years.

The 6th edition of JavaScript is referred to as ES6. Finalized in 2015, the latest versions of major browsers are still finishing adding support for ES6 as of 2017. ES6 is a significant update. It contains a whole host of new features for JavaScript, almost two dozen in total. JavaScript written in ES6 is tangibly different than JavaScript written in ES5.

ES7, a much smaller update that builds on ES6, was ratified in June 2016. ES7 contains only two new features.

[19] http://www.howtogeek.com/112455/how-to-install-iis-8-on-windows-8/

As the future of JavaScript, we want to write our code in ES6/ES7 today. But we also want our JavaScript to run on older browsers until they fade out of widespread use. We see later in this chapter how we can enjoy the benefits of ES6/ES7 today while still supporting the vast majority of the world's browsers.

This book is written with JavaScript ES7. Because ES6 ratified a majority of these new features, we'll commonly refer to these new features as ES6 features.

We've included an appendix on the ES6 syntax that we use, "Appendix B: ES6." We'll often refer to the appendix when encountering ES6 syntax for the first time, but if ever syntax seems unfamiliar to you it's worth checking Appendix B to see if it's new ES6 JavaScript syntax.

ES6 is sometimes referred to as ES2015, the year of its finalization. ES7, in turn, is often referred to as ES2016.

Getting started

Sample Code

All the code examples you find in each chapter are available in the code package that came with the book. In that code package you'll find completed versions of each app as well as boilerplates that we will use to build those apps together. Each chapter provides detailed instruction on how to follow along on your own.

While coding along with the book is not necessary, we highly recommend doing so. Playing around with examples and sample code will help solidify and strengthen concepts.

Previewing the application

We'll be building a basic React app that will allow us to touch on React's most important concepts at a high-level before diving into them in subsequent sections. Let's begin by taking a look at a working implementation of the app.

Open up the sample code folder that came with the book. Change to the `voting_app/` directory in the terminal:

```
$ cd voting_app/
```

If you're not familiar with `cd`, it stands for "change directory." If you're on a Mac, do the following to open terminal and change to the proper directory:

1. Open up `/Applications/Utilities/Terminal.app`.
2. Type `cd`, without hitting enter.
3. Tap the spacebar.
4. In the Finder, drag the `voting_app/` folder on to your terminal window.
5. Hit Enter.

Your terminal is now in the proper directory.

Throughout the book, a code block starting with a `$` signifies a command to be run in your terminal.

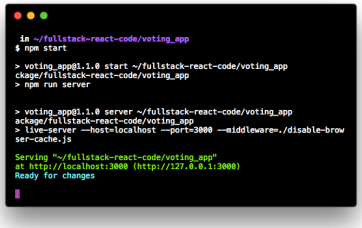
First, we'll need to use `npm` to install all our dependencies:

```
$ npm install
```

With our dependencies installed, we can boot the server using the `npm start` command

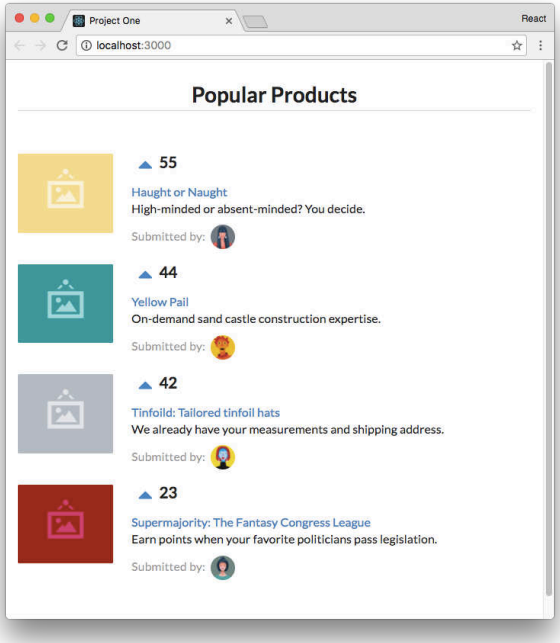
```
$ npm start
```

The boot process will print some text to the console:



Boot process output

In addition, your browser might automatically launch and open the app. If it doesn't, you can view the running application at the URL `http://localhost:3000`:



Completed version of the app

This demo app is a site like Product Hunt[20] or Reddit[21]. These sites have lists of links that users can vote on. Like those sites, in our app we can up-vote products. All

[20] http://producthunt.com
[21] http://reddit.com

products are sorted instantaneously by number of votes.

The keyboard command to quit a running Node server is CTRL+C.

Prepare the app

In the terminal, run `ls` to see the project's layout:

```
$ ls
README.md
disable-browser-cache.js
nightwatch.json
node_modules/
package.json
public/
tests/
```

If you're running on macOS or Linux, you can run `ls -1p` to format your output as we do above.

Node apps contain a `package.json` which specifies the dependencies of the project. When we ran `npm install`, npm used our `package.json` to determine which dependencies to download and install. It installed them to the folder `node_modules/`.

We explore the format of `package.json` in later chapters.

The code we'll be working with is inside the folder `public/`. Look inside that folder:

```
$ ls public
favicon.ico
images/
index.html
js/
semantic-dist/
style.css
vendor/
```

The general layout here is a common one for web apps. Inside `public/` is `index.html`, the file that we serve to browsers that request our website. As we'll see shortly, `index.html` is the centerpiece of our app. It loads in the rest of our app's assets.

Let's look inside `public/js` next:

```
$ ls public/js
app-1.js
app-2.js
app-3.js
app-4.js
app-5.js
app-6.js
app-7.js
app-8.js
app-9.js
app-complete.js
app.js
seed.js
```

Inside `public/js` is where we'll put our app's JavaScript. We'll be writing our React app inside `app.js`. `app-complete.js` is the completed version of the app that we're working towards, which we viewed a moment ago.

In addition, we've included each version of `app.js` as we build it up throughout this chapter (`app-1.js`, `app-2.js`, etc). Each code block in this chapter will reference which app version you can find it in. You can copy and paste longer code insertions from these app versions into your `app.js`.

All projects include a handy `README.md` that have instructions on how to run them.

To get started, we'll ensure `app-complete.js` is no longer loaded in `index.html`. We'll then have a blank canvas to begin work inside `app.js`.

Open up `public/index.html` in your text editor. It should look like this:

voting_app/public/index.html

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="utf-8">
    <title>Project One</title>
    <link rel="stylesheet" href="./semantic-dist/semantic.css" />
    <link rel="stylesheet" href="./style.css" />
    <script src="vendor/babel-standalone.js"></script>
    <script src="vendor/react.js"></script>
    <script src="vendor/react-dom.js"></script>
  </head>

  <body>
    <div class="main ui text container">
      <h1 class="ui dividing centered header">Popular Products</h1>
      <div id="content"></div>
    </div>
    <script src="./js/seed.js"></script>
    <script src="./js/app.js"></script>
    <!-- Delete the script tag below to get started. -->
    <script
      type="text/babel"
      data-plugins="transform-class-properties"
      src="./js/app-complete.js"
    ></script>
  </body>
```

```
</html>
```

We'll go over all the dependencies being loaded under the `<head>` tag later. The heart of the HTML document is these few lines here:

voting_app/public/index.html

```
    <div class="main ui text container">
      <h1 class="ui dividing centered header">Popular Products</h1>
      <div id="content"></div>
    </div>
```

For this project, we're using Semantic UI[22] for styling.

Semantic UI is a CSS framework, much like Twitter's Bootstrap[23]. It provides us with a grid system and some simple styling. You don't need to know Semantic UI in order to use this book. We'll provide all the styling code that you need. At some point, you might want to check out the docs Semantic UI docs[24] to get familiar with the framework and explore how you can use it in your own projects.

The `class` attributes here are just concerned with style and are safe to ignore. Stripping those away, our core markup is succinct:

```
<div>
  <h1>Popular Products</h1>
  <div id="content"></div>
</div>
```

We have a title for the page (`h1`) and a `div` with an `id` of `content`. **This `div` is where we will ultimately mount our React app**. We'll see shortly what that means.

The next few lines tell the browser what JavaScript to load. To start building our own application, let's remove the `./app-complete.js` script tag completely:

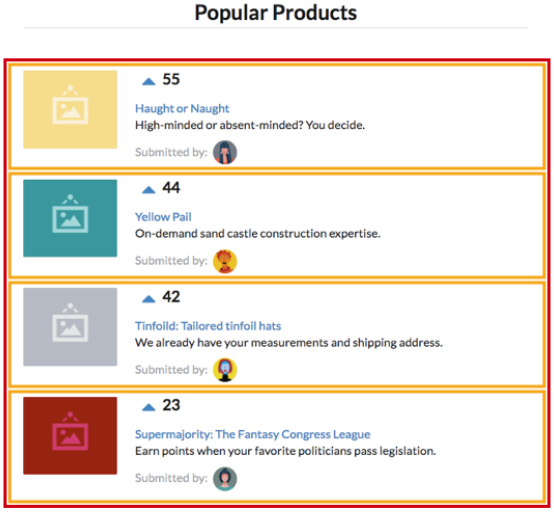
[22] http://semantic-ui.com/
[23] http://getbootstrap.com/
[24] http://semantic-ui.com/introduction/getting-started.html

```
<script src="./js/seed.js"></script>
<script src="./js/app.js"></script>
<!-- Delete the script tag below to get started. -->
<script
  type="text/babel"
  data-plugins="transform-class-properties"
  src="./js/app-complete.js"
></script>
```

After we save our updated `index.html` and reload the web browser, we'll see that our app has disappeared.

What's a component?

Building a React app is all about **components**. An individual React component can be thought of as a UI component in an app. We can break apart the interface of our app into two classes of components:



The app's components

We have a hierarchy of one parent component and many child components. We'll call these `ProductList` and `Product`, respectively:

1. `ProductList`: Contains a list of product components
2. `Product`: Displays a given product

Not only do React components map cleanly to UI components, but they are self-contained. The markup, view logic, and often component-specific style is all housed in one place. This feature makes React components reusable.

Furthermore, as we'll see in this chapter and throughout this book, React's paradigm for component data flow and interactivity is rigidly defined. In React, when the inputs for a component change, the framework simply re-renders that component. This gives us a robust UI consistency guarantee:

With a given set of inputs, the output (how the component looks on the page) will always be the same.

Our first component

Let's start off by building the `ProductList` component. We'll write all our React code for the rest of this chapter inside the file `public/js/app.js`. Let's open `app.js` and insert the component:

voting_app/public/js/app-1.js

```
class ProductList extends React.Component {
  render() {
    return (
      <div className='ui unstackable items'>
        Hello, friend! I am a basic React component.
      </div>
    );
  }
}
```

React components are **ES6 classes** that extend the class `React.Component`. We're referencing the `React` variable. `index.html` loads the React library for us so we're able to reference it here:

voting_app/public/index.html

```
    <script src="vendor/react.js"></script>
```

Our `ProductList` class has a single method, `render()`. **`render()` is the only required method for a React component**. React uses the return value from this method to determine what to render to the page.

While JavaScript is not a classical language, ES6 introduced a class declaration syntax. ES6 classes are syntactical sugar over JavaScript's prototype-based inheritance model.

We cover the important details you need to know about classes with respect to building React components. If you'd like to learn more about ES6 classes, refer to the docs on MDN[25].

There are two ways to declare React components:

(1) As ES6 classes (as above)

(2) Function components

An example of using an ES6 class:

```
class HelloWorld extends React.Component {
    render() { return <p>Hello, world!</p>; }
}
```

The same component written in a "functional component" style:

```
function HelloWorld() {
  return <p>Hello, world!</p>;
}
```

At the time of writing, both types of declarations are in widespread use. While they both do essentially the same thing, there are some important differences that we'll get to later in the book.

[25] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

> We'll be using ES6 class components through much of the book, but we'll be using function components from time-to-time as well. While it might seem cleaner to just pick one, you'll encounter (and use) both in your real-world work. So we'll cover both.

If you have some familiarity with JavaScript, the return value may be surprising:

voting_app/public/js/app-1.js

```
    return (
      <div className='ui unstackable items'>
        Hello, friend! I am a basic React component.
      </div>
    );
```

The syntax of the return value doesn't look like traditional JavaScript. We're using **JSX** (JavaScript eXtension syntax), a syntax extension for JavaScript written by Facebook. Using JSX enables us to write the markup for our component views in a familiar, HTML-like syntax. In the end, this JSX code compiles to vanilla JavaScript. Although using JSX is not a necessity, we'll use it in this book as it pairs really well with React.

If you don't have much familiarity with JavaScript, we recommend you follow along and use JSX in your React code too. You'll learn the boundaries between JSX and JavaScript with experience.

JSX

React components ultimately render HTML which is displayed in the browser. As such, the `render()` method of a component needs to describe how the view should be represented as HTML. React builds our apps with a fake representation of the Document Object Model (DOM). React calls this the *virtual DOM*. Without getting deep into details for now, React allows us to describe a component's HTML representation in JavaScript.

The Document Object Model (DOM) refers to the browser's HTML tree that makes up a web page.

JSX was created to make this JavaScript representation of HTML more HTML-like. To understand the difference between HTML and JSX, consider this JavaScript syntax:

```
React.createElement('div', {className: 'ui items'},
  'Hello, friend! I am a basic React component.'
)
```

Which can be represented in JSX as:

```
<div className='ui items'>
  Hello, friend! I am a basic React component.
</div>
```

The code readability is slightly improved in the latter example. This is exacerbated in a nested tree structure:

```
React.createElement('div', {className: 'ui items'},
  React.createElement('p', null, 'Hello, friend! I am a basic React com\
ponent.')
)
```

In JSX:

```
<div className='ui items'>
  <p>
    Hello, friend! I am a basic React component.
  </p>
</div>
```

JSX presents a light abstraction over the JavaScript version, yet the legibility benefits are huge. Readability boosts our app's longevity and makes it easier to onboard new developers.

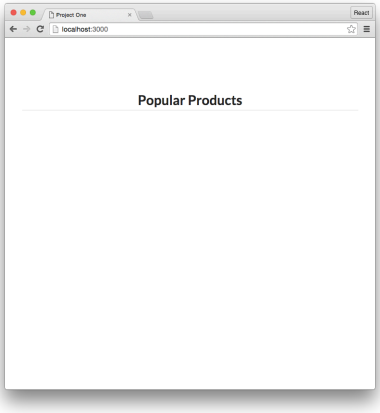
Even though the JSX above looks exactly like HTML, it's important to remember that JSX is actually just compiled into JavaScript (ex: `React.createElement('div')`).

During runtime React takes care of rendering the actual HTML in the browser for each component.

The developer console

Our first component is written and we now know that it uses a special flavor of JavaScript called JSX for improved readability.

After editing and saving our `app.js`, let's refresh the page in our web browser and see what changed:



Nothing?

Every major browser comes with a toolkit that helps developers working on JavaScript code. A central part of this toolkit is a console. Think of the console as JavaScript's primary communication medium back to the developer. If JavaScript encounters any errors in its execution, it will alert you in this developer console.

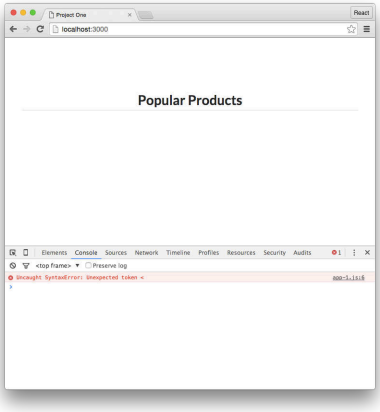
Our web server, `live-server`, should refresh the page automatically when it detects that `app.js` has changed.

To open the console in Chrome, navigate to View > Developer > JavaScript Console.

Or, just press `Command + Option + J` on a Mac or `Control + Shift + L` on Windows/Linux.

Opening the console, we are given a cryptic clue:

```
Uncaught SyntaxError: Unexpected token <
```



Error in the console

This `SyntaxError` prevented our code from running. A `SyntaxError` is thrown when the JavaScript engine encounters tokens or token order that doesn't conform to the syntax of the language when parsing code. This error type indicates some code is out of place or mistyped.

The issue? **Our browser's JavaScript parser tripped when it encountered the JSX**. The parser doesn't know anything about JSX. As far as it is concerned, this `<` is completely out of place.

As we discussed, JSX is an extension to standard JavaScript. Let's have our browser's JavaScript interpreter use this extension.

Babel

We mentioned at the beginning of the chapter that all the code in the book would be using ES6 JavaScript. However, most browsers in use today do not fully support ES6.

Babel is a JavaScript **transpiler**. **Babel turns ES6 code into ES5 code**. We call this process **transpiling**. So we can enjoy the features of ES6 today yet ensure our code still runs in browsers that only support ES5.

Another handy feature of Babel is that it understands JSX. Babel compiles our JSX into vanilla ES5 JS that our browser can then interpret and execute. We just need to instruct the browser that we want to use Babel to compile and run our JavaScript code.

The sample code's `index.html` already imports Babel in the `head` tags of `index.html`:

```
<head>
  <!-- ... -->
  <script src="vendor/babel-standalone.js"></script>
  <!-- ... -->
</head>
```

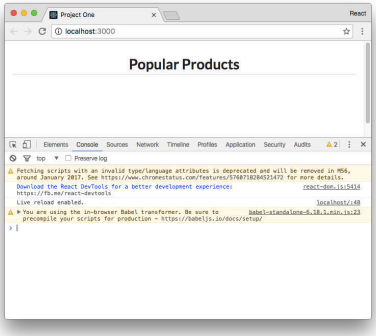
All we need to do is tell our JavaScript runtime that our code should be compiled by Babel. We can do this by setting the `type` attribute when we import the script in `index.html` to `text/babel`.

Open `index.html` and change the script tag that loads `./js/app.js`. We're going to add two attributes:

```
<script src="./js/seed.js"></script>
<script
  type="text/babel"
  data-plugins="transform-class-properties"
  src="./js/app.js"
></script>
```

The attribute `type="text/babel"` indicates to Babel that we would like it to handle the loading of this script. The attribute `data-plugins` specifies a special Babel plugin we use in this book. We discuss this plugin at the end of the chapter.

Save `index.html` and reload the page.



Still nothing. However, the console no longer has the error. Depending on your version of Chrome, you might see some warnings (highlighted in yellow as opposed to red). These warnings are safe to ignore.

Babel successfully compiled our JSX into JavaScript and our browser was able to run that JavaScript without any issues.

So what's happening? We've defined the component, **but we haven't told React to do anything with it yet**. We need to tell the React framework that our component should be inserted on this page.

Depending on your version of Chrome, you might see two errors.

The first:

```
Fetching scripts with an invalid type/language attributes is deprecated\
 and will be removed in M56, around January 2017.
```

This warning is misleading and safe to ignore. The second:

```
You are using the in-browser Babel transformer. Be sure to precompile y\
our scripts for production
```

Again, safe to ignore. To get up and running quickly, we're having Babel transpile **on-the-fly** in the browser. We explore other JavaScript transpiling strategies later in the book that are more suitable for production.

ReactDOM.render()

We need to instruct React to render this `ProductList` inside a specific DOM node.

Add the following code below the component inside `app.js`:

voting_app/public/js/app-1.js

```
class ProductList extends React.Component {
  render() {
    return (
      <div className='ui unstackable items'>
        Hello, friend! I am a basic React component.
      </div>
    );
  }
}

ReactDOM.render(
  <ProductList />,
  document.getElementById('content')
);
```

`ReactDOM` is from the `react-dom` library that we also include in `index.html`. We pass in two arguments to the `ReactDOM.render()` method. The first argument is *what* we'd like to render. The second argument is *where* to render it:

```
ReactDOM.render([what], [where]);
```

Here, for the "what," we're passing in a reference to our React component `ProductList` in JSX. For the "where," you might recall `index.html` contains a `div` tag with an `id` of `content`:

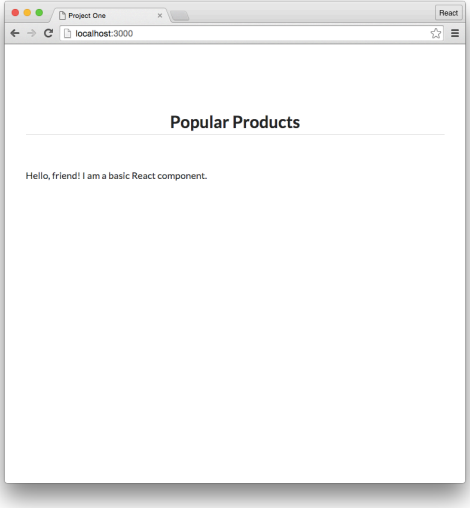
voting_app/public/index.html

```
    <div id="content"></div>
```

We pass in a reference to that DOM node as the second argument to `ReactDOM.render()`.

At this point, it's interesting to note that we use different casing between the different types of React element declarations. We have HTML DOM elements like `<div>` and a React component called `<ProductList />`. In React, native HTML elements *always* start with a lowercase letter whereas React component names *always* start with an uppercase letter.

With `ReactDOM.render()` now at the end of `app.js`, save the file and refresh the page in the browser:



Our component is rendered to the page

To recap, we wrote a React component using an ES6 class as well as JSX. We specified that we wanted Babel to transpile this code to ES5. We then used `ReactDOM.render()` to write this component to the DOM.

While an accomplishment, our current `ProductList` component is rather uninteresting. We eventually want `ProductList` to render a list of products.

Each product will be its own UI element, a fragment of HTML. We can represent each of these elements as their own component, `Product`. Central to its paradigm, React components can render other React components. We'll have `ProductList` render `Product` components, one for each product we'd like to show on the page. Each of these `Product` components will be a **child component** to `ProductList`, the **parent component**.

Building `Product`

Let's build our child component, `Product`, that will contain a product listing. Just like with the `ProductList` component, we'll declare a new ES6 class that extends `React.Component`. We'll define a single method, `render()`:

```
class Product extends React.Component {
  render() {
    return (
      <div>
        {/* ... todo ... */}
      </div>
    );
  }
}

ReactDOM.render(
  // ...
);
```

For every product, we'll add an image, a title, a description, and an avatar of the post author. The markup is below:

voting_app/public/js/app-2.js

```
class Product extends React.Component {
  render() {
    return (
      <div className='item'>
        <div className='image'>
          <img src='images/products/image-aqua.png' />
        </div>
        <div className='middle aligned content'>
          <div className='description'>
            <a>Fort Knight</a>
            <p>Authentic renaissance actors, delivered in just two week\
s.</p>
          </div>
          <div className='extra'>
            <span>Submitted by:</span>
            <img
              className='ui avatar image'
              src='images/avatars/daniel.jpg'
            />
          </div>
        </div>
      </div>
    );
  }
}

ReactDOM.render(
```

The title of the code block above references the location of this example in the book's code download (`voting_app/public/js/app-2.js`). This pattern will be common throughout the book.

If you want to copy and paste the markup into your `app.js`, refer to this file.

Again, we've used a bit of SemanticUI styling in our code here. As we discussed previously, this JSX code will be transpiled to regular JavaScript in the browser. Because it runs in the browser as JavaScript, we cannot use any reserved JavaScript words in JSX. `class` is a reserved word. Therefore, React has us use the attribute name `className`. Later, when the HTML element reaches the page, this attribute name will be written as `class`.

Structurally, the `Product` component is similar to the `ProductList` component. Both have a single `render()` method which returns information about an eventual HTML structure to display.

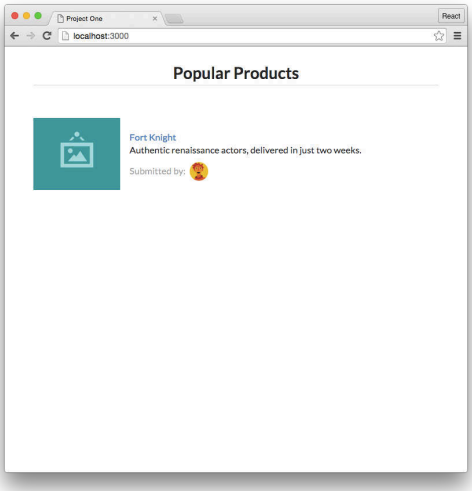
Remember, the JSX components return is *not* actually the HTML that gets rendered, but is the *representation* that we want React to render in the DOM.

To use the `Product` component, we can modify the render output of our parent `ProductList` component to include the child `Product` component:

voting_app/public/js/app-2.js

```
class ProductList extends React.Component {
  render() {
    return (
      <div className='ui unstackable items'>
        <Product />
      </div>
    );
  }
}
```

Save `app.js` and refresh the web browser.



With this update, we now have two React components being rendered in our app. The `ProductList` parent component is rendering the `Product` component as a child nested underneath its root `div` element.

While neat, at the moment the child `Product` component is static. We hard-coded an image, the name, the description, and author details. To use this component in a meaningful way, we'll want to change it to be data-driven and therefore dynamic.

Making `Product` data-driven

Driving the `Product` component with data will allow us to dynamically render the component based upon the data that we give it. Let's familiarize ourselves with the product data model.

The data model

In the sample code, we've included a file inside `public/js` called `seed.js`. `seed.js` contains some example data for our products (it will "seed" our app's data). The `seed.js` file contains a JavaScript object called `Seed.products`. `Seed.products` is an array of JavaScript objects, each representing a product object:

voting_app/public/js/seed.js

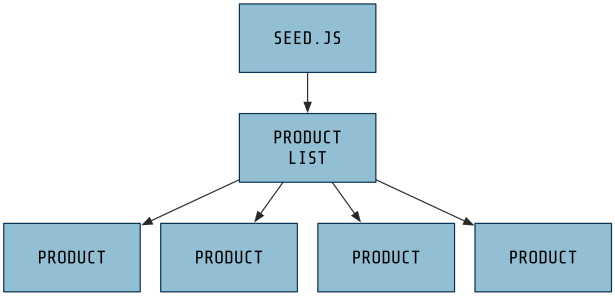
```
const products = [
  {
    id: 1,
    title: 'Yellow Pail',
    description: 'On-demand sand castle construction expertise.',
    url: '#',
    votes: generateVoteCount(),
    submitterAvatarUrl: 'images/avatars/daniel.jpg',
    productImageUrl: 'images/products/image-aqua.png',
  },
```

Each product has a unique `id` and a handful of properties including a `title` and `description`. `votes` are randomly generated for each one with the included function `generateVoteCount()`.

We can use the same attribute keys in our React code.

Using props

We want to modify our `Product` component so that it no longer uses static, hard-coded attributes. Instead, we want it to be able to accept data passed down from its parent, `ProductList`. Setting up our component structure in this way enables our `ProductList` component to dynamically render any number of `Product` components that each have their own unique attributes. Data flow will look like this:



The way data flows from parent to child in React is through **props**. When a parent renders a child, it can send along props the child depends on.

Let's see this in action. First, let's modify `ProductList` to pass down props to `Product`. `seed.js` will save us from having to create a bunch of data manually. Let's pluck the first object off of the `Seed.products` array and use that as data for a single product:

voting_app/public/js/app-3.js

```
class ProductList extends React.Component {
  render() {
    const product = Seed.products[0];
    return (
      <div className='ui unstackable items'>
        <Product
          id={product.id}
          title={product.title}
          description={product.description}
          url={product.url}
          votes={product.votes}
          submitterAvatarUrl={product.submitterAvatarUrl}
          productImageUrl={product.productImageUrl}
        />
      </div>
    );
  }
}
```

Here, the `product` variable is set to a JavaScript object that describes the first of our products. We pass the product's attributes along individually to the `Product` component using the syntax `[propName]=[propValue]`. The syntax of assigning attributes in JSX is exactly the same as HTML and XML.

There are two interesting things here. The first is the braces (`{}`) around each of the property values:

voting_app/public/js/app-3.js

```
            id={product.id}
```

In JSX, braces are a delimiter, signaling to JSX that what resides in-between the braces is a JavaScript expression. The other delimiter is using quotes for strings, like this:

```
id='1'
```

JSX attribute values **must** be delimited by either braces or quotes.

If type is important and we want to pass in something like a Number or a `null`, use braces.

If you've programmed with ES5 JavaScript before, you might be used to using `var` as opposed to `const` or `let`. See "Appendix B" for more on these new declarations.

Now the `ProductList` component is passing props down to `Product`. Our `Product` component isn't using them yet, so let's modify the component to use these props.

In React, a component can access all its props through the object **`this.props`**. Inside of `Product`, the `this.props` object will look like this:

```
{
  "id": 1,
  "title": "Yellow Pail",
  "description": "On-demand sand castle construction expertise.",
  "url": "#",
  "votes": 41,
  "submitterAvatarURL": "images/avatars/daniel.jpg",
  "productImageUrl": "images/products/image-aqua.png"
}
```

Let's swap out everywhere that we hard-coded data and use props instead. While we're here, we'll add a bit more markup like the description and the up-vote icon:

voting_app/public/js/app-3.js

```
class Product extends React.Component {
  render() {
    return (
      <div className='item'>
        <div className='image'>
          <img src={this.props.productImageUrl} />
        </div>
        <div className='middle aligned content'>
          <div className='header'>
            <a>
              <i className='large caret up icon' />
            </a>
            {this.props.votes}
          </div>
          <div className='description'>
            <a href={this.props.url}>
              {this.props.title}
            </a>
            <p>
              {this.props.description}
            </p>
          </div>
          <div className='extra'>
            <span>Submitted by:</span>
            <img
              className='ui avatar image'
              src={this.props.submitterAvatarUrl}
            />
          </div>
        </div>
      </div>
    );
  }
}
```

Again, everywhere inside of our JSX where we're interpolating a variable we delimit the variable with braces (`{}`). Note that we're inserting data both as text content inside of tags like this:

voting_app/public/js/app-3.js

```
          <div className='header'>
            <a>
              <i className='large caret up icon' />
            </a>
            {this.props.votes}
          </div>
```

As well as for attributes on HTML elements:

voting_app/public/js/app-3.js

```
          <img src={this.props.productImageUrl} />
```

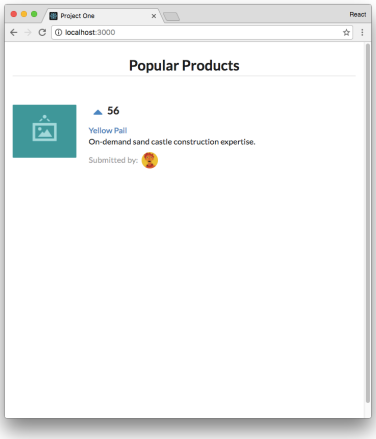
Interweaving props with HTML elements in this way is how we create dynamic, data-driven React components.

`this` is a special keyword in JavaScript. The details about `this` are a bit nuanced, but for the purposes of the majority of this book, **`this` will be bound to the React component class**. So, when we write `this.props` inside the component, we're accessing the `props` property on the component. When we diverge from this rule in later sections, we'll point it out.

For more details on `this`, check out this page on MDN[26].

With our updated `app.js` file saved, let's refresh the web browser again:

[26] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this



The `ProductList` component now shows a single product listed, the first object pulled from `Seed`.

We're getting somewhere interesting. Our `Product` component is now data-driven. Based on the props it receives it can render any product that we'd like.

Our code is poised to have `ProductList` render any number of products. We just need to configure the component to render some number of `Product` components, one for each product we'd like to represent on the page.

Rendering multiple products

To render multiple products, first we'll have `ProductList` generate an array of `Product` components. Each will be derived from an individual object in the `Seed` array. We'll use map to do so:

voting_app/public/js/app-4.js

```
class ProductList extends React.Component {
  render() {
    const productComponents = Seed.products.map((product) => (
      <Product
        key={'product-' + product.id}
        id={product.id}
        title={product.title}
        description={product.description}
        url={product.url}
        votes={product.votes}
        submitterAvatarUrl={product.submitterAvatarUrl}
        productImageUrl={product.productImageUrl}
      />
    ));
```

The function passed to `map` returns a `Product` component. This `Product` is created just as before with props pulled from the object in `Seed`.

We pass an arrow function to `map`. Arrow functions were introduced in ES6. For more info, see "Appendix B."

As such, the `productComponents` variable ends up being an array of `Product` components:

```
// Our `productComponents` array
[
  <Product id={1} ... />,
  <Product id={2} ... />,
  <Product id={3} ... />,
  <Product id={4} ... />
]
```

Notably, we're able to represent the `Product` component instance in JSX inside of `return`. It might seem odd at first that we're able to have a JavaScript array of

JSX elements, but remember that Babel will transpile the JSX representation of each `Product` (`<Product />`) into regular JavaScript:

```
// What `productComponents` looks like in JavaScript
[
  React.createElement(Product, { id: 1, ... }),
  React.createElement(Product, { id: 2, ... }),
  React.createElement(Product, { id: 3, ... }),
  React.createElement(Product, { id: 4, ... })
]
```

Array's `map()`

Array's `map` method takes a function as an argument. It calls this function with each item inside of the array (in this case, each object inside `Seed.products`) and builds a **new** array by using the return value from each function call.

Because the `Seed.products` array has four items, `map` will call this function four times, once for each item. When `map` calls this function, it passes in as the first argument an item. The return value from this function call is inserted into the new array that `map` is constructing. After handling the last item, `map` returns this new array. Here, we're storing this new array in the variable `productComponents`.

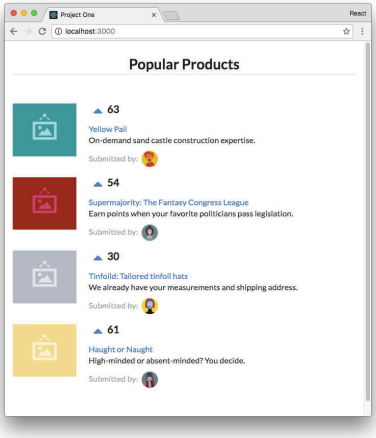
Note the use of the `key={'product-' + product.id}` prop. React uses this special property to create unique bindings for each instance of the `Product` component. The `key` prop is not used by our `Product` component, but by the React framework. It's a special property that we discuss deeper in the chapter "Advanced Component Configuration." For the time being, it's enough to note that this property needs to be unique per React component in a list.

Now, below the declaration of `productComponents`, we need to modify the return value of `render`. Before, we were rendering a single `Product` component. Now, we can render our array `productComponents`:

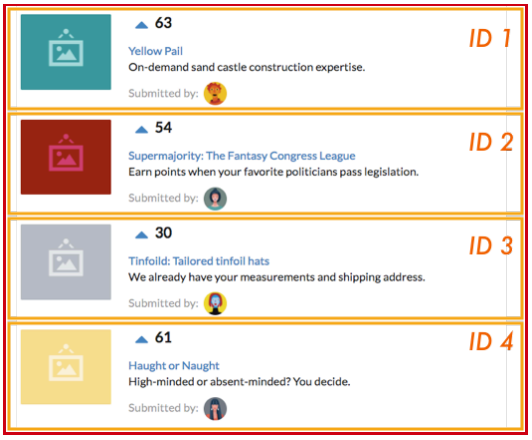
voting_app/public/js/app-4.js

```
    return (
      <div className='ui unstackable items'>
        {productComponents}
      </div>
    );
```

Refreshing the page, we'll see all four products from `Seed` listed:



We now have five total React components at work. We have a single parent component, `ProductList`. `ProductList` contains four child `Product` components, one for each product object in the `Seed.products` array in `seed.js`:



`Product` components inside of the `ProductList` component

At the moment, our products aren't sorted by the number of votes they have. Let's sort them. We'll use Array's sort method to do so. We'll sort the products first before the line where we build our `productComponents` array:

voting_app/public/js/app-5.js

```
class ProductList extends React.Component {
  render() {
    const products = Seed.products.sort((a, b) => (
      b.votes - a.votes
    ));
    const productComponents = products.map((product) => (
      <Product
```

Refreshing the page, we'll see our products are sorted.

`sort()` mutates the original array it was called on. While fine for now, elsewhere in the book we discuss why mutating arrays or objects can be a dangerous pattern.

In the markup for `Product` above, we added an 'up-vote' caret icon. If we click on one of these buttons, we'll see that nothing happens. We've yet to hook up an event to the button.

Although we have a data-driven React app running in our web browser, this page still lacks interactivity. While React has given us an easy and clean way to organize our HTML thus far and enabled us to drive HTML generation based on a flexible, dynamic JavaScript object, we've still yet to tap into its true power: creating dynamic interfaces.

The rest of this book digs deep into this power. Let's start with something simple: the ability to up-vote a given product.

Array's `sort()` method takes an optional function as an argument. If the function is omitted, it will just sort the array by each item's Unicode code point value. This is rarely what a programmer desires. If a function is supplied, elements are sorted according to the function's return value.

On each iteration, the arguments `a` and `b` are two subsequent elements in the array. Sorting depends on the return value of the function:

1. If the return value is less than `0`, `a` should come first (have a lower index).
2. If the return value is greater than `0`, `b` should come first.
3. If the return value is equal to `0`, leave order of `a` and `b` unchanged with respect to each other.

React the vote (your app's first interaction)

When the up-vote button on each one of the `Product` components is clicked, we expect it to update the `votes` attribute for that `Product`, increasing it by one.

But the `Product` component can't modify its `votes`. **`this.props` is immutable**.

While the child can read its props, it can't modify them. A child does not own its props. In our app, **the parent component `ProductList` owns the props given to `Product`**. React favors the idea of *one-way data flow*. This means that data changes come from the "top" of the app and are propagated "downwards" through its various components.

A child component does not own its `props`. Parent components own the `props` of child components.

We need a way for the `Product` component to let `ProductList` know that a click on its up-vote icon occurred. We can then have `ProductList`, the owner of the product's data, update the vote count for that product. The updated data will then flow downward from the `ProductList` component to the `Product` component.

In JavaScript, if we treat an array or object as **immutable** it means we cannot or should not make modifications to it.

Propagating the event

We know that parents communicate data to children through props. Because props are immutable, children need some way to communicate events to parents. The parents could then make whatever data changes might be necessary.

We can pass down *functions* as props too. We can have the `ProductList` component give each `Product` component a function to call when the up-vote button is clicked. Functions passed down through props are the canonical manner in which children communicate events with their parent components.

Let's see this in practice. We'll start by having up-votes log a message to the console. Later, we'll have up-votes increment the `votes` attribute on the target product.

The function `handleProductUpVote` in `ProductList` will accept a single argument, `productId`. The function will log the product's `id` to the console:

voting_app/public/js/app-6.js

```
class ProductList extends React.Component {
  handleProductUpVote(productId) {
    console.log(productId + ' was upvoted.');
  }

  render() {
```

Next, we'll pass this function down as a prop to each `Product` component. We'll name the prop `onVote`:

voting_app/public/js/app-6.js

```
    const productComponents = products.map((product) => (
      <Product
        key={'product-' + product.id}
        id={product.id}
        title={product.title}
        description={product.description}
        url={product.url}
        votes={product.votes}
        submitterAvatarUrl={product.submitterAvatarUrl}
        productImageUrl={product.productImageUrl}
        onVote={this.handleProductUpVote}
      />
    ));
```

We can now access this function inside `Product` via `this.props.onVote`.

Let's write a function inside `Product` that calls this new prop-function. We'll name the function `handleUpVote()`:

voting_app/public/js/app-6.js

```
  // Inside `Product`
  handleUpVote() {
    this.props.onVote(this.props.id);
  }

  render() {
```

We invoke the prop-function `this.props.onVote` with the `id` of the product. Now, we just need to call this function every time the user clicks the caret icon.

In React, we can use the special attribute **`onClick`** to handle mouse click events.

We'll set the `onClick` attribute on the `a` HTML tag that is the up-vote button. We'll instruct it to call `handleUpVote()` whenever it is clicked:

voting_app/public/js/app-6.js

```
        {/* Inside `render` for Product` */}
        <div className='middle aligned content'>
          <div className='header'>
            <a onClick={this.handleUpVote}>
              <i className='large caret up icon' />
            </a>
            {this.props.votes}
          </div>
```

When the user clicks the up-vote icon, it will trigger a chain of function calls:

1. User clicks the up-vote icon.
2. React invokes `Product` component's `handleUpVote`.
3. `handleUpVote` invokes its prop `onVote`. This function lives inside the parent `ProductList` and logs a message to the console.

There's one last thing we need to do to make this work. Inside the function `handleUpVote()` we refer to `this.props`:

voting_app/public/js/app-6.js

```
  handleUpVote() {
    this.props.onVote(this.props.id);
  }
```

Here's the odd part: When working inside `render()`, we've witnessed that `this` is always bound to the component. But inside our custom component method `handleUpVote()`, `this` is actually `null`.

Binding custom component methods

In JavaScript, the special `this` variable has a different **binding** depending on the context. For instance, inside `render()` we say that `this` is "bound" to the component. Put another way, `this` "references" the component.

Understanding the binding of `this` is one of the trickiest parts of learning JavaScript programming. Given this, it's fine for a beginner React programmer to not understand all the nuances at first.

In short, we want `this` inside `handleUpVote()` to reference the component, just like it does inside `render()`. But why does `this` inside `render()` reference the component while `this` inside `handleUpVote()` does not?

For the `render()` function, **React binds `this` to the component for us**. React specifies a default set of special API methods. `render()` is one such method. As we'll see at the end of the chapter, `componentDidMount()` is another. For each of these special React methods, React will bind the `this` variable to the component automatically.

So, **any time we define our own custom component methods, we have to manually bind `this` to the component ourselves**. There's a pattern that we use to do so.

Add the following `constructor()` function to the top of `Product`:

voting_app/public/js/app-6.js

```
class Product extends React.Component {
  constructor(props) {
    super(props);

    this.handleUpVote = this.handleUpVote.bind(this);
  }
```

`constructor()` is a special function in a JavaScript class. JavaScript invokes `constructor()` whenever an object is created via a class. If you've never worked with an object-oriented language before, it's sufficient to know that React invokes `constructor()` first thing when initializing our component. React invokes `constructor()` with the component's props.

Because `constructor()` is called when initializing our component, we'll use it for a couple different types of situations in the book. For our current purposes, it's enough to know that whenever we want to bind custom component methods to a React component class, we can use this pattern:

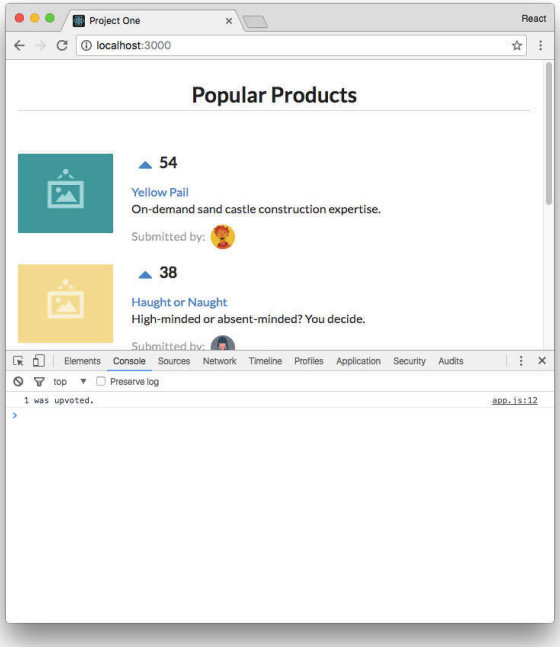
```
class MyReactComponent extends React.Component {
  constructor(props) {
    super(props); // always call this first

    // custom method bindings here
    this.someFunction = this.someFunction.bind(this);
  }
}
```

If you feel comfortable reading further details on this pattern, see the aside "Binding in `constructor()`".

At the end of the chapter, we'll use an experimental JavaScript feature that allows us to bypass this pattern. However, when working with regular ES7 JavaScript, it's important to keep this pattern in mind:

When defining custom methods on our React component classes, we must perform the binding pattern inside `constructor()` so that `this` references our component.

Saving our updated `app.js`, refreshing our web browser, and clicking an up-vote will log some text to our JavaScript console:



The events are being propagated up to the parent!

`ProductList` is the owner of the product data. And `Product` is now informing its parent whenever a user up-votes a product. Our next task is to update the vote count on the product.

But where do we perform this update? At the moment, our app doesn't have a place to store and manage data. `Seed` should be thought of as a seed of example data, not our app's datastore.

What our app is currently missing is **state**.

In fact, while we might be tempted to update the vote count in `Seed.products` like this:

```
// Would this work?
Seed.products.forEach((product) => {
  if (product.id === productId) {
    product.votes = product.votes + 1;
  }
});
```

Doing so wouldn't work. When updating `Seed`, *our React app would not be informed of the change*. On the user interface there would be no indication that the vote count was incremented.

Binding in `constructor()`

The first thing we do in `constructor()` is call `super(props)`. The `React.Component` class that our `Product` class is extending defines its own `constructor()`. By calling `super(props)`, we're invoking *that* `constructor()` function first.

Importantly, **the `constructor()` function defined by `React.Component` will bind `this` inside *our* `constructor()` to the component**. Because of this, it's a good practice to always call `super()` first whenever you declare a `constructor()` for your component.

After calling `super()`, we call `bind()` on our custom component method:

```
this.handleUpVote = this.handleUpVote.bind(this);
```

Function's `bind()` method allows you to specify what the `this` variable inside a function body should be set to. What we're doing here is a common JavaScript pattern. We're *redefining* the component method `handleUpVote()`, setting it to the same function but bound to `this` (the component). Now, whenever `handleUpVote()` executes, `this` will reference the component as opposed to `null`.

Does this `bind()` pattern feel a little weird? It turns out, you don't have to do it when you use "functional" components. Perhaps you've heard of React Hooks. This `bind()` pattern is unnecessary with functional components and Hooks.

That said, we haven't talked much about functional components or Hooks - and they introduce their own type of complexity. For now, using class components is, overall, easier to learn and use - so that's what we'll do here.

Using state

Whereas props are immutable and owned by a component's parent, **state is owned by the component**. `this.state` is private to the component and as we'll see can be updated with `this.setState()`.

Critically, **when the state or props of a component update, the component will re-render itself**.

Every React component is rendered as a function of its "props" and "state". This rendering is deterministic. This means that given a set of props and a set of state, a React component will always render a single way. As we mentioned at the beginning of the chapter, this approach makes for a powerful UI consistency guarantee.

Because we are mutating the data for our products (the number of votes), **we should consider this data to be stateful**. `ProductList` will be the owner of this state. It will then pass this state down as props to `Product`.

At the moment, `ProductList` is reading directly from `Seed` inside `render()` to grab the products. Let's move this data into the component's state.

When adding state to a component, the first thing we do is define what the **initial state** should look like. Because `constructor()` is called when initializing our component, it's the best place to define our initial state.

In React components, state is an object. The shape of our `ProductList` state object will look like this:

```
// Shape of the `ProductList` state object
{
  products: <Array>,
}
```

We'll initialize our state to an object with an empty `products` array. Add this `constructor()` to `ProductList`:

voting_app/public/js/app-7.js

```
class ProductList extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      products: [],
    };
  }
```

Like with our `constructor()` call in `Product`, the first line in `constructor()` is the `super(props)` call. The first line in any `constructor()` functions we write for React components will always be this same line.

Technically, because we don't supply `ProductList` any props, we don't need to propagate the `props` argument to `super()`. But it's a good habit to get into and helps avoid odd bugs in the future.

Next, with our state initialized, let's modify the `ProductList` component's `render` function so that it uses state as opposed to reading from `Seed`. We read the state with `this.state`:

voting_app/public/js/app-7.js

```
render() {
  const products = this.state.products.sort((a, b) => (
    b.votes - a.votes
  ));
```

Calling `.sort()` on `this.state.products` will *technically* mutate that array, which is generally considered bad practice. We're cutting a *slight* corner just to make the example clearer. A better (but longer) way to do this would be to copy this array when sorting - which we're going to talk about below (so don't worry too much about it now).

`ProductList` is driven by its own state now. If we were to save and refresh now, all our products would be missing. We don't have any mechanisms in `ProductList` that add products to its state.

Setting state with `this.setState()`

It's good practice to initialize components with "empty" state as we've done here. We explore the reasoning behind this when working asynchronously with servers in the chapter "Components & Servers."

However, after our component is initialized, we want to seed the state for `ProductList` with the data in `Seed`.

React specifies a set of **lifecycle methods**. React invokes one lifecycle method, `componentDidMount()`, after our component has mounted to the page. We'll seed the state for `ProductList` inside this method.

We explore the rest of the lifecycle methods in the chapter "Advanced Component Configuration."

Knowing this, we might be tempted to set the state to `Seed.products` inside `componentDidMount()` like this:

```
class ProductList extends React.Component {
  // ...
  // Is this valid ?
  componentDidMount() {
    this.state = Seed.products;
  }
  // ...
}
```

However, this is invalid. The only time we can modify the state in this manner is in `constructor()`. **For all state modifications after the initial state, React provides components the method `this.setState()`**. Among other things, this method triggers the React component to re-render which is essential after the state changes.

Never modify state outside of `this.setState()`. This function has important hooks around state modification that we would be bypassing.

We discuss state management in detail throughout the book.

Add `componentDidMount()` to `ProductList` now. We'll use `setState()` to seed the component's state:

voting_app/public/js/app-8.js

```
class ProductList extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      products: [],
    };
  }

  componentDidMount() {
    this.setState({ products: Seed.products });
  }
```

The component will mount with an empty state `this.state.products` array. After mounting, we populate the state with data from `Seed`. The component will re-render and our products will be displayed. This happens at a speed that is imperceptible to the user.

If we save and refresh now, we see that the products are back.

Updating state and immutability

Now that `ProductList` is managing the products in state, we're poised to make modifications to this data in response to user input. Specifically, we want to increment the `votes` property on a product when the user votes for it.

We just discussed that we can only make state modifications using `this.setState()`. So while a component can update its state, **we should treat the `this.state` object as immutable**.

As touched on earlier, if we treat an array or object as immutable we never make modifications to it. For example, let's say we have an array of numbers in state:

```
this.setState({ nums: [ 1, 2, 3 ]});
```

If we want to update the state's `nums` array to include a `4`, we might be tempted to use `push()` like this:

```
this.setState({ nums: (this.state.nums.push(4), this.state.nums) });
```

On the surface, it might appear as though we've treated `this.state` as immutable. However, the `push()` method *modifies the original array*:

```
console.log(this.state.nums);
// [ 1, 2, 3 ]
this.state.nums.push(4);
// 4 <-- the length of the array
console.log(this.state.nums);
// [ 1, 2, 3, 4] <-- Uh-oh!
```

Although we invoke `this.setState()` immediately after we push `4` onto the array, we're still modifying `this.state` outside of `setState()` and this is bad practice.

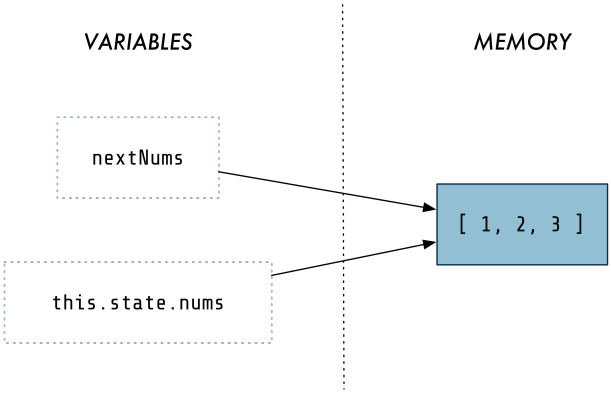
Part of the reason this is bad practice is because **`setState()` is actually asynchronous**. There is no guarantee *when* React will update the state and re-render our component. We touch on this in the "Advanced Component Configuration" chapter.

So, while we eventually called `this.setState()`, we unintentionally modified the state before that.

This next approach doesn't work either:

```
const nextNums = this.state.nums;
nextNums.push(4);
console.log(nextNums);
// [ 1, 2, 3, 4]
console.log(this.state.nums);
// [ 1, 2, 3, 4]  <-- Nope!
```

Our new variable `nextNums` references the same array as `this.state.nums` in memory:



Both variables reference the same array in memory

So when we modify the array with `push()`, we're modifying the same array that `this.state.nums` is pointing to.

Instead, we can use Array's `concat()`. **`concat()` creates a new array** that contains the elements of the array it was called on followed by the elements passed in as arguments.

With `concat()`, we can avoid mutating state:

```
console.log(this.state.nums);
// [ 1, 2, 3 ]
const nextNums = this.state.nums.concat(4);
console.log(nextNums);
// [ 1, 2, 3, 4]
console.log(this.state.nums);
// [ 1, 2, 3 ]  <-- Unmodified!
```

We touch on immutability throughout the book. While you might be able to "get away" with mutating the state in many situations, it's better practice to treat state as immutable.

Treat the state object as immutable. It's important to understand which Array and Object methods modify the objects they are called on.

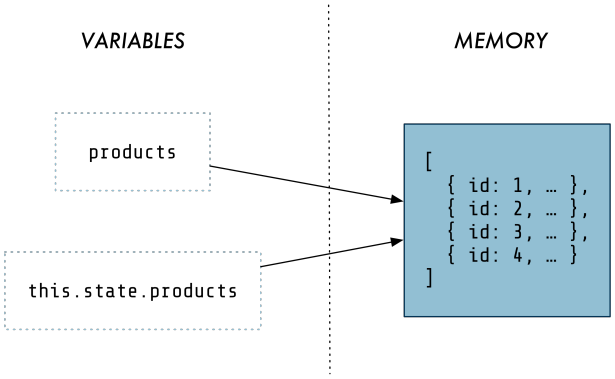
If an array is passed in as an argument to `concat()`, its elements are appended to the new array. For example:

```
> [ 1, 2, 3 ].concat([ 4, 5 ]);
=> [ 1, 2, 3, 4, 5 ]
```

Knowing that we want to treat the state as immutable, the following approach to handling up-votes would be problematic:

```
// Inside `ProductList`
// Invalid
handleProductUpVote(productId) {
  const products = this.state.products;
  products.forEach((product) => {
    if (product.id === productId) {
      product.votes = product.votes + 1;
    }
  });
  this.setState({
    products: products,
  });
}
```

When we initialize `products` to `this.state.products`, `products` references the same array in memory as `this.state.products`:



Both variables reference the same array in memory

So, when we modify a `product` object by incrementing its vote count inside `forEach()`, *we're modifying the original product object in state.*

Instead, we should create a *new* array of products. And if we modify one of the product objects, we should modify a *clone* of the object as opposed to the original one.

Let's see what a `handleProductUpVote()` implementation looks like that treats state as immutable. We'll see it in full then break it down:

voting_app/public/js/app-9.js

```
// Inside `ProductList`
handleProductUpVote(productId) {
  const nextProducts = this.state.products.map((product) => {
    if (product.id === productId) {
      return Object.assign({}, product, {
        votes: product.votes + 1,
      });
    } else {
      return product;
    }
  });
  this.setState({
    products: nextProducts,
```

```
    });
  }
```

First, we use `map()` to traverse the `products` array. Importantly, `map()` returns a *new* array as opposed to modifying the array `this.state.products`.

Next, we check if the current `product` matches `productId`. If it does, we create a *new* object, copying over the properties from the original product object. We then *overwrite* the `votes` property on our new product object. We set it to the incremented vote count. We do this using Object's `assign()` method:

voting_app/public/js/app-9.js

```
      if (product.id === productId) {
        return Object.assign({}, product, {
          votes: product.votes + 1,
        });
```

We use `Object.assign()` a lot for avoiding mutating objects. For more details on the method, check out "Appendix B."

If the current `product` is not the one specified by `productId`, we return it unmodified:

voting_app/public/js/app-9.js

```
      } else {
        return product;
      }
```

Finally, we use `setState()` to update the state.

`map()` is creating a new array. So you might ask: Why can't we modify the `product` object directly? Like this:

```
if (product.id === productId) {
  product.votes = product.votes + 1;
}
```

While we're creating a new array, **the variable `product` here still references the product object sitting on the original array in state**. Therefore, if we make changes to it we'll be modifying the object in state. So we use `Object.assign()` to clone the original into a new object and then modify the `votes` property on that new object.

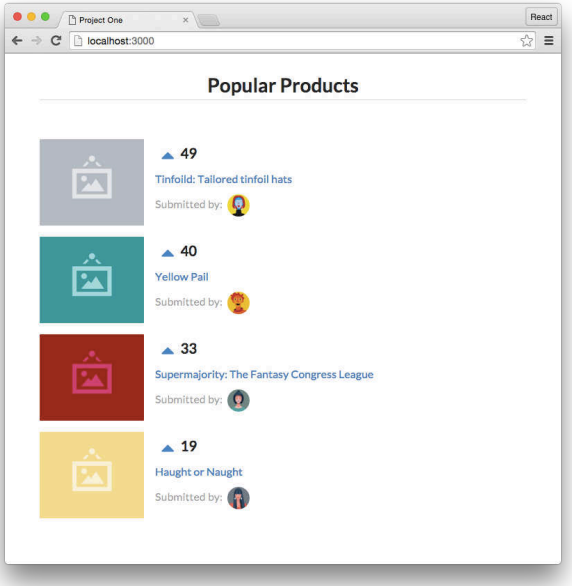
Our state update for up-votes is in place. There's one last thing we have to do: Our custom component method `handleProductUpVote()` is now referencing `this`. We need to add a `bind()` call like the one we have for `handleUpVote()` in `Product`:

voting_app/public/js/app-9.js

```
class ProductList extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      products: [],
    };

    this.handleProductUpVote = this.handleProductUpVote.bind(this);
  }
```

Now `this` in `handleProductUpVote()` references our component.

Our app should finally be responsive to user interaction. Save `app.js`, refresh the browser, and cross your fingers:



At last, the vote counters are working! Try up-voting a product a bunch of times and notice how it immediately jumps above products with lower vote counts.

Refactoring with the Babel plugin `transform-class-properties`

In this last section, we'll explore a possible refactor that we can make to our class components using an experimental JavaScript feature. For reasons you'll soon see, this feature is popular among React developers. Because the community is still adopting this feature, we expose you to both class component styles throughout the book.

We're able to access this feature using Babel's library of **plugins and presets**.

Babel plugins and presets

We've been using Babel in this project to give us the ability to write modern JavaScript that will run in a majority of browsers on the web. Specifically, our code

has been using Babel to convert ES6 syntax and JSX into vanilla ES5 JavaScript.

There's a few ways to integrate Babel into your project. We've been using `babel-standalone` which allows us to setup Babel quickly for use directly in the browser.

`babel-standalone` by default uses two **presets**. In Babel, **a *preset* is a set of *plugins* used to support particular language features**. The two presets Babel has been using by default:

- `es2015`[27]: Adds support for ES2015 (or ES6) JavaScript
- `react`[28]: Adds support for JSX

Remember: ES2015 is just another name used for ES6. We let Babel use the default `es2015` preset for this project because we don't need or use either of ES7's two new features.

JavaScript is an ever-changing language. At its current pace, new syntax will be ratified for adoption on a yearly basis.

Because JavaScript will continue to evolve, tools like Babel are here to stay. Developers want to take advantage of the latest language features. But it takes time for browsers to update their JavaScript engines. And it takes even more time for the majority of the public to upgrade their browsers to the latest versions. Babel closes this gap. It enables a codebase to evolve along with JavaScript without leaving older browsers behind.

Beyond ES7, proposed JavaScript features can exist in various **stages**. A feature can be an experimental proposal, one that the community is still working out the details for ("stage 1"). Experimental proposals are at risk of being dropped or modified at any time. Or a feature might already be "ratified," which means it will be included in the next release of JavaScript ("stage 4").

We can customize Babel with presets and plugins to take advantage of these upcoming or experimental features.

In this book, we generally avoid features that are experimental. However, there is one feature that looks to be ratified that we make an exception for: property initializers.

[27] https://babeljs.io/docs/plugins/preset-es2015/
[28] https://babeljs.io/docs/plugins/preset-react/

We avoid features that are experimental because we don't want to teach features that might be modified or dropped. For your own projects, it's up to you and your team to decide how "strict" you want to be about the JavaScript features that you use.

If you'd like to read more about the various Babel presets and plugins, check out the docs[29].

Property initializers

Property initializers are detailed in the proposal "ES Class Fields & Static Properties[30]." While an experimental feature that has yet to be ratified, property initializers offer a compelling syntax that greatly simplify React class components. This feature works so well with React that the Facebook team has written about using it internally[31].

Property initializers are available in the Babel plugin `transform-class-properties`. Recall that we specified this plugin for `app.js` inside `index.html`:

```
<script
  type="text/babel"
  data-plugins="transform-class-properties"
  src="./js/app.js"
></script>
```

Therefore, we're ready to use this feature in our code. The best way to understand what this feature gives us is to see it in action.

Refactoring `Product`

Inside `Product`, we defined the custom component method `handleUpVote`. As we discussed, because `handleUpVote` is not part of the standard React component API, React does not bind `this` inside the method to our component. So we had to perform a manual binding trick inside `constructor`:

[29] https://babeljs.io/docs/plugins/
[30] https://github.com/tc39/proposal-class-fields
[31] https://old.babeljs.io/blog/2015/06/07/react-on-es6-plus

voting_app/public/js/app-9.js

```
class Product extends React.Component {
  constructor(props) {
    super(props);

    this.handleUpVote = this.handleUpVote.bind(this);
  }

  handleUpVote() {
    this.props.onVote(this.props.id);
  }

  render() {
```

With the `transform-class-properties` plugin, **we can write `handleUpVote` as an arrow function. This will ensure `this` inside the function is bound to the component**, as expected:

voting_app/public/js/app-complete.js

```
class Product extends React.Component {
  handleUpVote = () => (
    this.props.onVote(this.props.id)
  );

  render() {
```

Using this feature, we can drop `constructor()`. There is no need for the manual binding call.

Note that methods that are part of the standard React API, like `render()`, will remain as class methods. If we write a custom component method in which we want `this` bound to the component, we write it as an arrow function.

Refactoring `ProductList`

We can give the same treatment to `handleProductUpVote` inside `ProductList`. In addition, property initializers give us an alternative way to define the initial state of a component.

Before, we used `constructor()` in `ProductList` to both bind `handleProductUpVote` to the component and define the component's initial state:

```
class ProductList extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      products: [],
    };

    this.handleProductUpVote = this.handleProductUpVote.bind(this);
  }
```

With property initializers, we no longer need to use `constructor`. Instead, we can define the initial state like this:

voting_app/public/js/app-complete.js

```
class ProductList extends React.Component {
  state = {
    products: [],
  };
```

And, if we define `handleProductUpVote` as an arrow function, `this` will be bound to the component as desired:

voting_app/public/js/app-complete.js

```
  handleProductUpVote = (productId) => {
    const nextProducts = this.state.products.map((product) => {
      if (product.id === productId) {
        return Object.assign({}, product, {
          votes: product.votes + 1,
        });
      } else {
        return product;
      }
    });
    this.setState({
      products: nextProducts,
    });
  }
```

In sum, we can use property initializers to make two refactors to our React components:

1. We can use arrow functions for custom component methods (and avoid having to bind `this`)
2. We can define the initial state outside of `constructor()`

We expose you to both approaches in this book as both are in widespread use. Each project will be consistent as to whether or not it uses `transform-class-properties`. You're welcome to continue to use vanilla ES6 in your own projects. However, the terseness afforded by `transform-class-properties` is often too attractive to pass up.

Using ES6/ES7 with additional presets or plugins is sometimes referred to by the community as "ES6+/ES7+."

Congratulations!

We just wrote our first React app. There are a ton of powerful features we've yet to go over, yet all of them build upon the core fundamentals we just covered:

1. We think about and organize our React apps as components
2. Using JSX inside the `render` method
3. Data flows from parent to children through props
4. Event flows from children to parent through functions
5. Utilizing React lifecycle methods
6. Stateful components and how state is different from props
7. How to manipulate state while treating it as immutable

Onward!

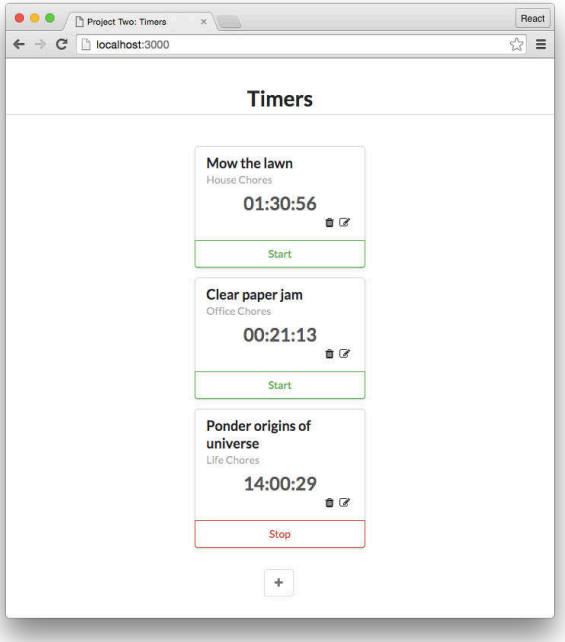
Components

A time-logging app

In the last chapter, we described how React organizes apps into components and how data flows between parent and child components. And we discussed core concepts such as how we manage **state** and pass data between components using **props**.

In this chapter, we construct a more intricate application. We investigate a pattern that you can use to build React apps from scratch and then put those steps to work to build an interface for managing timers.

In this time-tracking app, a user can add, delete, and modify various timers. Each timer corresponds to a different task that the user would like to keep time for:



This app will have significantly more interactive capabilities than the one built in the last chapter. This will present us with some interesting challenges that will deepen our familiarity with React's core concepts.

Getting started

As with all chapters, before beginning make sure you've downloaded the book's sample code and have it at the ready.

Previewing the app

Let's begin by playing around with a completed implementation of the app.

In your terminal, `cd` into the `time_tracking_app` directory:

GET THE FULL BOOK

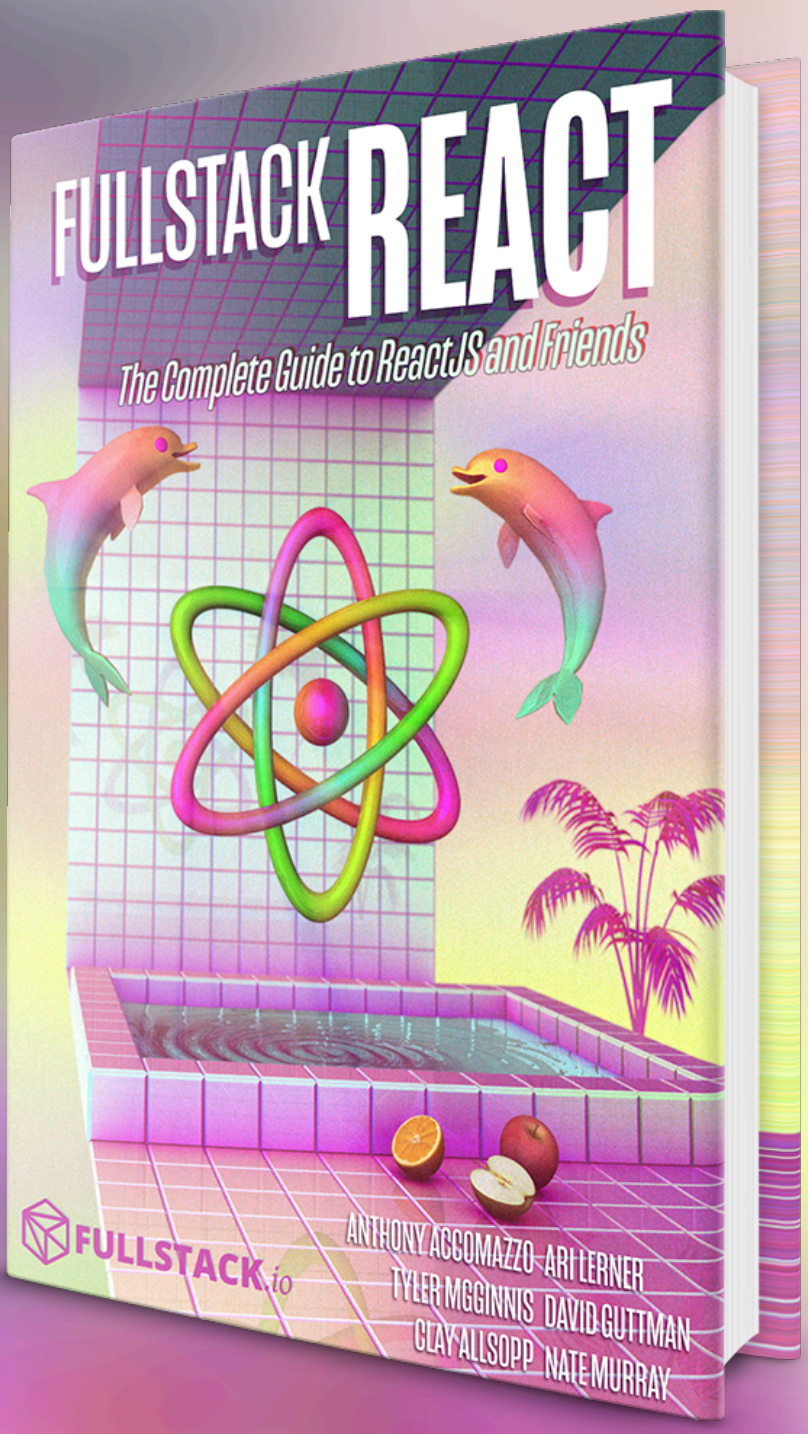
This is the end of the preview chapter!
Head over to:

https://fullstackreact.com

to download the full package!

Learn how to use:

- Redux
- Routing
- GraphQL
- Relay
- React Native
- and more!



GET IT NOW