

O'REILLY®

Early Release

RAW & UNEDITED



Learning React

FUNCTIONAL WEB DEVELOPMENT WITH REACT AND FLUX

Alex Banks & Eve Porcello

Learning React

*Functional Web Development with React and
Redux*

Alex Banks and Eve Porcello

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Learning React

by Alex Banks and Eve Porcello

Copyright © 2016-08-04 Alex Banks and Eve Porcello. All rights reserved.

Printed in the United States of America.

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

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

Editor: Allyson MacDonald

Production Editor: FILL IN PRODUCTION EDITOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition

2016-08-04: First Early Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491954553> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Learning React, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-95455-3

[FILL IN]

Table of Contents

Preface.....	vii
1. Welcome to React.....	11
History	12
React is not a Framework	14
React and MVC	15
React Ecosystem	16
Keeping up with the Changes	18
Working with the Files	18
File Repository	18
React Developer Tools	19
Installing Node.js	20
2. Emerging JavaScript.....	21
Declaring Variables in ES6	22
Const	22
Let	22
Template Strings	24
Default Parameters	26
Arrow Functions	26
Transpiling ES6	30
ES6 Objects and Arrays	31
Destructuring Assignment	31
Object Literal Enhancement	33
Spread Operator	34
Module Imports and Exports	36
Promises	36
Classes	39

3. Functional Programming with JavaScript.....	43
What it means to be Functional	44
Imperative vs Declarative	46
Functional Concepts	49
Immutability	49
Pure Functions	51
Data Transformations	53
Higher Order Functions	61
Recursion	62
Composition	65
Putting it all together	67
4. Pure React.....	73
Page Setup	73
The Virtual DOM	74
React Elements	75
ReactDOM	77
Children	78
Constructing Elements with Data	80
React Components	82
React.createClass()	83
React.Component	86
Stateless Functional Components	87
DOM Rendering	88
Factories	91
5. React with JSX.....	95
React Elements as JSX	95
JSX Tips	96
Babel	98
Recipes as JSX	99
Babel Presets	106
Intro to webpack	107
Webpack Loaders	108
Recipes App with Webpack Build	108
6. Props, State, and the Component Tree.....	121
Property Validation	121
Validating Props with createClass	122
Default Props	126
Custom Property Validation	127
ES6 Classes and Stateless Functional Components	128

Refs	131
Two-way Data Binding	133
Refs in Stateless Functional Components	134
React State Management	135
Introducing Component State	136
Initializing State from Properties	140
State within the component tree	142
Color Organizer App Overview	142
Passing properties down the component tree	144
Passing data back up the component tree	146

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

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

Constant width

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

Constant width bold

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

Constant width italic

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



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

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

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x.”

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

Safari® Books Online



Safari Books Online is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **plans and pricing** for **enterprise, government, education**, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kauf-

mann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds **more**. For more information about Safari Books Online, please visit us **online**.

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://www.oreilly.com/catalog/<catalog page>>.

To comment or ask technical questions about this book, send email to [*bookquestions@oreilly.com*](mailto:bookquestions@oreilly.com).

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

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

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

Acknowledgments

Welcome to React

React is a popular library for creating user interfaces. It was created at Facebook to address some of the challenges associated with large-scale, data-driven websites. When React was released in 2013, the project was initially viewed with some skepticism because the conventions of React are quite unique.

In an attempt to not scare people off, the core React team wrote an article called **Why React** that recommended that you “Give It [React] Five Minutes”. Their point was that they wanted to encourage people to work with React first before thinking that their approach was too crazy.

Yes, React is a small library that doesn’t come with everything you might need out of the box to build your application. Give it five minutes.

Yes, in React, you write code that looks like HTML right in your JavaScript code. And yes, those tags require pre-processing to run in a browser. And, you’ll probably need a build tool like webpack or browserify for that. Give it five minutes.

Reading this book won’t take you five minutes, but we do invite you to dive into React with an open mind.

A few companies that have given React more than five minutes, and use the library for large parts of their web interfaces, include Airbnb, Khan Academy, and the New York Times¹. Many of Facebook’s features and all of Instagram² are built on React and

¹ These companies were early adopters of React and used it in production as early as 2014. <https://facebook.github.io/react/blog/2014/05/29/one-year-of-open-source-react.html>

² “Why Did We Build React” by Pete Hunt, Facebook Blog: <https://facebook.github.io/react/blog/2013/06/05/why-react.html>

associated tools to manage the messages and pictures of lunch that over a billion users³ post every day.

The widespread use of React on large websites shows that it is stable enough to use at scale. React is ready, but nothing is set in stone. The unique opportunity we all have is that since it's so new, we can be part of building it. As the library and its tools evolve, we can suggest enhancements. When ideas come to mind about how to work with React more efficiently, we can build them. React is already great, but we can play an active role in building its even better future.

History

React was built at Facebook and Instagram, released initially in March 2013, then open-sourced on May 29, 2013. React is for your user interfaces or the view layer of your application. It was designed as a view-only library where you create user interface components that display data.

When it was released, React built steam quickly, and the community quickly contributed code and got involved in community events.

With all of the growth, Facebook decided to build and release a framework for building native applications in 2015⁴: React Native. You can build apps for iOS, Android, and Windows platforms using React Native. Unlike other platforms, React uses the native phone and tablet UI elements. The aim of React Native is to use the same programming language to build many types of apps - not necessarily the same codebase.

A design architecture that emerged around the same time as React from the Facebook team is **Flux**. Flux was built to deal with a problem in the Facebook messaging app. Users complained that when they read a message, they would still see a notification that they had an unread message⁵. To deal with these data inconsistencies, Flux introduced a new design where data flowed one way. This data flow works particularly well with React.

Building upon Flux's ideas, **Redux** was developed to simplify the process of managing data in React apps. It was released in 2015 and has picked up a lot of momentum as a less complex but similarly solid implementation of Flux. In addition, Falcorn and Relay have also emerged to tackle data handling challenges.

3 There are over 1.65 billion monthly active Facebook users as of 04/27/16. <https://zephoria.com/top-15-valuable-facebook-statistics/>

4 Tom Occhino introduces React Native in 2015 at the React Conference. [https://code.facebook.com/videos/786462671439502/react-js-conf-2015-keynote-introducing-react-native-/](https://code.facebook.com/videos/786462671439502/react-js-conf-2015-keynote-introducing-react-native/)

5 Facebook engineer Jing Chen discusses this in her 2014 talk, "Rethinking Web App Development at Facebook". <https://facebook.github.io/flux/docs/overview.html#content>

At the time we're writing this book, React isn't the most popular JavaScript library, but it's growing fast. Web searches for JavaScript frameworks and libraries indicate a huge rise in popularity.

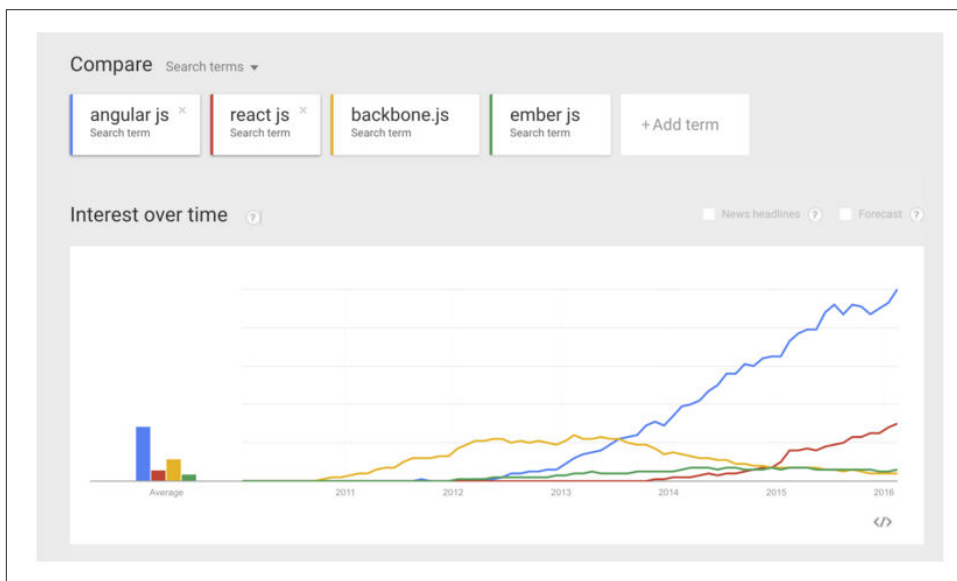


Figure 1-1.
Web Searches for JavaScript Frameworks & Libraries (Google Trends)

You've probably heard React described as the hot new library that everyone in the organization should learn to replace whatever everyone learned last month. Being popular isn't the reason to use React. Popularity is a side effect of being useful and time saving.

The influence of React is even felt in other MVC frameworks. Angular and Ember have been inspired by React's approach in newer versions of those frameworks. It's interesting to observe where there are overlaps and how the individual communities approach this work.

So, what does React do next? Perhaps we'll use React to build desktop apps. We might build console apps. We might use React to build robots or to make up the screens of our self-driving cars.

What will happen? None of us know, but you can join us. And, we need you to join us!

React is not a Framework

React is a library not a framework. A lot of people call it a framework or compare it to frameworks. We even compared it to frameworks in the previous section.

The reason is React is commonly mistaken as a framework is because it feels like frameworks are React's competitors. Most blog articles set up a classic bout between React vs. Angular or React vs. Ember. You can use React with Angular. You can use React with Ember. Although, for the most part, we typically do not combine Angular with React, you could.

Frameworks are bigger than libraries. They include everything that you may need to build applications. Think about the .NET framework. Not only does it have everything you need to build any type of Microsoft application, but there are specific design patterns in .NET that you would follow to construct applications. The .NET framework even comes with its own IDE⁶, Visual Studio.

Angular is a JavaScript framework that you can use to build large scale single page web applications. Angular comes with most everything you need to get some web development done. There are even design patterns to follow, primarily patterns based in MVC⁷. The Angular 2 JavaScript file that loads in the browser is 764k, but it is rich with functionality.

By comparison, the JavaScript file that is used to load React in the browser is 151k. React is small because it is simply a view library. It has some robust tools for managing views. It even has some tools to build full applications, but React relies a lot on pure JavaScript and other JavaScript libraries.

React does not come with any REST tools for making HTTP requests. React does not come with any tools to handle client-side routing. It is simply a library that helps you build efficient user interfaces.

React does not impose any restrictions on your data architecture. You can use React with MVC. You can use React with jQuery. You can use React with Flux or Redux. You can even build your own client data management library or design pattern and use React with that.

The reason that it is important to understand that React is only a library is because calling it a framework generates confusion about how to learn React. If you set out to learn Angular, the framework has everything you need to learn already included. It

⁶ IDE is an acronym for Integrated Development Environment and is a tool that you use to write code.

⁷ MVC is an object-oriented design pattern that can be used in any object-oriented language. MVC stands for Model View Controller. The model is the data, the view is the presentation layer, and the controller is the business logic.

also includes common design patterns that address how to build a web application. You can say, “I’m learning Angular 2”, and get started learning a single framework.

React is a little bit different. You could say, “I’m learning React”, learn how to build a React component, and then not know what to do next. The reason for this is because with React, you have some decisions that you need to make about your overall architecture. Is your app small enough to build with React and React alone? Is your application going to be a single page application, SPA? If so, what are you going to use for routing? If you’re using Flux, which flavor of Flux: Facebook Flux, Reflux, Redux, or your own implementation? How are you going to build or transpile your source code? Are you using emerging JavaScript like ES6 or ES7? You are going to need some answers to these questions before you get started learning, and that alone may feel daunting.

The good news is, if you answer these questions, working with React can be quite rewarding. To make learning React more approachable we are simply going to answer these questions for you and create a learning plan based on our answers. We are going to learn to use React alone and React with other libraries. We are not using MVC with React. We will build SPAs, and we’ll even build universal applications⁸. We will introduce functional programming and follow functional principles throughout the book. We will introduce the Flux pattern, and we’ll build applications using Redux, an implementation of the Flux pattern. We will use webpack and Babel to transpile our code. We are heavily using emerging JavaScript like ES6 and ES7.

React and MVC

In the previous section, we mentioned that React can be used with MVC. Originally, when React was first introduced, some developers even referred to it as the “V” in MVC. When we were first introduced to React, the first several applications that we developed used Backbone models, collections, and routes. If you are very familiar with MVC, you can probably hit the ground running and build some robust applications that use React with your favorite MVC framework.

One approach to learning React is to build some small components and swap them out with existing views in your project. React does not demand that you change your entire infrastructure. You can try React components out in your present MVC application without too much work.

React was developed to handle problems that can arise out of building MVC applications. If you use React with MVC, sooner or later you may encounter these problems. They start to show up when we have models that interact with multiple views, and

⁸ Universal apps are applications that use the same code on both the client and the server. See Chapter 12 for more.

views that listen to models to update UI when models change. In short MVC causes side effects. A certain view that you didn't expect to change could have changed when a certain model was updated. Sooner or later you may find complications within your React app that will push you to learn Flux. That is what happened to us

Learning Flux is a journey into functional JavaScript is going to deepen your knowledge of the JavaScript language. React is going to provide a user interface as data abstraction. Most of your work will be with the core datasets. These datasets will consist of JavaScript arrays and object literals. Functional programming demands that we keep this data immutable or unchanging.

To keep our arrays immutable, or unchanging, we are going to have to learn to use functions like `map` and `filter`. To keep objects immutable, we will need to learn how to duplicate and assign objects. You will need to be able to abstract objects from arrays with `Object.keys` and arrays from objects with `array.reduce`. All of this is pure JavaScript. All of this JavaScript is covered in this book. These practices can apply to any JavaScript application not just React.

It is our hope that you can use this book to skip confusion in the learning process by jumping straight into functional JavaScript development with React. We hope to get you up to speed with the React ecosystem faster by approaching it this way.

React Ecosystem

As we mentioned, React is a library. It's not a part of any overarching framework. There is an ecosystem of popular libraries and design patterns that we can use when building web applications with React. We get to choose our user interface stack out of this ecosystem of libraries to create our own stack.

A good rule of thumb when working with React is to use only what you need. For a lot of apps, you can use just React. React can manage views and even the data that is used in the views. That is powerful enough alone to build many types of applications. To that end, you don't have to use a ton of complicated tools.

You can build small apps with just React. As apps grow in scale, you may find the need to incorporate other solutions. If you start with a full stack app, it might be overkill.

Overly complex tooling has become a common complaint about React and JavaScript in general. Much of this is borrowed trouble. If you don't need the tool or you hate working with it, there's a way to not use it or use a tool you do like.

If you need extra tools, here is what you need them for. The purpose of a build tool is to take tasks that you commonly perform (SASS to CSS, code linting, testing, Babel transpiling, etc.) and automate them. Each of these links will give you additional information on how to set up a project with that tool.

- **Browserify** - Allows you to use require modules like you might do in Node.js. Browserify is often used with Grunt and Gulp.
- **Webpack** - The build system we'll use in this book. It has a steeper learning curve than Browserify, but it is adopted widely in the community. Also, once you get the hang of it, you will likely enjoy working with it more.

These build systems are often used in React projects, but they are widely used on all sorts of projects that have nothing to do with React. There are also a variety of tools that are intended to support React-specific projects.

React Router

The React Router provides a framework for handling routing in React applications. The router helps handle routing in single page applications. The website loads one page, and the router manages navigation from page to page.

React Motion

A framework for creating animations in React. It does so by interpolating the values used in CSS transforms for things like x and y values changing over time, opacity changes, etc.

React Addons

A package of opt-in features that can be used to enhance React applications. These utilities help improve performance like PureRenderMixin and Perf, manage animations with CSSTransitionGroup, and write test cases with TestUtils.

Enzyme

An increasingly popular unit testing framework for React created by Airbnb. Enzyme allows you to use whichever assertion library or test runner you'd like including Karma, Mocha, Jest, and more.

Redux

Redux is an implementation of Flux. Though commonly associated with React, Redux is library agnostic. You can use Redux with any UI: Angular, Ember, jQuery, or regular JavaScript. It includes React Redux which contains the official React bindings for Redux. It also includes Redux Thunk, middleware that provides a way of handling more complex architectures in Redux applications.

React Fire (Firebase for React)

Firebase is a pre-built back-end for features like user authentication, data storage, and hosting. React Fire is the React-specific implementation of Firebase that can be integrated into React applications.

As mentioned at the beginning of this section, it can be helpful to have some awareness of the large ecosystem of React-related tools, but it's important not to get too bogged down in whatever the trendy new thing is. New tools are popping up and changing all the time. Use only what you need, and keep it simple.

Keeping up with the Changes

As we've mentioned, React is still new. It has reached a place where core functionality is fairly stable but even that can change. The tools, libraries, and ecosystem are still being discovered, and the development of these tools is also changing.

As changes are made to React and related tools, sometimes these are breaking changes. In fact, some of the future versions of these tools may break some of the example code in this book. You can still follow along with the code samples. We'll provide exact version information in the `package.json` file, so that you can install these packages at the correct version.

Beyond this book, you can stay on top of changes by following along with the official [React blog](#). When new versions of React are released, the core team will write a detailed blog and changelog about what is new.

There are also a variety of popular React conferences that you can attend for the latest React information. If you can't attend these in-person, React conferences often release the talks on YouTube following the events.

- [React Conf](#) - Facebook sponsored conference in the Bay Area
- [React Rally](#) - Community conference in Salt Lake City
- [ReactiveConf](#) - Community conference in Bratislava, Slovakia
- [React.Amsterdam](#) - Community conference in Amsterdam

Working with the Files

In this section, we will discuss how to work with the files for this book and how to install some useful React tools.

File Repository

The GitHub repository associated with this book (<https://github.com/moonhighway/learning-react>) provides all of the code files organized by chapter. The repository is a mix of code files and JSBin samples. If you've never used JSBin before, it's an online code editor similar to CodePen and JSFiddle.

One of the main benefits of JSBin is that you can click the link and immediately start tinkering with the file. When you create or start editing a JSBin, it will generate a unique URL for your code sample.

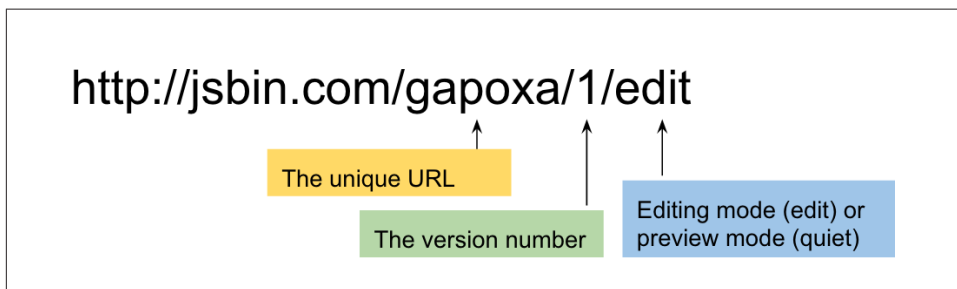


Figure 1-2. JSBin URL

The letters that follow `jsbin.com` represent the unique URL key. After the next slash is the version number. In the last part of the URL, there will be one of two words: `edit` for editing mode and `quiet` for preview mode.

React Developer Tools

There are several developer tools that can be installed as browser extensions or add-ons that you may find useful as well.

react-detector

The `react-detector` is a Chrome extension that lets you know which websites are using React and which are not. You can find the `react-detector` here: <https://chrome.google.com/webstore/detail/react-detector/jaaklebbonondhkanegppcca-nebkdljh?hl=en-US>

show-me-the-react

This is another tool that detects React as you browse the internet. It is available for both Firefox (<https://github.com/insin/show-me-the-react>) and Chrome (<https://github.com/cymen/show-me-the-react>).

React Developer Tools

The React Developer Tools are a plugin that can extend the functionality of the browser's developer tools. The React Developer Tools is installed as a new tab to let you view React elements.

If you prefer Chrome, you'll install as an extension: <https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en>. You can also install as an add-on for Firefox: <https://addons.mozilla.org/en-US/firefox/addon/react-devtools/>.

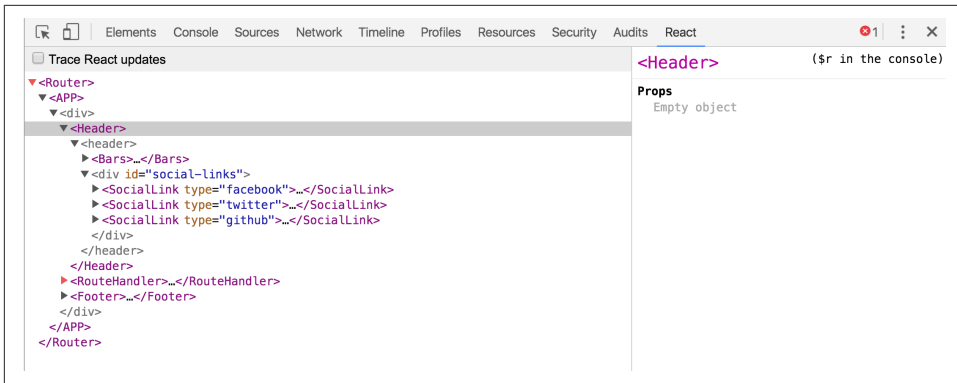


Figure 1-3. Viewing the React Developer Tools

Any time you see `react-detector` or `show-me-the-react` as active, you can open the developer tools and get an understanding of how React is being used on the site.

Installing Node.js

Node.js is JavaScript without the browser. It is a runtime environment used to build full-stack JavaScript applications. Node is open-source and can be installed on Windows, Mac, Linux, and other platforms.

You do not need to use Node to use React. We will be using Node in Chapter 12 when we build an Express server. However, when working with React, you need to use the Node package manager, `npm`, to install dependencies. This is automatically installed with the Node installation.

If you're not sure if Node.js is installed on your machine, you can open a Terminal or Command prompt window and type:

```
node -v
```

```
Output: v6.1.0
```

Ideally, you will have a Node version number of 4 or higher. If you type the command and see an error message that says “Command not found”, Node.js is not installed. This can be done directly from the website at nodejs.org. Just go through the automated steps of the installer, and when you type in the ‘`node -v`’ command again, you’ll see the version number.

Not only is React new and changing, but JavaScript is currently undergoing huge changes. We’ll start by getting up to speed with the latest changes in JavaScript before diving into React.

Emerging JavaScript

Since its release in 1995, JavaScript has gone through many changes. At first, it made adding interactive elements to web pages much simpler. Then it got more robust with DHTML and AJAX. Now with Node.js, JavaScript has become a language that is used to build full-stack applications. The committee that is and has been in charge of shepherding the changes to JavaScript is the ECMA, the European Computer Manufacture Association.

Changes to the language are community driven. They originate from proposals that community members write. Anyone **can submit a proposal** to the ECMA committee. The responsibility of the ECMA committee is to manage and prioritize these proposals to decide what is included each spec. Proposals are taken through clearly defined stages. Stage-0 represents the newest proposals up through Stage-4 which represents the finished proposals.

The most recent version of the specification was approved in June 2015 ¹ and is called by many names: ECMAScript 6, ES6, ES2015, Harmony, or ESNext. Based on current plans, new specs will be released on a yearly cycle. For 2016, the release will be relatively small ², but it already looks like ES2017 will include quite a few useful features. We'll be using many of these new features in the book and will opt to use emerging JavaScript whenever possible.

Many of these features are already supported by the newest browsers. We will also be covering how to convert your code from emerging JavaScript syntax to ES5 syntax that will work today in most all of the browsers. The **kangax compatibility table** is a

¹ "ECMAScript 2015 Has Been Released", InfoQ, June 17, 2015 <https://www.infoq.com/news/2015/06/ecmascript-2015-es6>

² ES2016 Spec Info: <https://tc39.github.io/ecma262/2016/>

great place to stay informed about the latest JavaScript features and their varying degrees of support by browsers.

In this chapter, we will show you all of the emerging JavaScript that we'll be using throughout the book. If you haven't made the switch to the latest syntax yet, then now will be a good time to get started. If you are already comfortable with ESNext language features, you can go ahead and skip to the next chapter.

Declaring Variables in ES6

Const

ES6 introduced constants, and they are already supported in most browsers. A constant is a variable that cannot be changed. It is the same concept as constants in other language.

Before constants, all we had were variables, and variables could be overwritten.

```
var pizza = true
pizza = false
console.log(pizza) // false
```

We cannot reset the value of a constant variable, and it will generate a console error if we try to overwrite the value.

```
const pizza = true
pizza = false
```




Figure 2-1. An attempt at overwriting a constant

Let

JavaScript now has lexical variable scoping. In JavaScript, we create code blocks with curly brackets. With functions, these curly brackets block off the scope of variables. On the other hand, think about if/else statements. If you come from other languages, you might assume that these blocks would also block variable scope. This is not the case.

If a variable is created inside of an if/else block, that variable is not scoped to the block.

```
var topic = "JavaScript"

if (topic) {
```



```

var topic = "React"
console.log('block', topic)    // block React
}

console.log('global', topic)    // global React

```

The topic variable inside the if block resets the value of topic.

With the let keyword, we can scope a variable to any code block. Using let protects the value of the global variable.

```

var topic = "JavaScript"

if (topic) {
  let topic = "React"
  console.log('block', topic)    // React
}

console.log('global', topic)    // JavaScript

```

The value of topic is not reset outside of the block.

Another area where curly brackets don't block off a variable's scope is in for-loops.

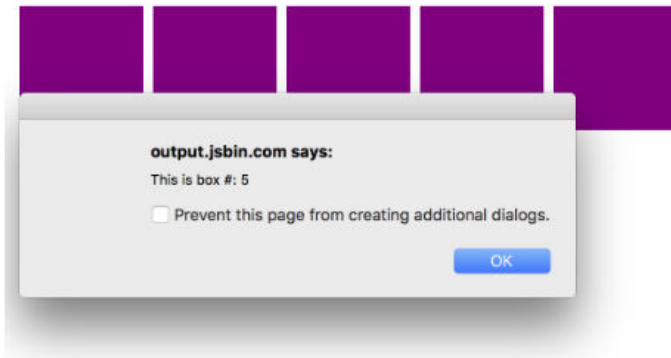
```

var div,
    container = document.getElementById('container')

for (var i=0; i<5; i++) {
  div = document.createElement('div')
  div.onclick = function() {
    alert('This is box #' + i)
  }
  container.appendChild(div)
}

```

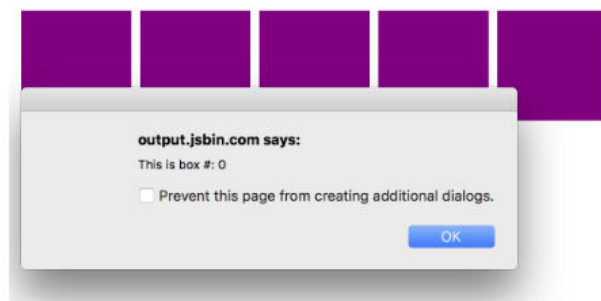
In this for-loop, we create 5 divs to appear within a container. Each div is assigned an onclick handler that alerts a message to displays the index. Declaring `i` in the loop creates a global variable named `i`, and then iterates it until it reaches 5. When you click on any of these boxes, the alert says that `i` is equal to 5 for all divs, because the current value for global `i` is 5.



Output

Declaring the loop counter `i` with `let` instead of `var` does block of the scope of `i`. Now clicking on any box will display the value for `i` that was scoped to the loop iteration.

```
var div, container = document.getElementById('container')
for (let i=0; i<5; i++) {
  div = document.createElement('div')
  div.onclick = function() {
    alert('This is box #: ' + i)
  }
  container.appendChild(div)
}
```



Output

Template Strings

Template strings provide us with an alternative to string concatenation. They also allow us to insert variables into a string.

Traditional string concatenation uses plus signs or commas to compose a string using variable values and strings.

```
console.log(lastName + ", " + firstName + " " + middleName)
```

With a template, we can create one string and insert the variable values by surrounding them with `${variable}`.

```
console.log(`${lastName}, ${firstName} ${middleName}`)
```

Any JavaScript that returns a value can be added to a template string between the `${ }` in a template string.

Template strings honor whitespace. They will make it easier to draft up email templates, code examples, or anything that contains whitespace. Now you can have a string that spans multiple lines without breaking your code.

Example 2-1. Template Strings Honor Whitespace

```
,
Hello ${firstName},

Thanks for ordering ${qty} tickets to ${event}.

Order Details
  ${firstName} ${middleName} ${lastName}
  ${qty} x ${price} = ${qty*price} to ${event}

You can pick your tickets up at will call 30 minutes before
the show.

Thanks,

${ticketAgent}
,
```

These tabs, line breaks, spaces, and variable names can be used in an email template.

Previously, using an HTML string directly in our JavaScript code was not so easy to reason about because we'd need to run it together on one line. Now the whitespace is recognized as text, and you can insert formatted HTML that is easy to understand.

```
document.body.innerHTML = `
<section>
  <header>
    <h1>The HTML5 Blog</h1>
  </header>
  <article>
    <h2>${article.title}</h2>
    ${article.body}
  </article>
`
```

```

</article>
<footer>
  <p>copyright ${new Date().getFullYear()} | The HTML5 Blog</p>
</footer>
</section>
`

```

Notice that we can include variables for the page title and article text as well.

Default Parameters

Languages including C++ and Python allow developers to declare default values for function arguments. Default parameters are included in the ES6 spec, so in the event that a value is not provided for the argument, the default value will be used.

For example, we can set up default strings.

```

function logActivity(name="Shane McConkey", activity="skiing") {
  console.log( `${name} loves ${activity}` )
}

```

If no arguments are provided to the favoriteActivity function, it will run correctly using the default values. Default arguments can be any type, not just strings.

```

var defaultPerson = {
  name: {
    first: "Shane",
    last: "McConkey"
  },
  favActivity: "skiing"
}

function logActivity(p=defaultPerson) {
  console.log(`${p.name.first} loves ${p.favActivity}`)
}

```

Arrow Functions

Arrow functions are a useful new feature of ES6. With arrow functions, you can create functions without using the function keyword. You also often do not have to use the return keyword.

Example 2-2. As a Traditional Function

```

var lordify = function(firstname) {
  return `${firstname} of Canterbury`
}

console.log( lordify("Dale") )      // Dale of Canterbury
console.log( lordify("Daryle") )   // Daryle of Canterbury

```

With an arrow function, we can simplify the syntax tremendously.

Example 2-3. As an Arrow Function

```
var lordify = firstname => `${firstname} of Canterbury`
```



Semi-colons throughout this Book

Semi-colons are optional in JavaScript. Our philosophy on JavaScript is why put semi-colons in that aren't required. This book takes a minimal approach that excludes unnecessary syntax.

With an arrow, we now have an entire function declaration on one line. The function keyword is removed. We also remove return because the arrow points to what should be returned. Another benefit is that if the function only takes one argument, we can remove the parentheses around the arguments.

More than one argument should be surrounded in parentheses.

```
// Old
var lordify = function(firstName, land) {
  return `${firstName} of ${land}`
}

// New
var lordify = (firstName, land) => `${firstName} of ${land}`

console.log( lordify("Dale", "Maryland") )    // Dale of Maryland
console.log( lordify("Daryle", "Culpeper") )  // Daryle of Culpeper
```

We can keep this as a one line function because there is only one statement that needs to be returned.

More than one line needs to be surrounded with brackets.

```
// Old
var lordify = function(firstName, land) {

  if (!firstName) {
    throw new Error('A firstName is required to lordify')
  }

  if (!land) {
    throw new Error('A lord must have a land')
  }

  return `${firstName} of ${land}`
}

// New
var _lordify = (firstName, land) => {
```

```

    if (!firstName) {
      throw new Error('A firstName is required to lordify')
    }

    if (!land) {
      throw new Error('A lord must have a land')
    }

    return `${firstName} of ${land}`
  }

  console.log( lordify("Kelly", "Sonoma") )    // Kelly of Sonoma
  console.log( lordify("Dave") )               // ! JAVASCRIPT ERROR

```

These if/else statements are surrounded with brackets but still benefit from the shorter syntax of the arrow function.

Arrow functions do not block off *this*. For example, *this* becomes something else in the setTimeout callback, not the tahoe object.

```

var tahoe = {
  resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
  print: function(delay=1000) {

    setTimeout(function() {
      console.log(this.resorts.join(","))
    }, delay)

  }
}

tahoe.print() // Cannot read property 'join' of undefined

```

This error is thrown because it's trying to use the .join method on what *this* is. In this case, it's the window object. Alternatively, we can use the arrow function to protect the scope of *this*.

```

var tahoe = {
  resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
  print: function(delay=1000) {

    setTimeout(() => {
      console.log(this.resorts.join(","))
    }, delay)

  }
}

tahoe.print() // Kirkwood, Squaw, Alpine, Heavenly, Northstar

```

This works correctly and we can .join the resorts with a comma. Be careful though that you're always keeping scope in mind. Arrow functions do not block off the scope of *this*.

```
var tahoe = {
  resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
  print: (delay=1000) => {

    setTimeout(() => {
      console.log(this.resorts.join(","))
    }, delay)

  }
}

tahoe.print(); // Cannot read property resorts of undefined
```

Chaining the print function to an arrow function means that *this* is actually the window.

To verify, let's change the console message to evaluate whether this is the window.

```
var tahoe = {
  resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
  print: function(delay=1000) {

    setTimeout(() => {
      console.log(this === window)
    }, delay)

  }
}

tahoe.print(); // true
```

It evaluates as true. To fix this, we can just use a regular function.

```
var tahoe = {
  resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
  print: function(delay=1000) {

    setTimeout(() => {
      console.log(this === window)
    }, delay)

  }
}

tahoe.print() // false
```

Transpiling ES6

Most web browsers don't support ES6, and even those that do, don't support everything. The only way to be sure that your ES6 code will work is to convert it to ES5 code before running it in the browser. This process is called transpiling. One of the most popular tools for transpiling is Babel (www.babeljs.io)

In the past, the only way to use the latest JavaScript features was to wait weeks, months, or even years until browsers supported them. Now, transpiling has made it possible to use the latest features of JavaScript right away. The transpiling step makes JavaScript similar to other languages. Transpiling is not compiling - our code isn't compiled to binary. Instead, it's transpiled into syntax that can be interpreted by a wider range of browsers. Also, JavaScript now has source code, meaning that there will be some files that belong to your project that don't run in the browser.

Here is some ES6 code. We have an arrow function, which we will cover in a bit, mixed with some default arguments for x and y.

Example 2-4. ES6 Code before Babel Transpiling

```
const add = (x=5, y=10) => console.log(x+y);
```

After we run the transpiler on this code, here is what the output would look like:

```
"use strict";

var add = function add() {
  var x = arguments.length <= 0 || arguments[0] === undefined ?
    5 : arguments[0];
  var y = arguments.length <= 1 || arguments[1] === undefined ?
    10 : arguments[1];
  return console.log(x + y);
};
```

The transpiler added a “use strict” declaration to run in strict mode. The variables x and y are defaulted using the arguments array, a technique you may be familiar with. The resulting JavaScript is more widely supported.

You can transpile JavaScript directly in the browser using the inline Babel transpiler. You just include the browser.js file, and any scripts with `type="text/babel"` will be converted. Even though Babel 6 is the current version of Babel, only the CDN for Babel 5 will work.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.js"> </script> <script>
```




Transpiling in the Browser

This approach means that the browser does the transpiling at run-time. This is not a good idea for production because it will slow your application down a lot. In chapter 5, we'll go over how to do this in production. For now, the CDN link will allow us to discover and use ES6 features.

You may think to yourself: “Great! When ES6 is supported by browsers, we won't have to use Babel anymore!” However, as soon as this happens, we will want to use ES7 and beyond features. Unless a tectonic shift occurs, we'll likely be using Babel in the foreseeable future.

ES6 Objects and Arrays

ES6 gives us new ways to work with objects and arrays and for scoping the variables within these data sets. These features include destructuring, object literal enhancement, and the spread operator.

Destructuring Assignment

The destructuring assignment allows you to locally scope fields within an object and to declare which values will be used.

Consider this sandwich object. It has four keys, but we only want to use the values of two. We can scope bread and meat to be used locally.

```
var sandwich = {  
  bread: "dutch crunch",  
  meat: "tuna",  
  cheese: "swiss",  
  toppings: ["lettuce", "tomato", "mustard"]  
}
```

```
var {bread, meat} = sandwich
```

```
console.log(bread, meat) // dutch crunch tuna
```

The code pulls bread and meat out of the object and creates local variables for them. Also, bread and meat variables can be changed.

```
var {bread, meat} = sandwich
```

```
bread = "garlic"  
meat = "turkey"
```

```
console.log(bread) // garlic  
console.log(meat) // turkey
```

```
console.log(sandwich.bread, sandwich.meat) // dutch crunch tuna
```

We can also destructure incoming function arguments. Consider this function that would log a person's name as a Lord.

```
var lordify = regularPerson => {  
  console.log(`${regularPerson.firstname} of Canterbury`)  
}  
  
var regularPerson = {  
  firstname: "Bill",  
  lastname: "Wilson"  
}  
  
lordify(regularPerson)      // Bill of Canterbury
```

Instead of using dot notation syntax to dig into objects, we can destructure the values that you need out of `regularPerson`.

```
var lordify = ({firstname}) => {  
  console.log(`${firstname} of canterbury`)  
}  
  
lordify(regularPerson)      // Bill of Canterbury
```

Destructuring is also more declarative, meaning that our code is more descriptive about what we are trying to accomplish. By destructuring `firstname`, we declare that we will only use the `firstname` variable. More on declarative programming in the next chapter.

Values can also be destructured from arrays. Imagine that we wanted to assign the first value of an array to a variable name

```
var [firstResort] = ["Kirkwood", "Squaw", "Alpine"]  
  
console.log(firstResort) // Kirkwood
```

You can also pass over unnecessary values with list matching using commas. List matching occurs when commas take the place of elements that should be skipped. With the same array, we can access the last value by replacing the first two values with commas.

```
var [, ,thirdResort] = ["Kirkwood", "Squaw", "Alpine"]  
  
console.log(thirdResort) // Alpine
```

Later in the section, we'll take this example a step further by combining array destructuring and the spread operator.

Object Literal Enhancement

Object literal enhancement is the opposite of destructuring. It is the process of restructuring or putting back together. With object literal enhancement, we grab variables from scope and turn them into an object.

```
var name = "Tallac"
var elevation = 9738

var funHike = {name,elevation}

console.log(funHike) // {name: "Tallac", elevation: 9738}
```

Name and elevation are now keys of the funHike object.

We can also create object methods with object literal enhancement or restructuring.

```
var name = "Tallac"
var elevation = 9738
var print = function() {
  console.log(`Mt. ${this.name} is ${this.elevation} feet tall`)
}

var funHike = {name,elevation,print}

funHike.print()    // Mt. Tallac is 9738 feet tall
```

Notice we use *this* to access the object keys.

When defining object methods, it is no longer necessary to use the function keyword.

Example 2-6. Old vs. New: Object Syntax

OLD

```
var skier = {
  name: name,
  sound: sound,
  powderYell: function() {
    var yell = this.sound.toUpperCase()
    console.log(`${yell} ${yell} ${yell}!!!`)
  },
  speed: function(mph) {
    this.speed = mph
    console.log('speed:', mph)
  }
}
```

NEW

```
const skier = {
  name,
  sound,
  powderYell() {
    let yell = this.sound.toUpperCase()
  }
}
```

```

    console.log(`${yell} ${yell} ${yell}!!!`);
  },
  speed(mph) {
    this.speed = mph;
    console.log('speed:', mph);
  }
}

```

Object literal enhancement allows us to pull global variables into objects and reduces typing by making the function keyword unnecessary.

Spread Operator

The spread operator is three dots (...) that perform several different tasks. First, the spread operator allows us to combine the contents of arrays. For example, if we had two arrays, we could make a third array that combines the two arrays into one.

```

var peaks = ["Tallac", "Ralston", "Rose"];
var canyons = ["Ward", "Blackwood"];
var tahoe = [...peaks, ...canyons];

console.log(tahoe.join(', ')) // Tallac, Ralston, Rose, Ward, Blackwood

```

All of the items from peaks and canyons are pushed into a new array called tahoe.

Let's take a look at how the spread operator can help us deal with a problem. Using the peaks array from the previous sample, let's imagine that we wanted to grab the last item from the array rather than the first. We can use the .reverse() method to reverse the array in combination with array destructuring.

```

var peaks = ["Tallac", "Ralston", "Rose"];
var [last] = peaks.reverse();

console.log(last) // Rose
console.log(peaks.join(', ')) // Rose, Ralston, Tallac

```

Look what happens though. The reverse function has actually altered or mutated the array. In a world with the spread operator, we don't have to mutate the original array, we can create a copy of the array and then reverse it.

```

var peaks = ["Tallac", "Ralston", "Rose"];
var [last] = [...peaks].reverse();

console.log(last) // Rose
console.log(peaks.join(', ')) // Tallac, Ralston, Rose

```

Since we use the spread operator to copy the array, the peaks array is still intact and can be used later in its original form.

The spread operator can also be used to get some, or the rest, of the items in the array.

```

var lakes = ["Donner", "Marlette", "Fallen Leaf", "Cascade"]

var [first, ...rest] = lakes

console.log(rest.join(", ")) // "Marlette, Fallen Leaf, Cascade"

```

We can also use the spread operator to collect function arguments as an array. We will build a function that takes in n number of arguments using the spread operator, and then uses those arguments to print some console messages.

```

function directions(...args) {
  var [start, ...remaining] = args
  var [finish, ...stops] = remaining.reverse()

  console.log(`drive through ${args.length} towns`)
  console.log(`start in ${start}`)
  console.log(`the destination is ${finish}`)
  console.log(`stopping ${stops.length} times in between`)
}

directions(
  "Truckee",
  "Tahoe City",
  "Sunnyside",
  "Homewood",
  "Tahoma"
)

```

The directions function takes in the arguments using the spread operator. The first argument is assigned to the start variable. The last argument is assigned to a finish variable using array.reverse(). We then use the length of the arguments array to display how many towns we're going through. The number of stops is the length of the arguments array minus the finish stop. This provides incredible flexibility because we could use the directions function to handle any number of stops.

The spread operator can also be used for objects. This is a stage-2 proposal³ in the current ES2017 pending specification. Using the spread operator with objects is similar. In this example, we'll use it the same way we combined two arrays into a third array, but instead of arrays, we'll use objects.

```

var morning = {
  breakfast: "oatmeal",
  lunch: "peanut butter and jelly"
}

var dinner = "mac and cheese"

var backpackingMeals = {

```

³ Spread Operator, <https://github.com/tc39/proposals>

```

    ...morning,
    dinner
  }

  console.log(backpackingMeals) // {breakfast: "oatmeal",
                                  lunch: "peanut butter and jelly",
                                  dinner: "mac and cheese"}

```

Module Imports and Exports

In the earlier days of JavaScript, we would write a script that had many different functions

Promises

Promises give us a way to make sense out of asynchronous behavior. When making an asynchronous request, one of two things can happen: everything goes as we hope or there's an error. This can happen in a number of different ways. For example, we could try several ways to obtain the data to reach success. We also could receive multiple types of errors. Promises give us a way to simplify back to a simple pass or fail.

We'll build a promise that handles all of the different ways you can win or lose craps. If you've never played the game craps, that's ok. The code sample will teach it to you. The goal here is to wrangle the multiple different possible results into pass or fail.

First, we'll define all of the ways to win and lose craps. On the first roll when the point is not yet set, we win by rolling a 7 or an 11. We'll lose by rolling a 2 or a 3. We set a point by rolling any other number.

On each additional roll when the point is set, we win by hitting the point or rolling the same number again. We lose by rolling a 7. If we roll any other number, nothing happens. We roll again.

After each roll, the game is either over because you won or lost, or you have to roll again. When the game is over, this function logs whether you've won or lost.

```

const gameOver = result =>
  console.log(`Game Over - ${result}`)

```

If you are still rolling, we'll use another function to tell you what happened and what the current point is.

```

const stillRolling = (message, currentPoint) =>
  console.log(`${message} - try again for ${currentPoint}`)

```

If we had a function called `craps` that returned a promise, we could send the function that defines what to do when the game is over, and the function that defines what to do if we are still rolling. The `craps` promise will figure out which one to use.

We then can call the function that represents the first roll because we do not send a point.

```
craps(7).then(gameOver, stillRolling)
craps(2).then(gameOver, stillRolling)
craps(8).then(gameOver, stillRolling)
```

Then we can send with an additional argument to represent that the point has been set.

```
craps(5,8).then(gameOver, stillRolling)
craps(7,8).then(gameOver, stillRolling)
craps(8,8).then(gameOver, stillRolling)
```

That second argument represents where the point is set: 8.

The `craps` function takes the roll and the point as arguments and returns a promise object. Promise objects have a `then` function. Promises send the promise constructor a callback. With this callback, we will check the roll and navigate the complexities of all the different ways that `craps` can be won or lost.

If the game is over, `gameOver` is invoked with the results. If the game is still going, `roll again` is invoked with a message.

```
const craps = (roll, point) => new Promise((gameOver, rollAgain) => {

  // If roll is not sent as a number between 2 and 12, rollAgain
  if (!roll || typeof roll !== "number" || roll < 2 || roll > 12) {
    rollAgain("to roll a number")

    // If a point is not set, then this must be the first roll, the come out roll
  } else if (!point) {

    // If you roll a 7 or 11 during the first role, gameOver, you loose
    if (roll === 7 || roll === 11) {
      gameOver("You win by natural")

      // If you roll a 2 or a 3, gameOver, you win
    } else if (roll === 2 || roll === 3) {
      gameOver("You lose, crapped out")

      // Otherwise the point is set, rollAgain
    } else {
      rollAgain(roll)
    }

    // It's not the first roll, and you rolled the point, gameOver. You win
  } else if (roll === point) {
```

```

    gameOver("You win, you hit the point!")

    // It's not the first roll
  } else {

    // And you rolled a 7, gameOver you loose
    if (roll === 7) {
      gameOver("You lose, craps")

      // Otherwise you missed, try again to hit the point
    } else {
      rollAgain(point)
    }
  }
}

})

```

There are many different outcomes for any roll. The promise defines all of them. We will then use the promise by using the `.then` function.

We haven't yet written a function called `end` that will log the outcome when the game is over.

```

const end = result =>
  console.log(`Game Over - ${result}`)

```

This just logs the outcome to the console. We also need to write the `stillRolling` function.

```

const stillRolling = point =>
  console.log(`The point is ${point}, try again`)

```

The game can be played using `craps.then`.

```

craps("foo").then(end, stillRolling) // The point is to roll a number, try again
craps(7).then(end, stillRolling) // Game Over - You win by natural
craps(2).then(end, stillRolling) // Game Over - You lose, crapped out
craps(8).then(end, stillRolling) // The point is 8, try again

craps(5,8).then(end, stillRolling) // The point is 8, try again
craps(7,8).then(end, stillRolling) // Game over - You lose, craps
craps(8,8).then(end, stillRolling) // Game over - You win, you hit the point

```

The promise can help us deal with the many different outcomes successfully. The issue here is that we're dealing with synchronous data. More often than not, promises are used with asynchronous data.

Let's create an asynchronous promise for loading data from the `randomuser.me` API. This API has information like email, name, phone number, location, etc. for fake members and is great to use as dummy data.

The `getFakeMembers` function returns a new promise. The promise makes a request to the API. If the promise is successful, the data will load. If the promise is unsuccessful, an error will occur.

```
const getFakeMembers = count => new Promise((resolves, rejects) => {
  const api = `http://api.randomuser.me/?nat=US&results=${count}`
  const request = new XMLHttpRequest()
  request.open('GET', api)
  request.onload = () =>
    (request.status === 200) ?
      resolves(JSON.parse(request.response).results) :
      reject(Error(request.statusText))
  request.onerror = (err) => rejects(err)
  request.send()
})
```

With that, the promise has been created, but it hasn't been used yet. We'll use it with a simple console log for now, but we'll use it in a larger project during Chapter 11.

We can use the promise by calling the `getFakeMembers` function and passing in the number of members who should be loaded. The `then` function can be chained on to do something once the promise has been fulfilled. This is called composition. We'll also use an additional callback that handles errors.

```
getFakeMembers(5).then(
  members => console.log(members),
  err => console.error(
    new Error("cannot load members from randomuser.me"))
)
```

Promises make dealing with asynchronous requests easier which is good because we have to deal with a lot of asynchronous data in JavaScript. You'll also see promises used heavily in Node.js, so a solid understanding of promises is essential for the modern JavaScript engineer.

Classes

Previously in JavaScript, there were no official classes. Types were defined by functions. We had to create a function and then define methods on the function object using the prototype.

```
function Vacation(destination, length) {
  this.destination = destination
  this.length = length
}

Vacation.prototype.print = function() {
  console.log(this.destination + " | " + this.length + " days")
}
```

```
var maui = new Vacation("Maui", 7);

maui.print(); // Maui | 7
```

If you were used to classical object orientation, this probably made you mad.

Now ES6 introduces class declaration, but JavaScript still works the same way. Functions are objects, and inheritance is handled through the prototype, but this syntax makes more sense if you come from classical object orientation.



Capitalization Conventions

The rule of thumb with capitalization is that all types should be capitalized. Due to that, we will capitalize all class names.

```
class Vacation {

  constructor(destination, length) {
    this.destination = destination
    this.length = length
  }

  print() {
    console.log(`${this.destination} will take ${this.length} days.`)
  }
}
```

Once we've created the class, we'll create a new instance of the class using the new keyword. Then you can call the custom method on the class.

```
const trip = new Vacation("Santiago, Chile", 7);

console.log(trip.printDetails()); // Chile will take 7 days.
```

Now that a class object has been created, you can use it as many times as you'd like to create new vacation instances. Classes can also be extended. When a class is extended, the subclass inherits the properties and methods of the super class. These properties and methods can be manipulated from here, but as a default, all will be inherited.

You can use vacation as an abstract class to create different types of vacations. For instance, an Expedition can extend the vacation class to include gear.

```
class Expedition extends Vacation {

  constructor(destination, length, gear) {
    super(destination, length)
    this.gear = gear
  }
}
```

```

print() {
  super.print()
  console.log(`bring your ${this.gear.join(" and your ")}`)
}
}

```

That's simple inheritance: the subclass inherits the properties of the super class. By calling the `printDetails` method of `Vacation`, we can append some new content onto what is printed in the `printDetails` method of `Expedition`. Creating a new instance works the exact same way: create a variable and use the new keyword.

```

const trip = new Expedition("Mt. Whitney", 3,
  ["sunglasses", "prayer flags", "camera"])

trip.print() // bring your sunglasses and your prayer flags and your camera

```



Classes and Prototypal Inheritance

Using a class still means that you are using JavaScript's prototypal inheritance. Log `Vacation.prototype`, and you'll notice the constructor and `printDetails` methods on the prototype.

```
console.log(Vacation.prototype)
```

We will use classes a bit in this book, but we're focusing on the functional paradigm. Classes have other features like getters, setters, and static methods, but this book favors functional techniques over object-oriented techniques. The reason we're introducing these is because we'll use them later when creating React components.

JavaScript is indeed moving quickly and adapting to the increasing demands that engineers are placing on the language. Browsers are quickly implementing the features of ES6 and beyond, so it's a good idea to use these features now without hesitation.

Functional Programming with JavaScript

There has been quite a buzz lately about functional programming. Purely functional programming languages such as Haskell, Clojure, and Scala are currently being used by tech giants including Google, Facebook, LinkedIn, and Netflix. Popular languages like JavaScript, Python, and Ruby support functional programming techniques although they wouldn't be considered fully functional languages. There is an explosion of libraries in all of these languages that religiously follow and encourage you to follow those techniques. Even traditionally object-oriented languages such as Java and C++ support functional programming with lambdas¹.

If you are wondering where this functional trend came from, the answer is the 1930's, with the invention of lambda calculus². Traditionally, functions have been a part of calculus since it emerged in the 17th century. Functions can be sent to functions as arguments or returned from functions as results. More complex functions called *higher order functions* can manipulate functions themselves and use them as either arguments or results or both. In the 1930's, Alonzo Church was at Princeton messing around with these higher order functions when he invented a universal model of computation called lambda calculus or λ -calculus³.

In the late 1950's⁴, John McCarthy took the concepts derived from λ -calculus and applied them to a new programming language called Lisp. Lisp, which is not purely functional, still belongs to the functional language paradigm in the same way that

1 "A lambda is a block of code that can be passed as an argument to a function call." - <http://martinfowler.com/bliki/Lambda.html>

2 Lambda Calculus Timeline, http://turing100.acm.org/lambda_calculus_timeline.pdf

3 Lambda Calculus Timeline, http://turing100.acm.org/lambda_calculus_timeline.pdf

4 Lambda Calculus Timeline, http://turing100.acm.org/lambda_calculus_timeline.pdf

JavaScript or Python does. Lisp implemented the concept of higher order functions and functions as *first class members* or first class citizens. A function is considered a first class member when it can be declared as a variable and sent to functions as arguments. These functions can even be returned from functions.

The functional programming trend is not new to computer science. It is a throwback. There is also a chance that you have already written functional JavaScript code without thinking about it. If you've mapped or reduced an array, then you're already on your way to becoming a functional programmer. React, Flux, and Redux all fit within the functional JavaScript paradigm. Understanding the basic concepts of functional programming will make you better at structuring React applications.

In this chapter, we are going to cover how to implement functional techniques with JavaScript, as well as how those techniques have inspired React and Flux. We will wrap this chapter up with introducing the Flux pattern, a design architecture that has sprung from these

What it means to be Functional

A great debate to get into with other engineers is whether or not languages that support functional programming techniques can be called “functional languages.” In order to be a functional programmer, you're going to have to take a side in this debate. Are languages that support functional programming alongside other paradigms considered functional languages?

Purely functional programming languages such as Haskell or Clojure strictly enforce the functional paradigm. Other languages, such as JavaScript and Python support functional techniques, but they also support other programming paradigms as well. Many engineers only define a language that enforce the functional paradigm as a “functional language”.

The same debate is true for object orientation. JavaScript is not a traditional object-oriented language like C++ or Java. Yet, JavaScript supports objects and inheritance so we've been able to incorporate object-oriented design patterns such as MVC into our applications. Although JavaScript is not an object-oriented language, JavaScript supports object orientation with objects and prototypal inheritance. ES5 beefed up this support by introducing `Object.create` and `Object.defineProperties`. ES6 takes it a little further by introducing classes.

JavaScript supports functional programming because JavaScript functions are first class citizens, meaning that functions can do the same things that variables can do. ES6 adds language improvements that can beef up your functional programming techniques including arrow functions, promises, and the spread operator.

We call JavaScript a functional language because it supports first class members, but what does it mean to be first class? It means that functions are variables; they too can represent data in your application. You may have noticed that you can declare functions with the `var` keyword the same way you can declare strings, numbers, or any other variable.

```
var log = function(message) {  
  console.log(message)  
};  
  
log("In JavaScript functions are variables")  
  
// In JavaScript, functions are variables.
```

With ES6, we can write the same function using an arrow function. Functional programmers write a lot of small functions, and the arrow function makes that much easier. Both of these statements do the same thing, they store a function in a variable called `log`. Additionally, the `const` keyword was used to declare the second function, this will prevent it from being overwritten.

```
const log = message => console.log(message)
```

Since functions are variables, we can add them to objects.

```
const obj = {  
  message: "They can be added to objects like variables",  
  log(message) {  
    console.log(message)  
  }  
}  
  
obj.log(obj.message)  
  
// They can be added to objects like variables
```

We can also add functions to arrays in JavaScript.

```
const messages = [  
  "They can be inserted into arrays",  
  message => console.log(message),  
  "like variables",  
  message => console.log(message)  
]  
  
messages[1](messages[0]) // They can be inserted into arrays  
messages[3](messages[2]) // like variables
```

Functions can be sent to other functions as arguments just like other variables.

```
const insideFn = logger =>  
  logger("They can be sent to other functions as arguments");
```

```
insideFn(message => console.log(message))

// They can be sent to other functions as arguments
```

They can also be returned from other functions... just like variables

```
var createScream = function(logger) {
  return function(message) {
    logger(message.toUpperCase() + "!!!")
  }
}

const scream = createScream(message => console.log(message))

scream('functions can be returned from other functions')
scream('createScream returns a function')
scream('scream invokes that returned function')

// FUNCTIONS CAN BE RETURNED FROM OTHER FUNCTIONS!!!
// CREATESCREAM RETURNS A FUNCTION!!!
// SCREAM INVOKES THAT RETURNED FUNCTION!!!
```

The last two examples were of higher order functions, functions that either take or return other functions. Using ES6 syntax, we could describe the same createScream higher order function with arrows.

```
const createScream = logger => message =>
  logger(message.toUpperCase() + "!!!")
```

From here on out, we need to pay attention to the number of arrows used during function declaration. More than one arrow means that we have a higher order function.

We can say that JavaScript is a functional language because its functions are first class citizens. This means that functions are data. They can be saved, retrieved, or flow through your applications just like variables.

Imperative vs Declarative

Functional programming is a part of a larger programming paradigm: declarative programming. Declarative programming is a style of programming where applications are structured in a way that prioritizes describing what should happen over defining how it should happen.

In order to understand declarative programming, we'll contrast it with imperative programming, or a style of programming that is only concerned with how to achieve results with code. Let's consider a common task: making a string URL friendly. Typically, this can be accomplished by replacing all of the spaces in a string with hyphens,

since spaces are not URL friendly. First, let's examine an imperative approach to this task.

```
var string = "This is the mid day show with Cheryl Waters";
var urlFriendly = "";

for (var i=0; i<string.length; i++) {
  if (string[i] === " ") {
    urlFriendly += "-";
  } else {
    urlFriendly += string[i];
  }
}

console.log(urlFriendly);
```

In this example, we loop through every character in the string replacing spaces as they occur. The structure of this program is only concerned with how such a task can be achieved. We use a for loop, an if statement, and set values with an equal operator. Just looking at the code alone does not tell us much. Imperative programs require lots of comments in order to understand what is going on.

Now let's look at a declarative approach to the same problem.

```
const string = "This is the mid day show with Cheryl Waters"
const urlFriendly = string.replace(/ /g, "-")

console.log(urlFriendly)
```

Here we are using `string.replace` along with a regular expression to replace all instances of spaces with hyphens. Using `string.replace` is a way of describing what is supposed to happen, spaces in the string should be replaced. The details of how spaces are dealt with are abstracted away inside the `.replace` function. In a declarative program, the syntax itself describes what should happen and abstract away the details of how things happen through abstraction.

Declarative programs are easy to reason about because the code itself describes what is happening. For example, read the syntax in the following sample, it details what happens after members are loaded from an API.

```
const loadAndMapMembers = compose(
  combineWith(sessionStorage, "members"),
  save(sessionStorage, "members"),
  scopeMembers(window),
  logMemberInfoToConsole,
  logFieldsToConsole("name.first"),
  countMembersBy("location.state"),
  prepStatesForMapping,
  save(sessionStorage, "map"),
  renderUSMap
);
```

```
getFakeMembers(100).then(loadAndMapMembers);
```

The declarative approach is more readable and, thus, easier to reason about. The details of how each of these functions is implemented are abstracted away. Those tiny functions are named well and combined in a way that describes how member data goes from being loaded to being saved and printed on a map. This approach does not require many comments. Declarative programming should produce applications that are easier to reason about. When it is easier to reason about an application, that application is easier to scale⁵.

Now, let's consider the task of building a DOM. An imperative approach would be concerned with how the DOM is constructed.

```
var target = document.getElementById('target');
var wrapper = document.createElement('div');
var headline = document.createElement('h1');

wrapper.id = "welcome";
headline.innerText = "Hello World";

wrapper.appendChild(headline);
target.appendChild(wrapper);
```

This code is concerned with creating elements, setting elements, and adding them to the document. It would be very hard to make changes, add features, or scale 10,000 lines of code where the DOM is constructed imperatively.

Now let's take a look at how we can construct a DOM declaratively using a React component.

```
const { render } = ReactDOM

const Welcome = () => (
  <div id="welcome">
    <h1>Hello World</h1>
  </div>
)

render(
  <Welcome />,
  document.getElementById('target')
)
```

React is declarative. Here, the welcome component describes the DOM that should be rendered. The render function uses the instructions declared in the component to build the DOM. The render function abstracts away the details of how the DOM is to

5 Additional detail about the declarative programming paradigm can be found here: <http://c2.com/cgi/wiki?DeclarativeProgramming>

be rendered. We can clearly see that we want to render our welcome component into the element with the id of “target”.

Functional Concepts

Now that we have been introduced to functional programming, and what it means to be “functional” or “declarative”, we will move on to introducing the core concepts of functional programming: immutability, purity, data abstraction, higher-order functions, recursion, and composition.

Immutability

To mutate is to change, so to be immutable is to be unchangeable. In a functional program, data is immutable, it never changes.

If you needed to share your birth certificate with the public, but wanted to redact or remove private information you essentially have two choices: you can take a big Sharpie to your original birth certificate and cross out private data, or you can find a copy machine. Finding a copy machine, making a copy of your birth certificate, and writing all over that copy with that big Sharpie would be preferable. This way you can have a redacted birth certificate, which you can share and your original which is still intact.

This is how immutable data works in an application. We will not change the original data structures. We will build changed copies of those data structures and use them instead.

To understand how immutability works, let’s take a look at what it means to mutate data. Consider an object that represents the color lawn.

```
let color_lawn = {  
  title: "lawn",  
  color: "#00FF00",  
  rating: 0  
}
```

We could build a function that would rate colors, and use that function to change the rating of the color object.

```
function rateColor(color, rating) {  
  color.rating = rating  
  return color  
}  
  
console.log(rateColor(color_lawn, 5).rating) // 5  
console.log(color_lawn.rating)              // 5
```

In JavaScript, function arguments are references to the actual data. Setting the color's rating would change or mutate the original color object. Imagine if you tasked a business with redacting and sharing your birth certificate and they returned your original birth certificate with black marker covering the important details. This rate color function is bad for business because it changing the original color.

You would hope that a business would use enough common sense to make a copy of your birth certificate and return the original unharmed. We can rewrite the rateColor function so that it does not harm the original goods, the color object.

```
var rateColor = function(color, rating) {  
  return Object.assign({}, color, {rating:rating})  
}  
  
console.log(rateColor(color_lawn, 5).rating)    // 5  
console.log(color_lawn.rating)                  // 4
```

Here, we used Object.assign to change the color rating. Object.assign is the copy machine. It takes a blank object, copies the color to that object, and overwrites the rating on the copy. Now we can have a newly rated color object without having to change the original.

We can write the same function using an ES6 arrow function along with the ES7 object spread operator. This rate color function uses the spread operator to copy the color into a new object and then overwrite its rating.

```
const rateColor = (color, rating) =>  
  ({  
    ...color,  
    rating  
  })
```

This emerging JavaScript version of the rateColor is exactly the same as the previous. It treats the color as an immutable object. It just does so with less syntax and looks a little bit cleaner. Notice that we wrap the returned object in parentheses. With arrow functions, this is a required step since the arrow can't just point to an object's curly braces.

Let's consider an array of color names.

```
let list = [  
  { title: "Rad Red"},  
  { title: "Lawn"},  
  { title: "Party Pink"}  
]
```

We could create a function that will add colors to that array using array.push().

```
var addColor = function(title, colors) {  
  colors.push({ title: title })  
  return colors;  
}
```

```

}

console.log(addColor("Glam Green", list).length) // 4
console.log(list.length) // 4

```

However, `array.push()` is not immutable. This `addColor` function changes the original array by adding another field to it. In order to keep the colors array immutable, we must use `array.concat` instead.

```

const addColor = (title, array) => array.concat({title})

console.log(addColor("Glam Green", list).length) // 4
console.log(list.length) // 3

```

`Array.concat` concatenates arrays. In this case, it takes a new object, with a new color title, and adds it to a copy of the original array.

You can also use the ES6 spread operator to concatenate arrays in the same way it can be used to copy objects. Here is the emerging JavaScript equivalent of the previous `addColor` function.

```

const addColor = (title, list) => [...list, {title}]

```

This function copies the original list to a new array and then adds a new object containing the color's title to that copy. It is immutable.

Pure Functions

A pure function is a function that returns a value that is computed based on its arguments. Pure functions take at least one argument and always return a value or another function. They do not cause side effects. They do not set global variables or change anything about application state. They treat their arguments as immutable data. If you send a specific argument to a pure function, you can expect a specific result.

In order to understand pure functions, we will first take a look at an impure function.

```

var frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
}

function selfEducate() {
  frederick.canRead = true
  frederick.canWrite = true
  return frederick
}

selfEducate()

```

```
console.log( frederick )

// {name: "Frederick Douglass", canRead: false, canWrite: false}
```

The `selfEducate` function is not a pure function. It does not take any arguments, and it does not return a value or a function. It also changes a variable outside of its scope: `frederick`. Once the `selfEducate` function is invoked, something about the “world” has changed. It causes side effects.

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
}

const selfEducate = (person) => {
  person.canRead = true
  person.canWrite = true
  return person
}

console.log( selfEducate(frederick) )
console.log( frederick )

// {name: "Frederick Douglass", canRead: false, canWrite: false}
// {name: "Frederick Douglass", canRead: false, canWrite: false}
```



Pure functions are testable

Pure functions are naturally testable. They do not change anything about their environment or “world”, and therefore do not require a complicated test setup or teardown. Everything a pure function needs to operate it accesses via arguments. When testing a pure function, you control the arguments, and thus you can estimate the outcome. More on testing in Chapter 11.

This `selfEducate` function is also impure. It causes side effects. Invoking this `selfEducate` function mutates the objects that are sent to it. If we can treat the arguments sent to this function as immutable data, then we would have ourselves a pure function.

Let’s have this function take in an argument.

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
}

const selfEducate = person =>
  ({
    ...person,
```

```

    canRead: true,
    canWrite: true
  })

  console.log( selfEducate(frederick) )
  console.log( frederick )

  // {name: "Frederick Douglass", canRead: false, canWrite: false}
  // {name: "Frederick Douglass", canRead: true, canWrite: true}

```

Finally, this version of the `selfEducate` is a pure function. It computes a value based on the argument that was sent to it - the person. It returns a new person object without mutating the argument sent to it and therefore has no side effects.

Now let's examine an impure function that mutates the DOM.

```

function Header(text) {
  let h1 = document.createElement('h1');
  h1.innerText = text;
  document.body.appendChild(h1);
}

Header("Header() caused side effects");

```

The `header` function creates a heading one element with specific text and adds it to the DOM. This function is impure. It does not return a function or a value, and it causes side effects: a changed DOM.

In React, UI is expressed with pure functions. In this sample, `Header` is a pure function that can be used to create heading one elements just like in the previous example. However this function on its own does not cause side effects because it does not mutate the DOM. This function will create a heading one element, and it is up to some other part of the application to use that element to change the DOM.

```

const Header = (props) => <h1>{props.title}</h1>

```

Pure functions are another core concept of functional programming. They will make your life much easier because they will not affect your application's state. When writing functions try to follow these three rules:

1. The function should take in at least one argument
2. The function should return a value or another function
3. The function should not change or mutate any of its arguments

Data Transformations

How does anything change in an application if the data is immutable? Functional programming is all about transforming data from one form to another. We will pro-

duce transformed copies using functions. These functions make our code less imperative and thus reduce complexity.

In order to use React, you have to learn how to use JavaScript.
- Dan Abramov

Much of functional programming is about transforming data. You do not need a special framework to understand how produce one dataset that is based upon another. JavaScript already has the necessary tools for this task built into the language. There are two core functions that you must master in order to be proficient with functional JavaScript. These functions are `array.map` and `array.reduce`.

In this section, we will take a look at how we can use some of these core functions to transform data from one type to another.

Consider this array of high schools.

```
const schools = [
  "Yorktown",
  "Washington & Lee",
  "Wakefield"
]
```

We can get a comma delimited list of these strings by using the `array.join` function.

```
console.log( schools.join(", ") )

// "Yorktown, Washington & Lee, Wakefield"
```

The `join` function is a built-in JavaScript array method that we can use to extract a delimited string from our array. The original array is still intact. `Join` simply provides a different take on it. The details of how this string is produced are abstracted away from the programmer.

If we wanted to create a function that creates a new array of the schools that begin with the letter “W” we could use the `array.filter` method.

```
const wSchools = schools.filter(school => school[0] === "W")

console.log( wSchools )

// ["Washington & Lee", "Wakefield"]
```

`Array.filter` is a built-in JavaScript function that produces a new array from a source array. This function takes a *predicate* as its only argument. A predicate is a function that always returns a boolean value: true or false. `Array.filter` invokes this predicate once for every item in the array. That item is passed to the predicate as an argument and used to decide if that item shall be added to the new array. In this case, `array.filter` is checking every school to see if it begins with a “W”.

When it is time to remove an item from an array we should use `array.filter` over `array.pop()` or `array.splice()` because `array.filter` is immutable. In this next sample, the `cutSchool` function returns new arrays that filter out specific school names.

```
const cutSchool = (cut, list) =>
  list.filter(school => school !== cut)

console.log(cutSchool("Washington & Lee", schools).join(" * "))

// "Yorktown * Wakefield"

console.log(schools.join("\n"))

// Yorktown
// Washington & Lee
// Wakefield
```

In this case, the `cutSchool` function was used to return a new array that does not contain “Washington & Lee”. Then the `join` function is used with this new array to create a star delimited string out of the remaining two schools. `CutSchool` is a pure function. It takes a list of schools and the name of the school that should be removed and returns the new array without that specific school. Additionally, the `join` function has been chained on to produce a star delimited string out of the returned array.

Another array function that is essential to functional programming is `array.map`. Instead of a predicate, the `array.map` method takes a function as its argument. This function will be invoked once for every item in the array, and whatever it returns will be added to the new array.

```
const highSchools = schools.map(school => `${school} High School`)

console.log(highSchools.join("\n"))

// Yorktown High School
// Washington & Lee High School
// Wakefield High School

console.log(schools.join("\n"))

// Yorktown
// Washington & Lee
// Wakefield
```

In this case, the `map` function was used to append “High School” to each school name. The `schools` array is still intact.

In the last example, we produced an array of strings from an array of strings. The `map` function could produce an array of objects, values, arrays, other functions - any JavaScript type. Here is an example of the `map` function returning an object for every school.

```
const highSchools = schools.map(school => ({ name: school })))

console.log( highSchools )

// [
//   { name: "Yorktown" },
//   { name: "Washington & Lee" },
//   { name: "Wakefield" }
// ]
```

An array containing objects was produced from an array that contains strings.

If you need to create a pure function that changes one object in an array of objects, the map function can be used. In the following example, we will change the school with the name of Stratford to HB Woodlawn without mutating the schools array.

```
let schools = [
  { name: "Yorktown" },
  { name: "Stratford" },
  { name: "Washington & Lee" },
  { name: "Wakefield" }
]

let updatedSchools = editName("Stratford", "HB Woodlawn", schools)

console.log( updatedSchools[1] ) // { name: "HB Woodlawn" }
console.log( schools[1] )        // { name: "Stratford" },
```

The schools array is an array of objects. The updatedSchools variable calls the editName function and we send it the school we want to update, the new school, and the schools array. This changes the new array but makes no edits to the original.

```
const editName = (oldName, name, arr) =>
  arr.map(item => {
    if (item.name === oldName) {
      return {
        ...item,
        name
      }
    } else {
      return item
    }
  })
```

Within editName, the map function is used to create a new array of objects based upon the original array. Array.map injects the index of each item into the callback as the second argument, the variable i. When i is not equal to the index of the item we wish to edit, we'll simply package the same item into the new array. When i is equal to the index of the item that we wish to edit, we replace the item at that index in the new array with a new object.

The editName function can be written entirely in one line. The is an example of the same function using a shorthand if/else statement

```
const editName = (oldName, name, arr) =>
  arr.map(item => (item.name === oldName) ?
    ({...item,name}) :
    item
  )
```

If you needed to transform an array into an object, you can use array.map in conjunction with Object.keys. Object.keys is a method that can be used to return an array of keys from an object.

Let's say we needed to transform an array of school objects from a hash of schools.

```
const schools = {
  "Yorktown": 10,
  "Washington & Lee": 2,
  "Wakefield": 5
}

const schoolArray = Object.keys(schools).map(key =>
  ({
    name: key,
    wins: schools[key]
  })
)

console.log(schoolArray)

// [
//   {
//     name: "Yorktown",
//     wins: 10
//   },
//   {
//     name: "Washington & Lee",
//     wins: 2
//   },
//   {
//     name: "Wakefield",
//     wins: 5
//   }
// ]
```

In this example, Object.keys returns an array of school names, and we can use map on that array to produce a new array of the same length. The name of the new object will be set using the key, and the wins is set equal to the value.

So far we've learned that we can transform arrays with array.map and array.filter. We've also learned that we can change arrays into objects by combining Object.keys

with `Array.map`. The final tool that that we need in our functional arsenal is the ability to transform arrays into primitives and other objects.

The `reduce` and `reduceRight` function can be used to transform an array into any value. Any value means a number, string, boolean, object, or even function.

Let's say we needed to find the maximum number in an array of numbers. We need to transform an array into a number; therefore, we can use `reduce`.

```
const ages = [21,18,42,40,64,63,34];

const maxAge = ages.reduce((max, age) => {
  console.log(`${age} > ${max} = ${age > max}`);
  if (age > max) {
    return age
  } else {
    return max
  }
}, 0)

console.log('maxAge', maxAge);

// 21 > 0 = true
// 18 > 21 = false
// 42 > 21 = true
// 40 > 42 = false
// 64 > 42 = true
// 63 > 64 = false
// 34 > 64 = false
// maxAge 64
```

The `ages` array has been reduced into a single value: the maximum age: 64. `Reduce` takes two arguments: a callback function and an original value. In this case, the original value is 0, which sets the initial maximum value to 0. The callback is invoked once for every item in the array. The first time this callback is invoked the age is equal to 21, the first value in the array, and `max` is equal to 0, the initial value. The callback returns the greater of the two numbers, 21, and that becomes the `max` value during the next iteration. Each iteration compares each age against the `max` value and returns the greater of the two. Finally, the last number in the array is compared and returned from the previous callback.

If we remove the `console.log` statement from the above function and use a shorthand `if/else` statement, we can calculate the `max` value in any array of numbers with the following syntax:

```
const max = ages.reduce(
  (max, value) => (value > max) ? value : max,
  0
)
```



array.reduceRight

Array.reduce right works the same way as array.reduce, the difference is that it starts reducing from the end of the array rather than the beginning.

Sometimes we need to transform an array into an object. The following example uses reduce to transform an array that contains colors into a hash.

```
const colors = [
  {
    id: '-xekare',
    title: "rad red",
    rating: 3
  },
  {
    id: '-jbwsof',
    title: "big blue",
    rating: 2
  },
  {
    id: '-prigbj',
    title: "grizzly grey",
    rating: 5
  },
  {
    id: '-ryhbhsl',
    title: "banana",
    rating: 1
  }
]

const hashColors = colors.reduce(
  (hash, {id, title, rating}) => {
    hash[id] = {title, rating}
    return hash
  },
  {}
)

console.log(hashColors);

// {
//   "-xekare": {
//     title:"rad red",
//     rating:3
//   },
//   "-jbwsof": {
//     title:"big blue",
//     rating:2
//   },
//   "-prigbj": {
```

```
//    title:"grizzly grey",
//    rating:5
//  },
//  "-ryhbhsl": {
//    title:"banana",
//    rating:1
//  }
// }
```

In the above example, the second argument sent to the reduce function is an empty object. This is our initial value for hash. During each iteration, the callback function adds a new key to the hash using bracket notation and sets the value for that key to the id field of the array. `Array.reduce` reduces can be used to reduce an array to a single value, in this case, an object.

We can even transform arrays into completely different arrays using reduce. There are cases where `array.reduce` is a better choice. Consider reducing an array with multiple instances of the same value to an array of distinct values. The reduce method can be used to accomplish this task.

```
const colors = ["red", "red", "green", "blue", "green"];

const distinctColors = colors.reduce(
  (distinct, color) =>
    (distinct.indexOf(color) !== -1) ?
      distinct :
      [...distinct, color],
  []
)

console.log(distinctColors)

// ["red", "green", "blue"]
```

In this example, the `colors` array is reduced to an array of distinct values. The second argument sent to the reduce function is an empty array. This will be the initial value for `distinct`. When the `distinct` array does not already contain a specific color, it will be added. Otherwise, it will be skipped, and the current `distinct` array will be returned.

Map and reduce are the main weapons of any functional programmer, and JavaScript is no exception. If you want to be a proficient JavaScript engineer, then you must master these functions. The ability to create one data set from another is a required skill and is useful for any type of programming paradigm.

Higher Order Functions

The use of higher order functions is also essential to functional programming. We've already mentioned higher order functions several times over, and we've even used a few in this chapter. Higher order functions are functions that can manipulate other functions. They can either take functions in as arguments or return functions or both.

The first category of higher order functions are functions that expect other functions as arguments. `Array.map`, `array.filter`, and `array.reduce` all take functions as arguments. They are higher order functions.⁶

Let's take a look at how we can implement a higher order function. In the following example, we will create an `invokeIf` callback function that will test a condition and invoke a callback function when it is true and another callback function when that condition is false.

```
const invokeIf = (condition, fnTrue, fnFalse) =>
  (condition) ? fnTrue() : fnFalse()

const showWelcome = () =>
  console.log("Welcome!!!")

const showUnauthorized = () =>
  console.log("Unauthorized!!!")

invokeIf(true, showWelcome, showUnauthorized) // "Welcome"
invokeIf(false, showWelcome, showUnauthorized) // "Unauthorized"
```

`InvokeIf` expects two functions: one for true, and one for false. This is demonstrated by sending both `showWelcome` and `showUnauthorized` to `invokeIf`. When the condition is true, `showWelcome` is invoked. When it is false, `showUnauthorized` is invoked.

Higher order functions that return other functions can help us handle the complexities associated with asynchronicity in JavaScript. They can help us create functions that can be used or reused at our convenience.

Currying is a functional technique that involves the use of higher order functions. Currying is the practice of holding on to some of the values needed to complete an operation until the rest can be supplied at a later point in time. This is achieved through the use of a function that returns another function, the curried function.

The following is an example of currying. The `userLogs` function hangs on to some information - the user name - and returns a function that can be used and reused when the rest of the information - the message - is made available. In this example,

⁶ For more on higher-order functions, check out *Eloquent JavaScript*, Chapter 5. http://eloquentjavascript.net/05_higher_order.html

log messages will all be prepended with the associated username. Notice that we're using the `getFakeMembers` function that returns a promise from chapter 2.

```
const userLogs = userName => message =>
  console.log(`${userName} -> ${message}`)

const log = userLogs("grandpa23")

log("attempted to load 20 fake members")
getFakeMembers(20).then(
  members => log(`successfully loaded ${members.length} members`),
  error => log("encountered an error loading members")
)

// grandpa23 -> attempted to load 20 fake members
// grandpa23 -> successfully loaded 20 members

// grandpa23 -> attempted to load 20 fake members
// grandpa23 -> encountered an error loading members
```

UserLogs is the higher order function. The log function is produced from userLogs, and every time the log function is used, “grandpa23” is prepended to the message.

Recursion

Recursion is a technique that involves creating functions that recall themselves. Often when faced with a challenge that involves a loop, a recursive function can be used instead. Consider the task of counting down from 10. We could create a for loop to solve this problem, or we could alternatively use a recursive function. In this example, countdown is the recursive function.

```
const countdown = (value, fn) => {
  fn(value)
  return (value > 0) ? countdown(value-1, fn) : value
}

countdown(10, value => console.log(value));

// 10
// 9
// 8
// 7
// 6
// 5
// 4
// 3
// 2
// 1
// 0
```


Countdown expects a number and a function as arguments. In this example, countdown is invoked with a value of 10 and a callback function. When countdown is invoked, the callback is invoked which logs the current value. Next, countdown checks the value to see if it is greater than 0. If it is, countdown recalls itself with a decremented value. Eventually, the value will be 0 and countdown will return that value all the way back up the call stack.



Browser Call Stack Limitations

Recursion should be used over loops wherever possible, but not all JavaScript engines are optimized for a large amount of recursion. Too much recursion can cause JavaScript errors. These errors can be avoided by implementing advanced techniques to clear the call stack and flatten out recursive calls. Future JavaScript engines plan to eliminate any call stack limitations entirely..

Recursion is another functional technique that works well with asynchronous processes. Functions can recall themselves when they are ready.

The countdown function can be modified to count down with a delay. This modified version of the countdown function can be used to create a countdown clock.

```
const countdown = (value, fn, delay=1000) => {  
  fn(value)  
  return (value > 0) ?  
    setTimeout(() => countdown(value-1, fn), delay) :  
    value  
}  
  
const log = value => console.log(value)  
countdown(10, log);
```

In this example, we create a 10-second countdown by initially invoking countdown once with the number 10 in a function that logs the countdown. Instead of recalling itself right away, the countdown function waits one second before recalling itself, thus creating a clock.

Recursion is a good technique for searching data structures. You can use recursion to iterate through subfolders until the folder, the one that contains only files, is identified. You can use recursion to iterate through the HTML DOM until you find the one that does not contain any children. In the next example, we will use recursion to iterate deeply into an object to retrieve a nested value.

```
var dan = {  
  type: "person",  
  data: {  
    gender: "male",  
    info: {
```

```

    id: 22,
    fullname: {
      first: "Dan",
      last: "Deacon"
    }
  }
}
}

deepPick("type", dan); // "person"
deepPick("data.info.fullname.first", dan); // "Deacon"

```

DeepPick, can be used to access dan's type, stored immediately in the first object, or dig down into nested objects to locate dan's first name. Sending a string that uses dot notation, we can specify where to locate values that are nested deep within an object.

```

const deepPick = (fields, object={}) => {
  const [first, ...remaining] = fields.split(".")
  return (remaining.length) ?
    deepPick(remaining.join("."), object[first]) :
    object[first]
}

```

The deepPick function is either going to return a value or recall itself, until it eventually returns a value. First, this function splits the dot notated fields string into an array and uses array destructuring to separate the first value from the remaining values. If there are remaining values, deepPick recalls itself with slightly different data, allowing it to dig one level deeper.

This function continues to call itself until the field string no longer contains dots, meaning that there are no more remaining fields. In this sample, you can see how the values for first, remaining, and object[first] change as deepPick iterates through match.

```

deepPick("data.info.fullname.first", dan); // "Deacon"

// First Iteration
// first = "data"
// remaining.join(".") = "info.fullname.first"
// object[first] = { gender: "male", {info} }

// Second Iteration
// first = "info"
// remaining.join(".") = "fullname.first"
// object[first] = {id: 22, {fullname}}

// Third Iteration
// first = "fullname"
// remaining.join(".") = "first"
// object[first] = {first: "Dan", last: "Deacon" }

// Finally...

```

```
// first = "first"
// remaining.length = 0
// object[first] = "Deacon"
```

Recursion is a powerful functional technique that is fun to implement. Use recursion over looping whenever possible.

Composition

Functional programs break their logic up into small pure functions that are focused on specific tasks. Eventually, you will need to put these smaller functions together. Specifically, you may need to combine them, call them in series or parallel, or compose them into larger functions until you eventually have an application.

When it comes to composition, there are a number of different implementations, patterns, and techniques. One that you may be familiar with is chaining. In JavaScript functions can be chained together using dot notation to act on the return value of the previous function.

Strings have a replace method. The replace method returns a template string which also will have a replace method. Therefore, we can chain together replace methods with dot notation to transform a string.

```
const template = "hh:mm:ss tt"
const clockTime = template.replace("hh", "03")
                           .replace("mm", "33")
                           .replace("ss", "33")
                           .replace("tt", "PM")

console.log(clockTime)

// "03:33:33 PM"
```

In this example, the template is a string. By chaining replace methods to the end of the template string, we can replace hours, minutes, seconds, and time of day in the string with new values. The template itself remain intact and can be reused to create more clock time displays.

Chaining is one composition technique, but there are others. The goal of composition is to “generate a higher order function by combining simpler functions.”⁷

```
const both = date => appendAMPM(civilianHours(date))
```

The both function is one function that pipes a value through two separate functions. The output of civilian hours becomes the input for appendAMPM, and we can

⁷ Functional.js Composition, <http://functionaljs.com/functions/compose/>

change a date using both of these functions combined into one. However, this syntax is hard to comprehend and therefore tough to maintain or scale. What happens when we need to send a value through 20 different functions?

A more elegant approach is to create a higher order function that we can use to compose functions into larger functions.

```
const both = compose(  
  civilianHours,  
  appendAMPM  
)  
  
both(new Date())
```

This approach looks much better. It is easy to scale because we can add more functions at any point. This approach also makes it easy to change the order of the composed functions.

The compose function is a higher order function. It takes functions as arguments and returns a single value.

```
const compose = (...fns) =>  
  (arg) =>  
    fns.reduce(  
      (composed, f) => f(composed),  
      arg  
    )
```

Compose takes in functions as arguments and returns a single function. In this implementation, the spread operator is used to turn those function arguments into an array called `fns`. A function is then returned that expects one argument, `arg`. When this function is invoked, the `fns` array is piped starting with the argument we want to send through the function. The argument becomes the initial value for `composed` and then each iteration of the reduced callback returns. Notice that the callback takes two arguments: `composed` and a function `f`. Each function is invoked with `composed` which is the result of the previous functions output. Eventually, the last function will be invoked and the last result returned.

This is a simple example of a compose function designed to illustrate composition techniques. This function becomes more complex when it is time to handle more than one argument or deal with arguments that are not functions. Other implementations of `compose`⁸ may use `reduceRight` which would compose the functions in reverse order.

⁸ Another implementation of `compose` is found in Redux: <http://redux.js.org/docs/api/compose.html>

Putting it all together

Now that we've been introduced to the core concepts of functional programming, let's put those concepts to work for us and build a small JavaScript application.

Since JavaScript will let you slip away from the functional paradigm, and you do not have to follow the rules, you will need to stay focused. Following these three simple rules will help you stay on target.

1. Keep Data Immutable
2. Keep Functions Pure : accept at least one argument, return data or another function
3. Use Recursion over looping (wherever possible)

Our challenge is to build a ticking clock. The clock needs to display hours, minutes, seconds and time of day in civilian time. Each field must always have double digits, that means leading zeros need to be applied to single digit values like 1 or 2. The clock must also tick and change the display every second.

First, let's review an imperative solution for the clock.

```
// Log Clock Time every Second
setInterval(logClockTime, 1000);

function logClockTime() {

  // Get Time string as civilian time
  var time = getClockTime();

  // Clear the Console and log the time
  console.clear();
  console.log(time);
}

function getClockTime() {

  // Get the Current Time
  var date = new Date();
  var time = "";

  // Serialize clock time
  var time = {
    hours: date.getHours(),
    minutes: date.getMinutes(),
    seconds: date.getSeconds(),
    ampm: "AM"
  }
```

```

    // Convert to civilian time
    if (time.hours == 12) {
        time.ampm = "PM";
    } else if (time.hours > 12) {
        time.ampm = "PM";
        time.hours -= 12;
    }

    // Prepend a 0 on the hours to make double digits
    if (time.hours < 10) {
        time.hours = "0" + time.hours;
    }

    // prepend a 0 on the minutes to make double digits
    if (time.minutes < 10) {
        time.minutes = "0" + time.minutes;
    }

    // prepend a 0 on the seconds to make double digits
    if (time.seconds < 10) {
        time.seconds = "0" + time.seconds;
    }

    // Format the clock time as a string "hh:mm:ss tt"
    return time.hours + ":"
        + time.minutes + ":"
        + time.seconds + " "
        + time.ampm;
}

```

This solution is pretty straight forward. It works, the comments help us understand what is happening. However, these functions are large and complicated. Each function does a lot. They are hard to comprehend, they require comments and they are tough to maintain. Let's see how a functional approach can produce a more scalable application.

Our goal will be to break the application logic up into smaller parts, functions. Each function will be focused on a single task, and we will compose them into larger functions that we can use to create the clock.

First, let's create some functions that give us values and manage the console. We'll need a function that gives us one second, a function that gives us the current time, and a couple of functions that will log messages on a console and clear the console. In functional programs, we should use functions over values wherever possible. We will invoke the function to obtain the value when needed.

```

const oneSecond = () => 1000
const getCurrentTime = () => new Date()
const clear = () => console.clear()
const log = message => console.log(message)

```

Next we will need some functions for transforming data. These three functions will be used to mutate the Date object into an object that can be used for our clock.

serializeClockTime

Takes a date object and returns a object for clock time that contains hours minutes and seconds.

civilianHours

Takes the clock time object and returns an object where hours are converted to civilian time. For example: 1300 becomes 1 o'clock

appendAMPM

Takes the clock time object and appends time of day, AM or PM, to that object.

```
const serializeClockTime = date =>
({
  hours: date.getHours(),
  minutes: date.getMinutes(),
  seconds: date.getSeconds()
})

const civilianHours = clockTime =>
({
  ...clockTime,
  hours: (clockTime.hours > 12) ?
    clockTime.hours - 12 :
    clockTime.hours
})

const appendAMPM = clockTime =>
({
  ...clockTime,
  ampm: (clockTime.hours >= 12) ? "PM" : "AM"
})
```

These three functions are used to transform data without changing the original. They treat their arguments as immutable objects.

Next we'll need a few higher order functions.

display

Takes a target function and returns a function that will send a time to the target. In this example the target will be console.log.

formatClock

Takes a template string and uses it to return clock time formatted based upon the criteria from the string. In this example, the template is "hh:mm:ss tt". From there, formatClock will replace the placeholders with hours, minutes, seconds, and time of day.

prependZero

Takes an object's key as an argument and prepends a zero to the value stored under that object's key. It takes in a key to a specific field and prepends values with a zero if the value is less than 10.

```
const display = target => time => target(time)

const formatClock = format =>
  time =>
    format.replace("hh", time.hours)
           .replace("mm", time.minutes)
           .replace("ss", time.seconds)
           .replace("tt", time.ampm)

const prependZero = key => clockTime =>
  ({
    ...clockTime,
    [key]: (clockTime[key] < 10) ?
      "0" + clockTime[key] :
      clockTime[key]
  })
```

These higher order functions will be invoked to create the functions that will be reused to format the clock time for every tick. Both format clock and prependZero will be invoked once, initially setting up the required template or key. The inner functions that they return will be invoked once every second to format the time for display.

Now that we have all of the functions required to build a ticking clock, we will need to compose them. We will use the compose function that we defined in the last section to handle composition.

convertToCivilianTime

A single function that will take clock time as an argument and transforms it into civilian time by using both civilian hours.

doubleDigits

A single function that will take civilian clock time and make sure the hours, minutes, and seconds display double digits by prepending zeros where needed.

startTicking

Starts the clock by setting an interval that will invoke a callback every second. The callback is composed using all of our functions. Every second the console is cleared, currentTime obtained, converted, civilianized, formatted, and displayed.

```
const convertToCivilianTime = clockTime =>
  compose(
    appendAMPM,
    civilianHours
  )(clockTime)
```



```

const doubleDigits = civilianTime =>
  compose(
    prependZero("hours"),
    prependZero("minutes"),
    prependZero("seconds")
  )(civilianTime)

const startTicking = () =>
  setInterval(
    compose(
      clear,
      getCurrentTime,
      serializeClockTime,
      convertToCivilianTime,
      doubleDigits,
      formatClock("hh:mm:ss tt"),
      display(log)
    ),
    oneSecond()
  )

startTicking()

```

This declarative version of the clock achieves the same results as the imperative version. However, there are quite a few benefits to this approach. First, all of these functions are easily testable and reusable. They can be used in future clocks or other digital displays. Also, this program is easily scalable. There are no side effects. There are no global variables outside of functions themselves. There could still be bugs, but they will be easier to find.

In this chapter, we've introduced functional programming principles. Throughout the book when we discuss best practices in React and Flux, we'll demonstrate how these libraries are based in functional techniques. In the next chapter, we will dive into React officially, with an improved understanding of the principles that guided its development.

Pure React

In order to understand how React runs in the browser, we will be working purely with React. We will not introduce JSX, or JavaScript as XML, until the next chapter. You may have worked with React in the past without ever looking at the pure React code that is generated when we transpile JSX into React. You can be successful at working with React without ever looking at that code. However, if you take the time to understand what is going on behind the scenes you will be more efficient, especially when it comes time to debug. That is our goal in this chapter: to look under the hood and understand how React works.

Page Setup

In order to work with React in the browser, we need to include two libraries: React and ReactDOM. React is the library for creating views. ReactDOM is the library used to actually render the UI in the browser.

ReactDOM

React and ReactDOM were split into two packages for version 0.14 with the release notes stating “the beauty and the essence of React has nothing to do with browsers or the DOM...This [splitting into two packages] paves the way to writing components that can be shared between the web version of React and React Native”¹. Instead of assuming that React will render only in the browser, the future will aim to support rendering for a variety of platforms.

¹ React v0.14 by Ben Alpert. <https://facebook.github.io/react/blog/2015/10/07/react-v0.14.html>

We also need an HTML element that ReactDOM will use to render the UI. You can see how the scripts and HTML elements are added below. Both libraries are available as scripts from the Facebook CDN.

Example 4-1. HTML Document Setup with React

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Pure React Samples</title>
</head>
<body>

  <!-- Target Container -->
  <div class="react-container"></div>

  <!-- React Library & React DOM-->
  <script src="https://fb.me/react-15.1.0.js"></script>
  <script src="https://fb.me/react-dom-15.1.0.js"></script>

  <script>

    // Pure React and JavaScript Code

  </script>

</body>
</html>
```

These are the minimum requirements for working with React in the browser. You can place your JavaScript in a separate file, but it must be loaded somewhere in the page after React has been loaded.

The Virtual DOM

The Virtual DOM is a JavaScript object that tells React how to construct UI in the browser.

(New Virtual DOM Content here)

HTML is simply a set of instructions that a browser will follow when constructing the Document Object Model, or DOM. Let's say that you have to construct an HTML hierarchy for a recipe. A possible solution for such a task may look something like this:

Example 4-2. Recipe HTML

```
<section id="baked-salmon">
  <h1>Baked Salmon</h1>
  <ul class="ingredients">
    <li>1 lb Salmon</li>
    <li>1 cup Pine Nuts</li>
    <li>2 cups Butter Lettuce</li>
    <li>1 Yellow Squash</li>
    <li>1/2 cup Olive Oil</li>
    <li>3 cloves of Garlic</li>
  </ul>
  <section class="instructions">
    <h2>Cooking Instructions</h2>
    <p>Preheat the oven to 350 degrees.</p>
    <p>Spread the olive oil around a glass baking dish.</p>
    <p>Add the salmon, Garlic, and pine nuts to the dish</p>
    <p>Bake for 15 minutes.</p>
    <p>Add the Butternut Squash and put back in the oven for 30 mins.</p>
    <p>Remove from oven and let cool for 15 minutes.
      Add the lettuce and serve.</p>
  </section>
</section>
```

In HTML, elements relate to each other in a hierarchy that resembles a family tree. We could say that the root element has three children, an `<h1>`, an unordered list of ingredients, and a section with a class of ingredients. Constructing this hierarchy of HTML elements has always been a pretty common task for a web developer. With the introduction of AJAX and single page applications, developers must rely on the DOM API to make changes to the UI.



DOM API

The API that JavaScript uses to interact with a browser DOM. If you have used `document.createElement` or `document.appendChild`, you have worked with the DOM API.

With React, we do not interact with the DOM API directly. We define a Virtual DOM using React elements instead of HTML elements. The Virtual DOM provides a description of what the DOM should look like. The Virtual DOM also describes how data is used to construct the browser DOM. React will use these instructions and interact with the DOM API as efficiently as possible.

React Elements

The browser DOM is made up of DOM elements. Similarly, the React DOM is made up of React elements. DOM elements and React elements may look the same, but

they are actually quite different. A React element is a description of what the actual DOM should look like. In other words, React elements are the instructions for how the browser DOM should be created.

We can create a React Element to represent an `<h1>` using `React.createElement`.

```
React.createElement("h1", null, "Baked Salmon")
```

The first argument defines the type of element that we wish to create. In this case, we want to create a heading one element. The third argument represents the element's children, any nodes that are inserted between the opening and closing tag. The second argument represents that element's properties. This heading one currently does not have any properties.

When React is rendered, React will convert this element to an actual DOM element.

```
<h1>Baked Salmon</h1>
```

When an element has attributes, they can be described with properties. Here is a sample of an HTML heading one tag that has an `id` and `data-type` attribute.

```
React.createElement("h1",  
  {id: "recipe-0", 'data-type': "title"},  
  "Baked Salmon"  
)
```

```
<h1 data-reactroot id="recipe-0" data-type="title">Baked Salmon</h1>
```

The properties are similarly applied to the new DOM element. The properties are added to the tag as attributes. The child text is added as element text. You'll also notice `data-reactroot`, which identifies that this is the root element of your React component.

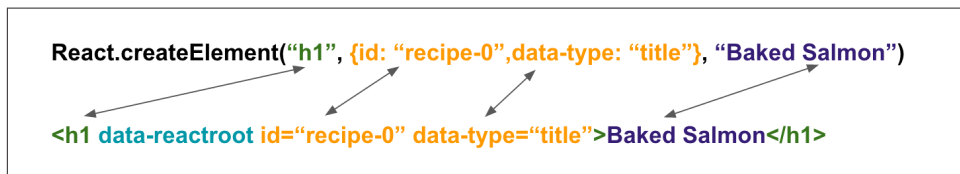


Figure 4-1. Relationship between `createElement` and DOM element



data-reactroot

`data-reactroot` will always appear as an attribute of the root element of your React component. Prior to version 15, React `id`'s were added to each node that was a part of your component. This helped with rendering and keeping track of which elements needed to be updated. Now, there is only an attribute added to the root, and rendering is kept track of based on the hierarchy of elements.

What is a React element? It's just a JavaScript literal that tells React how to construct the DOM element. This is the element that the `createElement` actually creates.

Example 4-3. Logging the Title Element

```
{
  $$typeof: Symbol(React.element),
  "type": "h1",
  "key": null,
  "ref": null,
  "props": {"children": "Baked Salmon"},
  "_owner": null,
  "_store": {}
}
```

This is a React element. There are fields that are used by React: `_owner`, `_store`, `$$typeof`. The `key` and `ref` are important to React elements, but we'll introduce those later in Chapter 5. Let's take a closer look at the `type` and `props` fields.

Example 4-4. Type & Props Fields

```
{
  ...
  "type": "h1",
  "props": {"children": "Baked Salmon"},
}
```

The `type` property of the React element tells React what type of HTML or SVG element to create. The `props` property represents the data and child elements required to construct a DOM. The `children` property is for other nested elements as text.



A Note on Creating Elements

We are taking a peek at the object that `React.createElement` returns. There is never a case where you would create elements by hand typing literals that look like this. You must always create React elements with the `React.createElement` function or factories, which are discussed at the end of this chapter.

ReactDOM

ReactDOM contains the tools necessary to render React elements in the browser. ReactDOM is where we will find the `render` method as well as the `renderToString` or `renderToStaticMarkup` methods that are used on the server. These will be discussed in greater detail in Chapter 12. The tools necessary to generate HTML from the Virtual DOM are found in this library.

We can render a React element, including its children, to the DOM with ReactDOM.render. The element that we wish to render is passed as the first argument and the second argument is the target node, where we should render the element.

```
var dish = React.createElement("h1", null, "Baked Salmon");

ReactDOM.render(dish, document.getElementById('react-container'));
```

Rendering the title element to the DOM would add a heading one element to the div with the id of react-container, which we would already have defined in our HTML. Below, we build this div inside the body tag.

Example 4-5. React added the h1 element to the target: react-container

```
<body>
  <div id="react-container">
    <h1>Baked Salmon</h1>
  </div>
</body>
```

All of the DOM rendering functionality in React has been moved to ReactDOM because we can use React to build native applications as well. The browser is one target for React.

That's all you need to do. You create an element, and then you render it on the DOM. In the next section, we'll get an understanding of how to use props.children.

Children

The ReactDOM allows you to render a single element to the DOM². React tags this as data-reactroot. All other React elements are composed into a single element using nesting.

React renders child elements using props.children. We rendered a text element as a child of the h1 element, and thus props.children was set to “Baked Salmon”. Additionally, we could render other React elements as children which creates a tree of elements. This is why we use the term component trees. The tree has one root component from which many branches grow.

Let's consider the unordered list that contains ingredients:

² <https://facebook.github.io/react/docs/displaying-data.html#components-are-just-like-functions>

Example 4-6. Ingredients List

```
<ul>
  <li>1 lb Salmon</li>
  <li>1 cup Pine Nuts</li>
  <li>2 cups Butter Lettuce</li>
  <li>1 Yellow Squash</li>
  <li>1/2 cup Olive Oil</li>
  <li>3 cloves of Garlic</li>
</ul>
```

In this sample, the unordered list is the root element, and it has 6 children. We can represent this ul and its children with `React.createElement`.

Example 4-7. Unordered List as React Elements

```
React.createElement(
  "ul",
  null,
  React.createElement("li", null, "1 lb Salmon"),
  React.createElement("li", null, "1 cup Pine Nuts"),
  React.createElement("li", null, "2 cups Butter Lettuce"),
  React.createElement("li", null, "1 Yellow Squash"),
  React.createElement("li", null, "1/2 cup Olive Oil"),
  React.createElement("li", null, "3 cloves of Garlic")
)
```

Every additional argument sent to the `createElement` function is another child element. React creates an array of these child elements and sets the value of `props.children` to that array.

If we were to inspect the resulting React element we would see each list item represented by a React element and added to an array called `props.children`. Let's take a look at the element that this created

Example 4-8. Resulting React element

```
{
  "type": "ul",
  "props": {
    "children": [
      { "type": "li", "props": { "children": "1 lb Salmon" } ... },
      { "type": "li", "props": { "children": "1 cup Pine Nuts" } ... },
      { "type": "li", "props": { "children": "2 cups Butter Lettuce" } ... },
      { "type": "li", "props": { "children": "1 Yellow Squash" } ... },
      { "type": "li", "props": { "children": "1/2 cup Olive Oil" } ... },
      { "type": "li", "props": { "children": "3 cloves of Garlic" } ... }
    ]
  }
  ...
}
```

```

    }
  }
}

```

We can now see that each list item is a child. Earlier in this chapter, we introduced HTML for an entire recipe rooted in a section element. If we want to create this using React, we'll use a series of `createElement` calls.

Example 4-9. React Element Tree

```

React.createElement("section", {id: "baked-salmon"},
  React.createElement("h1", null, "Baked Salmon"),
  React.createElement("ul", {"className": "ingredients"},
    React.createElement("li", null, "1 lb Salmon"),
    React.createElement("li", null, "1 cup Pine Nuts"),
    React.createElement("li", null, "2 cups Butter Lettuce"),
    React.createElement("li", null, "1 Yellow Squash"),
    React.createElement("li", null, "1/2 cup Olive Oil"),
    React.createElement("li", null, "3 cloves of Garlic")
  ),
  React.createElement("section", {"className": "instructions"},
    React.createElement("h2", null, "Cooking Instructions"),
    React.createElement("p", null, "Preheat the oven to 350 degrees."),
    React.createElement("p", null, "Spread the olive oil around a glass baking dish."),
    React.createElement("p", null, "Add the salmon, Garlic, and pine..."),
    React.createElement("p", null, "Bake for 15 minutes."),
    React.createElement("p", null, "Add the Butternut Squash and put..."),
    React.createElement("p", null, "Remove from oven and let cool for 15 ....")
  )
)

```



class Name in React

Notice that any element that has an HTML class attribute is using `className` for that property instead of `class`. Since `class` is a reserved word in JavaScript we have to use `className` to define the class attribute of an HTML element.

This sample is what pure React looks like. Pure React is ultimately what runs in the browser. The Virtual DOM is a tree of React elements all stemming from a single root element. React elements are the instructions that React will use to build a UI in the browser.

Constructing Elements with Data

The major advantage of using React is its ability to separate data from UI elements. Since React is just JavaScript, we can add JavaScript logic to help us build the React component tree. For example, ingredients can be stored in an array, and we can map

that array to the React elements.

Let's go back and think about this unordered list for a moment:

Example 4-10. Unordered List

```
React.createElement("ul", { "className": "ingredients"},
  React.createElement("li", null, "1 lb Salmon"),
  React.createElement("li", null, "1 cup Pine Nuts"),
  React.createElement("li", null, "2 cups Butter Lettuce"),
  React.createElement("li", null, "1 Yellow Squash"),
  React.createElement("li", null, "1/2 cup Olive Oil"),
  React.createElement("li", null, "3 cloves of Garlic")
);
```

The data used in this list of ingredients can be easily represented using a JavaScript array.

Example 4-11. Items Array

```
var items = [
  "1 lb Salmon",
  "1 cup Pine Nuts",
  "2 cups Butter Lettuce",
  "1 Yellow Squash",
  "1/2 cup Olive Oil",
  "3 cloves of Garlic"
]
```

We could construct a virtual DOM around this data using the array's map function.

Example 4-12. Mapping an array to elements

```
React.createElement(
  "ul",
  { className: "ingredients" },
  items.map(ingredient =>
    React.createElement("li", null, ingredient)
  )
)
```

This syntax creates a React element for each ingredient in the array. Each string is displayed in the list item's children as text. The value for each ingredient is displayed as the list item.



Console Warning

When running this code, you'll see a console error.

✖ ▶ Warning: Each child in an array or iterator should have a unique "key" prop. Check the top-level render call using . See <https://fb.me/react-warning-keys> for more information. runner-3.36.10.min.js:1

When we build a list of child elements by iterating through an array, React likes each of those elements to have a key property. The key property is used by React to help it update the DOM efficiently. We will be discussing keys and why we need them in Chapter 5, but for now you can make this warning go away by adding a unique key property to each of the list item elements. We can use the array index for each ingredient as that unique value. Adding this key property will cause the console warning to go away.

Example 4-13. Adding a Key Property

```
React.createElement("ul", {className: "ingredients"},
  items.map((ingredient, i) =>
    React.createElement("li", { key: i }, ingredient)
  )
)
```

React Components

Every user interface is made up of parts. This recipe example has a few recipes. Each will be made up of recipes, and each recipe will be made up of parts. Components allow us to use the same DOM structure.

<div><h3>Chicken Noodle Soup</h3><ul style="list-style-type: none">• 2 tablespoons extra-virgin olive oil• 1 yellow onion• 2 medium carrots• 2 stalks celery• 8 cups chicken broth• 1 16 oz package wide egg noodles• 1 cup cooked chicken<p>Chop and sautee onion, carrots, and celery in olive oil until soft. Add chicken broth and bring to boil. Add egg noodles and cook until soft. Add chicken and simmer.</p></div>	<div><h3>Curried Egg Salad</h3><ul style="list-style-type: none">• 12 hard boiled eggs• 1/2 cup mayonnaise• 1/4 cup whole grain mustard• 1 Tablespoon curry powder• 1 teaspoon garlic powder• 1/4 cup finely chopped onion<p>Chop hard boiled eggs. Combine with other ingredients. Chill for at least 30 minutes before serving.</p></div>	<div><h3>Chocolate Chip Cookies</h3><ul style="list-style-type: none">• 12 hard boiled eggs• 1/2 cup mayonnaise• 1/4 cup whole grain mustard• 1 Tablespoon curry powder• 1 teaspoon garlic powder• 1/4 cup finely chopped onion<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam.</p></div>
--	--	---

Figure 4-2. Recipes App

In React, we describe each of these parts as a component. Components allow us to reuse the same DOM structure for different recipes or different sets of data.

When considering a user interface that you want to build with React, look for opportunities to break down your elements into reusable pieces. For example, the recipes below have a title, ingredients list, and instructions. All are part of a larger recipe or app component. We could create a component for each of the highlighted parts: an ingredient, instructions, etc.

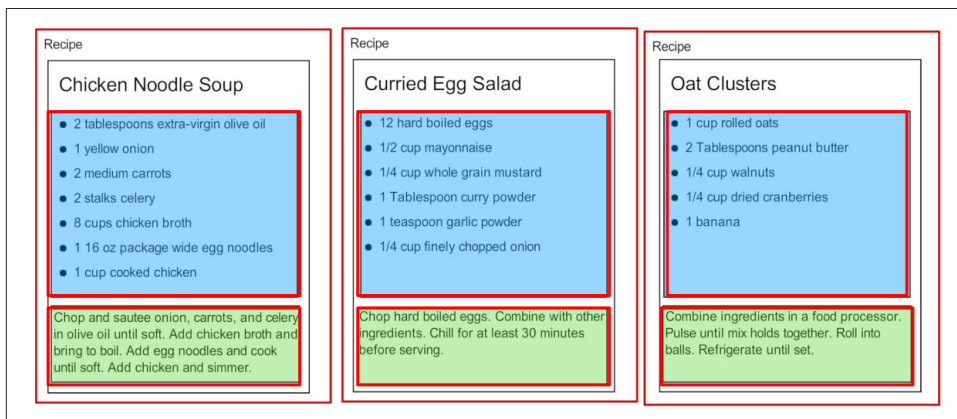


Figure 4-3. Each component is outlined: App, IngredientsList, Instructions.

Think about how scalable this is. If we want to display one recipe, our component structure will support this. If we want to display 10,000 recipes, we'll just create new instances of that component.

Let's investigate the three different ways to create components: `createClass()`, ES6 classes, and stateless functional components.

React.createClass()

When React was first introduced in 2013, there was only one way to create a component: the `createClass` function.

Just like another hit of 2013, "Get Lucky" by Daft Punk, `React.createClass` remains solid. Even though new methods of creating components have emerged, `createClass` is still a very valid way to create components.

Let's consider the list of ingredients that are included in each recipe. We can create a React component using `React.createClass` that returns a single unordered list element that contains a child list item for each ingredient in an array.

Components allow us to use data to build reusable UI. In the render function, we

can use the “this” keyword to refer to the component instance, and properties can be accessed on that instance with this.props .

Example 4-14. Ingredients List as a React Component

```
const IngredientsList = React.createClass({
  displayName: "IngredientsList",
  render() {
    return React.createElement("ul", {className: "ingredients"},
      React.createElement("li", null, "1 lb Salmon"),
      React.createElement("li", null, "1 cup Pine Nuts"),
      React.createElement("li", null, "2 cups Butter Lettuce"),
      React.createElement("li", null, "1 Yellow Squash"),
      React.createElement("li", null, "1/2 cup Olive Oil"),
      React.createElement("li", null, "3 cloves of Garlic")
    )
  }
})

const list = React.createElement(IngredientsList, null, null)

ReactDOM.render(
  list,
  document.getElementById('react-container')
)
```

Explain: Look at the React DOM in the react tools panel. We have created an element using our component and named it ingredients list.

```
<IngredientsList>
  <ul className="ingredients">
    <li>1 lb Salmon</li>
    <li>1 cup Pine Nuts</li>
    <li>2 cups Butter Lettuce</li>
    <li>1 Yellow Squash</li>
    <li>1/2 cup Olive Oil</li>
    <li>3 cloves of Garlic</li>
  </ul>
</IngredientsList>
```

Explain: Data can be passed to React Components as properties. We can create a reusable list of ingredients by passing that data to the list as an array.

```
const IngredientsList = React.createClass({
  displayName: "IngredientsList",
  render() {
    return React.createElement("ul", {className: "ingredients"},
      this.props.items.map((ingredient, i) =>
        React.createElement("li", { key: i }, ingredient)
      )
    )
  }
})
```

```

    }
  })

  const items = [
    "1 lb Salmon",
    "1 cup Pine Nuts",
    "2 cups Butter Lettuce",
    "1 Yellow Squash",
    "1/2 cup Olive Oil",
    "3 cloves of Garlic"
  ]

  ReactDOM.render(
    React.createElement(IngredientsList, {items}, null),
    document.getElementById('react-container')
  )

```

Explain: Now look at React DOM. The data property items is an array with 6 ingredients. Because we made the tags using a loop we added a unique key using the index of the loop.

```

<IngredientsList items={...}>
  <ul className="ingredients">
    <li key="0">1 lb Salmon</li>
    <li key="1">1 cup Pine Nuts</li>
    <li key="2">2 cups Butter Lettuce</li>
    <li key="3">1 Yellow Squash</li>
    <li key="4">1/2 cup Olive Oil</li>
    <li key="5">3 cloves of Garlic</li>
  </ul>
</IngredientsList>

```

Explain: Components are object. They can be used to encapsulate code just like classes. We can create a method that renders a single list item and use that to build out list.

Example 4-15. With a Custom Method

```

const IngredientsList = React.createClass({
  displayName: "IngredientsList",
  renderListItem(ingredient, i) {
    return React.createElement("li", { key: i }, ingredient)
  },
  render() {
    return React.createElement("ul", {className: "ingredients"},
      this.props.items.map(this.renderListItem)
    )
  }
})

```

This is also the idea of views in MVC languages. Everything that is associated with the UI for `IngredientsList` is encapsulated into one component. Everything we need right there.

Now we can create a React element using our component and passing it to the list of elements as a property. Notice that the element's type is now a string - it's the component class directly.



Component Classes as Types

When rendering HTML or SVG elements, we use strings. When creating elements with components we use the component class directly. This is why `IngredientsList` is not surrounded in quotation marks, we are passing the class to `createElement` because it is a component. React will create an instance of our component with this class and manage it for us.

Using the `IngredientsList` component with this data would render the following unordered list to the DOM.

```
<ul data-react-root class="ingredients">
  <li>1 lb Salmon</li>
  <li>1 cup Pine Nuts</li>
  <li>2 cups Butter Lettuce</li>
  <li>1 Yellow Squash</li>
  <li>1/2 cup Olive Oil</li>
  <li>3 cloves of Garlic</li>
</ul>
```

React.Component

As discussed in Chapter 2, one of the key features included in the ES6 spec was class syntax. `React.Component` is an abstract class that we can use to build new React components. We can create custom components through inheritance by extending this class with ES6 syntax. We can create `IngredientsList` using the same syntax.

Example 4-16. *IngredientsList* as ES6 Class

```
class IngredientsList extends React.Component {

  renderListItem(ingredient, i) {
    return React.createElement("li", { key: i }, ingredient)
  }

  render() {
    return React.createElement("ul", {className: "ingredients"},
      this.props.items.map(this.renderListItem)
    )
  }
}
```



```
}  
  
}
```

Stateless Functional Components

Stateless functional components are functions not objects, therefore, they do not have a “this” scope. Because they are simple, pure functions, we’ll use them as much as possible in our applications. There will come a point where the stateless functional component isn’t robust enough and we might fall back to using class or `createClass`. Other than that, the more you can use these, the better.

Stateless functional components are functions that take in properties and return a DOM. Stateless functional components are a good way to practice the rules of functional programming. You should strive to make each stateless functional component a pure function. They should take in props and return a DOM without causing side effects. This encourages simplicity and makes the code base extremely testable.

If you need to encapsulate functionality or have a “this” scope, you can’t use stateless functional components. They will keep your application architecture simple, and the React team promises some performance gains by using them.

Below, we combine the functionality of `renderListItem` and `render` into this single function.

Example 4-17. Creating a Stateless Functional Component

```
const IngredientsList = props =>  
  React.createElement("ul", {className: "ingredients"},  
    props.items.map((ingredient, i) =>  
      React.createElement("li", { key: i }, ingredient)  
    )  
  )
```

We would render this component with `ReactDOM.render` the exact same way we render components created with `createClass` or ES6 class syntax. This is just a function. The function collects data through the props arguments and returns an unordered list for each item that is sent to the props data.

One way we can improve this stateless functional component is through destructuring the properties argument. Using ES6 destructuring syntax, we can scope the list property directly to this function, reducing the repetitive dot syntax. Now we’d use the ingredients list the same way we render component classes.

Example 4-18. Destructuring Properties Argument

```
const IngredientsList = ({items}) =>  
  React.createElement("ul", {className: "ingredients"},  
    items.map((ingredient, i) =>  
      React.createElement("li", { key: i }, ingredient)  
    )  
  )  
)
```



Const with Stateless Functional Components

Each of these stateless functional components uses `const` instead of `var` when creating a component. This is a common practice but not a requirement. `Const` declares this function as a constant and prevents us from redefining that variable later.

Aside from being slightly cleaner syntax, Facebook has hinted that in the future, stateless functional components might be faster than `createClass` or ES6 class syntax.

DOM Rendering

Since we are able to pass data to our components as props, we are able to separate our application's data from the logic that is used to create the UI. This gives us an isolated set of data that is much easier to work with and manipulate than the document object model. When we change any of the values in this isolated in our data set, we change the state of our application.

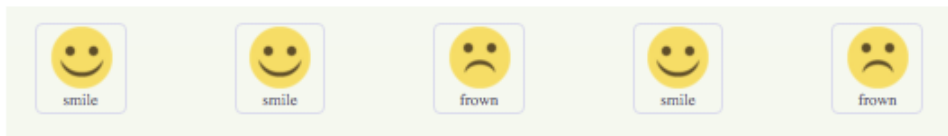
Imagine storing all of the data in your application in a single JavaScript object. Every time we make a change to this object, we could send it to a component as props and re-render the UI. This means that `ReactDOM.render` is going to be doing a lot of heavy lifting.

In order for React to work in a reasonable amount of time, `ReactDOM.render` has to work smart, and it does. Instead of emptying and reconstructing the entire DOM, `ReactDOM.render` leaves the current DOM in place and only applies the minimal amount of changes required to mutate the DOM.

Let's say we had an app that displayed the mood of 5 of our five team members using either a smiley face or a frowny face. We can represent the mood of all five individuals in a single JavaScript array:

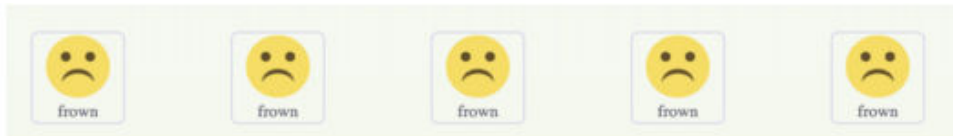
```
["smile", "smile", "frown", "smile", "frown"];
```

This array of data may be used to construct a UI that looks something like:



When a build breaks and the team has to work all weekend, we can reflect the team's new mood simply by changing the data in this array.

```
[“frown”, “frown”, “frown”, “frown”, “frown”];
```



How many changes do we have to make to the first array to make it look like the second array of all frowns?

```
[“smile”, “smile”, “frown”, “smile”, “frown”];
```

```
[“frown”, “frown”, “frown”, “frown”, “frown”];
```

We would need to change the first, second, and fourth values from a smile to a frown.

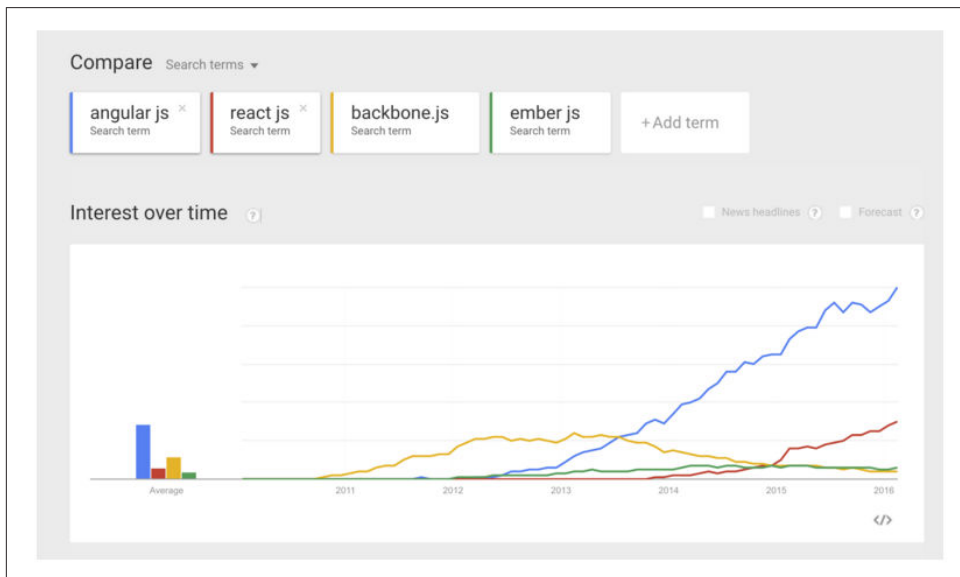


Figure 4-4. This is an image caption

Therefore, we can say that it would take 3 mutations to change the first array of data to match the second.

Now consider how we can update the DOM to reflect these changes. One inefficient solution to applying these changes to the UI is to erase the entire DOM and rebuild it.

Example 4-19. Start with the Current List

```
<ul>
  <li class="smile">smile</li>
  <li class="smile">smile</li>
  <li class="frown">frown</li>
  <li class="smile">smile</li>
  <li class="frown">frown</li>
</ul>
```

Step 1: Empty the current data

```
<ul>
</ul>
```

Step 2: Begin looping through data and build the first list item.

```
<ul>
  <li class="frown">frown</li>
</ul>
```

Step 3: Build and add the second list item.

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

Step 4: Build and append the third list item.

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

Step 5: Build and append the fourth list item.

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

Step 6: Build and append the fifth list item.

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
```

```
<li class="frown">frown</li>
<li class="frown">frown</li>
<li class="frown">frown</li>
</ul>
```

If we change the UI by erasing and rebuilding the DOM, we are creating and inserting 5 new DOM elements. Inserting an element into the DOM is one of the most costly DOM API operations - it's slow. In contrast, updating DOM elements that are already in place performs much faster than inserting new ones.

The way ReactDOM.render makes changes is by leaving the current DOM in place and simply updating the DOM elements that need to be updated. In our example, there are only 3 mutations, so ReactDOM.render only needs to update 3 DOM elements.

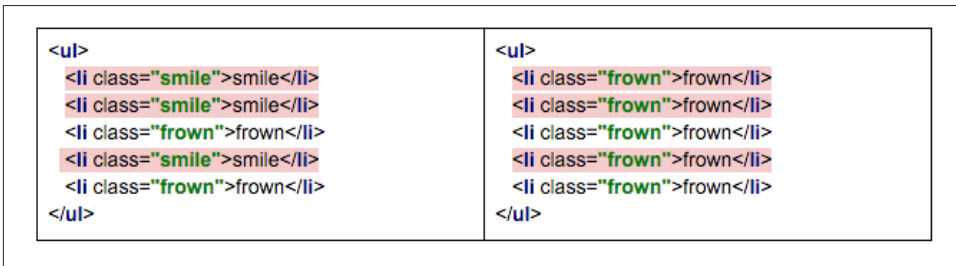


Figure 4-5. Three DOM elements are updated.

If new DOM elements need to be inserted, ReactDOM will insert them, but it tries to keep DOM insertion, the most costly operation, to a minimum.

This smart DOM rendering is necessary for React to work in a reasonable amount of time because our application state changes a lot, and every time we change that state we are going to rely on ReactDOM.render to efficiently re-render the UI.

Factories

So far the only way we have created elements has been with React.createElement. Another way to create a React Element is to use factories. A factory is a special object that can be used to abstract away the details of instantiating objects. In React, we use factories to help us create React element instances.

React has built-in factories for all commonly supported HTML and SVG DOM elements, and you can use the React.createFactory function to build your own factories around specific components.

Why would we use a factory? Factories are an alternative to using createElement calls. For example, we can use HTML element factories that come with React to create our heading one element listed earlier in this chapter.

```
<h1>Baked Salmon</h1>
```

Instead of using `createElement`, we can create a React element with a factory.

Example 4-20. Using `createFactory` to Create an `h1`

```
React.DOM.h1(null, "Baked Salmon")
```

In this case, the first argument is for the properties and the second argument is for the children. We can also use a DOM factory to build an unordered list.

Example 4-21. Building an Unordered List with DOM Factories

```
React.DOM.ul({ "className": "ingredients" },
  React.DOM.li(null, "1 lb Salmon"),
  React.DOM.li(null, "1 cup Pine Nuts"),
  React.DOM.li(null, "2 cups Butter Lettuce"),
  React.DOM.li(null, "1 Yellow Squash"),
  React.DOM.li(null, "1/2 cup Olive Oil"),
  React.DOM.li(null, "3 cloves of Garlic")
)
```

In this case the first argument is for the properties, where we define the `className`. Every additional argument are elements that will be added to the children array of the unordered list. We can also separate out the ingredient data and improve the above definition using factories.

Example 4-22. Using `.map()` with Factories

```
var items = [
  "1 lb pizza",
  "1 cup Pine Nuts",
  "2 cups Butter Lettuce",
  "1 Yellow Squash",
  "1/2 cup Olive Oil",
  "3 cloves of Garlic"
]

var list = React.DOM.ul(
  { className: "ingredients" },
  items.map((ingredient, key) =>
    React.DOM.li({key}, ingredient)
  )
)

ReactDOM.render(
  list,
  document.getElementById('react-container')
)
```

Using Factories with Components

If you would like to simplify your code by calling components as functions, you need to explicitly create a factory.

Example 4-23. Creating a Factory with IngredientsList and App

```
const { render } = ReactDOM;

const IngredientsList = ({ list }) =>
  React.createElement('ul', null,
    list.map((ingredient, i) =>
      React.createElement('li', {key: i}, ingredient)
    )
  )

const Ingredients = React.createFactory(IngredientsList)

const list = [
  "1 lb pizza",
  "1 cup Pine Nuts",
  "2 cups Butter Lettuce",
  "1 Yellow Squash",
  "1/2 cup Olive Oil",
  "3 cloves of Garlic"
]

render(
  Ingredients({list}),
  document.getElementById('react-container')
)
```

In this example, we can quickly render a React element with the Ingredients factory. Ingredients is a function that takes in properties and children as arguments just like DOM factories.

If you are not working with JSX, you may find factories preferable to numerous React.createElement calls. However, the easiest and most common way to define React elements is with JSX tags. If you use JSX with React, chances are you will never use a factory.

Throughout this chapter, we've used createElement and createFactory to build React components. In chapter 5, we'll take a look at how to simplify component creation by using JSX.

React with JSX

In the last chapter, we looked at how the Virtual DOM is a set of instructions that React will follow when creating and updating a user interface. These instructions are made up of JavaScript objects called React elements. So far, we've learned two ways to create React elements: using `React.createElement` and using factories. In this chapter, we are going to discuss how you can use JSX to construct a virtual DOM with React elements.

The alternative to typing out verbose `React.createElement` calls is JSX, a JavaScript extension that allows us to define React elements using syntax that looks similar to HTML.

React Elements as JSX

Facebook's React team released JSX when they released React to provide a concise syntax for creating complex DOM trees with attributes. They also hoped to make React more readable like HTML and XML.

In JSX, an element's type is specified with a tag. The tag's attributes represent the properties. The element's children can be added between the opening and closing tags.

You can also add other JSX elements as children. If you have an unordered list, you can add child list item elements to it with JSX tags. It looks very similar to HTML.

Example 5-1. JSX for Unordered list

```
<ul>
  <li>1 lb Salmon</li>
  <li>1 cup Pine Nuts</li>
  <li>2 cups Butter Lettuce</li>
```

```

<li>1 Yellow Squash</li>
<li>1/2 cup Olive Oil</li>
<li>3 cloves of Garlic</li>
</ul>

```

JSX works with components as well. Simply define the component using the class name. In this example, we pass an array of ingredients to the `IngredientsList` as a property with JSX.

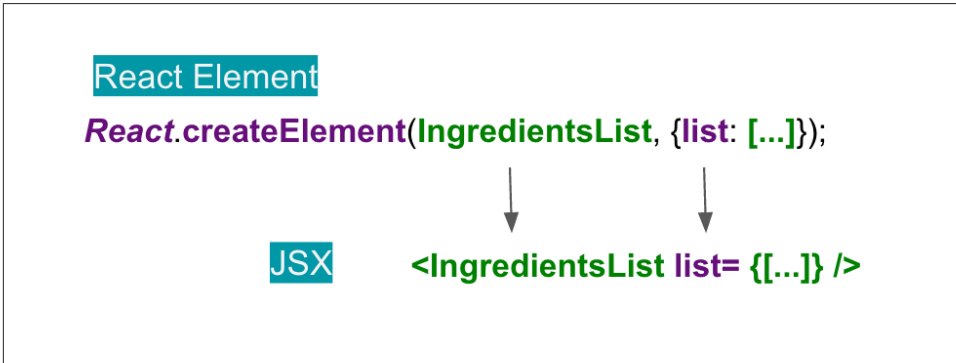


Figure 5-1.
Creating the `IngredientsList` with JSX

When we pass the array of ingredients to this component, we need to surround it with curly brackets. This is called a JavaScript expression and we must use these when passing JavaScript values to components as properties. Component properties will take two types: either a string or a JavaScript expression. JavaScript expressions can include arrays, objects, even functions - any JavaScript type, and in order to include them, you must surround them in curly brackets.

JSX Tips

JSX might look familiar, and most of the rules result in syntax that is similar to HTML. However, there are a few quirks that you should understand when working with JSX.

Nested Components

JSX allows you to add components as children of other components. Inside of the `IngredientsList`, we can render another component called `Ingredient` multiple times.

Example 5-2. `IngredientsList` with Three Nested `Ingredient` Components

```

<IngredientsList>
  <Ingredient />
  <Ingredient />

```

```
<Ingredient />
</IngredientsList>
```

className

Since `class` is a reserved word in JavaScript, `className` is used to define the class attribute instead.

```
<h1 className="fancy">Baked Salmon</h1>
```

JavaScript Expressions

JavaScript expressions are wrapped in curly braces and indicate where variables shall be evaluated and their resulting values returned. For example, if we want to display the value of the `title` property in an element, we can insert that value using a JavaScript expression. The variable will be evaluated, and its value is returned.

```
<h1>{this.props.title}</h1>
```

Values of types other than strings should also appear as a JavaScript expression.

```
<input type="checkbox" defaultChecked={false} />
```

Evaluation

The JavaScript that is added in between the curly brackets will get evaluated. This means that operations such as concatenation or addition will occur. This also means that functions found in JavaScript expressions will be invoked.

```
<h1>["Hello " + this.props.title]</h1>

<h1>{this.props.title.toLowerCase().replace}</h1>

function appendTitle({this.props.title}) {
  console.log(`${this.props.title} is great!`)
}
```

Mapping Arrays to JSX

JSX is JavaScript, so you can incorporate JSX directly inside of JavaScript functions. For example, you can map an array to JSX elements.

Example 5-3. `.map()` with JSX

```
<ul>
  {this.props.ingredients.map((ingredient, i) =>
    <li key={i}>{ingredient}</li>
  )}
</ul>
```

JSX looks clean and readable, but it can't be interpreted with a browser. All JSX must be converted into `createElement` calls or factories. Luckily, there is an excellent tool for this task: Babel.

Babel

Most software languages allow you to compile your source code. JavaScript is an interpreted language, the browser interprets the code as text so there is no need to compile JavaScript. However, browsers do not yet support the latest ES6 and ES7 syntax, and no browser supports JSX syntax. Since we want to use the latest features of JavaScript along with JSX, we are going to need a way to convert our fancy source code into something that the browser can interpret. This process is called transpiling, and it is what **Babel** is designed to do.

Sebastian McKenzie was in high school in Australia when he first created Babel. The first version of the project was called 6to5, and it was released in September 2014. 6to5 was a tool that could be used to convert ES6 syntax to ES5 syntax, which is more widely supported by the browser. As the project grew, it aimed to be a platform to support all of the latest changes in ECMAScript. It also grew to support transpiling JSX into pure React. The project was more appropriately renamed to Babel in February of 2015.

Babel is used in production at Facebook, Netflix, PayPal, Airbnb and more. Previously, Facebook had created a JSX transformer that was their standard, but soon retired that in favor of Babel.

There are many ways of working with Babel. The easiest way to get started is to include a link to the babel-core transpiler directly in your HTML which will transpile any code in script blocks that have a type of “text/babel”. Babel will transpile the source code on the client before running it. Although this may not be the best solution for production, it is a great way to get started with JSX.

Example 5-4. Including babel-core

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>React Examples</title>
  </head>
  <body>
    <div class="react-container"></div>

    <script src="//fb.me/react-15.1.0.js"></script>
    <script src="//fb.me/react-dom-15.1.0.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.29/browser.js"></script>

    <script type="text/babel">

      // JSX code here. Or link to separate JavaScript file that contains JSX.
```

```
</script>

</body>
</html>
```



Babel v5.8 required

To transpile code in the browser, use Babel v. 5.8. Using Babel 6.0+ will not work as an in-browser transformer.

Later in the chapter, we'll look at how we can incorporate Babel to transpile our JavaScript files statically. For now, using the in browser transpiler will do.

Recipes as JSX

One of the reasons that we have grown to love React is that allows us to express web applications with beautiful code. It is extremely rewarding to create beautifully written modules that clearly communicate how the application functions. JSX provides us with a nice clean way to express React elements in our code that makes sense to us and is immediately readable by the engineers that make up our community. The drawback of JSX is that it is not readable by the browser. Before our code can be interpreted by the browser, it needs to be converted from JSX into pure React.

The following array contains two recipes, and they represent our application's current state.

Example 5-5. Sample data : Array of Recipes

```
var data = [
  {
    "name": "Baked Salmon",
    "ingredients": [
      { "name": "Salmon", "amount": 1, "measurement": "l lb" },
      { "name": "Pine Nuts", "amount": 1, "measurement": "cup" },
      { "name": "Butter Lettuce", "amount": 2, "measurement": "cups" },
      { "name": "Yellow Squash", "amount": 1, "measurement": "med" },
      { "name": "Olive Oil", "amount": 0.5, "measurement": "cup" },
      { "name": "Garlic", "amount": 3, "measurement": "cloves" }
    ],
    "steps": [
      "Preheat the oven to 350 degrees.",
      "Spread the olive oil around a glass baking dish.",
      "Add the salmon, Garlic, and pine nuts to the dish",
      "Bake for 15 minutes.",
      "Add the Butternut Squash and put back in the oven for 30 mins.",
      "Remove from oven and let cool for 15 minutes. Add the lettuce and serve."
    ]
  }
]
```

```

    },
    {
      "name": "Fish Tacos",
      "ingredients": [
        { "name": "Whitefish", "amount": 1, "measurement": "l lb" },
        { "name": "cheese", "amount": 1, "measurement": "cup" },
        { "name": "Iceberg Lettuce", "amount": 2, "measurement": "cups" },
        { "name": "Tomatoes", "amount": 2, "measurement": "large" },
        { "name": "Tortillas", "amount": 3, "measurement": "med" }
      ],
      "steps": [
        "Cook the Fish on the Grill until Hot",
        "Place the fish on the 3 tortillas",
        "Top them with lettuce, tomatoes, and cheese"
      ]
    }
  ];

```

The data is expressed in an array of two JavaScript objects. Each object contains the name of the recipe, a list of the ingredients required, and a list of steps necessary to cook each recipe.

Example 5-6. Recipe App code structure

```

// The Data, an array of recipes objects
var data = [ ... ];

// A stateless functional component for an individual Recipe
const Recipe = (props) => (
  ...
);

// A stateless functional component for the Menu of recipes
const Menu = (props) => (
  ...
);

// A call to ReactDOM.render to render our Menu into the current DOM
ReactDOM.render(<Menu recipes={data} title="Delicious Recipes" />,
  document.getElementById("react-container"));

```

We can create a UI for these recipes with two components: a Menu component for listing the recipes, and a Recipe component that describes the UI for each recipe. It's the Menu component that we will render to the DOM. We will pass our data to the Menu component as a property called recipes.



ES6 support

We will be using ES6 in this file as well. When we transpile our code from JSX to pure React, Babel will also convert ES6 into common ES5 Javascript that is readable by the browser. Any ES6 features used have been discussed in Chapter 2.

Example 5-7. Menu Component Structure

```
const Menu = (props) =>
  <article>
    <header>
      <h1>{props.title}</h1>
    </header>
    <div className="recipes">
    </div>
  </article>
```

The React elements within the Menu component are expressed as JSX. Everything is contained within an article element. A `<header>` element, an `<h1>` element, and a `<div.recipes>` element are used to describe the DOM for our menu. The value for the title property will be displayed as text within the `<h1>`. Inside of the `div.recipes` element, we will need to add a component for each recipe.

Example 5-8. Mapping recipe data

```
<div className="recipes">
  {props.recipes.map((recipe, i) =>
    <Recipe key={i} name={recipe.name}
      ingredients={recipe.ingredients}
      steps={recipe.steps} />
  )}
</div>
```

In order to list the recipes within the `div.recipes` element, we are going to use curly brackets to add a JavaScript expression that will return an array of children. We can use the `map` function on the `props.recipes` array to return a component for each object within the array. Each recipe contains a name, some ingredients, and cooking instructions. We will need to pass this data to each recipe as props. Also remember, we should use the `key` property to uniquely identify each element.

Using the JSX spread operator can improve our code. The JSX spread operator works like the ES7 object spread operator discussed in Chapter 2. It will add each field of the recipe object as a property of the Recipe component. This syntax accomplishes the same results:

Example 5-9. Enhancement: JSX spread operator

```
{props.recipes.map((recipe, i) =>
  <Recipe key={i} {...recipe} />
)}
```

Another place we can make an ES6 improvement to our Menu component is where we take in the props argument. We can use object destructuring to scope the properties variables to this function. This allows us to access the title and recipes variables directly, no longer having to prefix them with props.

Example 5-10. Refactored Menu Component

```
const Menu = ({ title, recipes }) => (
  <article>
    <header>
      <h1>{title}</h1>
    </header>
    <div className="recipes">
      {recipes.map((recipe, i) =>
        <Recipe key={i} {...recipe} />
      )}
    </div>
  </article>
);
```

Now let's code the component for each individual recipe.

Example 5-11. Complete Recipe Component

```
const Recipe = ({ name, ingredients, steps }) =>
  <section id={name.toLowerCase().replace(/ /g, "-")}>
    <h1>{name}</h1>
    <ul className="ingredients">
      {ingredients.map((ingredient, i) =>
        <li key={i}>{ingredient.name}</li>
      )}
    </ul>
    <section className="instructions">
      <h2>Cooking Instructions</h2>
      {steps.map((step, i) =>
        <p key={i}>{step}</p>
      )}
    </section>
  </section>
```

This component is also a Stateless Functional Component. Each recipe has a string for the name, an array of objects for ingredients, and an array of strings for the steps. Using ES6 object destructuring, we can tell this component to locally scope those

fields by name so we can access them directly without having to use `props.name`, or `props.ingredients`, or `props.steps`.

The first Javascript expression that we see is being used to set the `id` attribute for the root section element. It is converting the recipe's name to a lower case string that globally replaces spaces with dashes. The result is that "Baked Salmon" would be converted to "baked-salmon" and likewise, if we had a recipe with the name "Boston Baked Beans" it would be converted to "boston-baked-beans" before it is used as the `id` attribute in our UI. The value for `name` is also being displayed in an `h1` as a text node.

Inside of the unordered list, a JavaScript expression is mapping each ingredient to a `li` element that displays the name of the ingredient. Within our instructions section we see the same pattern being used to return a paragraph element where each step is displayed. These map functions are returning arrays of children elements.

All of the code for the application should look like:

Example 5-12. Finished Code for Recipe App

```
const data = [
  {
    "name": "Baked Salmon",
    "ingredients": [
      { "name": "Salmon", "amount": 1, "measurement": "l lb" },
      { "name": "Pine Nuts", "amount": 1, "measurement": "cup" },
      { "name": "Butter Lettuce", "amount": 2, "measurement": "cups" },
      { "name": "Yellow Squash", "amount": 1, "measurement": "med" },
      { "name": "Olive Oil", "amount": 0.5, "measurement": "cup" },
      { "name": "Garlic", "amount": 3, "measurement": "cloves" }
    ],
    "steps": [
      "Preheat the oven to 350 degrees.",
      "Spread the olive oil around a glass baking dish.",
      "Add the salmon, Garlic, and pine nuts to the dish",
      "Bake for 15 minutes.",
      "Add the Butternut Squash and put back in the oven for 30 mins.",
      "Remove from oven and let cool for 15 minutes. Add the lettuce and serve."
    ]
  },
  {
    "name": "Fish Tacos",
    "ingredients": [
      { "name": "Whitefish", "amount": 1, "measurement": "l lb" },
      { "name": "cheese", "amount": 1, "measurement": "cup" },
      { "name": "Iceberg Lettuce", "amount": 2, "measurement": "cups" },
      { "name": "Tomatoes", "amount": 2, "measurement": "large" },
      { "name": "Tortillas", "amount": 3, "measurement": "med" }
    ],
    "steps": [
```

```

    "Cook the Fish on the Grill until Hot",
    "Place the fish on the 3 tortillas",
    "Top them with lettuce, tomatoes, and cheese"
  ]
}
]

const Recipe = ({ name, ingredients, steps }) =>
  <section id={name.toLowerCase().replace(/ /g, "-")}>
    <h1>{name}</h1>
    <ul className="ingredients">
      {ingredients.map((ingredient, i) =>
        <li key={i}>{ingredient.name}</li>
      )}
    </ul>
    <section className="instructions">
      <h2>Cooking Instructions</h2>
      {steps.map((step, i) =>
        <p key={i}>{step}</p>
      )}
    </section>
  </section>

const Menu = ({ title, recipes }) =>
  <article>
    <header>
      <h1>{title}</h1>
    </header>
    <div className="recipes">
      {recipes.map((recipe, i) =>
        <Recipe key={i} {...recipe} />
      )}
    </div>
  </article>

ReactDOM.render(
  <Menu recipes={data}
    title="Delicious Recipes" />,
  document.getElementById("react-container")
)

```

When we run this code in the browser, React will construct a UI using our instructions with the recipe data.

Delicious Recipes

Baked Salmon

- Salmon
- Pine Nuts
- Butter Lettuce
- Yellow Squash
- Olive Oil
- Garlic

Cooking Instructions

Preheat the oven to 350 degrees.

Spread the olive oil around a glass baking dish.

Add the salmon, Garlic, and pine nuts to the dish

Bake for 15 minutes.

Add the Butternut Squash and put back in the oven for 30 mins.

Remove from oven and let cool for 15 minutes. Add the lettuce and serve.

Fish Tacos

- Whitefish
- cheese
- Iceberg Lettuce
- Tomatos
- Tortillas

Cooking Instructions

Cook the Fish on the Grill until Hot

Place the fish on the 3 tortillas

Top them with lettuce, tomatos, and cheese

Figure 5-2. Delicious Recipes Resulting Output

If you are using Google Chrome, and you have the React developer tools installed, you can take a look at the present state of the virtual DOM. To do this, open the JavaScript tools and select the React tab to see the Virtual DOM.

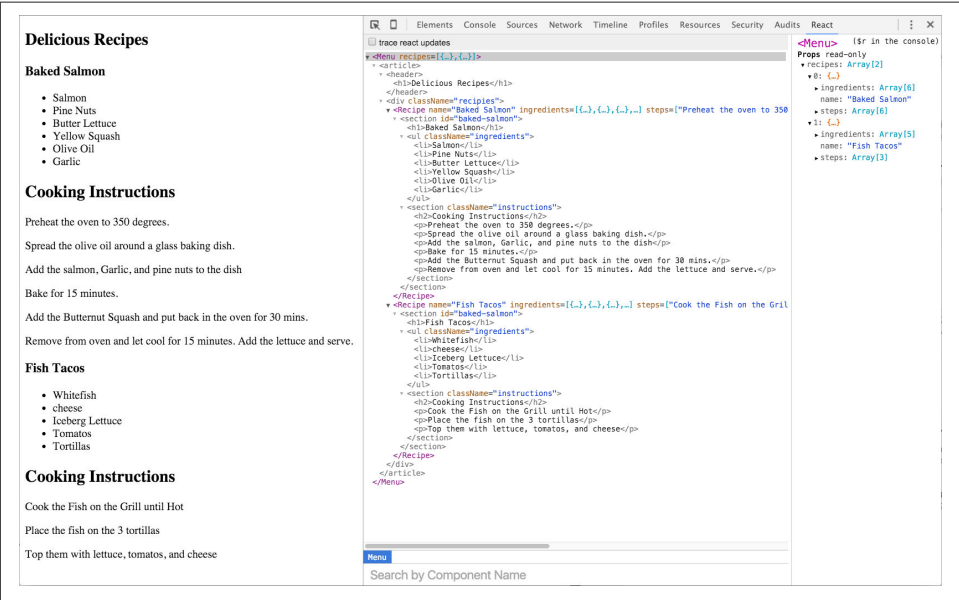


Figure 5-3. Resulting Virtual DOM in React Developer Tools

Here we can see our Menu and its child elements. The data array contains two objects for recipes, and we have two Recipe elements. Each recipe element has properties for the recipe name, ingredients, and steps.

The Virtual DOM is constructed based on the applications state data being passed to the Menu component as a property. If we change the recipes array, and re-render our Menu component, React will change this DOM as efficiently as possible.

Babel Presets

Babel 6 breaks possible transformations up into modules called presets. It requires engineers to explicitly define which transformations should be run by specifying which presets to use. The goal was to make everything more modular to allow developers to decide which syntax should be converted. The plugins fall into a few categories, and all are opt-in based on the needs of the application:

babel-preset-es2015

Compiles ES2015, or ES6, to ES5

babel-preset-react

Compiles JSX to `React.createElement` calls

Stage Presets

When a new feature is proposed for inclusion in the ECMAScript spec, it goes through stages of acceptance from Stage 0 Strawman (newly proposed and very experimental), to Stage 4, Finished (accepted as part of the standard). Babel provides presets for each of these stages, so you choose which stage you want to allow in your application.

- `babel-preset-stage-0`: Strawman
- `babel-preset-stage-1`: Proposal
- `babel-preset-stage-2`: Draft
- `babel-preset-stage-3`: Candidate

Intro to webpack

Once we start working in production with React, there are a lot of questions to consider: How do we want to deal with JSX and ES6+ transformation? How can we manage our dependencies? How can we optimize our images and CSS?

Many different tools have emerged to answer these questions including Browserify, Gulp, and Grunt. Due to its features and the widespread adoption by large companies, webpack has also emerged as one of the leading tools for bundling CommonJS modules (see Chapter 1 for more on CommonJS).

Webpack is billed as a module bundler. A module bundler takes all of our different files (JavaScript, LESS, CSS, JSX, ES6, etc.) and turns it into a single file. The two main benefits of modular bundling are *modularity* and *network performance*.

Modularity will allow you to break down your source code into parts, or modules, that are easier to work with, especially in a team environment.

Network performance is gained by only needing to load one dependency in the browser, the bundle. Each script tag makes an HTTP request, and there is a latency penalty for each HTTP request. Bundling all of the dependencies into a single file allows you to load everything with one HTTP request, thereby avoiding additional latency.

Aside from handling transpiling, webpack also can handle:

- **Code Splitting:** Split up your code into different chunks that can be loaded when you need them. Sometimes these are called rollups or layers and aim to break up code as needed for different pages or devices.

- **Minification:** Removing whitespace, linebreaks, lengthy variable names, and unnecessary code to reduce the file size.
- **Feature Flagging:** Send code to one or more - but not all - environments when testing out features.
- **Hot Module Replacement (HMR):** Watches for changes in source code. Changes only the updated modules immediately.

Webpack Loaders

A loader is a function that handles the transformations that we want to put our code through during the build process. If our application uses ES6, JSX, CoffeeScript, and other languages that can't be read natively by the browser, we'll specify the necessary loaders in the `webpack.config.js` file to do the work of converting the code into syntax that can be read natively by the browser.

Webpack has a huge number of loaders that fall into a few categories. The most common use case for loaders is transpiling from one dialect to another. **For example, ES6 and React code is transpiled by including the babel-loader.** We specify the types of files that Babel should be run on, then Webpack will take care of the rest.

Another popular category of loaders is for styling. The `sass-loader` looks for files with the `.scss` extension and compiles to CSS. The `css-loader` can be used to include CSS modules in your bundle. All CSS is bundled as JavaScript and automatically added when the bundled JavaScript file is included. No need to use `link` elements to include stylesheets.

Check out the full **list of loaders** if you'd like to see all of the different options.

Recipes App with Webpack Build

The Recipes app that we have built at the beginning of this chapter has some limitations that webpack will help us alleviate. Using a tool like webpack to statically build your client JavaScript makes it possible for teams to work together on large-scale web applications. We can also gain the following benefits by incorporating the Webpack module bundler:

Modularity

Using the CommonJS module pattern in order to export modules that will later be imported or required by another part of the application makes our source more approachable. It allows development teams to easily work together by allowing them to create and work with separate files that will later be statically combined into a single file for runtime.

Composing

With modules, we can build small, simple, reusable, React components that we can efficiently compose into applications. Smaller components are easier to comprehend, easier to test, and easier to reuse. They are also easier to replace down the line when enhancing your applications.

Speed

Packaging all of the applications modules and dependencies into a single client bundle will reduce the load time of your application because there is latency associated with each HTTP request. Packaging everything together in a single file means that the client will only need to make a single request. Minifying the code in the bundle will improve load time as well.

Consistency

Since webpack will transpile JSX into React and ES6, even ES7, into universal JavaScript means that we can start using tomorrow's JavaScript syntax today. Babel supports a wide range of ES6 and ES7 syntax which means we do not have to worry about whether the browser supports our code. It allows developers to consistently use cutting edge JavaScript syntax.

Breaking Components into Modules

Approaching the recipes app with the ability to use webpack and Babel allows us to break our code down into modules that use ES6 syntax. Let's take a look at our Stateless Functional Component for Recipes.

Example 5-13. Current Recipe Component

```
const Recipe = ({ name, ingredients, steps }) =>
  <section id="baked-salmon">
    <h1>{name}</h1>
    <ul className="ingredients">
      {ingredients.map((ingredient, i) =>
        <li key={i}>{ingredient.name}</li>
      )}
    </ul>
    <section className="instructions">
      <h2>Cooking Instructions</h2>
      {steps.map((step, i) =>
        <p key={i}>{step}</p>
      )}
    </section>
  </section>
```

This component is doing quite a bit. We are displaying the name of the recipe, constructing an unordered list of ingredients, and displaying the instructions with each step getting it's own paragraph element.

A more functional approach to the recipe component would be to break it up into smaller more focused stateless functional components and compose them together. We can start by pulling the instructions out into its own stateless functional component and creating a module in a separate file that we can use for any set instructions.

Example 5-14. Instructions Component

```
const Instructions = ({ title, steps }) =>
  <section className="instructions">
    <h2>{title}</h2>
    {steps.map((s, i) =>
      <p key={i}>{s}</p>
    )}
  </section>
```

```
module.exports = Instructions
```

Here we have created a new component called instructions. We will pass the title of the instructions and the steps to this component. This way we can reuse this component for “Cooking Instructions”, “Baking Instructions”, “Prep Instructions”, or a “Pre-cool checklist”, we can use this component for anything that has steps.

Think about the ingredients. In the recipe component above, we are only displaying the ingredients name, but each ingredient in the data for the recipe has an amount and measurement as well. We could create a stateless functional component to represent a single ingredient.

Example 5-15. Ingredient Component

```
const Ingredient = ({ amount, measurement, name }) =>
  <li>
    <span className="amount">{amount}</span>
    <span className="measurement">{measurement}</span>
    <span className="name">{name}</span>
  </li>
```

```
module.exports = Ingredient
```

Here we assume each ingredient has an amount, a measurement, and a name. We will destructure those values from our props object and display them each in independent classed span elements.

Using the Ingredient component, we can construct an IngredientsList component that can be used any time we need to display a list of ingredients.

Example 5-16. IngredientsList using Ingredient Component

```
import Ingredient from './Ingredient'

const IngredientsList = ({ list }) =>
  <ul className="ingredients">
    {list.map((ingredient, i) =>
      <Ingredient key={i} {...ingredient} />
    )}
  </ul>

module.exports = IngredientsList
```

In this file, we first import the Ingredient component because we are going to use it for each ingredient. The ingredients are passed to this component as an array in a property called list. Each ingredient in the list array will be mapped to the ingredient component. The JSX spread operator is used to pass all of the data to the ingredient component as props.

Given an ingredient with these fields:

```
let ingredient = {
  amount: 1,
  measurement: 'cup',
  name: 'sugar'
}
```

The spread operator

```
<Ingredient {...ingredient} />
```

is another way of expressing

```
<Ingredient amount={ingredient.amount}
  measurement={ingredient.measurement}
  name={ingredient.name} />
```

or in this case

```
<Ingredient amount={1}
  measurement="cup"
  name="sugar" />
```

Now that we have components for Ingredients and Instructions we can compose recipes using these components.

Example 5-17. Refactored Recipe Component

```
import IngredientsList from './IngredientsList'
import Instructions from './Instructions'

const Recipe = ({ name, ingredients, steps }) =>
```

```

<section id={name.toLowerCase().replace(/ /g, '-')}>
  <h1>{name}</h1>
  <IngredientsList list={ingredients} />
  <Instructions title="Cooking Instructions"
    steps={steps} />
</section>

```

```
module.exports = Recipe
```

The first thing is to import the components that we are going to use, `IngredientsList` and `Instructions`. Now we can use them to create the `Recipe` Component. Instead of a bunch of complicated code building out the entire recipe details in one place, we have expressed our recipe more declaratively by composing smaller components. Not only is the code nice and simple it reads well. This shows us that a recipe should show the name of the recipe, a list of `Ingredients`, and some cooking instructions. We've abstracted away what it means to display `Ingredients` and `Instructions` into smaller simple components.

In a modular approach with `CommonJS`, the `Menu` component would look pretty similar. The key difference being that it would live in its own file, import the modules that it needs to use, and export itself.

Example 5-18. Completed Menu Component

```

import Recipe from './Recipe'

export const Menu = ({ recipes }) =>
  <article>
    <header>
      <h1>Delicious Recipes</h1>
    </header>
    <div className="recipes">
      { recipes.map((recipe, i) =>
        <Recipe key={i} {...recipe} />)
      }
    </div>
  </article>

module.exports = Menu

```

We still need to use `ReactDOM` to render the menu component, so we would still have a `main.js` file, but it would look much different.

Example 5-19. Completed index.js File

```

import React from 'react'
import { render } from 'react-dom'
import Menu from './components/Menu'
import data from './data/recipes'

```

```
window.React = React

render(
  <Menu recipes={data} />,
  document.getElementById("react-container")
)
```

The first four statements import the necessary modules for our app to work. Instead of loading react and react-dom via the script tag, we are going to import them so webpack can add them to our bundle. We also need the Menu component, and a sample data array which has been moved to a separate module. It still contains two recipes: Baked Salmon and Fish Tacos.

All of our imported variables are local to the Main.js file. Setting window.React to React, exposes the React library globally in the browser. This way all calls to React.createElement are assured to work.

When we render the Menu component, we pass the array of recipe data to this component as a property. This single ReactDOM.render call will mount and render our Menu component.

Now that we have pulled our code apart into separate modules and files, let's create a static build process with webpack that will put everything back together into a single file.

Installing Webpack Dependencies

In order to create a static build process with webpack, we'll need to install a few things. Everything that we need can be installed with npm. First, we might as well install webpack globally so we can use the webpack command anywhere.

```
$ sudo npm install -g webpack
```

Webpack is also going to work with Babel to transpile our code from JSX and ES6 to JavaScript that runs in the browser. We are going to use a few loaders along with a few presets to accomplish this task.

```
$ npm install babel-loader babel-preset-es2015 babel-preset-react babel-preset-stage-0
```

Our application uses React and ReactDOM. We've been loading these dependencies with the script tag. Now we are going to let webpack add them to our bundle. We'll need to install the dependencies for React and ReactDOM locally.

```
$ npm install react react-dom
```

This adds the necessary scripts for react and react-dom to the ./node_modules folder. Now we have everything needed to setup a static build process with webpack.

Webpack Configuration

For this modular recipes app to work we are going to need to tell webpack how to bundle our source code into a single file. We can do this with configuration files, and the default webpack configuration file is always **webpack.config.js**

The start file for our recipes app is `main.js`. It imports `React`, `ReactDOM`, and the `Menu.js`. This is what we want to run in the browser first. Wherever Webpack finds an import statement it will find the associated module in the file system and include it in the bundle. `Main.js` imports `Menu.js`, `Menu.js` imports `Recipe.js`, `Recipe.js` imports `Instructions.js` and `IngredientsList.js`, `IngredientsList.js` imports `Ingredient.js`. Webpack will follow this import tree and include all of these necessary modules in your bundle.



ES6 Import Statements

We are using ES6 import statements which are not presently supported by most browsers or by Node.js. The reason ES6 import statements work is because Babel will convert them into `require('module/path');` statements in our final code. The require function is how CommonJS modules are typically loaded.

As webpack builds our bundle, we need to tell webpack to transpile JSX to pure `React.Elements`. We also need to convert any ES6 syntax to ES5 syntax. Our build process will initially have 3 steps.

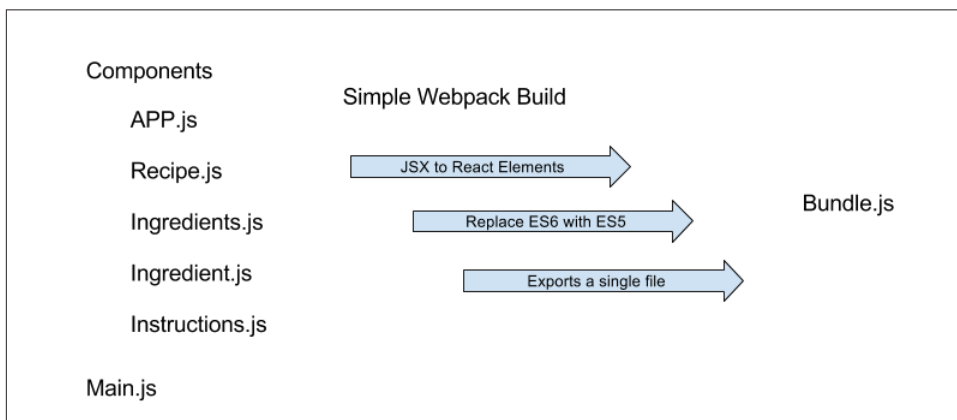


Figure 5-4. Recipe App Build process

The `webpack.config` file is just another module that exports a JavaScript literal object that describes the actions that webpack should take. This file should be saved to the root folder of the project right next to the `main.js` file.

Example 5-20. webpack.config.js

```
module.exports = {
  entry: "./main.js",
  output: {
    path: "dist/assets",
    filename: "bundle.js"
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: ['babel'],
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  }
}
```

First we tell webpack that our client entry file is `main.js`. It will automatically build the dependency tree based upon import statements starting in that file. Next we specify that we want to output a bundled JavaScript file to `./dist/assets/bundle.js`. This is where webpack will place the final packaged JavaScript.

The next set of instructions for webpack consists of a list of loaders to run on specified modules. Notice the `loaders` field is an array, this is because there are many types of loaders that you can incorporate with webpack, in this example we are only incorporating `babel`.

Each loader is a JavaScript object. The `test` field is a regular expression that matches the file path of each module that the loader should operator on. In this case we are running the `babel` loader on all imported JavaScript files except those found in the `node_modules` folder. When the `babel` loader runs, it will use presents for ES2015 (ES6) and React to transpile any ES6 or JSX syntax into JavaScript that will run in most browsers.

Webpack is run statically. Typically bundles are created before the app is deployed to the server. Since you have installed webpack globally, you can run it from the command line.

```
$ webpack
Time: 1727ms
Asset      Size  Chunks             Chunk Names
bundle.js  693 kB    0 [emitted]  main
  + 169 hidden modules
```

Webpack will either succeed and create a bundle or fail and show you an error. Typically most errors have to do with broken import references. When debugging webpack errors look closely at the file names and file paths used in import statements

Loading the Bundle

You have a bundle, so now what? We exported the bundle to the dist folder. This folder contains the files that we want to run on the web server. The dist folder is where the index.html file should be placed. This file needs to include a target div element where the React Menu Component will be mounted. It also requires a single script tag that will load your bundled JavaScript.

Example 5-21. index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>React Flux Recipes</title>
</head>
<body>
  <div id="react-container"></div>
  <script src="/assets/bundle.js"></script>
</body>
</html>
```

This is the home page for your app. It will load everything it needs from one file, one HTTP request, the bundle.js. You will need to deploy these files to your webserver or build a web server application that will serve these files with something like Node.js or Ruby on Rails.

Source Mapping

Bundling our code into a single file can cause some setbacks when it comes time to debug your application in the browser. We can eliminate this problem by providing a source map. A source map is a file that maps your bundle to the original source files. With webpack, they are easy to create, all we have to do is add a couple of lines to our webpack.config.js.

Example 5-22. webpack.config.js with source mapping

```
module.exports = {
  entry: './main.js',
  output: {
    path: 'dist/assets',
    filename: 'bundle.js',
    sourceMapFilename: 'bundle.map'
  },
}
```

```

devtool: '#source-map',
module: {
  loaders: [
    {
      test: /\.js$/,
      exclude: /(node_modules)/,
      loader: ['babel'],
      query: {
        presets: ['es2015', 'react']
      }
    }
  ]
}
}

```

Setting the devtool property to `'#source-map'` tells webpack that we want to use source-mapping. A `sourceMapFilename` is required. It is always a good idea to name your source map file after the target dependency. Webpack will associate the bundle with the source map during the export.

The next time you run webpack you will see that two output files are generated and added to the assets folder: the original `bundle.js` and the `bundle.map`.

The source map is going to let us debug using our original source files. In the sources tab of your browser's developer tools you should find a folder named **webpack://**. Inside of this folder you will see all of the source files in your bundle.

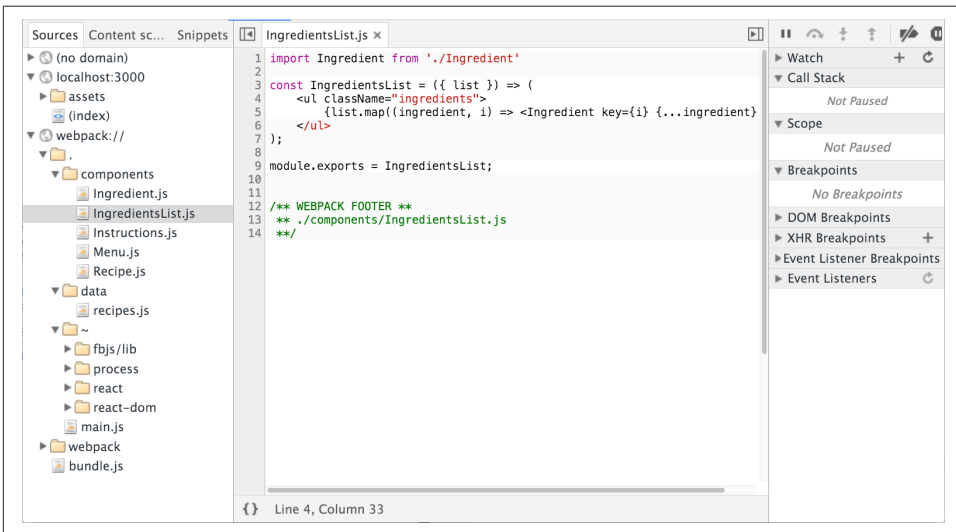


Figure 5-5. Sources panel of Chrome Developer Tools

You can debug from these files using the browser's step through debugger. Clicking on any line number adds a breakpoint. Refreshing the browser will pause JavaScript

processing when any breakpoints are reached in your source file. You can inspect scoped variables in the scopes panel or add variables to watch in the watch panel.

Optimizing the Bundle

The output bundle file is still simply a text file, so reducing the amount of text in this file will reduce the file size and cause it to load faster over HTTP. Some things that can be done to reduce the file size include removing all whitespace, reducing variable names to a single character, and removing any lines of code that the interpreter will never reach. Reducing the size of your JavaScript file with these tricks is referred to as minifying or uglifying your code.

Webpack has a built in plugin that you can use to uglify the bundle. In order to use it you will need to install webpack locally.

```
$ npm install webpack
```

We can add extra steps to the build process using webpack plugins. In this example, we are going to add a step to our build process to uglify our output bundle which will significantly reduce the file size.

Example 5-23. webpack.config.js with Uglify Plugin

```
var webpack = require("webpack");

module.exports = {
  entry: "./index.js",
  output: {
    path: "dist/assets",
    filename: "bundle.js",
    sourceMapFilename: 'bundle.map'
  },
  devtool: '#source-map',
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: ['babel'],
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin({
      sourceMap: true,
      warnings: false,
      mangle: true
    })
  ]
}
```



```
    })  
  ]  
}
```

To use the uglify plugin we need to require webpack, this is why we needed to install webpack locally. We can add any number of additional steps to our build process using a plugins array. Here we are going to add the UglifyJs step.

The UglifyJsPlugin is a function that gets instructions from its arguments. Once we uglify our code it will become unrecognizable, we are going to need a source map, which is why sourcemap is set to true. Setting warnings to false will remove any console warnings from the exported bundle. Mangling our code means that we are going to reduce long variable names like recipes or ingredients to a single letter.

The next time you run webpack you will see that the size of your bundled output has been significantly reduced and is no longer recognizable. Including a source map will still allow you to debug from your original source even though your bundle has been minified.

Props, State, and the Component Tree

In the last chapter, we talked about how to create components. We primarily focused on how to build a user interface by composing React components. This chapter is filled with techniques that you can use to better manage data and reduce time spent debugging applications.

Data handling within component trees is one of the key advantages of working with React. There are techniques that you can use when working with data in React components that will make your life much easier in the long run. Our applications will be easier to reason about and scale if we can manage data from a single location and construct UI based on that data.

Property Validation

JavaScript is a loosely typed language, which means that variables can change the datatypes for their values. For example, you can initially set a JavaScript variable as a string and later change its value to an array and JavaScript will not complain. Managing our variable types inefficiently can lead to a lot of time spent debugging applications.

React components provide a way to specify and validate property types. Using these features will greatly reduce the amount of time spent debugging applications. Supplying incorrect property types triggers warnings that can help us find bugs that may have otherwise slipped through the cracks.

React has built in automatic property validation for these variable types:

Table 6-1. React Property Validation^a

Arrays	React.PropTypes.array
--------	-----------------------

Boolean	React.PropTypes.bool
Functions	React.PropTypes.func
Numbers	React.PropTypes.number
Objects	React.PropTypes.object
Strings	React.PropTypes.string
^a React Documentation, Prop Validation: https://facebook.github.io/react/docs/reusable-components.html#prop-validation	

In this next section, we will create a Summary component for our recipes. The recipe summary component will display the title of the recipe along with counts for both ingredients and steps.



Figure 6-1. Summary Component Output for Baked Salmon

In order to display this data, we must supply the Summary component with three properties: a title, an array of ingredients, and an array of steps. We want to validate these properties to make sure they are arrays and supply default titles when they are unavailable. How to implement property validation depends upon how components are created. Stateless functional components and ES6 classes have a different way of implementing property validation.

First, let's look at why we should use property validation and how to implement it in components created with `React.createClass()`.

Validating Props with `createClass`

We need to understand why it is important to validate component property types. Consider the following implementation for the Summary component:

```
const Summary = createClass({
  displayName: "Summary",
  render() {
    const {ingredients, steps, title} = this.props
    return (
      <div className="summary">
        <h1>{title}</h1>
        <p>
```

```

        <span>{ingredients.length} Ingredients</span> |
        <span>{steps.length} Steps</span>
      </p>
    </div>
  )
}
})

```

The Summary component destructures ingredients, steps, and title from the properties object and then constructs a UI to display that data. Since we expect both ingredients and steps to be arrays, we'll use `Array.length()` to count the array's items.

What if we rendered this Summary component accidentally using strings?

```

render(
  <Summary title="Peanut Butter and Jelly"
    ingredients="peanut butter, jelly, bread"
    steps="spread peanut butter and jelly between bread" />,
  document.getElementById('react-container')
)

```

JavaScript will not complain, but finding the length will count the number of characters in each string.



Figure 6-2. Summary Component Output for Peanut Butter and Jelly

The output of this code is odd. No matter how fancy your peanut butter and jelly might be, it's doubtful that you are going to have 27 ingredients and 44 steps. Instead of seeing the correct number of steps and ingredients, we are seeing the length in characters of each string. A bug like this is easy to miss. If we validated the property types when we created the Summary component, React could catch this bug for us.

```

const Summary = createClass({
  displayName: "Summary",
  propTypes: {
    ingredients: PropTypes.array,
    steps: PropTypes.array,
    title: PropTypes.string
  },
  render() {
    const {ingredients, steps, title} = this.props
    return (

```

```

        <div className="summary">
          <h1>{title}</h1>
          <p>
            <span>{ingredients.length} Ingredients | </span>
            <span>{steps.length} Steps</span>
          </p>
        </div>
      )
    }
  })
}

```

Using React's built-in property type validation we can make sure that both ingredients and steps are arrays. Additionally, we can make sure that the title value is a string. Now when we pass incorrect property types, we will see an error.

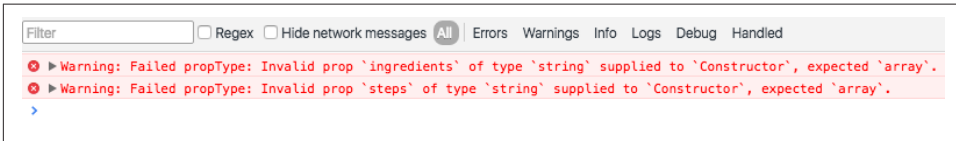


Figure 6-3. Property Type Validation Warning

What would happen if we rendered the Summary component without sending it any properties?

```

render(
  <Summary />,
  document.getElementById('react-container')
)

```

Rendering the Summary component without any properties causes a JavaScript error that takes down the web app.

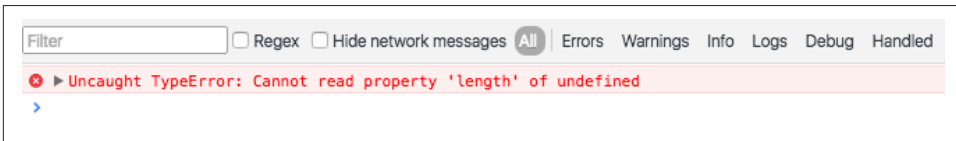


Figure 6-4. Error generated from missing array

This error occurs because the type of the ingredients property is undefined, and undefined is not an object that has a length property like an array or a string. React has a way to specify required properties. When those properties are not supplied, React will trigger a warning in the console.

```

const Summary = createClass({
  displayName: "Summary",
  propTypes: {
    ingredients: PropTypes.array.isRequired,
    steps: PropTypes.array.isRequired,
  }
})

```

```

        title: PropTypes.string
      },
      render() {
        ...
      }
    })
  })
}

```

Now when we render the Summary component without any properties React clearly directs our attention to the problem with a console warning just before the error occurs. This makes it easier to figure out what when wrong.

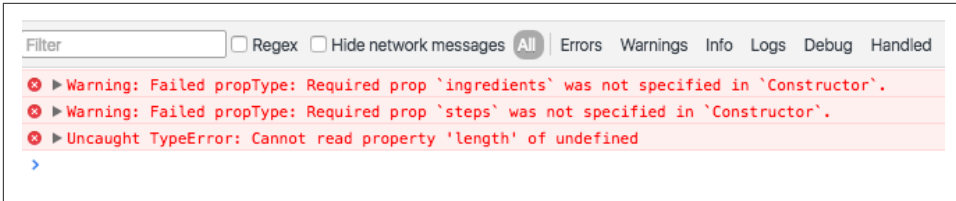


Figure 6-5. React warnings for missing properties

The Summary component expects an array for ingredients and an array for steps, but it only uses the length property of each array. This component is designed to display counts, numbers, for each of those values. It may make more sense to refactor our code to expect numbers instead, since the component doesn't actually need arrays.

```

import { createClass, PropTypes } from 'react'

export const Summary = createClass({
  displayName: "Summary",
  propTypes: {
    ingredients: PropTypes.number.isRequired,
    steps: PropTypes.number.isRequired,
    title: PropTypes.string
  },
  render() {
    const {ingredients, steps, title} = this.props
    return (
      <div className="summary">
        <h1>{title}</h1>
        <p>
          <span>{ingredients} Ingredients</span> |
          <span>{steps} Steps</span>
        </p>
      </div>
    )
  }
})

```

Instead of passing arrays, we can pass numbers to the Summary component using the length of the ingredients array. Using numbers for this component is a more flexible

approach. Now the Summary component simply displays UI, it sends the burden of actually counting ingredients or steps further up the component tree to a parent or ancestor.

Default Props

Another way to improve the quality of components is to assign default properties¹. The validation behavior is similar to what you might expect: the default values you establish will be used if other values are not provided.

Let's say we want the Summary component to work even when the properties are not supplied.

```
import { render } from 'react-dom'

render(<Summary />, document.getElementById('react-container'))
```

With `createClass`, we can add a method called `getDefaultProps` that returns default values for properties that are not assigned.

```
const Summary = createClass({
  displayName: "Summary",
  propTypes: {
    ingredients: PropTypes.number,
    steps: PropTypes.number,
    title: PropTypes.string
  },
  getDefaultProps() {
    return {
      ingredients: 0,
      steps: 0,
      title: "[recipe]"
    }
  },
  render() {
    const {ingredients, steps, title} = this.props
    return (
      <div className="summary">
        <h1>{title}</h1>
        <p>
          <span>{ingredients}</span> Ingredients | </span>
          <span>{steps}</span> Steps</span>
        </p>
      </div>
    )
  }
})
```

¹ React Documentation, Default Prop Values: <https://facebook.github.io/react/docs/reusable-components.html#default-prop-values>

Now when we try to render this component without properties, we will see some default data instead.



Figure 6-6. Summary Component output with default properties

Using default properties can extend the flexibility of your component and prevent errors from occurring when your users do not explicitly require every property.

Custom Property Validation

React's built-in validators are great for making sure that your variables are required and typed correctly. There are instances that require more robust validation. For example, you may want to make sure that a number is within a specific range or that a value contains a specific string. React provides a way to build your own custom validation.

Custom validation in React is implemented with a function. This function should either return an error when a specific validation requirement is not met or null when the property is valid.

With basic property type validation, you can only validate a property based on one condition. The good news is that the custom validator will allow us to test the property in many different ways. In the custom function, we'll first check that the property is a string. Then we'll limit this property to 20 characters.

Example 6-2. Custom Prop Validation

```
propTypes: {
  ingredients: PropTypes.number,
  steps: PropTypes.number,
  title: (props, propName) =>
    (typeof props[propName] !== 'string') ?
      new Error("A title must be a string") :
      (props[propName].length > 20) ?
        new Error(`title is over 20 characters`) :
        null
}
```

All property type validators are functions. To implement a custom validator, we will set the value of the title property, under the propTypes object, to a callback function.

When rendering the component, React will inject the props object and the name of the current property into the function as arguments. We can use those arguments to check the specific value for a specific property.

In this case, we are first checking the title to make sure it is a string. If the title is not a string the validator returns a new error with the message: “A title must be a string”. If the title is a string, then we will check its value to make sure it is not longer than 20 characters. If the title is under 20 errors, the validator function returns null. If the title is over 20 characters then the validator function will return an error. React will capture the returned error and display it in the console as a warning.

Custom validators allow you to implement specific validation criteria. A custom validator can perform multiple validations and only return errors when specific criteria is not met. Custom validators are a great way to prevent errors when using and reusing your components.

ES6 Classes and Stateless Functional Components

In the previous section, we discovered that `propTypes` and `defaultProps` can be added to our component classes using `React.createClass`. This type checking also works for ES6 classes and stateless functional components, but the syntax is slightly different.

When working with ES6 classes, `propTypes` and `defaultProps` declarations are defined on the class instance, outside of the class body. Once a class is defined, we can set the `propTypes` and `defaultProps` object literals.

Example 6-3. ES6 Class

```
class Summary extends React.Component {
  render() {
    const {ingredients, steps, title} = this.props
    return (
      <div className="summary">
        <h1>{title}</h1>
        <p>
          <span>{ingredients}</span> Ingredients | </span>
          <span>{steps}</span> Steps</span>
        </p>
      </div>
    )
  }
}

Summary.propTypes = {
  ingredients: PropTypes.number,
  steps: PropTypes.number,
  title: (props, propName) =>
```

```

    (typeof props[propName] !== 'string') ?
      new Error("A title must be a string") :
      (props[propName].length > 20) ?
        new Error(`title is over 20 characters`) :
        null
  }

  Summary.defaultProps = {
    ingredients: 0,
    steps: 0,
    title: "[recipe]"
  }

```

The `propTypes` and `defaultProps` object literals can also be added to stateless functional component.

Example 6-4. Stateless Functional Component

```

const Summary = ({ ingredients, steps, title }) => {
  return <div>
    <h1>{title}</h1>
    <p>{ingredients} Ingredients | {steps} Steps</p>
  </div>
}

Summary.propTypes = {
  ingredients: React.PropTypes.number.isRequired,
  steps: React.PropTypes.number.isRequired
}

Summary.defaultProps = {
  ingredients: 1,
  steps: 1
}

```

With a stateless functional component, you also have the option of setting default properties directly in the function arguments. We can set default values for ingredients, steps, and the title when we destructure the properties object in the function arguments.

```

const Summary = ({ ingredients=0, steps=0, title='[recipe]' }) => {
  return <div>
    <h1>{title}</h1>
    <p>{ingredients} Ingredients | {steps} Steps</p>
  </div>
}

```

```

    </div>
  }

```

Class Static Properties

In the previous section, we looked at how `defaultProps` and `propTypes` are defined outside of the class. An alternative to this is emerging in one of the latest proposals to the ECMAScript spec: Class Fields and Static Properties.

Class static properties allow us to encapsulate `propTypes`, `defaultProps` inside of the class declaration. Property initializers also provide encapsulation and cleaner syntax.

```

class Summary extends React.Component {

  static propTypes = {
    ingredients: PropTypes.number,
    steps: PropTypes.number,
    title: (props, propName) =>
      (typeof props[propName] !== 'string') ?
        new Error("A title must be a string") :
        (props[propName].length > 20) ?
          new Error(`title is over 20 characters`) :
          null
  }

  static defaultProps = {
    ingredients: 0,
    steps: 0,
    title: "[recipe]"
  }

  render() {
    const {ingredients, steps, title} = this.props
    return (
      <div className="summary">
        <h1>{title}</h1>
        <p>
          <span>{ingredients}</span> Ingredients | </span>
          <span>{steps}</span> Steps</span>
        </p>
      </div>
    )
  }
}

```

Property validation, custom property validation, and the ability to set default property values should be implemented in every component. This makes the component easier to reuse because any problems with component properties will show up as console warnings.

Refs

References, or refs, are a feature that allow React components to interact with child elements. The most common use case for refs is to interact with UI elements that collect input from the user. Consider an HTML form element. These elements are initially rendered, but the user can interact with them. When they do, the component should respond appropriately.

For the rest of this chapter, we are going to be working with an application that allows users to save and manage specific hexadecimal color values. This application, the color organizer, allows users to add colors to a list. Once a color is in the list, it can be rated or removed by the user.

We will need a form to collect information about new colors from the user. The user can supply the color's title and hex value in the corresponding fields. The AddColorForm renders the HTML with a text input and a color input for collecting hex values from the color wheel.

Example 6-6. Add Color Form

```
import { Component } from 'react'

class AddColorForm extends Component {
  render() {
    return (
      <form onSubmit={e=>e.preventDefault()}>
        <input type="text"
          placeholder="color title..." required/>
        <input type="color" required/>
        <button>ADD</button>
      </form>
    )
  }
}
```

The AddColorForm component renders an HTML form that contains three elements: a text input for the title, a color input for the color's hex value, and a button to submit the form. When the form is submitted, a handler function is invoked where the default form event is ignored. This prevents the form from trying to send a GET request once submitted.

Once we have the form rendered, we will need to interact with it. Specifically, when the form is first submitted, we need to collect the new color information and reset the form's fields so that the user can add more colors. Using refs, we can refer to the title and color elements and interact with them.

Example 6-7. Add Color Form

```
import { Component } from 'react'

class AddColorForm extends Component {
  constructor(props) {
    super(props)
    this.submit = this.submit.bind(this)
  }
  submit(e) {
    const { _title, _color } = this.refs
    e.preventDefault();
    alert(`New Color: ${_title.value} ${_color.value}`)
    _title.value = '';
    _color.value = '#000000';
    _title.focus();
  }
  render() {
    return (
      <form onSubmit={this.submit}>
        <input ref="_title"
              type="text"
              placeholder="color title..." required/>
        <input ref="_color"
              type="color" required/>
        <button>ADD</button>
      </form>
    )
  }
}
```

We need to add a constructor to this ES6 component class because we moved submit to its own function. With ES6 component classes, we must bind the scope of the component to any methods that need to access that scope with `this`.

Next, in the render method, we've set the form's `onSubmit` handler by pointing it to the component's `submit` method. We've also added `ref` fields to the components that we want to reference. A *ref* is an identifier that React uses to reference DOM elements. Adding `_title` and `_color` to the input's `ref` attribute means that we can access those elements with `this.refs._title` or `this.refs_color`.

When the user adds a new title, selects a new color, and submits the form, the component's `submit` method will be invoked to handle the event. After we prevent the form's default submit behavior, we send the user an alert that echoes back the data collected via refs. After the user dismisses the alert, refs are used again to reset the form values and focus on the title field.



Binding 'this' scope

When using `React.createClass` to create your components there is no need to bind the `this` scope to your component methods. `React.createClass` automatically binds the `this` scope for you.

Two-way Data Binding

It's nice to have a form that echoes back input data in an alert, but there is really no way to make money with such a product. What we need to do is collect data from the user and send it somewhere else to be handled. This means that any data collected may eventually make its way back to the server, which we will cover in Chapter 12. First, we need to collect the data from the `AddForm` component and pass it on.

A common solution for collecting data from a React component is two-way data binding. This involves sending a callback function to the component as a property that the component can use to pass data back as arguments. It's called two-way data binding because we send the component a function as a property, and the component sends data back as function arguments.

Let's say we wanted to use the color form, but when a user submits a new color we want to collect that information and log it to the console.

```
const logColor = (title, color) =>
  console.log(`New Color: ${title} | ${value}`)

<AddColorForm onNewColor={logColor} />
```

With two-way data binding, we can create a function called `logColor` that receives the title and color as arguments. The values of those arguments can be logged to the console. When we use the `AddColorForm`, we simply add a function property for `onNewColor` and set it to our `logColor` function. When the user adds a new color, `logColor` is invoked, and we've sent a function as a property.

To implement two-way data binding, all we need to do is invoke `onNewColor` from props with the appropriate data.

```
submit() {
  const {_title, _color} = this.refs
  this.props.onNewColor(_title.value, _color.value)
  _title.value = ''
  _color.value = '#000000'
  _title.focus()
}
```

In our component, this means that we'll replace the alert call with a call to `this.props.onNewColor()` and pass the new title and new color values that we have obtained through refs.

The role of the `AddColorForm` component is to collect data and pass it on. It is not concerned with what happens to that data. We can now use this form to collect color data from users and pass it on to some other component or method to handle the collected data.

```
<AddColorForm onNewColor={({title, color}) => {  
  console.log(`TODO: add new ${title} and ${color} to the list`)  
  console.log(`TODO: render UI with new Color`)  
}} />
```

When we are ready, we can collect the information from this component and add the new color to our list of colors.



Optional Function Properties

In order to make two-way data binding optional, you must first check to see if the function property exists before trying to invoke it. In the last example, not supplying an `onNewColor` function property would lead to a JavaScript error because the component will try to invoke an undefined value.

This can be avoided by first checking for the existence of the function property.

```
if (this.props.onNewColor) {  
  this.props.onNewColor(_title.value, _color.value)  
}
```

A better solution is to define the function property in the component's `propTypes` and `defaultProps`.

```
AddColorForm.propTypes = {  
  onNewColor: PropTypes.func  
}  
  
AddColorForm.defaultProps = {  
  onNewColor: f=>f  
}
```

Now when the property supplied is some type other than function, React will complain. If the `onNewColor` property is not supplied, it will default to this dummy function `f=>f`. This is simply a placeholder function that returns the first argument sent to it. Although this placeholder function doesn't do anything, it can be invoked by JavaScript without causing errors.

Refs in Stateless Functional Components

Refs can also be used in stateless functional components. These components do not have `this`, so it's not possible to use `this.refs`. Instead of using string attributes, we

will set refs using functions. This function will pass us the input instance as an argument. We can capture that instance and save it to a local variable.

Let's refactor the AddColorForm as a stateless functional component.

```
const AddColorForm = ({onNewColor=f=>f}) => {
  let _title, _color
  const submit = e => {
    e.preventDefault()
    onNewColor(_title.value, _color.value)
    _title.value = ''
    _color.value = '#000000'
    _title.focus()
  }
  return (
    <form onSubmit={submit}>
      <input ref={input => _title = input}
        type="text"
        placeholder="color title..." required/>
      <input ref={input => _color = input}
        type="color" required/>
      <button>ADD</button>
    </form>
  )
}
```

In this stateless functional component, refs are set with a callback function instead of a string. The callback function passes the element's instance as an argument. This instance can be captured and saved into a local variable like `_title` or `_color`. Once we've saved the refs to local variables they are easily accessed when the form is submitted.

React State Management

Thus far we've only used properties to handle data in React components. Properties are immutable. Once rendered, a component's properties do not change. In order for our UI to change, we would need some other mechanism that can re-render the component tree with new properties. React state is a built in option for managing data that will change within a component. When application state changes, the UI is re-rendered to reflect those changes.

Users interact with applications. They navigate, search, filter, select, add, update, and delete. When a user interacts with an application, the state of that application changes, and those changes are reflected back to the user as UI. Screens and menus appear and disappear. Visible content changes. Indicators light up or are turn off. In React, UI is a reflection of application state.

State can be expressed in React components with a single JavaScript object. When the state of a component changes, the component renders a new UI that reflects those

changes. What can be more functional than that? Given some data a React component will represent that data as UI. Given a change to that data, a React will update the UI as efficiently as possible to reflect that change.

Let's take a look at how we can incorporate state within our React components.

Introducing Component State

State represents data that we may wish to change within a component. To demonstrate this, we will take a look at a StarRating component.

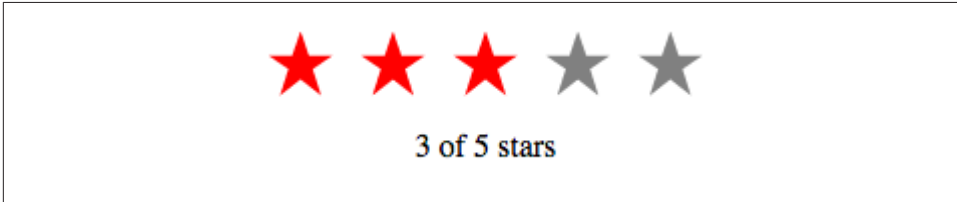


Figure 6-7. The StarRating component

The Star Rating Component requires two critical pieces of data: the total or the number of stars to display, and the rating, or the number of stars to highlight.

We'll need a clickable Star component that has a selected property. A stateless functional component can be used for each Star.

```
const Star = ({ selected=false, onClick=f=>f }) =>
  <div className={{(selected) ? "star selected" : "star"}
    onClick={onClick}>
    </div>

Star.propTypes = {
  selected: PropTypes.bool,
  onClick: PropTypes.func
}
```

Every Star element will consist of a div that includes the class 'star'. If the Star is selected it will additionally add the class 'selected'. This component also has an optional onClick property. When a user clicks on any Star div, the onClick property will be invoked. This will tell the parent component, the StarRating, that a Star has been clicked.

The Star is a stateless functional component. It says it right in the name: you cannot use state in a stateless functional component. Stateless functional components are meant to be the children of more complex, stateful components. It's a good idea to -

as the React documentation recommends - “try to keep as many of your components as possible stateless”².



The Star is in the CSS

Our star rating component uses CSS to construct and display a star. Specifically, using a clip-path, we can clip the area of our div to look like a star. The clip path is collection of points that make up a polygon.

```
.star {  
  cursor: pointer;  
  height: 25px;  
  width: 25px;  
  margin: 2px;  
  float: left;  
  background-color: grey;  
  clip-path: polygon(  
    50% 0%,  
    63% 38%,  
    100% 38%,  
    69% 59%,  
    82% 100%,  
    50% 75%,  
    18% 100%,  
    31% 59%,  
    0% 38%,  
    37% 38%  
  );  
}  
  
.star.selected {  
  background-color: red;  
}
```

A regular star has a background color of grey, but a selected star will have a background color of red.

Now that we have a Star we can use it to create a StarRating. StarRating will obtain the total number of stars to display from component properties. The rating, the value that the user can change, will be stored in state.

First, let's look at how to incorporate state into a component defined with `createClass`.

```
const StarRating = createClass({  
  displayName: 'StarRating',  
  propTypes: {
```

² “Thinking in React” by Pete Hunt: <https://facebook.github.io/react/docs/thinking-in-react.html#step-3-identify-the-minimal-but-complete-representation-of-ui-state>

```

        totalStars: PropTypes.number
      },
      getDefaultProps() {
        return {
          totalStars: 5
        }
      },
      getInitialState() {
        return {
          starsSelected: 0
        }
      },
      change(starsSelected) {
        this.setState({starsSelected})
      },
      render() {
        const {totalStars} = this.props
        const {starsSelected} = this.state
        return (
          <div className="star-rating">
            {[...Array(totalStars)].map((n, i) =>
              <Star key={i}
                selected={i<starsSelected}
                onClick={() => this.change(i+1)}
              />
            )}
            <p>{starsSelected} of {totalStars} stars</p>
          </div>
        )
      }
    })
  })
}

```

When using `createClass`, state can be initialized by adding `getInitialState()` to the component configuration and returning a JavaScript object that initially sets the state variable, `selectedStars`, to zero.

When the component renders, `totalStars` is obtained from component properties and used to render a specific number of `Star` elements. Specifically, the spread operator is used with the `Array` constructor to initialize a new array at a specific length that is mapped to `Star` elements.

The state variable, `starsSelected`, is destructured from `this.state` when the component renders. It is used to display the rating as text in a paragraph element. It is also used to calculate the number of selected stars to display. Each `Star` element obtains its `selected` property by comparing its index to the number of stars that are selected. If 3 stars are selected then the first 3 star elements will set the `selected` property to `true` and any remaining stars would have a `selected` property of `false`.

Finally, when a user clicks a single star, the index of that specific Star element is incremented and sent to the change function. This value is incremented because it is assumed that the first star would have a rating of 1 even though it has an index of 0.

Initializing state in an ES6 component class is slightly different than using create Class. In these classes, state can be initialized in the constructor.

```
class StarRating extends Component {

  constructor(props) {
    super(props)
    this.state = {
      starsSelected: 0
    }
    this.change = this.change.bind(this)
  }

  change(starsSelected) {
    this.setState({starsSelected})
  }

  render() {
    const {totalStars} = this.props
    const {starsSelected} = this.state
    return (
      <div className="star-rating">
        {[...Array(totalStars)].map((n, i) =>
          <Star key={i}
            selected={i<starsSelected}
            onClick={() => this.change(i+1)}
          />
        )}
        <p>{starsSelected} of {totalStars} stars</p>
      </div>
    )
  }

  static propTypes = {
    totalStars: PropTypes.number
  }

  static defaultProps = {
    totalStars: 5
  }
}
```

When an ES6 component is mounted, its constructor is invoked with properties injected as the first argument. Those properties are in turn sent to the superclass by invoking `super()`. In this case, the superclass is `React.Component`. Invoking `super()` initializes the component instance, and `React.Component` decorates that instance

with functionality that includes state management. After invoking `super()` , we can initialize our component state variables.

Once state is initialized, it operates as it does in `createClass` components. State can only be changed by calling `this.setState()` , which updates specific parts of the state object. After every `setState` call, the render function is called, updating the state with the new UI.

Initializing State from Properties

We can initialize our state values using incoming properties. There are only a few necessary cases for this pattern. The most common case for this is when we create a reusable component that we would like to use across applications in different component trees.

When using `createClass`, a good way to initialize state variables based on incoming properties is to add a method called `componentWillMount()`. The `componentWillMount()` method is invoked once when the component mounts, and you can call `this.setState()` from this method. It also has access to `this.props`, so you can use values from `this.props` to help you initialize state.

```
const StarRating = createClass({
  displayName: 'StarRating',
  propTypes: {
    totalStars: PropTypes.number
  },
  getDefaultProps() {
    return {
      totalStars: 5
    }
  },
  getInitialState() {
    return {
      starsSelected: 0
    }
  },
  componentWillMount() {
    const { starsSelected } = this.props
    if (starsSelected) {
      this.setState({starsSelected})
    }
  },
  change(starsSelected) {
    this.setState({starsSelected})
  },
  render() {
    const {totalStars} = this.props
    const {starsSelected} = this.state
    return (
      <div className="star-rating">
```

```

        {...Array(totalStars)].map((n, i) =>
          <Star key={i}
            selected={i<starsSelected}
            onClick={() => this.change(i+1)}
          />
        )}
      <p>{starsSelected} of {totalStars} stars</p>
    </div>
  )
}
})

render(
  <StarRating totalStars={7} starsSelected={3} />,
  document.getElementById('react-container')
)

```

`componentWillMount()` is a part of the component lifecycle. It can be used to help you initialize state based on property values in components created with `createClass` or ES6 class components. We will dive deeper into the component lifecycle in the next chapter.

There is an easier way to initialize state within an ES6 class component. The constructor receives properties as an argument, so you can simply use the props argument passed to the constructor.

```

constructor(props) {
  super(props)
  this.state = {
    starsSelected: props.starsSelected || 0
  }
  this.change = this.change.bind(this)
}

```

For the most part, you want to avoid setting state variables from properties. Only use these patterns when they are absolutely required. You should find this goal easy to accomplish because when working with React components you want limit the number of components that have state.³

³ React Documentation, Interactivity and Dynamic UIs: <https://facebook.github.io/react/docs/interactivity-and-dynamic-uis.html>



Updating component properties

When initializing state variables from component properties, you may need to re-initialize component state when a parent component changes those properties. The `componentWillReceiveProps()` lifecycle method can be used to solve this issue. Chapter 7 goes into greater detail on this issue and the available methods of the component lifecycle.

State within the component tree

All of your React components can have their own state, but should they? The joy of using React does not come from chasing down state variables all over your application. The joy of using React comes from building scalable applications that are easy to understand. The most important thing that you can do to make your application easy to understand is limit the number of components that use state as much as possible.

In many React applications it is possible to group all state data in the root component. State data can be passed down the component tree via properties, and data can be passed back up the tree to the root via two-way function binding. The result is that all of the state for your entire application exists in one place. This is often referred to as having a “single source of truth”.⁴

In this chapter, we will look at how to architect presentation layers where all of the state is stored in one place, the root component.

Color Organizer App Overview

The Color Organizer allows users to add, name, rate, and remove colors from their customized list. The entire state of the color organizer can be represented with a single array.

```
{
  colors: [
    {
      "id": "0175d1f0-a8c6-41bf-8d02-df5734d829a4",
      "title": "ocean at dusk",
      "color": "#00c4e2",
      "rating": 5
    },
    {
      "id": "83c7ba2f-7392-4d7d-9e23-35adbe186046",
      "title": "lawn",
      "color": "#26ac56",

```

⁴ Hacking with React, “State and the Single Source of Truth”: <http://www.hackingwithreact.com/read/1/12/state-and-the-single-source-of-truth>


```

    "rating": 3
  },
  {
    "id": "a11e3995-b0bd-4d58-8c48-5e49ae7f7f23",
    "title": "bright red",
    "color": "#ff0000",
    "rating": 0
  }
]
}

```

The array tells us that we need to display 3 colors: ocean as dusk, lawn, and bright red. It gives us the color's hex values and the current rating for each color in the display. It also provides a way to uniquely identify each color.

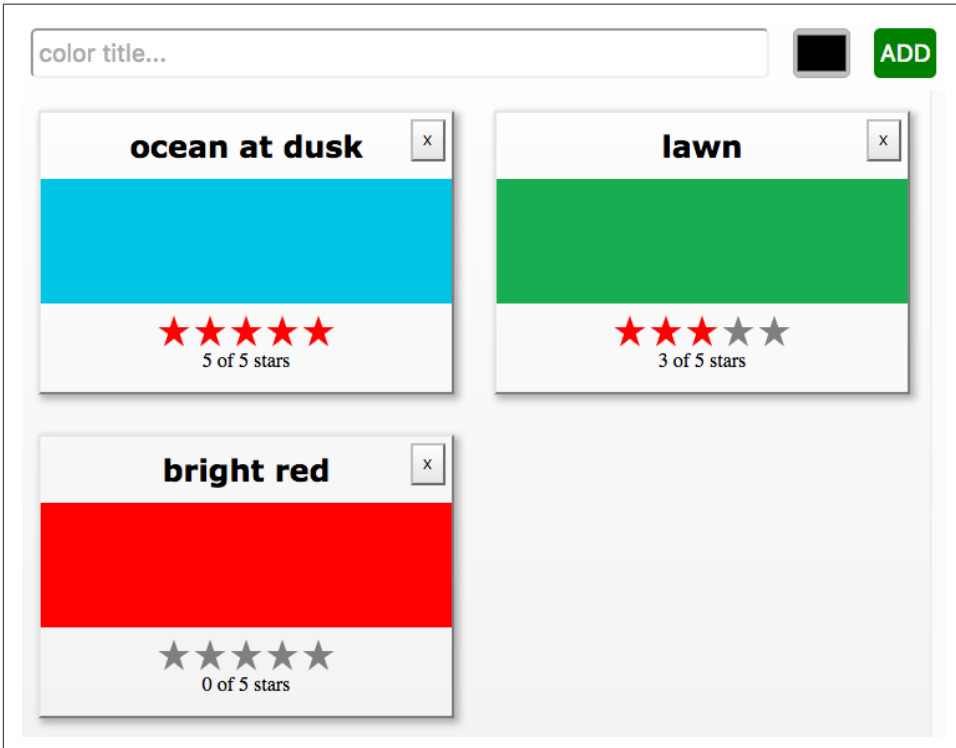


Figure 6-8. Color Organizer with 3 colors in state

This state data will drive our application. It will be used to construct the UI every time this object changes. When users add or remove colors, they will be added or removed from this array in state. When users rate colors, their rating will change in the array.

Passing properties down the component tree

Earlier in this chapter, we created a `StarRating` component that saved the rating in state. In the color organizer, the rating is stored in each color object. It makes more sense to treat the `StarRating` as a *presentational component*⁵ and declare it with a stateless functional component. Presentational components are only concerned with how things look in the application. They only render DOM Elements or other presentational components. All data is sent to these components via properties and passed out of these components via callback functions.

In order to make the `StarRating` component purely presentational, we need to remove state, presentational components only use props. Since we are removing state from this component, when a user changes the rating, that data will be passed out of this component via a callback function.

```
const StarRating = ({starsSelected=0, totalStars=5, onRate=f=>f}) =>
  <div className="star-rating">
    {[...Array(totalStars)].map((n, i) =>
      <Star key={i}
        selected={i<starsSelected}
        onClick={() => onRate(i+1)}/>
    )}
    <p>{starsSelected} of {totalStars} stars</p>
  </div>
```

First, `starsSelected` is no longer a state variable, it is a property. Second, an `onRate` callback property has been added to this component. Instead of calling `setState` when the user changes the rating this component now invokes `onRate` and sends the rating as an argument.



State in Reusable Components

You may need to create stateful UI components for distribution and re-use across many different applications. It is not absolutely required that you remove every last state variable from components that are only used for presentation. It is a good rule to follow, but sometimes it may make sense to keep state in a presentation component.

Restricting state to a single location, the root component, means that all of the data must be passed down to child components as properties.

⁵ “Smart and Dumb Components”, Dan Abramov: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

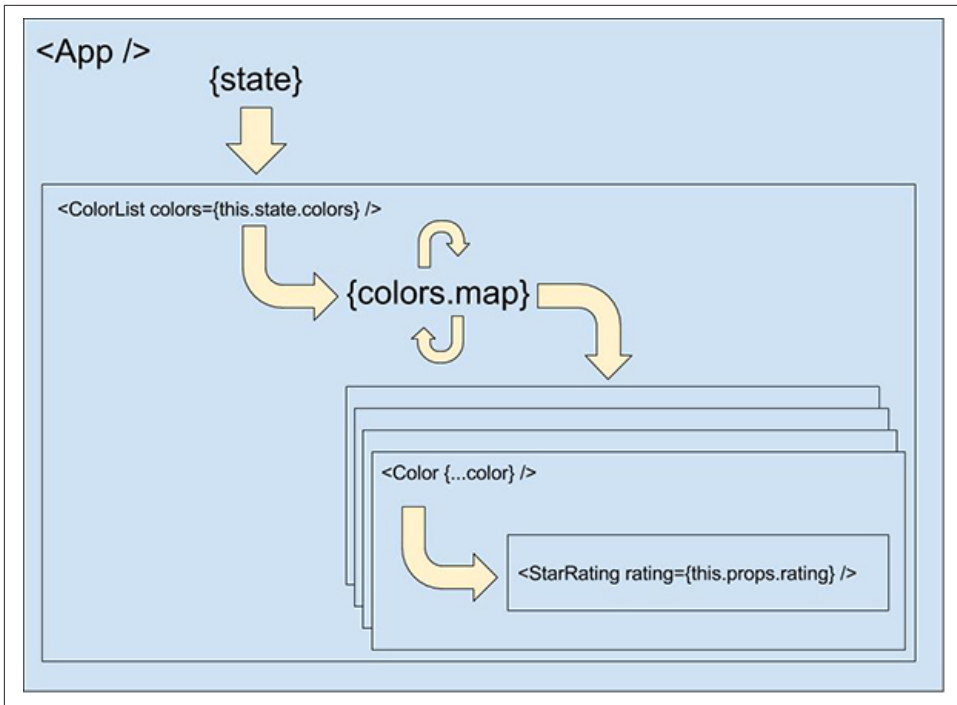


Figure 6-9. State is passed from the App component to child components as properties.

In the color organizer, state consists of an array of colors that is declared in the App component. Those colors are passed down to the ColorList component as a property.

```
class App extends Component {
  constructor(props) {
    super(props)
    this.state = {
      colors: []
    }
  }

  render() {
    const { colors } = this.state
    return (
      <div className="app">
        <AddColorForm />
        <ColorList colors={colors} />
      </div>
    )
  }
}
```

Initially the colors array is empty, so the ColorList component will display a message instead of each color. When there are colors in the array, data for each individual color is passed to the Color component as properties.

```
const ColorList = ({ colors=[] }) =>
  <div className="color-list">
    {(colors.length === 0) ?
      <p>No Colors Listed. (Add a Color)</p> :
      colors.map(color =>
        <Color key={color.id} {...color} />
      )
    }
  </div>
```

Now the color component can display the color's title and hex value and pass the color's rating down to the StarRating component as a property.

```
const Color = ({ title, color, rating=0 }) =>
  <section className="color">
    <h1>{title}</h1>
    <div className="color"
      style={{ backgroundColor: color }}>
    </div>
    <div>
      <StarRating starsSelected={rating} />
    </div>
  </section>
```

The number of starsSelected in the star rating comes from each color's rating. All of the state data for every color has been passed down the tree to child components as properties. When there is a change to the data in the root component, React will change the UI as efficiently as possible to reflect the new state.

Passing data back up the component tree

State in the color organizer can only be updated by calling setState from the app component. If the user initiates any change from the UI, their input would need to be passed back up the component tree to the App component in order to update the state. This can be accomplished through the use of callback function properties.

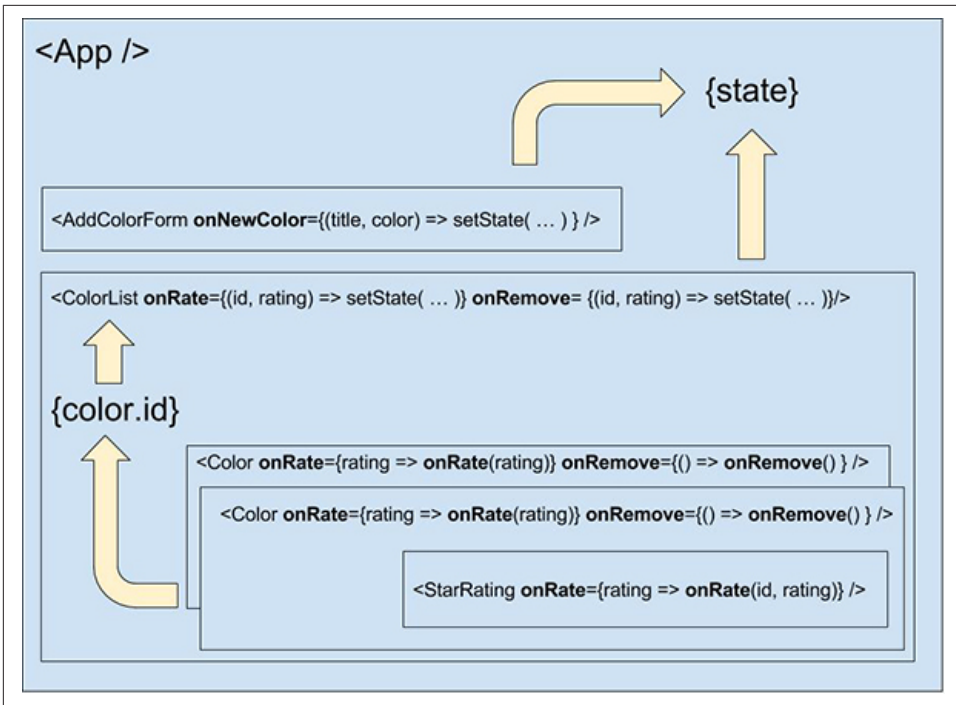


Figure 6-10. Passing data back up to the root component when there are UI events

In order to add new colors, we need a way to uniquely identify each color. This identifier will be used to locate colors within the state array. We can use the `node-uuid` library to create absolutely unique ids.

```
npm install node-uuid --save
```

All new colors will be added to the color organizer from the `AddColorForm` component that we constructed in the Refs section of this chapter. That component has an optional callback function property called `onNewColor`. When the user adds a new color and submits the form, the `onNewColor` callback function is invoked with the new title and color hex value obtained from the user.

```
import { Component } from 'react'
import { v4 } from 'node-uuid'
import AddColorForm from './AddColorForm'
import ColorList from './ColorList'

export class App extends Component {

  constructor(props) {
    super(props)
    this.state = {
      colors: []
    }
  }
}
```

```

    }
    this.addColor = this.addColor.bind(this)
  }

  addColor(title, color) {
    const colors = [
      ...this.state.colors,
      {
        id: v4(),
        title,
        color,
        rating: 0
      }
    ]
    this.setState({colors})
  }

  render() {
    const { addColor } = this
    const { colors } = this.state
    return (
      <div className="app">
        <AddColorForm onNewColor={addColor} />
        <ColorList colors={colors} />
      </div>
    )
  }
}

```

All new colors can be added from the `addColor()` method in the App Component. This function is bound to the component in the constructor, which means that it has access to `this.state` and `this.setState`.

New colors are added by concatenating the current colors array with a new color object. The id for the new color object is set using node-uuid's `v4()` function, this creates a unique identifier for each color. The title and color has been passed to the `addColor()` method from the `AddColorForm` component. Finally, the initial value for each color's rating will be 0.

When the user adds a color with the `AddColorForm` component, the `addColor()` method updates the state with a new list of colors. Once the state has been updated the App component re-renders the component tree with the new list of colors. The render method is invoked after every `setState` call. The new data is passed down the tree as properties and is used to construct the UI.

If the user wishes to rate or remove a color, we need to collect information about that color. Each color will have a remove button, if the user clicks the remove button we'll know they wish to remove that color. Also, if the user changes the color's rating with the `StarRating` component, we want to change the rating of that color.

```

const Color = ({title,color,rating=0,onRemove=f=>f,onRate=f=>f}) =>
  <section className="color">
    <h1>{title}</h1>
    <button onClick={onRemove}>X</button>
    <div className="color"
      style={{ backgroundColor: color }}>
    </div>
    <div>
      <StarRating starsSelected={rating} onRate={onRate} />
    </div>
  </section>

```

The information that will change in this app is all stored in the list of colors. Therefore, onRemove and onRate callback properties will have to be added to each color to pass those events back up the tree. The Color component will also have an onRate and onRemove callback function properties. When colors are rated or removed, the ColorList component will need to notify its parent, the App, that the color should be rated or removed.

```

const ColorList = ({ colors=[], onRate=f=>f, onRemove=f=>f }) =>
  <div className="color-list">
    {(colors.length === 0) ?
      <p>No Colors Listed. (Add a Color)</p> :
      colors.map(color =>
        <Color key={color.id}
          {...color}
          onRate={(rating) => onRate(color.id, rating)}
          onRemove={() => onRemove(color.id)} />
      )
    }
  </div>

```

The ColorList will invoke onRate() if any colors are rated and onRemove() if any colors are removed. This component manages the collection of colors by mapping them to individual color components. When individual colors are rated or removed the ColorList identifies which color was rated or removed and passes that info to its parent via callback function properties.

The ColorList's parent is the App. In the App Component, methods for rateColor() or removeColor() can be added and bound to the component instance in the constructor. Anytime a color needs to be rated or removed, these methods will update the state. They are added to the ColorList component as callback function properties.

```

class App extends Component {
  constructor(props) {
    super(props)
    this.state = {
      colors: []
    }
    this.addColor = this.addColor.bind(this)
  }

```

```

    this.rateColor = this.rateColor.bind(this)
    this.removeColor = this.removeColor.bind(this)
  }

  addColor(title, color) {
    const colors = [
      ...this.state.colors,
      {
        id: v4(),
        title,
        color,
        rating: 0
      }
    ]
    this.setState({colors})
  }

  rateColor(id, rating) {
    const colors = this.state.colors.map(color =>
      (color.id !== id) ?
        color :
        {
          ...color,
          rating
        }
    )
    this.setState({colors})
  }

  removeColor(id) {
    const colors = this.state.colors.filter(
      color => color.id !== id
    )
    this.setState({colors})
  }

  render() {
    const { addColor, rateColor, removeColor } = this
    const { colors } = this.state
    return (
      <div className="app">
        <AddColorForm onNewColor={addColor} />
        <ColorList colors={colors}
          onRate={rateColor}
          onRemove={removeColor} />
      </div>
    )
  }
}

```


Both `rateColor()` and `removeColor()` expect the id of the color to rate or remove. The id is captured in the `ColorList` component and passed as an argument to `rateColor` or `removeColor`. The `rateColor` method finds the color to rate and changes its rating in state. The `removeColor` method uses `Array.filter` to create a new state array that removes the removed color.

Once `setState` is called, the UI is re-rendered with the new state data. All data that changes in this app is managed from a single component, the `App`. This approach makes it much easier to understand what data the application uses to create state and how that data will change.

React components are quite robust. They provide us with a clean way to manage and validate properties, communicate with child elements, and manage state data from within a component. These features make it possible to construct beautifully scalable presentation layers.

We have mentioned many times that state data is for data that changes. You can also use state to cache data in your application. For instance, if you had a list of records that the user could search, the records list could be stored in state until they are searched.

Reducing state to root components is often recommended, and you will encounter this approach in many React applications. Once your application reaches a certain size, two-way data binding and explicitly passing properties can become quite a nuisance. The Flux design pattern and Flux libraries like `Redux` can be used to manage state and reduce boilerplate in these situations.

React is a relatively small library, and thus far we've reviewed much of its functionality. The major features of React Components that we have yet to discuss include: the component lifecycle and higher order components which we will cover in the next chapter.