



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Mastering React

Master the art of building modern web applications using React

Adam Horton
Ryan Vice

[PACKT] open source*
PUBLISHING community experience distilled

Mastering React

Master the art of building modern web applications
using React

Adam Horton

Ryan Vice



BIRMINGHAM - MUMBAI

Mastering React

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2016

Production reference: 1170216

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78355-856-8

www.packtpub.com

Credits

Authors

Adam Horton

Ryan Vice

Reviewer

Tung Dao

Commissioning Editor

Veena Pagare

Acquisition Editor

Kirk D'costa

Content Development Editor

Rashmi Suvarna

Technical Editor

Vivek Pala

Copy Editor

Neha Vyas

Project Coordinator

Judie Jose

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Graphics

Disha Haria

Production Coordinator

Nilesh Mohite

Cover Work

Nilesh Mohite

Foreword

We've all heard the old phrase, "Don't reinvent the wheel."

On the surface, I understand the wisdom of this ancient idiom, especially in the way it relates to software craftsmanship. Programmers are expected to always work within known patterns and get it shipped as fast as possible. We have so many words in software engineering to disparage the act of seemingly needless experimentation and rework—stop yak shaving, bikeshedding, gold plating, tinkering, configuring, fiddling, experimenting, reworking, or creating special snowflake architectures. Also, we have heard "stop chasing waterfalls and stick to the rivers and lakes that you're used to." Indeed, the noblest of software developers proudly stand on the shoulders of giants by implementing best practices and established standards. By contrast, the epitome of software self-indulgence is a shop with Not Invented Here Syndrome. Stick to the plan, stay focused, stop wasting time, and do what we already know works.

If ever a community of software developers rejected the total adoption of this worldview, it is the serious practitioners of JavaScript. The constantly moving target of browser capability, the never-ending inflow of developers with varied backgrounds, and the ever-evolving standards of JavaScript itself conspire to forge an expectation of mutability in the stack.

Reinvention is commonplace and always has been. When interacting with the DOM on a browser was problematic, a target for reinvention was set. Sizzle, jQuery, and eventually the native implementation, `querySelectorAll`, were born of a fundamental dissatisfaction with existing standards. From the ashes of the best practice of XML, JSON ascended as the dominant standard for web communication. Download a JavaScript framework today, and it could be using any number of patterns. Look upon the wheels of varying shapes and sizes: MVVM, MVC, MVW, MVP, Chain of Responsibility, PubSub, Event-Driven, Declarative, Functional, Object-Oriented, Modules, Prototypes. There is no one true way to architect a program. Furthermore, even a cursory glance at the world of preprocessors, such as CoffeeScript, LiveScript, Babel, Typescript, and ArnoldC, proves that developers are feverishly reinventing even JavaScript itself. Nothing is sacred, and perhaps that is why JavaScript has progressed so rapidly.

I remember the first time I learned about React. I was attending a fairly well-known conference in San Francisco, and during lunch, I had the fortune of sitting next to some developers from Facebook and Khan Academy, who made for some lively conversation. At the time, the most popular tools were Ember, Backbone, and — of course — Angular (there were something to the tune of thirty talks on that topic at the conference). We began to discuss the pros and cons of the existing tooling, and some of the difficulties, we felt, were because of the prevailing opinions on how to abstract a web application. It was then that the person sitting next to me said, "Perhaps you should join the React family," and he invited me to see his one and only talk that day. Of course, I went. It ended up being the most valuable (and at the time, controversial) presentation I attended.

This lunchtime conversationalist, who had introduced himself as Pete Hunt, turned out to be a core contributor to a new way of thinking about web applications. I attended his talk and knew immediately that I was looking upon the next great reinvention of the wheel in JavaScript. Normal two-way data binding techniques were eschewed for a clearer one-way data flow, and the standard MVC pattern of application organization had been rethought and re-forged into actions, stores, and dispatchers. However, the most interesting and radical feature of React was its way of dealing with the troublesome DOM — completely and unapologetically rebuilding it from the ground up in JavaScript.

If you've picked up this book, you are already interested in the future of JavaScript. This recurring theme of reinvention has been more relevant than ever in the last few years. React, ES6, modern build systems, scaffolding, and many more are the new tools populating the JavaScript landscape. This book is important because it teaches React alongside this modern ecosystem. After reading this book, you'll understand the principles needed to plan, design, and, ultimately, write a real application.

I can think of no better teacher for this exciting journey into the frontier of application design than Adam. I first met him when I was a student and have since had the pleasure of seeing him speak at Thunder Plains, a conference focused on the latest and greatest in the world of web development. He presented a whimsical collection of his personal projects, such as a lander game-based midpoint displacement and a completely rebuilt 3D ray casting engine in vanilla JavaScript.

Adam is a unique flavor of programmer. He works like a scientist, tinkerer, and craftsman. He is neither afraid to rebuild an existing system to better understand it, nor is he afraid to experiment in new ways to seek better ways of achieving his goals. When navigating these exciting new happenings in the world of JavaScript, you need a guide that encourages critical thinking, exploration, and discovery.

Your other guide is Ryan Vice, who has, over the years, thrice held the title of Microsoft MVP, published books on enterprise architecture, spoken frequently at industry events, and worked in the battle-hardened trenches of software development. More importantly though, Ryan created his own shop, Vice Software LLC, that puts React at the center of their webstack to solve their problems. His real-world experience in production of React projects qualifies him as an excellent teacher to help you on your way to building your own applications on the bleeding-edge of the web.

Reinventing the wheel is necessary. If you disagree, then I challenge you to attach to your car the first wheels ever invented. Stick to your convictions and roll mirthfully along the highway propelled by cumbersome stone disks. I will be dreaming of flying cars and betting on JavaScript.

Jesse Harlin

<http://jesseharlin.net/>

JavaScript Architect and Community Leader

About the Authors

Adam Horton is an avid retro gamer as well as a creator, destroyer, and rebuilder of all things Web, computing, and gaming. He started his career as a firmware developer for the high-end Superdome server division at Hewlett Packard. There, the JavaScript and C he wrought directed the power, cooling, health, and configuration of those behemoth devices. Since then, he has been a web developer for PayPal, utilizing cross-domain JavaScript techniques with an emphasis on user identity. Lately, at ESO Solutions, he's a lead JavaScript developer for next-generation, pre-hospital electronic health record (EHR) field collection applications.

Adam believes in an inclusive, ubiquitous, and open Web. He values pragmatism and practice over dogma in the design and implementation of computing applications and education.

I'd like to thank my wife for her enduring patience and support. She is the wind at my back that presses forward all of my endeavors, including this book. I'd also like to thank my parents for constantly fueling a stray rocket of a child while he tuned his guidance system.

Ryan Vice is the founder and chief architect of Vice Software, which specializes in practical, tailored solutions for clients, whether they are looking to get their MVP to market or modernize existing applications. On top of offering more competitive prices across the board, Vice Software offers skill-based pricing, which means you only pay architect rates when needed and pay much lower production rates for simpler feature work. Ryan has also been awarded Microsoft's MVP award three times, has published one other book on software architecture, and frequently speaks at conferences and events in Texas. Additionally, Ryan is lucky enough to be married to his wife, Heather, and spends most of his free time trying to keep up with their three kids, Grace, Dylan, and Noah.

About the Reviewer

Tung Dao is a full-stack developer with several years of experience building websites and services.

Currently, he works as a software engineer at FPT Software, Vietnam, where he builds RESTful web services involving NoSQL and Elasticsearch. In his free time, he is busy building web apps in Clojure/Go or hacking his Raspberry Pi.

Nowadays, his front-end work is mostly done in ClojureScript/Reagent (React binding in Clojure). Working over a binding did hide some of the great ideas in React. This book is a refresher to him, as he works with the next generation of JavaScript (ES6) and the re-explorer core React philosophy.

Many thanks to the authors and the staff at Packt Publishing for all their hard work and support.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	vii
Chapter 1: Introduction to React	1
Hello React	1
JSX	6
How it works	6
Decompiling JSX	8
Structure of render result	9
props	11
How it works	12
propTypes	13
getDefaultProps	15
state	16
How it works	17
Summary	18
Chapter 2: Component Composition and Lifecycle	19
How to compose simple components	19
Composing components with behavior	21
How it works	23
Accessing a component's children	27
Component lifecycle - mounting and unmounting	32
Component lifecycle – updating events	35
How it works	38
Summary	41
Chapter 3: Dynamic Components, Mixins, Forms, and More JSX	43
Dynamic components	43
How it works	45
Mixins	47
How it works	49

Forms	51
Controlled components - the read-only input	51
How it works	52
Controlled components - the read and write input	52
How it works	53
Isn't that harder than it needs to be?	54
Controlled components – a simple form	55
How it works	57
But what about the best practices?	58
Refactoring the form to be data driven	59
How it works	60
Validation	60
Validation types	61
The react-validation-mixin example	62
Summary	75
Chapter 4: Anatomy of a React Application	77
What is a single-page application?	78
Three aspects of a SPA design	79
Build systems	80
Choosing a build system	81
Module systems	83
CSS preprocessors	85
Compiling the modern JS syntax and JSX templates	85
Front-end architecture components	86
The front-end router	87
Front-end models	87
Views, view models, and view controllers	88
Messaging and eventing	88
Other utility needs	88
The application design	89
Creating wireframes	90
Main data entities and the API	92
Main views, site map, and routes	93
Summary	94
Chapter 5: Starting a React Application	95
Application design	95
Creating wireframes	95
User-related views	96
Post-related views	98
Data entities	99
Main views and the sitemap	100
Preparing the development environment	101
Installing Node and its dependencies	101
Installing and configuring Webpack	103

The Webpack configuration	104
Considerations before starting	109
React and rendering	109
Starting the app	111
The directory structure	111
The mock database	112
index.html	112
js/app.jsx	113
Main views	115
Linking views with React Router	116
Summary	118
Chapter 6: React Blog App Part 1 – Actions and Common Components	119
Reflux actions	120
Reusable components and base styles	121
Base styles	121
Inputs and loading indicator	125
The BasicInput component	125
The loader component	126
The application header	128
Summary	128
Chapter 7: React Blog App Part 2 – Users	129
Code manifest	130
Application runtime configuration	131
Mixins and dependencies	131
Reading and writing cookies	131
The form utilities mixin	132
User-related stores	135
The session context store	135
The user store	137
User views	139
The log in view	139
The create user view	141
Mixins and lifecycle methods	146
The user profile image	147
Form validation and submission	147
The user view component	149
The user list view	150
The user view	152
Other affected views	152
The app header	152
Summary	153

Chapter 8: React Blog App Part 3 – Posts	155
Code manifest	155
The posts store	156
Post views	158
Post create/edit	158
Mixins and lifecycle methods	163
Form submission	164
The post view	164
The post list component	169
The post list view	171
Other affected views	172
The user view	172
Summary	173
Chapter 9: React Blog App Part 4 – Infinite Scroll and Search	175
Infinite scroll loading	176
Infinite scroll code manifest	177
Modifying the posts store	177
Modifying the post list component	180
Searching posts	184
Search feature code manifest	184
The search store	185
Modifying the posts store	185
Modifying the application header	188
Modifying the post list component	190
Final thoughts	193
Suggested improvements	193
Level up the blog app	194
Moving forward	194
Chapter 10: Animation in React	195
Animation terms	196
CSS transitions using class switching	196
JavaScript code	197
CSS source	198
Animating DOM enter and exit	199
Popover menus	200
JavaScript source	200
CSS source	202
List filtering	205
JavaScript source	206
CSS source	209

Using the React-Motion animation library	210
How React-Motion works	210
Clock animation	211
JavaScript source	211
CSS source	218
Summary	220
Index	221

Preface

The book before you is a collaboration between myself — a web-focused, day-to-day code slinger — and Ryan Vice — a veteran .NET expert turned web application Maven and entrepreneur. I met Ryan at my current company while creating very complex web applications for emergency response outfits, such as EMS. At that time, the company was building a revolution of its flagship product. While it was still a very early time for React, it offered something above and beyond all the current MV* frameworks: a fresh approach to blazing-fast rendering. Since speed is paramount in the emergency response industry, we embraced React. When an opportunity came along to write about React, Ryan tapped me for assistance, and this book is the result of that.

It's great timing for a technology such as React. The open web platform has succeeded because of a simple tenet: never break the Web. As a result, the DOM, the in-browser data representation of a rendered web page, has grown to be enormous and somewhat unwieldy. Every small touch on the DOM cascades into a flurry of calculation and reconciliation in order to update pixels on the screen. React treats the DOM as the expensive resource that it is and makes operations on the DOM minimal. The time saved is better spent running your complex application logic.

In this book, you'll learn the fundamentals of React as well as a pragmatic approach to making web applications. You'll also learn how to choose the right tools for your needs. In the first three chapters, we'll start by thoroughly covering basic React topics such as state, props, and JSX, as well as the more complex subject of forms and validation. *Chapter 4, Anatomy of a React Application*, is something unique to a book like this: a tour of web application anatomy and some design techniques that can help you formulate a clear plan to build your apps. In chapters 5 through 9, we'll build a multiuser blog application using the design techniques and supporting libraries explored in the previous two chapters. Finally, in *Chapter 10, Animation in React*, we'll have some fun by navigating through the many ways you can create cool animations using React.

What this book covers

Chapter 1, Introduction to React, explains the basics of React JS, starting with a simple Hello World example and moving forward thru types of React entities and their definitions.

Chapter 2, Component Composition and Lifecycle, explores nesting components and managing their state as they come and go from the DOM.

Chapter 3, Dynamic Components, Mixins, Forms, and More JSX, explores React form basics and patterns for validation in React.

Chapter 4, Anatomy of a React Application, teaches how to approach web application design and how to choose from the vast menu of web technologies available within the context of React being your primary choice. Practice technical design and learn how to generate artifacts that guide development.

Chapter 5, Starting a React Application, begins with a fully featured React multiuser blog application. Prepare a development environment. Install all of the tools. Scaffold the application views.

Chapter 6, React Blog App Part 1 – Actions and Common Components, establishes a application communication strategy using Reflux Actions. Here, we will create some common components.

Chapter 7, React Blog App Part 2– Users, explains how to implement user account management for our prototype application.

Chapter 8, React Blog App Part 3 – Posts, covers creating and viewing blog posts.

Chapter 9, React Blog App Part 4 – Infinite Scroll and Search, explains how to add two features: infinite scroll loading and search.

Chapter 10, Animation in React, reveals web animation techniques in React.

What you need for this book

All of the software used in this book is open source, so it's free to use. You'll need a web browser (obviously) and the ability to install Node and npm. Some command-line access will be needed. For the command-line tasks, Bash is recommended. It is readily available for OSX and Linux as well as Windows through Git (git-bash). In this book, we'll use Node 4.x and React 0.14.

Who this book is for

This book is focused on web professionals who already understand JavaScript, CSS, and working in the web-browser environment. Previous experience with other web application frameworks isn't required, but may help. Comfort with the command line will also make things easier.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We then defined a second object literal that defines a `componentWillMount` method that calls `console.log` and passes `componentWillMount` from `ReactMixin2`."


Blocks of code and inline code uses the following format:


```
var path    = require('path')
,   webpack = require('webpack')
;
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
PostStore.getPostsByPage(
  this.state.page,
  Object.assign({}, this.state.search ? {q: this.state.search} :
{}, this.props)
).then(function (results) {
  var data = results.results;
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the **OK** button to see the output."

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

About the code samples

The first three chapters in this book are a React primer. These chapters are example driven and contain concise working examples of each concept which can be easily run in an Internet connected browser. A similar format is used in *Chapter 10, Animation in React* for the animation examples. The rest of the chapters with code listings, 4 thru 9, are contained in ZIP files included with the book and obtained from the Packt website.


Where to get the code samples

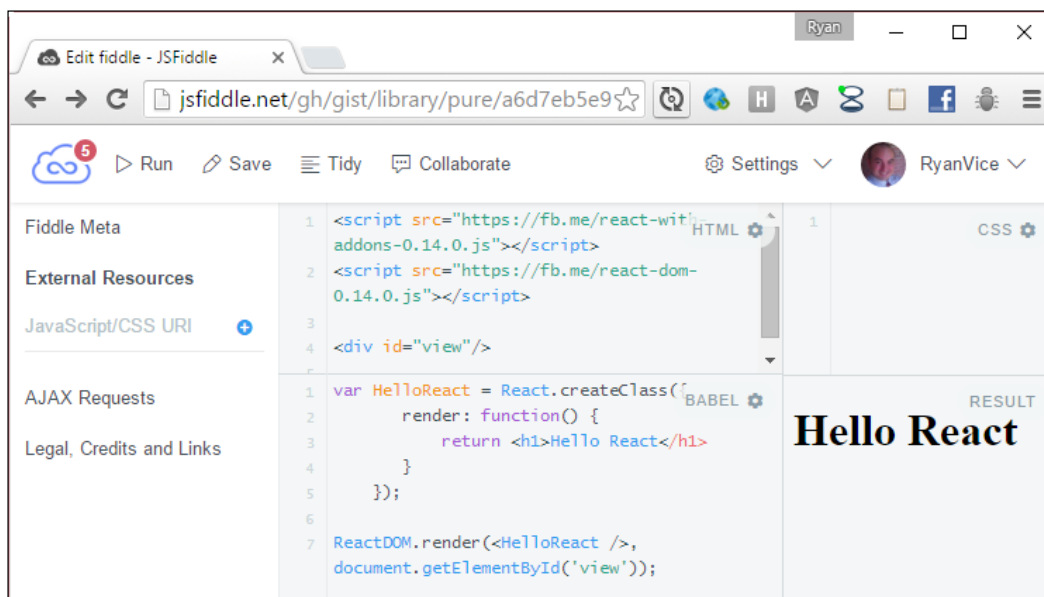
Each code sample in the first 3 chapters is hosted on Ryan's GitHub (RyanAtViceSoftware) as a Gist (<https://gist.github.com/RyanAtViceSoftware>). A Gist is a small Git repository that is meant to store only a few files and they are a perfect fit for the kind of small example we will be looking at in the first three chapters.

Similarly, the *Chapter 10, Animation in React* animation examples can be found on Adam's GitHub Gists (<https://gist.github.com/digitalicarus>).

How to run the code

After the first few examples, we will take advantage of the integration that exists between JsFiddle (<http://JsFiddle.net/>), a JavaScript online sandbox, and GitHub Gists. The integration allows for each Gist referenced in the text to be opened as a "Fiddle" and run live in a browser. Below is a screen shot of a Fiddle.

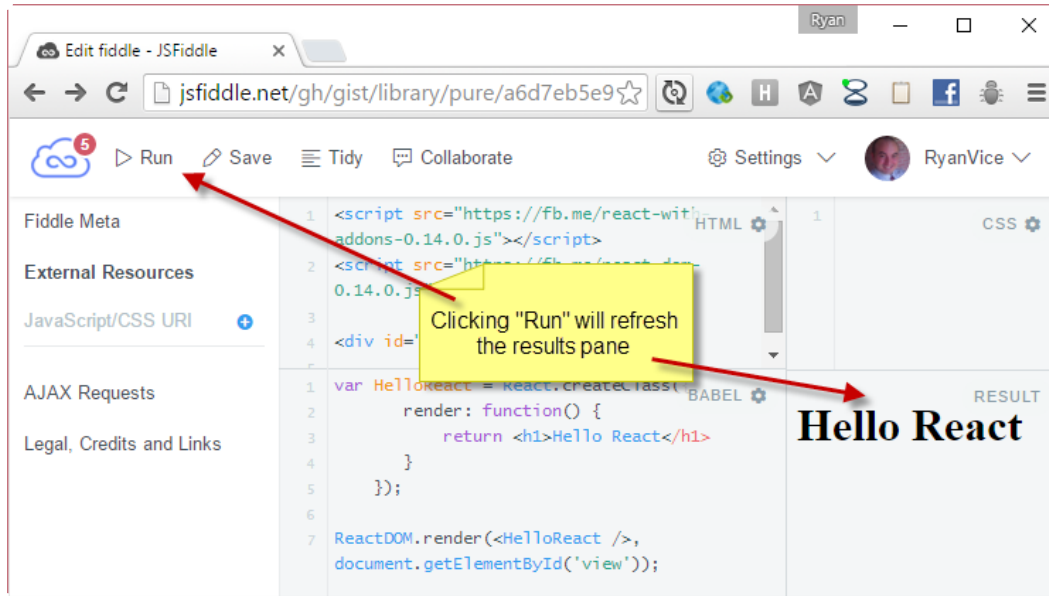
[ Fiddle: <http://j.mp/MasteringReact-1-2-1-Fiddle>]



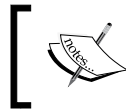
As you can see, there are four sections to a JsFiddle:

1. HTML: The HTML markup that will be used to generate the result view.
2. CSS: The CSS markup that will be used to generate the result view.
3. Babel: The JavaScript that will be run in the result view. Note that Babel is a JavaScript compiler will compile JSX to JavaScript.
4. Result: The results of linking the HTML to the CSS and JavaScript and running them all together.

I encourage you to open each fiddle as you follow along and run the fiddle by clicking the **Run** button shown as follows:



Once you have opened a code sample, play around with the code a bit to make sure you understand how everything works.



We will cover some of the basics of using JsFiddle as we go along but would encourage you to take a look at the documentation if you haven't used JsFiddle before (<http://doc.jsfiddle.net/>).

JsFiddle is designed to allow for experimenting with code and you will find it's a great tool to allow for you to quickly learn the concepts presented here in an interactive fashion.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Introduction to React

This book is an advanced book on React but we wanted to provide a primer on the basics so that this book could be both comprehensive and accessible. We will not spend a lot of time on all of the subtleties of each technique that we will look at here. We will instead look at concise samples that illustrate the tools and techniques that we are covering. We will also have links to where you can easily access and run the code samples as you follow along.

In this chapter, we will be covering the following concepts:

- About the code samples
- Hello World sample in React
- JSX
- props
- state

Hello React

For the first example we will create a fully working **Hello World**-style example using React by undertaking the following steps:

1. Create a new HTML file and call it `hello-react.html`.
2. Paste the following code in `hello-react.html`:

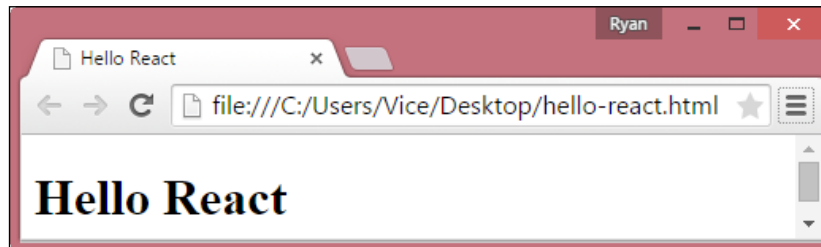
```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Hello React</title>
  <script src="https://fb.me/react-with-addons-0.14.0.js">
```

```
</script>
  <script src="https://fb.me/react-dom-0.14.0.js">
</script>
</head>
<body>

<script>
  var HelloReact = React.createClass({
    render: function() {
      return React.DOM.h1(null, 'Hello React');
    }
  });

  ReactDOM.render(React.createElement(HelloReact), document.
body);
</script>
</body>
</html>
```

3. Open `HelloReactJs.html` in a browser and you should see something similar to the image that follows:



React is a component-based framework that allows creating composable view components. The first thing that we have to do to be able to create a component in React is to include the React source as shown below:

```
<script src="https://fb.me/react-with-addons-0.14.0.js"></script>
<script src="https://fb.me/react-dom-0.14.0.js"></script>
```



As of version 0.13.0, React splits its API into two files. Now, there is one file that contains the browser-specific DOM code and another file that contains the rest of React's API. This was done because React is being used in more and more places and currently is used by React Native to build mobile applications. It can also be used to build Windows and Mac desktop applications using platforms such as Electron..

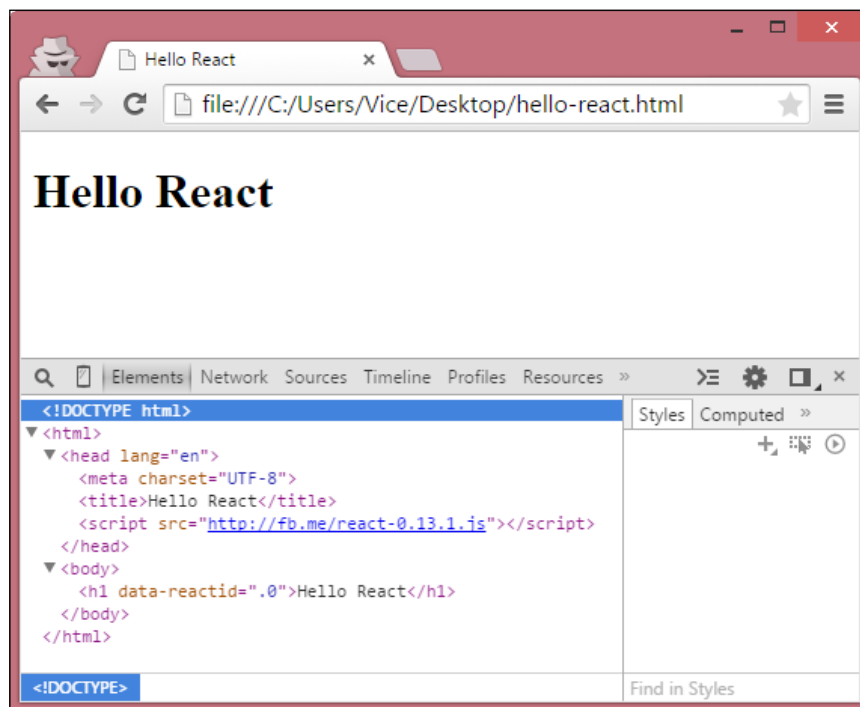
This will bring in version v0.14.0 of React, and then the code in the script block that is shown in the following code creates a `ReactDOM` parameter of type `h1` and sets children to be the string `Hello React`:

```
React.createElement('h1', null, 'Hello React');
```

We pass the `ReactDOM` parameter that is created by this call as the first argument to the `ReactDOM.render()` method shown as follows:

```
ReactDOM.render(  
  React.createElement('h1', null, 'Hello React'),  
  document.body);
```

React's render method is taking a `ReactDOM` parameter as its first argument and a `document.body` DOM element as its second argument. The render method will then write the HTML generated by the first argument, the `ReactDOM`, as a child of the second argument, the `document.body` DOM **element**. You can see the results in Chrome's Elements tab, as shown in the following screenshot:



As you can see, we now have an `h1` element in our DOM that contains the string `Hello React` as a child.

However, you might be wondering how this is component based and you'd be right in being skeptical as we haven't created a component yet. Components are one of the things that really makes React a powerful and flexible framework so let's see what our example looks like if we update it to create a component. To do this, update the script in `hello-react.html` as shown in the following code, and refresh the browser to verify that our program still works the same:

```
var HelloReact = React.createClass({
  render: function() {
    return React.DOM.h1(null, 'Hello React');
  }
});

ReactDOM.render(React.createElement(HelloReact), document.body);
```

Now we are creating a `HelloReact` JavaScript variable and assigning it a newly created React component that is created using the `React.createClass()` method as shown below:

```
var HelloReact = React.createClass({
  render: function() {
    return React.DOM.h1(null, 'Hello React');
  }
});
```

The `createClass()` method takes an object that must specify a render method. The render method is responsible for returning a single `ReactClass` to be rendered. Here we are creating a `ReactClass` that represents an `h1` DOM element that contains the string `Hello React`.

Note that we are now using the `React.DOM` API to create our `h1 ReactElement` instance. The `React.DOM` API provides convenience methods for creating common HTML elements and internally calls the `React.createElement()` method and passes the needed parameters for us. It may seem odd that the `React.DOM` part of the API was not moved to `ReactDOM` like `ReactDOM.Render`. However, it appears that the React team has decided that the HTML semantics and UI widgets are part of their universal approach to building UIs, while the actual rendering is platform specific. Here's an excerpt from the React documentation about the restructuring.



"As we look at packages such as `react-native`, `react-art`, `react-canvas`, and `react-three`, it has become clear that the beauty and essence of React has nothing to do with browsers or the DOM.

To make this more clear and to make it easier to build more environments that React can render, we're splitting the main React package into two: `react` and `react-dom`. This paves the way to writing components that can be shared between the Web version of React and React Native. We don't expect all the code in an app to be shared, but we want to be able to share the components that do behave in the same manner across platforms.

The React package contains `React.createElement`, `.createClass`, `.Component`, `.PropTypes`, `.Children`, and the other helpers related to elements and component classes. We think of these as isomorphic or universal helpers that you need to build components."

As shown in the following code, we then call the `ReactDOM.render()` method and pass the results of calling `React.createElement(HelloReact)`:

```
ReactDOM.render(React.createElement(HelloReact), document.body);
```

Now we've updated our code to create a React component and, as we will see, these components will offer a lot of power and flexibility to our web applications.



Source code: <http://bit.ly/MasteringReact-1-1-Gist>

JSX

In order to make its component API easier to use, React has its own syntax called JSX, which combines JavaScript and HTML. Let's take a look at updating our sample code to use JSX by copying the following code into `hello-react.html` and refreshing our browser:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Hello React</title>
  <script src="https://fb.me/react-with-addons-0.14.0.js"></script>
  <script src="https://fb.me/react-dom-0.14.0.js"></script>
  <script src="http://fb.me/JSXTransformer-0.13.1.js"></script>
</head>
<body>

<script>
  var HelloReact = React.createClass({
    render: function() {
      return React.DOM.h1(null, 'Hello React');
    }
  });

  ReactDOM.render(React.createElement(HelloReact), document.body);
</script>
</body>
</html>
```



Note that we are assigning a type of "text/jsx" to our script tag in the preceding sample.

How it works

The first thing we had to do to make this work was to add a reference to the JSX Transformer file shown in the following code:

```
<script src="http://fb.me/JSXTransformer-0.13.1.js"></script>
```



Note that using an in browser transformer is only recommended for testing. We will look into more appropriate ways to transform the JSX code to JavaScript in later chapters. Also note that as of 0.14 React recommends using Babel for doing JSX transforms and has deprecated it's JSX Transformer.

Next we updated our component creation code as shown below:

```
var HelloReact = React.createClass({
  render: function() {
    return <h1>Hello React</h1>;
  }
});
```

Now instead of calling into the React API to define our components DOM structure, we just write the desired DOM structure inline within our return statement.



As we will see later, we can even reference other react components as HTML elements!

And now we can just pass the HTML tag version of our component into the `React.render()` method to have it rendered to the DOM:

```
ReactDOM.render(<HelloReact />, document.body);
```

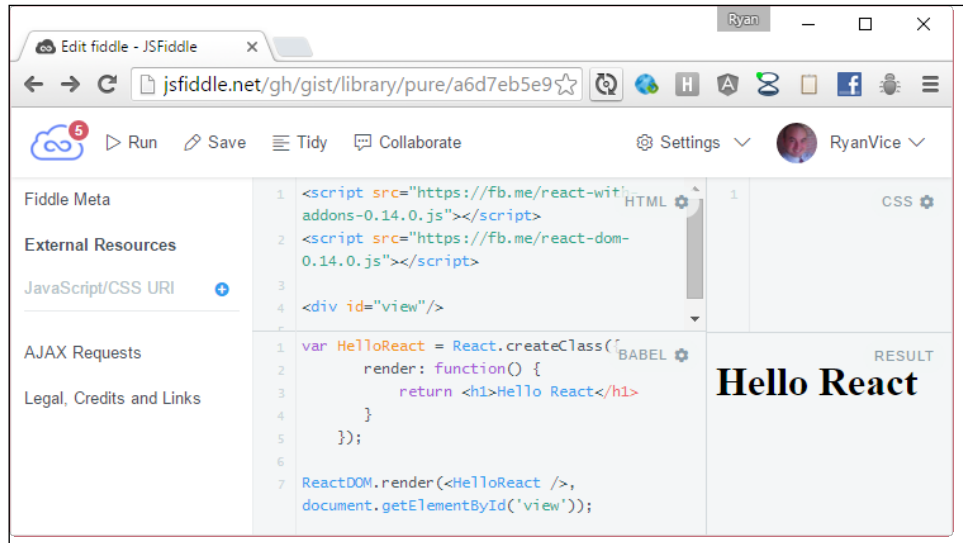
Notice how we no longer need to call `React.createElement()`. This is because the JSX compiler will make the call for us.



Source code: <http://j.mp/MasteringReact-1-2-Gist>

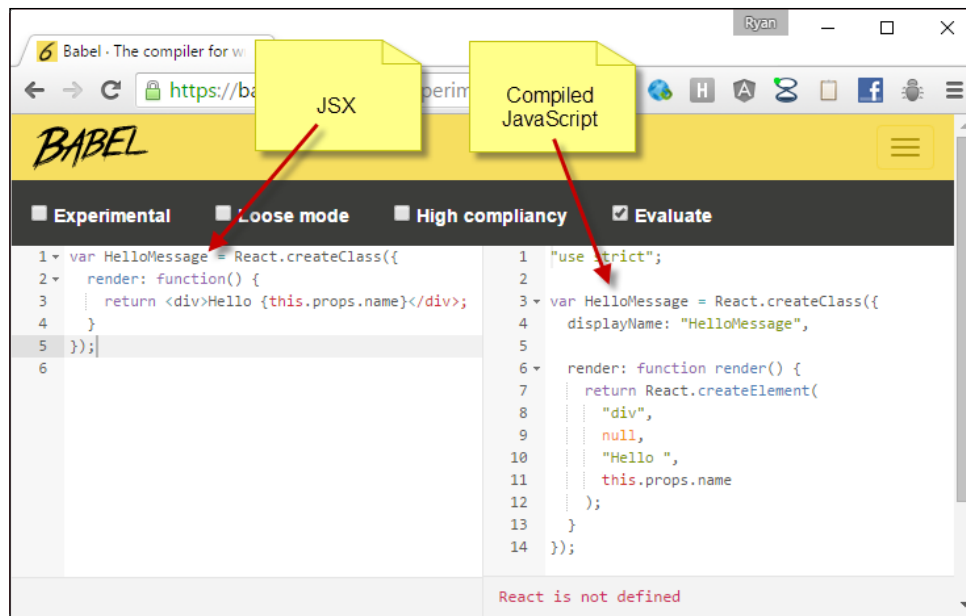
Now that we have seen how to set up a page to host the React application, we will start using JsFiddle to look at the examples. The example we just looked at is in the fiddle (<http://j.mp/MasteringReact-1-2-1-Fiddle>).

Follow that link and click the **Run** button to run the code. You should see the following output:



Decompiling JSX

Babel has created a tool that allows seeing the decompiled JavaScript that would be created during JSX transformation. The Babel tool is shown in the following screenshot, and you can find the tool at <https://babeljs.io/repl/>:



The JSX Compiler has two code windows. The code window on the left is where you can put JSX, and the results of compiling it to React's API is shown in the code window on the right.

Structure of render result

There are a few things we should take note of about using JSX to create the `ReactDOM` parameter that you return from your component's `render()` method. Let's say that we want to write Hello React code in two lines. You might be tempted to structure your code like the code that follows:

```
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello React</div> // error
    <div>How are you?</div>;
  }
});
```

However, if we paste this code into the JSX Compiler we will see the following error:

```
Error: Parse Error: Line 1: Adjacent JSX elements must be wrapped in
an enclosing tag
```

The error here is indicating that we need to wrap our JSX in a single root element shown in the following code:

```
var HelloMessage = React.createClass({
  render: function() {
    return <div> // works
      <div>Hello React</div>
      <div>How are you?</div>
    </div>;
  }
});
```

If we paste this in the JSX compiler we won't see any errors and, instead, we will see the compiled React API code as shown below:

```
var HelloMessage = React.createClass({displayName: "HelloMessage",
  render: function() {
    return React.createElement("div", null, " // works",
      React.createElement("div", null, "Hello React"),
      React.createElement("div", null, "How are you?")
    );
  }
});
```



Source code: <http://j.mp/MasteringReact-1-3-Gist>



Looking at the code generated by the compiler we can see why our previous code structure wouldn't work as our JavaScript return statement can only return a single `ReactDOMElement`. Because of this we have to return a single root node and can't return multiple adjacent sibling nodes as the error pointed out. However, now that we have added a root node, everything works and we can see in the compiled output that we are creating a `div` that contains two `<div>` tags and returning a single `ReactDOMElement`.


Now we might want to format our code better for readability and we could be tempted to do something, as shown in the following code:

```
var HelloReact = React.createClass({
  render: function() {
    return {
      <div>
        <div>Hello React</div>
        <div>How are you?</div>
      </div>
    };
  }
});
```

This will not throw an error in the JSX Compiler but it will throw the runtime error shown in the following code:

```
Uncaught Error: Invariant Violation: HelloReact.render(): A valid
ReactComponent must be returned. You may have returned undefined, an
array or some other invalid object.
```

[




Source code: <http://j.mp/MasteringReact-1-4-Gist>
 Fiddle: <http://j.mp/MasteringReact-1-4-Fiddle>
 You will need to open the developer tools in your browser and look in the console output to see the error.

]

All is not lost though because simply wrapping our JSX in parenthesis will allow us more flexibility when formatting our code. The following code sample uses this approach and we will no longer get an error when we run it. I like using this approach as it lets me keep my code nice and neatly formatted:

```
var HelloReact = React.createClass({
  render: function() {
    return (
      <div>
        <div>Hello React</div>
        <div>How are you?</div>
      </div>
    );
  }
});
```

[



Source code: <http://j.mp/MasteringReact-1-5-Gist>
 Fiddle: <http://j.mp/MasteringReact-1-5-Fiddle>

]

props

So far our components have not been configurable. Clearly we would want to be able to define a component that can take arguments via the component's HTML element attributes. The following code shows how we can do this:

```
var HelloReact = React.createClass({
  render: function() {
    return (
      <div>
        <div>Hello React</div>
```

```
        <div>{this.props.message}</div>
      </div>
    );
  }
});

ReactDOM.render(
  <HelloReact message='Message from props' />,
  document.getElementById('view')
);
```



Note that props are immutable and not dynamic. To change the value of a prop requires rerendering the component.

How it works

React components have props collections that will be populated from the component's HTML attributes that the component is declared with. So, in the following code, we are setting the message attribute of the prop collection to Message from props:

```
React.render(
  <HelloReact message='Message from props' />,
  document.getElementById('view')
);
```

Now, in our component's definition, we can access the value that message was set to by using `this.props.message`.

Note that if we want to access this in JSX markup, we need to surround the code with brackets, shown in the following code:

```
var HelloReact = React.createClass({
  render: function() {
    return (
      <div>
        <div>Hello React</div>
        <div>{this.props.message}</div>
      </div>
    );
  }
});
```



Source code: <http://j.mp/Mastering-React-1-6-Gist>

Fiddle: <http://j.mp/Mastering-React-1-6-Fiddle>

We can also copy prop values to local variables as shown in the following code, and then reference the local variables in our JSX markup shown in the following code:

```
var HelloReact = React.createClass({
  render: function() {

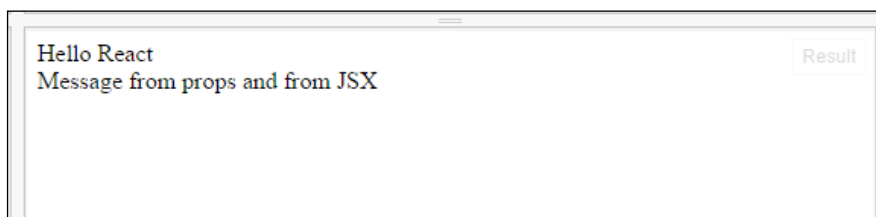
    var localMessage = this.props.message;

    return (
      <div>
        <div>Hello React</div>
        <div>{localMessage}</div>
      </div>
    );
  }
});
```

We can use any valid JavaScript expression between the brackets in our JSX so we could update the previous example shown in the following code:

```
{localMessage + ' and from JSX'}
```

This will concatenate the value contained in the `localMessage` variable with the string 'and from JSX', resulting in the output shown in the following screenshot:



propTypes

We would also like to be able to validate our props and we can do some basic validations using `propTypes` as shown in the following code:

```
var HelloReact = React.createClass({
  propTypes: {
    message: React.PropTypes.string,
```

```
    number: React.PropTypes.number,
    requiredString: React.PropTypes.string.isRequired
  },
  render: function() {
    return (
      <div>
        <div>Hello React</div>
        <div>{this.props.message}</div>
      </div>
    );
  }
});

ReactDOM.render(
  <HelloReact message='How are you' number='not a number' />,
  document.getElementById('view'));
```

Now, if we run this code and look at the warnings in the console of our browser's debug tools we will see the warnings shown in the following code. These warnings indicate that we have two invalid props:

```
Warning: Failed propType: Invalid prop `number` of type `string`
supplied to `HelloReact`, expected `number`.
```

```
Warning: Failed propType: Required prop `requiredString` was not
specified in `HelloReact`.
```

By defining the `propTypes` property of the component's class we were able to configure validations to enforce that our message prop is a string, our number is a number, and our `requiredString` is a string that is required. Then, when we define our component as shown in the following code, we violate some of our validation rules and get appropriate warning messages:

```
React.render(
  <HelloReact message='How are you' number='not a number' />,
  document.getElementById('view'));
```



Source code: <http://j.mp/Mastering-React-1-7-Gist>
Fiddle: <http://j.mp/Mastering-React-1-7-Fiddle>

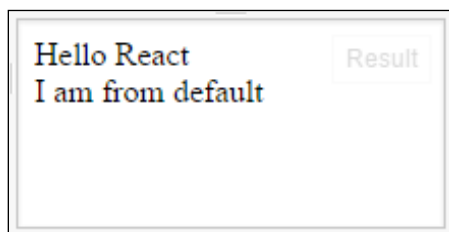
getDefaultProps

We can also provide default property values that will get used if an attribute isn't specified in the HTML markup declaring our component. The following code shows how we can create a default value for the message prop by defining a `getDefaultProps` method:

```
var HelloReact = React.createClass({
  getDefaultProps: function() {
    return {
      message: 'I am from default'
    };
  },
  render: function() {
    return (
      <div>
        <div>Hello React</div>
        <div>{this.props.message}</div>
      </div>
    );
  }
});

ReactDOM.render(
  <HelloReact />,
  document.getElementById('view'));
```

Now when we run the code we will see the output as shown below:



Source code: <http://j.mp/Mastering-React-1-8-Gist>
 Fiddle: <http://j.mp/Mastering-React-1-8-Fiddle>

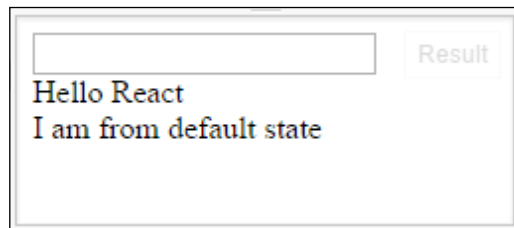
state

So far our components are static and don't allow for dynamic behavior, which isn't very interesting and we would need to be able to make our components dynamic for them to be useful. React components have the concept of state to allow for dynamic behavior. The following code shows how we can use state to create some simple dynamic behavior:

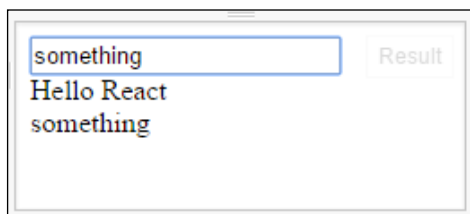
```
var HelloReact = React.createClass({
  getInitialState: function() {
    return {
      message: 'I am from default state'
    };
  },
  updateMessage: function(e) {
    this.setState({message: e.target.value});
  },
  render: function() {
    return (
      <div>
        <input type='text' onChange={this.updateMessage}/>
        <div>Hello React</div>
        <div>{this.state.message}</div>
      </div>
    );
  }
});

ReactDOM.render(
  <HelloReact />,
  document.getElementById('view'));
```

Go ahead and run the code and you will see the output as shown below:



Now type something into the text box and you will see that the **I am from default state** will change to something dynamically as you type, as shown in the following screenshot:



Source code: <http://j.mp/Mastering-React-1-9-Gist>
 Fiddle: <http://j.mp/Mastering-React-1-9-Fiddle>

How it works

We are seeing a few new concepts here. First, we are wiring up our dynamic property from the state collection using the following code:

```
<div>{this.state.message}</div>
```

Now, anytime `this.state.message` changes we will see that change reflected in the browser because React will rerender our component. Next, we need to wire up our UI to allow us to update `this.state.message` in response to user input. To do this we will take advantage of the synthetic events' capabilities of React's virtual DOM.



The virtual DOM is described on React's website, as shown below:

"React abstracts away the DOM from you, giving a simpler programming model and better performance."

The virtual DOM exposes synthetic events and you can learn more about React's synthetic events here:

<https://facebook.github.io/react/docs/events.html>


We use the following code to subscribe the `this.updateMessage` method to the `onChange` synthetic event:

```
<input type='text' onChange={this.updateMessage}/>
```

Now, anytime we change the text in our textbox, `this.updateMessage` will be called, which is shown in the following code:

```
updateMessage: function(e) {  
    this.setState({message: e.target.value});  
},
```

Here we are capturing the synthetic event's argument, `e`, and then calling `this.setState` and passing in a JavaScript object with a `message` property that is set to the value of `e.target` (our text box). The `this.setState()` method is added to our React component by React and it will update the component's state with the properties that are defined in the JSON object that is passed in, and then rerender the component using the new state. Components in React are meant to be state machines and changing the state transitions the UI from one visual state to another.

 Note that `this.setState()` method will merge the existing `this.state` with the object that is passed in. This means that you only need to specify the properties that you want to update as it will not delete any properties that are not defined in the JSON object, which are currently defined on `this.state`.

The only remaining detail in the code sample is how we are able to declare a default state by defining a `getInitialState()` method:

```
getInitialState: function() {  
    return {  
        message: 'I am from default state'  
    };  
},
```

The object we return from the `getInitialState()` method will be used to initialize our component's state.

Summary

In this chapter we looked at the most basic and fundamental concepts in React. We saw how to create components, how to pass in data to them using props, and how to make them dynamic using state.

In the next chapter we will look at component composition and component lifecycle.

2

Component Composition and Lifecycle

Now that we've covered the basics of creating components let's look at how we can compose our components to make more complex views. We will also look at how we can hook into a component's lifecycle events so that we can execute code before and after our component renders as well as prevent rendering based on incoming changes to state and props.

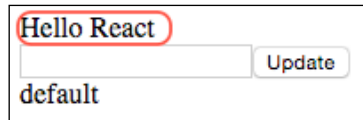
In this chapter we will be covering the following concepts:

- Component composition
 - How to compose simple components
 - Composing components with behavior
 - How to access child components
- Component lifecycle
 - Mounting and unmounting events
 - Updating events

How to compose simple components

One of the best things about React is that it is component based allowing us to easily compose our application from small autonomous components. Let's break up our Hello React application into smaller components to see how we can take advantage of React's component system.

Let's start by separating our app into two components. We will make the Hello React title its own component called `HelloMessage` as shown below:



The code to create our `HelloMessage` component is shown in the following code:

```
var HelloMessage = React.createClass({
  render: function() {
    return <div>{this.props.message}</div>;
  }
});
```

This code defines a new component that simply writes out `this.props.message` in a `<div>` tag. Next let's update the rest of our code to use this new component to write out `Hello React` as shown in the following code:


```
var HelloReact = React.createClass({
  getInitialState: function() {
    return { message: 'default' }
  },
  updateMessage: function () {
    console.info('updateMessage');
    this.setState({
      message: this.refs.messageTextBox.value
    });
  },
  render: function() {
    return (
      <div>
        <HelloMessage message='Hello React'></HelloMessage>
        <input type='text' ref='messageTextBox' />
        <button onClick={this.updateMessage}>Update</button>
        <div>{this.state.message}</div>
      </div>
    );
  }
});

ReactDOM.render(
  <HelloReact/>,
  document.getElementById('view'));
```

In this code we are simply referencing our `HelloMessage` component and then setting `message` to `Hello React` as shown again in the following code:

```
<HelloMessage message='Hello React'></HelloMessage>
```

This example is simple and contrived but it shows how easy it is to start breaking our application into smaller reusable chunks. This allows us to write our application by creating and using a **Domain Specific Language (DSL)**. With our new DSL our JSX markup is made up of readable custom tags like `HelloMessage` instead of blocks of HTML markup. This will allow us to improve the readability and organization of our code dramatically.

[ Source code: <http://j.mp/Mastering-React-2-1-Gist>
Fiddle: <http://j.mp/Mastering-React-2-1-Fiddle>]

Composing components with behavior

Now let's make things more interesting by updating our app to be composed of components that have behavior. We are going to create a view with a form that allows for updating our `HelloMessage` as shown in the following screenshot:



If we click an **Edit** button then that button will change to an **Update** button and the associated text input box will be enabled. This will allow us to set the **First Name** or **Last Name** that is displayed in our `HelloMessage` component. After setting a **First Name** or **Last Name** we can then click the associated **Update** button and the `HelloMessage` will be updated to display the new first and last name. The preceding image shows what the component looks like after putting **Ryan** for **First Name** and **Vice** for **Last Name** and clicking an **Update** button.

Let's take a look at the following code:

```
var HelloMessage = React.createClass({  
  render: function() {  
    return <h2>{this.props.message}</h2>;  
  }  
});
```

```
    }
  });

  var TextBox = React.createClass({
    getInitialState: function() {
      return { isEditing: false }
    },
    update: function() {
      this.props.update(this.refs.messageTextBox.value);
      this.setState(
        {
          isEditing: false
        }
      );
    },
    edit: function() {
      this.setState({ isEditing: true });
    },
    render: function() {
      return (
        <div>
          {this.props.label}<br/>
          <input type='text' ref='messageTextBox'
disabled={!this.state.isEditing}/>
          {
            this.state.isEditing ?
              <button onClick={this.update}>Update</button>
            :
              <button onClick={this.edit}>Edit</button>
          }
        </div>
      );
    }
  });

  var HelloReact = React.createClass({
    getInitialState: function () {
      return { firstName: '', lastName: '' }
    },
    update: function(key, value) {
      var newState = {};
      newState[key] = value;
      this.setState(newState);
    },
    render: function() {
```


```

    return (
      <div>
        <HelloMessage
          message={'Hello ' + this.state.firstName + ' ' +
this.state.lastName}>
        </HelloMessage>
        <TextBox label='First Name' update={this.update.
bind(this, 'firstName')}>
        </TextBox>
        <TextBox label='Last Name'
          update={this.update.bind(this, 'lastName')}>
        </TextBox>
      </div>
    );
  }
});

ReactDOM.render(
  <HelloReact/>,
  document.getElementById('view'));

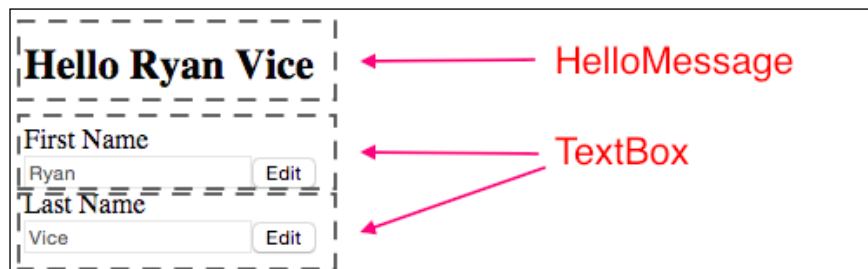
```

Run the code and get a feel for how it works.

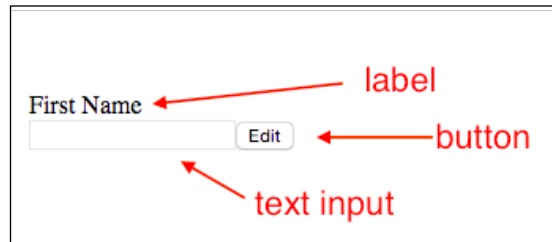

 Source code: <http://j.mp/Mastering-React-2-2-Gist>
 Fiddle: <http://j.mp/Mastering-React-2-2-Fiddle>

How it works

We have divided our view into the components shown in the following screenshot:



The `HelloMessage` component is the same one that we created in the previous example but now we've added a new `TextBox` component to the mix. Each `TextBox` component has a label, text input and button as shown in the following screenshot:



We declare two instances of our `TextBox` component in our `HelloReact` component's render method shown in the following code:

```
var HelloReact = React.createClass({
  getInitialState: function () {
    return { firstName: '', lastName: '' }
  },
  update: function(key, value) {
    var newState = {};
    newState[key] = value;
    this.setState(newState);
  },
  render: function() {
    return (
      <div>
        <HelloMessage
          message={'Hello '
+ this.state.firstName + ' '
+ this.state.lastName}>
          </HelloMessage>
          <TextBox label='First Name'
update={this.update.bind(null, 'firstName')}>
          </TextBox>
          <TextBox label='Last Name'
update={this.update.bind(null, 'lastName')}>
          </TextBox>
        </div>
      );
    }
  });
```

Here we are creating two `TextBox` component instances and setting their label and `update` properties. The `update` property needs to be set to the callback function that will be called when the input component's `onChange` event fires (we will look more closely at this in the following code). We are setting the `update` property to the new method created by calling Javascript's `bind` method on `this.update` shown in the following code:

```
update={this.update.bind(null, 'lastName')}
```

If you are not familiar with JavaScript's `bind` method it will return a new method that allows us to do two things. First, it allows us to set the function's context which is the value of the `this` variable in the function's scope. Second, it allows us to curry the method's arguments, which allows us to prepend arguments to the argument array that will be used to call the method when it's invoked. We are passing `null` for the first parameter as we are not interested in changing the functions context and this will result in `this.update` being called in the context of our `HelloReact` component instance meaning that `this.setState` will refer to `HelloReact.setState` which is what we want. More interesting to our goal, we are using JavaScript's `bind` method to curry the `this.update` function's arguments. Doing this allows us to provide the `this.update` method a key argument from `HelloReact` render's method. This technique allows us to configure how our callback method will be called. Here we are using JavaScript's `bind` method to let the consuming component pass the key argument when it invokes the `update` callback method in response to an `onChange` synthetic event.

In the `HelloReact` component's `update` method, as shown in the following code, we are expecting to be passed a key and value. As we just discussed, the key was sent via the `bind` method and we will use the key along with the value that was passed from the react synthetic event to update the `HelloReact` component's state. We update the state by first creating a new object called `newState`. Then we use JavaScript's index operator on the `newState` object with our key to create a new property on the `newState` object using JavaScript's index operator. We then assign value to the new property that was created on the `newState` object. Finally we call `this.setState` and pass in `newState` which will merge `newState` with `this.state` causing our component to rerender with the updated value.

```
update: function(key, value) {  
  var newState = {};  
  newState[key] = value;  
  this.setState(newState);  
},
```

After updating our state our `HelloMessage.message` property will be updated shown in the following code:

```
{'Hello ' + this.state.firstName + ' ' + this.state.lastName}
```

This will make it so that our `HelloMessage` will get rerendered and updated every time the `HelloReact` components state is updated from the call to `this.setState` method from the `this.update` method.

Next let's look at the `TextBox` component that will call the `HelloReact` component's update method. The `TextBox` component's code is shown below:

```
var TextBox = React.createClass({
  getInitialState: function() {
    return { isEditing: false }
  },
  update: function() {
    this.props.update(this.refs.messageTextBox.value);
    this.setState(
      {
        isEditing: false
      }
    );
  }
});
```

Here we first pass the value in our text input via `this.refs.messageTextBox.value` into the `this.props.update` method. We then update our state so that `isEditing` is false. As we saw in the preceding code, we used JavaScript's `bind` method to wire up the `TextBox.update` property. Now when we call `this.props.update(value)` this will result in the call being `this.props.update(key, value)` where `key` was assigned in the `bind` call in the `HelloReact` component's render method. The remaining code in `TextBox` deals with controlling the components enabled and disabled state and the text displayed in button shown in the following code:

```
edit: function() {
  this.setState({ isEditing: true });
},
render: function() {
  return (
    <div>
      {this.props.label}<br/>
      <input type='text'
ref='messageTextBox'
disabled={!this.state.isEditing}/>
      {
        this.state.isEditing ?
          <button onClick={this.update}>
Update
          </button>
:
          <button onClick={this.edit}>
```

```

      Edit
    </button>
      }
    </div>
  );
}
});

```

We are defining an edit method that simply sets `this.state.isEditing` to `true` by calling the `this.setState` method. Then we are defining a render method that creates a label, an input text box and a button. We are using JavaScript's ternary operator to conditionally create a different button depending on the value of `this.state.isEditing`. If `this.state.isEditing` is `true` then we create an Update button while if `this.state.isEditing` is `false` we will create an Edit button. We also set our input component's `disabled` property to `!this.state.isEditing` so that our input will be disabled when we are not editing.

Accessing a component's children

In React when we want to access the inner HTML of a component or a component that's been embedded inside of a component we can use `this.props.children`. This feature is very similar to Angular's Transclusion, WebComponent's Contents or Ember's Yield. To demonstrate this we are going to update the button from our previous example to be the Button component shown in the following code:

```

var Button = React.createClass({
  render: function() {
    return (
      <button onClick={this.props.onClick}>
        {this.props.children}
      </button>
    );
  }
});

```

What we have created here is Button component that can have its opening and closing tags wrapped around the HTML elements and React components that it wants to display within the button that it will render. To demonstrate this we will create a component for displaying glyph icons using bootstrap shown in the following code:

```

var GlyphIcon = React.createClass({
  render: function() {
    return (
      <span className={'glyphicon glyphicon-'

```

```
+ this.props.icon}>
</span>
);
}
});
```

[ Note that to make this work we updated the JsFiddle references to include a reference to Twitter's Bootstrap framework. For more information about Bootstrap see the documentation here: <http://getbootstrap.com/>]

Our `GlyphIcon` component will simplify displaying a bootstrap Glyphicon if we simply configure it by specifying the last part of the Glyphicon's style name. In the following screenshot, we have shown a few of the Glyphicon styles:



So for example we can display a pencil by specifying `pencil` for our `GlyphIcon` component's `icon` property. Next we will update our `TextBox` component to use our `GlyphIcon` class shown in the following code:

```
render: function() {
  return (
    <div>
      {this.props.label}<br/>
      <input
        type='text'
        ref='messageTextBox'
        disabled={!this.state.isEditing}/>
    </div>
  );
}
```

```

        this.state.isEditing ?
            <Button onClick={this.update}>
<GlyphIcon icon='ok' /> Update
            </Button>
            :
            <Button onClick={this.edit}>
<GlyphIcon icon='pencil' /> Edit
            </Button>
        }
    </div>
  );
}

```

In this code we are using our button component to wrap both a `GlyphIcon` component and also some text which will allow us to display buttons with both text and icon's as shown in the following screenshot:

The screenshot shows a form with the title "Hello Ryan". Below the title are two input fields. The first field is labeled "First Name" and contains the text "Ryan". To the right of this field is a button labeled "Edit" with a pencil icon. The second field is labeled "Last Name" and contains the text "Vice". To the right of this field is a button labeled "Update" with a checkmark icon.

The full code is shown below:

```

var HelloMessage = React.createClass({
  render: function() {
    return <h2>{this.props.message}</h2>;
  }
});

var Button = React.createClass({
  render: function() {
    return (
      <button onClick={this.props.onClick}>
        {this.props.children}
      </button>
    );
  }
});

var GlyphIcon = React.createClass({

```

```
    render: function() {
      return (
        <span className={'glyphicon glyphicon-'
+ this.props.icon}>
        </span>
      );
    }
  });

var TextBox = React.createClass({
  getInitialState: function() {
    return { isEditing: false, text: this.props.label };
  },
  update: function() {
    this.setState(
      {
        text: this.refs.messageTextBox.getDOMNode().value,
        isEditing: false
      });
    this.props.update();
  },
  edit: function() {
    this.setState({ isEditing: true });
  },
  render: function() {
    return (
      <div>
        {this.props.label}<br/>
        <input
type='text'
ref='messageTextBox'
disabled={!this.state.isEditing}/>
        {
          this.state.isEditing ?
            <Button onClick={this.update}>
<GlyphIcon icon='ok' /> Update
          </Button>
          :
            <Button onClick={this.edit}>
<GlyphIcon icon='pencil' /> Edit
          </Button>
        }
      </div>
    );
  }
});
```

```

    }
  });

  var HelloReact = React.createClass({
    getInitialState: function () {
      return { firstName: '', lastName: '' }
    },
    update: function () {
      this.setState({
        firstName:
          this.refs.firstName.refs.messageTextBox.getDOMNode().
value,
        lastName:
          this.refs.lastName.refs.messageTextBox.getDOMNode().
value});
    },
    render: function() {
      return (
        <div>
          <HelloMessage
            message={'Hello ' + this.state.firstName + ' ' +
this.state.lastName}>
          </HelloMessage>
          <TextBox label='First Name' ref='firstName'
            update={this.update}>
          </TextBox>
          <TextBox label='Last Name' ref='lastName'
            update={this.update}>
          </TextBox>
        </div>
      );
    }
  });

ReactDOM.render(
  <HelloReact/>,
  document.getElementById('view'));

```



Source code: <http://j.mp/Mastering-React-2-3-Gist>
 Fiddle: <http://j.mp/Mastering-React-2-3a-Fiddle>

Component lifecycle - mounting and unmounting

Components in React have a lifecycle of events that we can easily subscribe to by defining the associated methods on our component definition object. Let's go ahead and update our previous example to see this feature in action.

```
var HelloMessage = React.createClass({
  componentWillMount: function() {
    console.log('componentWillMount');
  },
  componentDidMount: function() {
    console.log('componentDidMount');
  },
  componentWillUnmount: function() {
    console.log('componentWillUnmount');
  },
  render: function() {
    console.log('render');
    return <h2>{this.props.message}</h2>;
  }
});
```

Here we have updated our `HelloMessage` component to log to the console the following three React component lifecycle events:

- `componentWillMount`: This event will be called right before a component mounts
- `componentDidMount`: This event will be called right after a component mounts
- `componentWillUnmount`: This event will be called right before a component unmounts

We are also logging our `render` method to the console so that we can see when the various lifecycle events occur relative to render.

Let's also update our `HelloReact` component to add a button that will reload our `HelloMessage` component allowing us to see what happens when it unmounts. We've added this button to the `render` method shown in the following code:

```
render: function() {
  return (
    <div>
      <HelloMessage
        message='Hello '
      />
    </div>
  );
}
```

```

+ this.state.firstName + ' '
+ this.state.lastName}>
    </HelloMessage>
    <TextBox label='First Name' ref='firstName'
      update={this.update}>
    </TextBox>
    <TextBox label='Last Name' ref='lastName'
      update={this.update}>
    </TextBox>
    <button onClick={this.reload}>Reload</button>
  </div>
);
}

```

And then let us add a reload method to our `HelloReact` component that will call `ReactDOM.unmountComponentAtNode` which will unmount our component. We then call `ReactDOM.render` to mount our component.

```

reload: function() {
  ReactDOM.unmountComponentAtNode(
    document.getElementById('view'));
  ReactDOM.render(
    <HelloReact/>,
    document.getElementById('view'));
},

```







Source code: <http://j.mp/Mastering-React-2-4a-Gist>
 Fiddle: <http://j.mp/Mastering-React-2-4a-Fiddle>





Now let's go ahead and run the code in JsFiddle and open our browsers debugging tools (*F12* in Chrome) so that we can see the console output. After running the code we see the following output:

Console	Search	Emulation	Rendering
<div> <top frame> ▼ <input type="checkbox"/> Preserve log </div>			
<div> You are using the in-browser JSX compiler. Be sure to precompile your JSX for production - http://facebook.github.io/react/docs/tooling-integration.html#jsx </div>			
componentWillMount			Inline JSX script:7
render			Inline JSX script:16
componentDidMount			Inline JSX script:10
<div> </div>			

And as we can see we get a call to `componentWillMount` right before `Render` is called and then we get a call to `componentDidMount` right after `render` is called. This gives us an opportunity to run code both before and after our `render` method. Next let's add a **First Name** and we can see what happens in the console as shown below:

Console	Search	Emulation	Rendering
		<top frame>	 <input type="checkbox"/> Preserve log
 You are using the in-browser JSX transformer. Be sure to precompile your JSX for production - http://facebook.github.io/react/docs/tooling-integration.html#jsx			
componentWillMount			Inline JSX script:7
render			Inline JSX script:16
componentDidMount			Inline JSX script:10
render			Inline JSX script:16
>			

We get one more call to `render` but `componentWillMount` and `componentDidMount` are not called because our `HelloMessage` component is already mounted and we are simply causing `React` to call the `HelloMessage.render` method. Let's set a last name and look at the console output:

Console	Search	Emulation	Rendering
		<top frame>	 <input type="checkbox"/> Preserve log
 You are using the in-browser JSX transformer. Be sure to precompile your JSX for production - http://facebook.github.io/react/docs/tooling-integration.html#jsx			
componentWillMount			Inline JSX script:7
render			Inline JSX script:16
componentDidMount			Inline JSX script:10
2 render			Inline JSX script:16
>			

Probably no surprise here but we find that `render` is called again but that none of the lifecycle events are called. Next let's click the **Reload** button and look at the following output:

Console	Search	Emulation	Rendering
<div> <div> <div></div> <div></div> </div> <div><top frame></div> <div> <div></div> <div>▼</div> <div> <input type="checkbox"/> Preserve log </div> </div> </div>			
<div> <div>⚠</div> <div> You are using the in-browser JSX transformer. Be sure to precompile your JSX for production – http://facebook.github.io/react/docs/tooling-integration.html#jsx </div> </div>			
componentWillMount			Inline JSX script:7
render			Inline JSX script:16
componentDidMount			Inline JSX script:10
2 render			Inline JSX script:16
componentWillUnmount			Inline JSX script:13
componentWillMount			Inline JSX script:7
render			Inline JSX script:16
componentDidMount			Inline JSX script:10

Now we see that `componentWillUnmount` is called as our component is unmounted and then we repeat the same sequence we saw earlier as the component is mounted again.

Component lifecycle – updating events

There are also events that will allow us to execute code relative to when our component's state and properties get updated. To demonstrate this we will look at the sample application shown below:



This is an extremely contrived example that is intended to help us see updating events in action. This application has two buttons:

- **Like** button: This button will increase the like count
- **Unlike** button: This button will decrease the like count

The application also has the following features:

- It displays a total count of likes
- It has a `GlyphIcon` component that will show an up arrow if the like count is increasing or a down arrow if the like count is decreasing
- It will not update the view until after we have two or more likes

Let's take a look at how we can implement these features by taking advantage of the updating lifecycle events as shown in the following code:

```
var Button = React.createClass({
  render() {
    return (
      <button onClick={this.props.onClick}>
        {this.props.children}
      </button>
    );
  }
});

var GlyphIcon = React.createClass({
  render() {
    return (
      <span className={'glyphicon glyphicon-'
        + this.props.icon}>
      </span>
    );
  }
});

var HelloReact = React.createClass({
  getDefaultProps() {
    return {likes: 0};
  },
  getInitialState() {
    return {isIncreasing: false};
  },
  componentWillReceiveProps(nextProps) {
    this._logPropsAndState('componentWillReceiveProps()');
    console.log('nextProps.likes: ' + nextProps.likes);

    this.setState({
      isIncreasing: nextProps.likes > this.props.likes
    });
  }
});
```

```

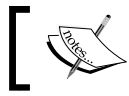
    },
    shouldComponentUpdate(nextProps, nextState) {
      this._logPropsAndState('shouldComponentUpdate()');
      console.log(
        'nextProps.likes: ',
        nextProps.likes,
        ' nextState.isIncreasing: ',
        nextState.isIncreasing);
      return nextProps.likes > 1;
    },
    componentDidUpdate(prevProps, prevState) {
      this._logPropsAndState('componentDidUpdate');
      console.log(
        'prevProps.likes: ',
        prevProps.likes,
        ' prevState.isIncreasing:',
        prevState.isIncreasing);
      console.log('componentDidUpdate() gives an opportunity to
        execute code after react is finished updating the DOM.');
```

```

    },
    _logPropsAndState(callingFunction) {
      console.log('=> ' + callingFunction);
      console.log('this.props.likes: ' + this.props.likes);
      console.log('this.state.isIncreasing: '
+ this.state.isIncreasing);
    },
    like() {
      this.setProps({likes: this.props.likes+1});
    },
    unlike() {
      this.setProps({likes: this.props.likes-1});
    },
    render() {
      this._logPropsAndState("render()");
      return (
        <div>
          <Button onClick={this.like}>
            <GlyphIcon icon='thumbs-up' /> Like
          </Button>
          <Button onClick={this.unlike}>
            <GlyphIcon icon='thumbs-down' /> Unlike
          </Button>
          <br/>
          Likes {this.props.likes}
        </div>
      );
    }
  }
);
```

```
        <GlyphIcon icon={
  (this.state.isIncreasing)
    ? 'circle-arrow-up' : 'circle-arrow-down'}/>
      </div>
    );
  }
});

ReactDOM.render(
  <HelloReact/>,
  document.getElementById('view'));
```



Source code: <http://j.mp/Mastering-React-2-5a-Gist>
Fiddle: <http://j.mp/Mastering-React-2-5-Fiddle>

How it works

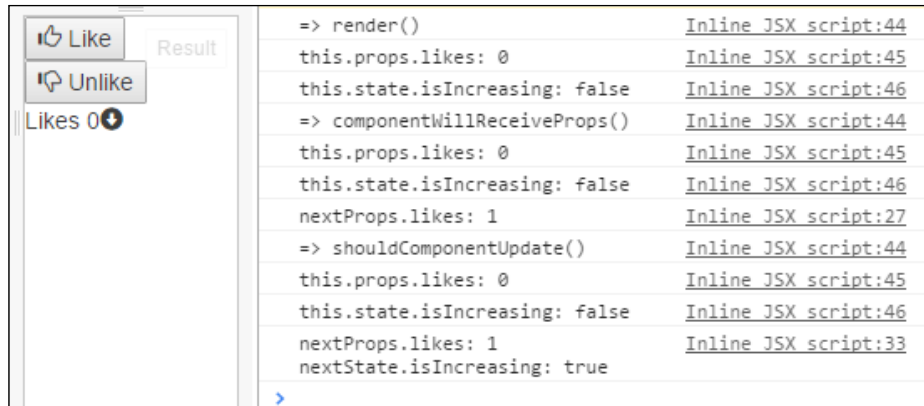
We've implemented the following component lifecycle events in our `HelloReact` component in the preceding code:

- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentDidUpdate`

We've also added a good bit of logging code that will allow us to see the state and properties of our component in these lifecycle event methods. We've added the method shown in the following code that we can call and write out the calling method name along with the current value of `this.props.likes` and `this.state.isIncreasing`.

```
_logPropsAndState(callingFunction) {
  console.log('=> ' + callingFunction);
  console.log('this.props.likes: ' + this.props.likes);
  console.log('this.state.isIncreasing: ' + this.state.isIncreasing);
},
```

Let's run the code and confirm that it works as described in the preceding code. First let's click the **Like** button. We will see that clicking the **Like** button does not have any effect on the UI because of the rule we added to the `shouldComponentUpdate` method as shown below:



Here we are looking at the console log and can see that the `componentWillReceiveProps` method is called after the `render` method but before our component's state is updated. When the `componentWillReceiveProps` method is called the props haven't changed from what we saw in the `render` method and `this.props.likes` is 0 and `this.state.isIncreasing` is false.

We also see that the `componentWillReceiveProps` is passed the future value of `this.props` in the `nextProps` argument and we can see that `nextProps.likes` is 1 as we would expect.

The `componentWillReceiveProps` method also gives us an opportunity to apply our business rule to determine if the component's like count is increasing or decreasing as shown in the following code:

```
componentWillReceiveProps(nextProps) {
  this._logPropsAndState('componentWillReceiveProps()');
  console.log('nextProps.likes: ' + nextProps.likes);

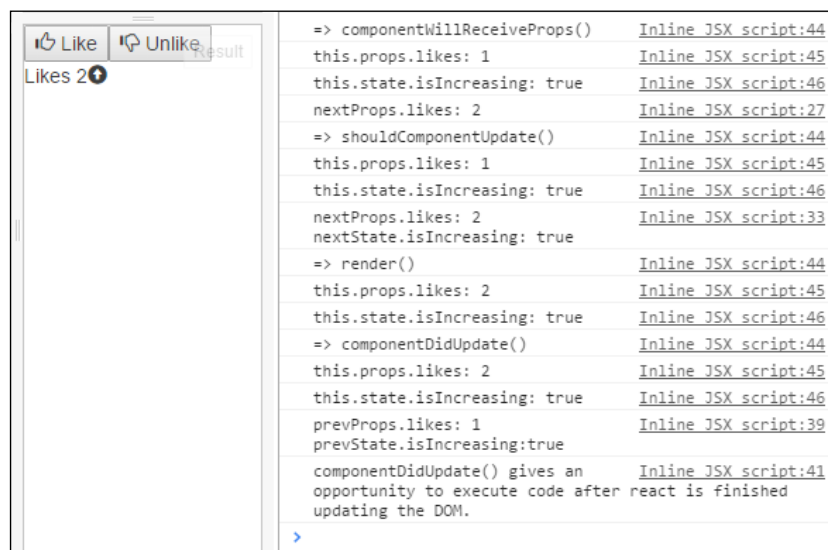
  this.setState({
    isIncreasing: nextProps.likes > this.props.likes});
}
```


We also see in the console that the `shouldComponentUpdate` method is called. The `shouldComponentUpdate` gets all the information available in the `componentWillReceiveProps` method but is also passed the future value of the `this.state` property via the `nextState` argument. Looking at the `nextState.isIncreasing` property we can see that it is true meaning that `this.state.isIncreasing` will be true when the component renders which is what we would expect. The updated value of `this.state.isIncreasing` reflects the call to `this.setState` from the `componentWillReceiveProps` method shown in the preceding code. The `shouldComponentUpdate` method also gives us an opportunity to apply our business rule that prevents the component from updating if the `this.props.likes` property is less than 2 as shown in the following code:

```
shouldComponentUpdate(nextProps, nextState) {
  this._logPropsAndState('shouldComponentUpdate()');
  console.log('nextProps.likes: '
+ nextProps.likes
          + ' nextState.isIncreasing: '
+ nextState.isIncreasing);
  return nextProps.likes > 1;
}
```

By returning `false` when evaluating `nextProps.likes > 1`, we are preventing our component from updating.

Next clear the console and click the **Like** button for a second time. We will see that the UI now updates to show two likes, an up arrow to indicate that likes are increasing as well as the log statements as shown below:

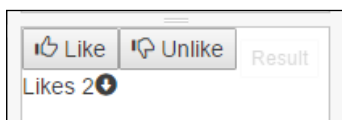


=> componentWillReceiveProps()	Inline JSX script:44
this.props.likes: 1	Inline JSX script:45
this.state.isIncreasing: true	Inline JSX script:46
nextProps.likes: 2	Inline JSX script:27
=> shouldComponentUpdate()	Inline JSX script:44
this.props.likes: 1	Inline JSX script:45
this.state.isIncreasing: true	Inline JSX script:46
nextProps.likes: 2	Inline JSX script:33
nextState.isIncreasing: true	
=> render()	Inline JSX script:44
this.props.likes: 2	Inline JSX script:45
this.state.isIncreasing: true	Inline JSX script:46
=> componentDidUpdate()	Inline JSX script:44
this.props.likes: 2	Inline JSX script:45
this.state.isIncreasing: true	Inline JSX script:46
prevProps.likes: 1	Inline JSX script:39
prevState.isIncreasing: true	
componentDidUpdate() gives an opportunity to execute code after react is finished updating the DOM.	

Here we see in our log statements that we now get a call to the `componentDidUpdate` method which gets the previous properties and previous state passed to it allowing us to execute business rules and logic after our component updates. We are not implementing any rules at this time and are simply writing out some values to demonstrate this feature. The `componentDidUpdate` method is called now because we are returning `true` from `componentShouldUpdate` when evaluating `nextProps.likes > 1`.

This expression now evaluates to `true` because the `nextProps.likes` property is 2.

Now let's click the **Like** button again followed by clicking the **Unlike** button. We will now see that our `Glyphicon` arrow is pointing down as shown in the following screenshot:



This is because we've set `this.state.isIncreasing` to `false` in our `componentWillReceiveProps` method because this expression `nextProps.likes > this.props.likes` now evaluates to `false`.

Summary

In this chapter we looked at how to compose components and how to access child components and/or the inner HTML of our components. We then looked at how to hook into the component lifecycle events to allow us to execute logic relative to mounting events and updating events.

In the next chapter will look at `mixin`'s, dynamic components, property validation and forms.

3

Dynamic Components, Mixins, Forms, and More JSX

In this chapter we are going to wrap up our coverage of the basics of React by looking at the following concepts.

- Dynamic components
- Mixins
- Forms
- Validation

We will see how dynamic components allow us to easily compose our application out of the component ecosystem we create for application by mostly using imperative JavaScript. Next we will look at how we can share functionality via the mixins feature that allows us to hook into our component's lifecycle events and fire custom logic as well as allowing us to extend our components API by extending them with new methods. After that we will look at some idiosyncrasies when working with forms in React and we will follow that up by looking at one options for validating our forms.

Dynamic components

We will often have the need to have components dynamically create child components at runtime based on the runtime state of the application. Let's take a look at how we can do this using the code that follows:

```
var UserRow = React.createClass({
  render: function() {
    return (
      <tr>
```

```
        <td>{this.props.user.userName}</td>
        <td>
            <a href={'mailto:' + this.props.user.email}>
                {this.props.user.email}
            </a>
        </td>
    </tr>
    );
}
});

var UserList = React.createClass({
  getInitialState: function() {
    return {
      users: [
        {
          id: 1,
          userName: 'RyanVice',
          email: 'ryan@vicesoftware.com'
        },
        {
          id: 2,
          userName: 'AdamHorton',
          email: 'digitalicarus@gmail.com'
        }
      ]
    };
  },
  render: function() {
    var users = this.state.users.map(
      function(user) {
        // key prevents react warning
        return (
          <UserRow user={user}
            key={user.id}/>)
      });


    return (
      <table>
        <tr>
          <th>User Name</th>
          <th>Email Address</th>
        </tr>
        {users}
      </table>
    );
  }
});
```

```

    );
  }
});

ReactDOM.render(
  <UserList/>,
  document.getElementById('view'));

```


 Source code: <http://j.mp/Mastering-React-3-1-Gist>
 Fiddle: <http://j.mp/Mastering-React-3-1-Fiddle>

Let's run this code and you will see output like shown as follows:

User Name	Email Address
RyanVice	ryan@vicesoftware.com
AdamHortondigitalcarus	adamhortondigitalcarus@gmail.com

How it works

As shown in the following screenshot, the view is made up of two components, first is `UserList` and the second is `UserRow`:



The `UserRow` component that is shown below simply writes out a table row with two columns that contain the user name and user email which are taken as input to the component via props.

```

var UserRow = React.createClass({
  render: function() {
    return (
      <tr>
        <td>{this.props.user.userName}</td>
        <td>
          <a href={'mailto:' + this.props.user.email}>
            {this.props.user.email}
          </a>
        </td>
      </tr>
    );
  }
});

```

```
        </td>
      </tr>
    );
  }
});
```

Next the `UserList` component's render function iterates over `this.state.users` by using the JavaScript `map` function and returns a `UserRow` component for each element in `this.state.users` and initializes `UserRow.user` and `UserRow.key` appropriately as shown below. We then return `{users}` in as part of our markup in the return statement and React at runtime will add our list of `UserRow` components to the rendered output as expected.

```
render: function() {
  var users = this.state.users.map(
    function(user) {
      // key prevents react warning
      return (
        <UserRow user={user}
                      key={user.id}/>)
    });

  return (
    <table>
      <tr>
        <th>User Name</th>
        <th>Email Address</th>
      </tr>
      {users}
    </table>
  );
}
```



Note that setting a `key` property on collections of child components, as we did above, allows React to uniquely identify the child components during virtual DOM rendering and this will help React more efficiently batch updates to the DOM for you. Also note that not providing a `key` property on child components will result in a warning being written to the console.

Mixins

mixins is a React feature that allows you to share cross cutting concerns with components. A mixin is simply an Object Literal that is used to add behavior to a component. It's an implementation of the decorator pattern and the mixin you create can provide implementations of React's component lifecycle events (`componentWillMount`, `componentDidMount`, and so on) and those will be called during your component's lifecycle along with the component's lifecycle methods.



Here are the details from React's documentation:

A nice feature of mixins is that if a component is using multiple mixins and several mixins define the same lifecycle method (i.e. several mixins want to do some cleanup when the component is destroyed), all of the lifecycle methods are guaranteed to be called. Methods defined on mixins run in the order mixins were listed, followed by a method call on the component.

Let's take a look at a code sample below.

```
var ReactMixin1 = {
  log: function(message) {
    console.log(message);
  },
  componentWillMount: function() {
    this.log('componentWillMount from ReactMixin1');
  }
};

var ReactMixin2 = {
  componentWillMount: function() {
    console.log('componentWillMount from ReactMixin2');
  }
};

var HelloMessage = React.createClass({
  mixins: [ReactMixin1, ReactMixin2],
  componentWillMount: function() {
    this.log('componentWillMount from HelloMessage');
  },
  render: function() {
    return <h2>{this.props.message}</h2>;
  }
});
```



```
var Button = React.createClass({
  mixins: [ReactMixin2, ReactMixin1],
  clicked: function() {
    this.log(this.props.text + ' clicked');
  },
  componentWillMount: function() {
    this.log('componentWillMount from Button');
  },
  render: function() {
    return <button onClick={this.clicked}>{this.props.text}</
button>
  }
});

var HelloReact = React.createClass({
  render: function() {
    return (
      <div>
        <HelloMessage message='Hi' />
        <Button text='OK' />
      </div>
    );
  }
});

ReactDOM.render(
  <HelloReact />,
  document.getElementById('view'));
```



Source code: <http://j.mp/Mastering-React-3-2-Gist>
Fiddle: <http://j.mp/Mastering-React-3-2-Fiddle>

Let's go ahead and run this code with the console open so we can see the console.log output. Click the **OK** button and then you will see the output shown as follows:

```
componentWillMount from ReactMixin1
componentWillMount from ReactMixin2
componentWillMount from HelloMessage
componentWillMount from ReactMixin2
componentWillMount from ReactMixin1
componentWillMount from Button
OK clicked
> |
```

As you can see we have two logs for `componentWillMount` from `ReactMixin1` and `componentWillMount` from `ReactMixin2`. One pair of calls is for our `HelloMessage` component shown as follows:

<code>componentWillMount from ReactMixin1</code>
<code>componentWillMount from ReactMixin2</code>
<code>componentWillMount from HelloMessage</code>

And one pair for our `Button` component is shown as follows:

<code>componentWillMount from ReactMixin2</code>
<code>componentWillMount from ReactMixin1</code>
<code>componentWillMount from Button</code>

Take note of how each of these pairs of calls is then followed by a call from each component's `componentWillMount` (`HelloMessage` and `Button`). Also note that for `HelloMessage` we get `ReactMixin1` and then `ReactMixin2` while for `Button` we get the opposite order. Then note that we get one message sent from the `Button` instance declared in our `HelloReact` component shown as follows:

<code>OK clicked</code>
<code>> </code>

How it works

To take advantage of mixins we have created an object literal that defines a `log` method and a `componentWillMount` method that in turn calls `this.log` and logs out `componentWillMount` from `ReactMixin1`. We then assign that object literal to a `ReactMixin1` variable shown as follows:

```
var ReactMixin1 = {
  log: function(message) {
    console.log(message);
  },
  componentWillMount: function() {
    this.log('componentWillMount from ReactMixin1');
  }
};
```

We then defined a second object literal that defines a `componentWillMount` method that calls `console.log` and passes `componentWillMount` from `ReactMixin2` shown as follows:

```
var ReactMixin2 = {
  componentWillMount: function() {
    console.log('componentWillMount from ReactMixin2');
  }
};
```

We then use our object literals, `ReactMixin1` and `ReactMixin2`, as mixins in our `HelloMessage` component by adding them to the `HelloMessage.mixins` array shown as follows:

```
var HelloMessage = React.createClass({
  mixins: [ReactMixin1, ReactMixin2],
  componentWillMount: function() {
    this.log('componentWillMount from HelloMessage');
  },
  render: function() {
    return <h2>{this.props.message}</h2>;
  }
});
```

After being added to the `HelloMessage.mixins` array our `ReactMixin1` will do two things:

- It will decorate our `HelloMessage.componentWillMount` lifecycle method by adding additional behavior. As we saw `ReactMixin1.componentWillMount` will be called followed by a call to `ReactMixin2.componentWillMount` followed by the call to `HelloMessage.componentWillMount`.
- It will make the `log` method available to the `HelloMessage` component. We are then able to use `this.log` in the `HelloMessage.componentWillMount` method.

We then wire up `ReactMixin2` followed by `ReactMixin1` in our `Button` component shown as follows:

```
var Button = React.createClass({
  mixins: [ReactMixin2, ReactMixin1],
  clicked: function() {
    this.log(this.props.text + ' clicked');
  },
  componentWillMount: function() {
    this.log('componentWillMount from Button');
  },
});
```

```

    render: function() {
      return <button onClick={this.clicked}>{this.props.text}</button>
    }
  });

```

Adding the components in the reverse order from `HelloMessage` shows that the order that our mixins will be called is the same order as the order they are added to the mixins array in.

We also use `this.log` from `ReactMixin1` mixin in the `Button.clicked` method to log out `this.props.text + ' clicked'` which is why we see **OK** clicked in the console output.

Forms

Form components such as `<input/>`, `<textarea/>` and `<option/>` are handled differently by React as they allow for being mutated by the users input unlike static components like `<div/>` or `<h1/>`. As we will the dynamic nature of form components combined with the determination of React components can lead to some unexpected things when you are first learning React.

Controlled components - the read-only input

Let's start by exploring the concept of controlled components by looking at the following code:

```

var TextBox = React.createClass({
  render: function() {
    return <input type='text' value='Read only' />;
  }
});

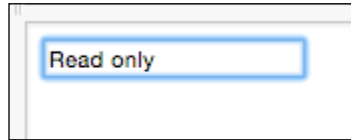
ReactDOM.render(
  <TextBox />,
  document.getElementById('view'));

```



Source code: <http://j.mp/Mastering-React-3-3-Gist>
 Fiddle: <http://j.mp/Mastering-React-3-3-Fiddle>

Let's now run this code and try and change the text displayed in the text box.



As you can see the `TextBox` element doesn't allow the text for being updated and as you type the text displayed, "Read only", doesn't change.

How it works

The reason that our `TextBox` element doesn't change as we type is because in React an input with its value prop set like the one below is a controlled component.

```
var TextBox = React.createClass({
  render: function() {
    return <input type='text' value='Read only' />;
  }
});
```

What it means to be a controlled component is that the input will always display the value that is currently assigned to the input's value prop. In our code we haven't provided a way for the input's value to change so our component always displays "Read only" and it ignores incoming input from the keyboard. This is because React form components are not wired up to respond to the peripheral input like keyboards. And this is because, as we just discussed, React form components are only wired to display what is set on the input component's value prop.

Controlled components - the read and write input

Let's now see what happens when we wire up controlled components to state and props by looking at the following code:

```
var ExampleForm = React.createClass({
  getInitialState: function() {
    return { message: 'Read and write' }
  },
  getDefaultProps: function () {
    return { message: 'Read only' }
  },
  onChange: function(event) {
```

```

        this.setState({message: event.target.value});
    },
    render: function() {
        return (
<div>
            <input id='readOnly' className='form-control'
type='text'
                value={this.props.message}/>
            <input id='readAndWrite' className="form-control"
type='text'
                value={this.state.message}
                onChange={this.onChange}/>
        </div>
        );
    }
});

ReactDOM.render(
    <ExampleForm/>,
    document.getElementById('view'));

```



Source code: <http://j.mp/Mastering-React-3-4-Gist>
 Fiddle: <http://j.mp/Mastering-React-3-4-Fiddle>

Run the code and you will see the output shown as follows:

If you try and change the values in the two text input boxes you will see that you can't change the **Read only** text box but you can change the **Read and write** text box.

How it works

The **Read only** text box is the input with id of `readOnly` and has its value set to `this.props.message` shown as follows:

```

<input id='readOnly' className='form-control' type='text'
        value={this.props.message}/>

```

Note that `this.props.message` is given a default value of **Read only** from the `ExampleForm.getDefaultProps` method shown as follows:

```
getDefaultProps: function () {  
    return { message: 'Read only' }  
},
```

Because React component props are immutable and because in our example `this.props.message` is only set inside the `ExampleForm` component that declares our **Read only** input box our **Read only** text input box can't be changed.

However, the **Read and write** input box with the id set to `readAndWrite` is set to `this.state.message` and its `onChange` synthetic event is set to the `this.onChange` method shown as follows:

```
<input id='readAndWrite' className="form-control" type='text'  
      value={this.state.message}  
      onChange={this.onChange}/>
```

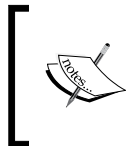
In the `onChange` method we are then taking the event that is passed in from the React synthetic event and then calling `this.setState({message: event.target.value})`. This call will update `this.state.message` reflect the input that the user sends in via a keyboard or some other input device.

```
onChange: function(event) {  
    this.setState({message: event.target.value});  
},
```

Updating state in this way will cause a rerender and when the component's render method is called it will use the current value from `this.state.message` allowing our component to be dynamic to update its displayed value.

Isn't that harder than it needs to be?

If you are familiar with two way data binding frameworks then this probably seems like a lot of work to do something in React that is really simple in frameworks that support two way databinding. However, the React team chose to follow a one way data flow model below.



In React, data flows one way: from owner to child. This is because data only flows one direction in the Von Neumann model of computing. You can find the source at <https://facebook.github.io/react/docs/two-way-binding-helpers.html>.

React's philosophy is all about performance and maintainability. Both of those goals benefit from using one way data flows down the component hierarchy. If you are using a large complex web application then it becomes much easier to reason about the application when data flows in one direction. It gets even better when state is minimized to the smallest set of state needed and maintained as high in the component hierarchy as possible. Below is some guidance from the React team taken from the documentation about organizing data flows and state in React applications.

Remember: React is all about one-way data flow in the component hierarchy. It may not be immediately clear which component should own what state. This is often the most challenging part for newcomers to understand, so follow these steps to figure it out:

For each piece of state in your application identify every component that renders something based on that state. Then, find a common owner component (a single component above all the components that need the state in the hierarchy). Either the common owner or another component higher up in the hierarchy should own the state.

If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component. The source code can be found at: <https://facebook.github.io/react/docs/thinking-in-react.html>.

That said, I think that once you start building larger applications you will appreciate the maintainability provided by this approach and we look at more complex examples in the upcoming chapters so you will get to see a better example of this aspect of React. For now though, let's take a look at how we can organize a simple form around these best practices.

Controlled components – a simple form

Let's take a look at a simple form now using controlled components.

```
var TextBox = React.createClass({
  render: function() {
    return (
      <input className='form-control'
        name={this.props.name}
        type='text'
        value={this.props.value}
        onChange={this.props.onChange}/>
    );
  }
});
```



```
var ExampleForm = React.createClass({
  getInitialState: function () {
    return { form: { firstName: 'Ryan', lastName: 'Vice' } }
  },
  onChange: function(event) {
    this.state.form[event.target.name] = event.target.value;

    this.setState({form: this.state.form});
  },
  onSubmit: function(event) {
    event.preventDefault();

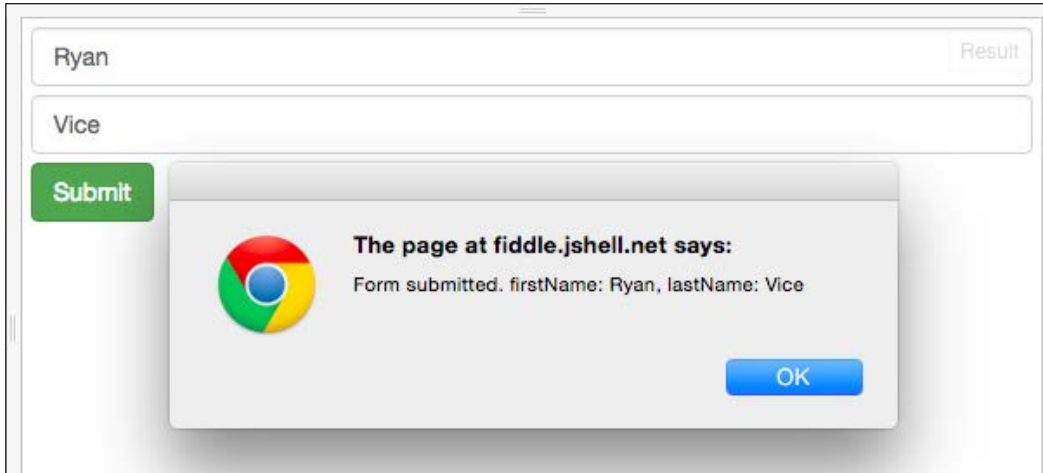
    alert('Form submitted. firstName: ' +
      this.state.form.firstName +
      ', lastName: ' +
      this.state.form.lastName);
  },
  render: function() {
    var self = this;
    return (
      <form onSubmit={this.onSubmit}>
        <TextBox name='firstName'
          value={this.state.form.firstName}
          onChange={this.onChange}/>
        <TextBox name='lastName'
          value={this.state.form.lastName}
          onChange={this.onChange}/>
        <button className='btn btn-success'
          type='submit'>Submit</button>
      </form>
    );
  }
});

ReactDOM.render(
  <ExampleForm/>,
  document.getElementById('view'));
```



Source code: <http://j.mp/Mastering-React-3-5-Gist>
Fiddle: <http://j.mp/Mastering-React-3-5-Fiddle>

Run the Fiddle, click **Submit** and you will see the following output:



How it works

This code creates a simple first name, last name form by doing the following.

Create a reusable `TextBox` component that allows for wiring up name, value and `onChange` in a consistent way.

```
var TextBox = React.createClass({
  render: function() {
    return (
      <input className='form-control'
        name={this.props.name}
        type='text'
        value={this.props.value}
        onChange={this.props.onChange}/>
    );
  }
});
```

1. In our `ExampleForm` component we create a simple form for the first name and last name using our `TextBox` component. We also wire up form's `onSubmit` to the `this.onSubmit` method and wire up each `TextBox` instance `onChange` to `this.onChange`.

```
render: function() {
  var self = this;
  return (
    <form onSubmit={this.onSubmit}>
```

```
        <TextBox name='firstName'
            value={this.state.form.firstName}
            onChange={this.onChange}/>
        <TextBox name='lastName'
            value={this.state.form.lastName}
            onChange={this.onChange}/>
        <button className='btn btn-success'
            type='submit'>Submit</button>
    </form>
    );
}
```

2. We wire up our `ExampleForm.onChange` method to allow our controlled components to be dynamic and to reflect our users input in the UI. Note that we are taking advantage of JavaScript's implementation of objects as dictionaries here to set the property on `this.state.form`. Using this kind of approach will greatly reduce boiler plate that you need to write to wire up inputs.

```
    onChange: function(event) {
        this.state.form[event.target.name] = event.target.value;

        this.setState({form: this.state.form})
    }
```

3. We then wire up `ExampleForm.onSubmit` method to first suppress the default form behavior of HTML which will prevent a server side postback and then we show an alert with the first name and last name values that were entered into our form.

```
    onSubmit: function(event) {
        event.preventDefault();

        alert('Form submitted. firstName: ' +
            this.state.form.firstName +
            ', lastName: ' +
            this.state.form.lastName);
    }
```

But what about the best practices?

Now we've looked at how it works but let's take a minute to focus on how we followed React's best practices around state. The following is a quick refresher on the best practices.

For each piece of state in your application, you have to identify every component that renders something based on that state. Then, find a common owner component (a single component above all the components that need the state in the hierarchy). Either the common owner or another component higher up in the hierarchy should own the state.

If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component.

In the example above we applied these best practices in a way that is often called **Smart** and **Dumb** components but also called fat and skinny, stateful and pure, screens and components, and so on. The approach involves dividing your components into two categories, Smart components that contain state and Dumb components that are immutable and only use props. Organizing your components in this way aligns really well with the React best practices and will make your app easier to understand and reason about.

In our example we have created the `ExampleForm` smart component that contains all the state for our application and the `TextBox` dumb component that is immutable and just provides a seam for our text input components allowing us to easily provide consistency in how we layout and wire up our text inputs. By using this approach we've moved the state out of the `TextBox` component and into the `ExampleForm`. The example form is then able to store the state for all the `TextBox` instances and will update the `TextBox` instances with any changes in state through `TextBox` props.

Refactoring the form to be data driven

The modular design of our application makes it trivial to make our form data driven by changing our render method as shown below.

```
render: function() {
  var self = this;
  return (
    <form onSubmit={this.onSubmit}>
      {Object.keys(this.state.form).map(
        function(key) {
          return (
            <TextBox name={key}
              value={self.state.form[key]}
              onChange={self.onChange}/>
          )
        })
      }
    )
}
```

```
        <button className='btn btn-success'
type='submit'>Submit</button>
      </form>
    );
```



Source code: <http://j.mp/Mastering-React-3-6-Gist>
Fiddle: <http://j.mp/Mastering-React-3-6a-Fiddle>



How it works

All we did here was replace the static `TextBox` component instances with code that dynamically generates the `TextBox` components based on `this.state.form` shown as follows:

```
(Object.keys(this.state.form).map(
  function(key) {
    return (
      <TextBox name={key}
        value={self.state.form[key]}
        onChange={self.onChange}/>
    )
  })
)
```

The `Object.keys` method will return a collection of all the property names on `this.state.form` and we then call `map` on that collection to generate our `TextBox` instances. From here there are many exciting things we could do. We could make our `TextBox` component more generic so that it takes an input type from the data that it's generated from and instead of just being a `TextBox` it could be a `FormInput` component that could be text, checkbox, and so on. There are micro frameworks, like `Formsy` that take this idea and add great features like validation to the mix. Speaking of validation, let's take a closer look at validation in React.

Validation

In this chapter we looked at forms but how do we handle validating the user input? There's good and bad news on this for React users. The bad news is that this is not a concern that React address for us. However, the good news is that we have a lot of options when it comes to validation because React is focuses only on displaying and modifying the screen and not validating it. Our first options is that we could simply write our validation logic into our components or other JavaScript modules.



Note we will look at using the CommonJs pattern in react to create modules in upcoming chapters.

If we started out writing our validation logic this way then we'd likely see some patterns emerging and want to write some library style components, mixins and modules to reduce repetition and provide consistency. We could definitely roll our own solution to validating our apps as we saw at end of the last section where we looked at creating data driven forms. We could use generic components and/or custom mixins to create our own library that would make writing forms and validation code easier. However, people have already done this and there are also open source libraries out there that can make things easier when it comes to validation. One of the key benefits of working in React is that you are working with a micro framework that only cares about view concerns and you can mix and match it with other tools allowing for a lot more flexibility than you will find in larger frameworks like AngularJs or EmberJs

Validation types

There are several places in a client-server style application that we can put validation logic. If we are using a relational database we can have validation that enforces the structure of our data for data coming into the database. On the server if we are using an N-Tier architecture we can have logic in our domain model and domain services, we can also have validation logic in our repositories or **Data Access Objects (DAO)**. No matter what patterns we are using we can add validation logic to each tier of our application. In the same way we can have validation logic at the transport layer and validating that the data coming into our REST API is valid. However, none of those concerns are things that address with our React code as React is a view concern. This means that in a Client Server application we would only be looking at how to address Client Side validation within the context of React.



Note that React is very flexible and it is possible to create applications with React that don't follow a Client Server architecture. While you can build web applications with React you can also use React in Thick Client applications using technologies like NW.js or Electron. You can also use React to write native mobile applications using React Native. And that's just some of the options available at the time of this writing. I'm sure we will see many other options come and go over time. However, for our conversation about React validation options we will refer to React's area of concern as being on the Client Side.

When writing client side validations and validating a form there are two scopes that we need to be able to validate at on the client side. We need to be able to do simple field level validations and more complex form level validations. Let's now take a closer look at these two concepts.

Field-level validation

Field level validation is validating a single input, in isolation, for simple things like whether a field is required, whether a field's length is under a maximum length or over a minimum, or whether a field satisfies a regular expression for things like emails, social security numbers, and so on.

Form-level validation

In addition to validating a form's various fields against simple rules in isolation we also need the ability to validate our form against complex rules that consider more than one field. For example, we may need to make sure that two fields are the same when confirming an email address or password doesn't have an obvious typo. Or we could have fields that are required when another field has a certain value, like requiring filling out a shipping address when it's not the same as a billing address. There's a whole host of complex business rules that we could need to validate that would require more than one field of data to process and these more complex, multi field rules fall into the scope of form level validations.

The react-validation-mixin example

Let's now take a look at one tool that we can use to do validations in React, the react-validation-mixin. This library takes advantage of React's mixin functionality to allow you to easily support both field and form level validations.

Getting the code

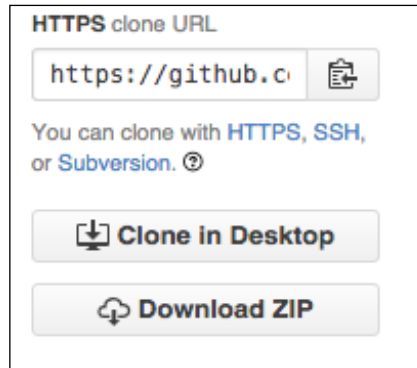
For this example we won't have a Fiddle because we need to use the React-validation-mixin which is easily installed using **NPM (Node Package Manager)** and accessed via the CommonJs require syntax. We will look in detail into setting up a React application to allow for consuming Node package dependencies in up coming chapters so I won't dive into those details here. For this code I've created a GitHub repository that you can either clone using Git or simply download as a ZIP file.



The following is the repository URL:

[https://github.com/RyanAtViceSoftware/
MasteringReactJsValidationExample](https://github.com/RyanAtViceSoftware/MasteringReactJsValidationExample)

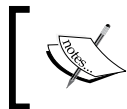
If you haven't used GitHub before you will find the clone and download options on the right side of the page shown as follows:



Running the code

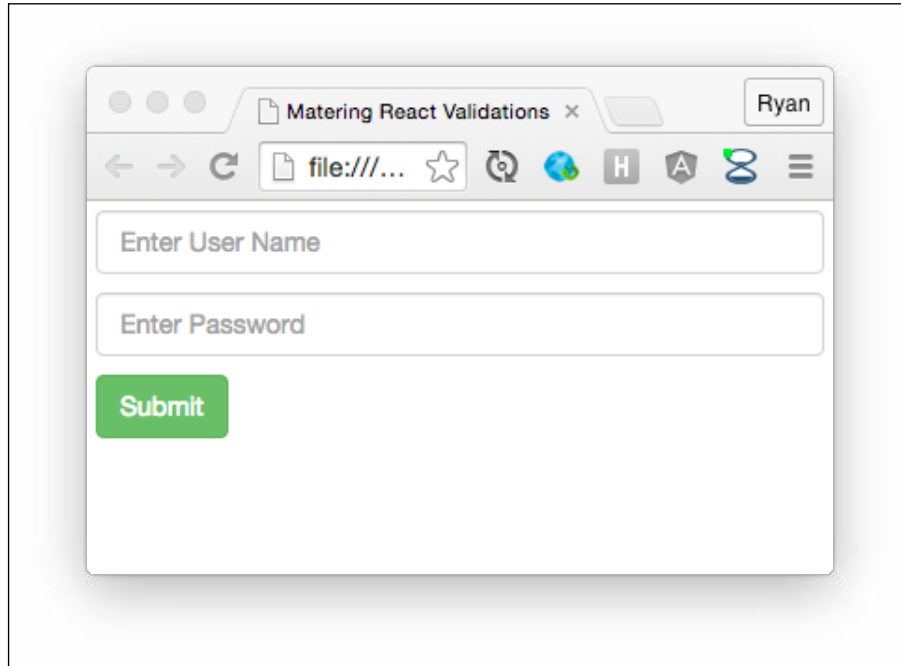
Once you get the code cloned or downloaded and extracted you can run the code by doing the following:

1. Execute from command prompt (Windows) or terminal (Mac)
`npm install`
2. Execute from command prompt (Windows) or terminal (Mac)
`npm start`
3. Open `index.html` in a browser.



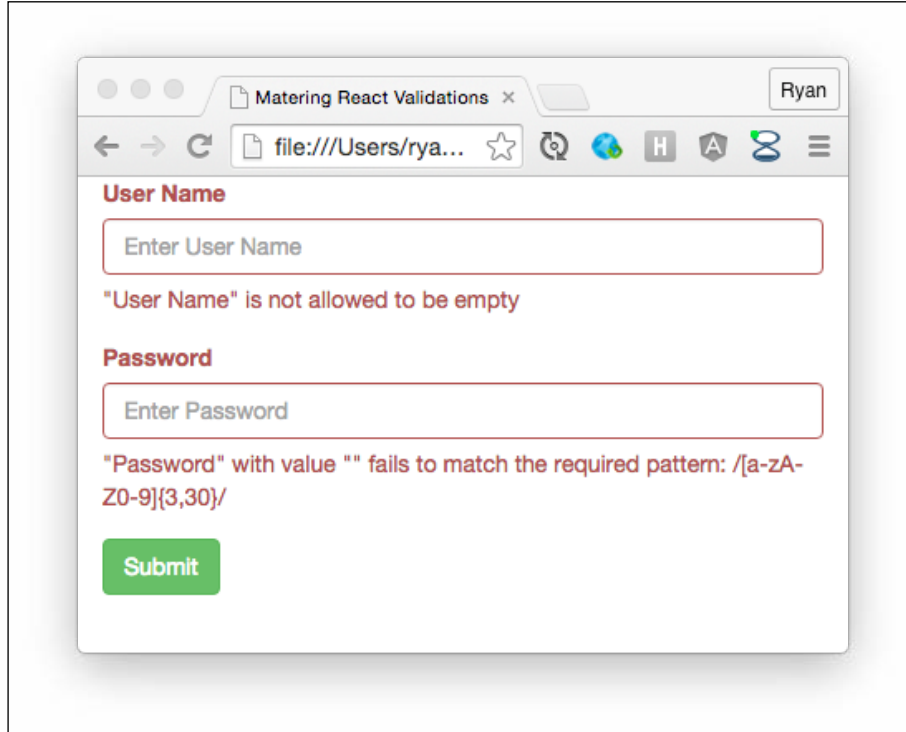
You open `index.html` by simply right clicking on the file in Windows Explorer (PC) or Finder (Mac) and then selecting to open the file with your favorite browser.

Now you should see a page like the one shown as follows:




Note that these steps will run watchify that will rebuild `dist/bundle.js` anytime you change the code in `app.jsx`. The `bundle.js` file is what is being linked to in `index.html` file. This setup allows you can change the code in `app.jsx` and save your changes to regenerate `bundle.js`. Feel free to experiment with the code and just make sure that you save and then let the build finish before you refresh the browser. You can tell the build is finished by watching the terminal\ console window that you ran the `npm start` command in.

We have implemented several field level validation rules here shown as follows:

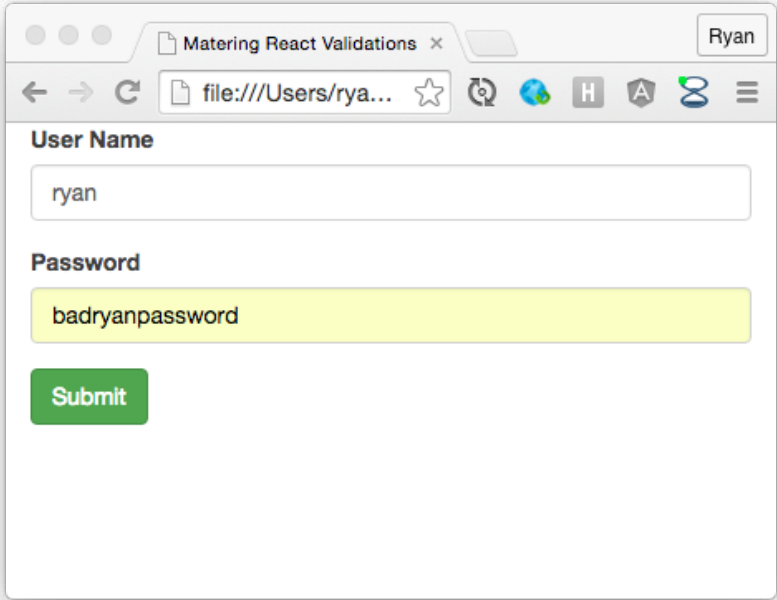


The screenshot shows a web browser window titled "Matering React Validations" with a user profile icon "Ryan". The address bar shows a file path. The form contains two input fields. The first field, labeled "User Name", has a placeholder "Enter User Name" and a red error message below it: "User Name" is not allowed to be empty. The second field, labeled "Password", has a placeholder "Enter Password" and a red error message below it: "Password" with value "" fails to match the required pattern: /[a-zA-Z0-9]{3,30}/. A green "Submit" button is located at the bottom of the form.

If you submit the form without adding any text to the boxes you will see the validations shown above. These validations fire when a text box loses focus so that the user isn't alerted until they've had a chance to provide valid data. Here we are getting the React-validation-mixin default messaging for a required field violation and a regular expression violation. The **User Name** field is indicating that it is a required field and the **Password** field is indicating that it is not matching a regular expression that requires that our password be between 3 and 30 characters long and only contains letters and numbers.

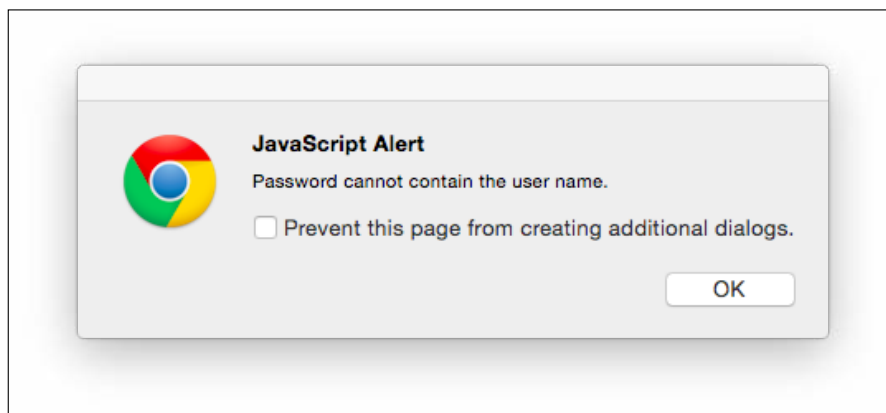
 Note that here we are using the default error messages and that in real code we would want to provide more user appropriate messages. We will look at how to do that in a moment.

Next let's take a look at how this web application implements a more complex form level validation. Enter **ryan** for the **User Name** field and then **badryanpassword** for Password shown as follows:



The screenshot shows a web browser window titled "Matering React Validations" with a tab labeled "Ryan". The address bar shows a file path: "file:///Users/rya...". The form contains two input fields: "User Name" with the value "ryan" and "Password" with the value "badryanpassword". A green "Submit" button is located below the password field.

Now click the **submit** button and you will see the alert box shown below and the form won't be submitted.



This is an example of a more complex form level validation as we are looking at more than one field, **User Name** and **Password**, and applying a rule against both of them. Note that you will also see alert boxes if you click **submit** and any of the field validations fail as we are re-running those rules as part of our form validations as you would want to do.

Getting the code

Note that the first thing we had to do was install the React validation mixin module using NPM. We won't cover those details here but you can find installation instructions on their site at. Now let's look at the code found in `app.jsx` file which is the only file in our solution with logic. There is also an `index.html` that simply allows from displaying our components and references the needed JavaScript files. However, the heart of what we need to focus on here is all in the `app.jsx` file so let's take a look at that code now.

Because this example is longer we will look at the code in a few parts. The first thing we are doing in `app.jsx` is to bring in the dependencies that our code needs using CommonJs syntax as shown below.

```
'use strict';
var React = require('react');
var Joi = require('joi');
var JoiValidationStrategy = require('joi-validation-strategy');
var ReactValidationMixin = require('react-validation-mixin');
```

We will cover CommonJs more in upcoming chapters but for now what is important is to know that `require("dependency-name")` allows us to pull in a dependency and assign it to a variable that we can then use in the file that we are in. Here we have pulled in React, joi, joi validation strategy and react-validation-mixin and assigned them all to local variables.

Next we create a `ValidatedInput` component to wrap our fields so that we can easily implement a consistent form layout and provide a consistent field API for our form through our `ValidatedInput` component.

```
var ValidatedInput = React.createClass({
  renderHelpText: function(message) {
    return (
      <span className='help-block'>
        {message}
      </span>
    );
  },
  render: function() {
    var error
```

```
        = this.props.getValidationMessages(
            this.props.name);

    var formClass = "form-group";

    if (error.length > 0) {
        formClass = formClass + " has-error";
    }

    return (
        <div className={formClass}>
            <label className="control-label" htmlFor={this.props.
name}>
                {this.props.label}
            </label>
            <input className="form-control" {...this.props}/>
            {this.renderHelpText(error)}
        </div>
    );
}
});
```

All we are doing here is allowing for all the props that are applied to the `ValidatedInput` to be wired to our input as shown below.

```
<input className="form-control" {...this.props}/>
```

Using `{...this.props}` allows us to easily use this component to handle all the bootstrap styles and layout while delegating the controlled component wire up to the consuming component. We are adding an input with a label that will reference the input element's name by setting the label element's `for={this.props.name}` value. We've also added an error that will be displayed if:

```
this.props.getValidationMessages(this.props.name)
```

Next let's look at the Demo component below which contains our form.

```
var Demo = React.createClass({
    validatorTypes: {
        userName: Joi.string().required().label('User Name'),
        password: Joi.string().required().regex(/[a-zA-Z0-9]{3,30}/).
label('Password')
    },

    getValidatorData: function() {
        return this.state;
    },
```

```
getInitialState: function() {
  return {
    userName: "",
    password: ""
  };
},

onSubmit(event) {
  event.preventDefault();

  // Handle field level validations
  var onValidate = function(error) {

    if (error) {
      if (error.userName) {
        alert(error.userName);
      }

      if (error.password) {
        alert(error.password);
      }
    }

    // Handle form level validations
    var passwordContainsUserName
      = this.state.password.indexOf(
        this.state.userName) > -1;

    if (this.state.userName
      && passwordContainsUserName) {
      alert("Password cannot contain the user name.");
      return;
    }

    if (!error) {
      alert("Account created!");
    }
  };

  this.props.validate(onValidate.bind(this));
},

onChange: function(event) {
  var state = {};
  state[event.target.name] = event.target.value;
  this.setState(state);
},

render: function() {
```

```
    return (
      <div className="container">
        <form onSubmit={this.onSubmit}>
          <ValidatedInput
            name="userName"
            type="text"
            ref="userName"
            placeholder="Enter User Name"
            label="User Name"
            value={this.state.userName}
            onChange={this.onChange}
            onBlur={this.props.handleValidation("userName")}
            getValidationMessages=
              {this.props.getValidationMessages}/>
          <ValidatedInput
            name="password"
            className="form-control"
            type="text"
            ref="password"
            placeholder="Enter Password"
            label="Password"
            value={this.state.password}
            onChange={this.onChange}
            onBlur={this.props.handleValidation("password")}
            getValidationMessages=
              {this.props.getValidationMessages}/>
          <button className="btn btn-success" type="submit">
            Submit
          </button>
        </form>
      </div>
    );
  }
});
```

We start by implementing React-validation-mixin's `validatorTypes` property that defines our validation rules shown as follows:

```
var Demo = React.createClass({
  validatorTypes: {
    userName: Joi.string().required().label("User Name"),
    password: Joi.string().required().regex(/[a-zA-Z0-9]{3,30}/).
      label("Password")
  },
});
```

Here we are using `Joi` to make `userName` a required string with a label of `User Name`. We are also defining `password` to be a required string. We then use a regular expression so that our password must be between 3 and 30 characters that are either letters or numbers and we set the label to `Password`.

Next we define `react-validation-mixin`'s `getValidatorData` method which returns the data that the validation rules will be applied too. In the `getValidatorData` method we simply return `this.state` and we are initializing `this.state` in `getInitialState` to return empty strings for `userName` and `password` shown as follows:

```
getValidatorData: function() {
  return this.state;
},
getInitialState: function() {
  return {
    userName: "",
    password: ""
  };
},
```

One of the things I like about `react-validation-mixin` is that it is very small and focused and relies on another library, `Joi`, for defining simple field level validation rules as we saw earlier in the code.

Next let's look at the `render` method that lays out and wires up our form shown as follows:

```
render: function() {
  return (
    <form onSubmit={this.onSubmit}>
      <ValidatedInput
        name="userName"
        type="text"
        ref="userName"
        placeholder="Enter User Name"
        label="User Name"
        value={this.state.userName}
        onChange={this.onChange}
        onBlur={this.props.handleValidation("userName")}
        getValidationMessages=
          {this.props.getValidationMessages}/>
      <ValidatedInput
        name="password"
        className="form-control"
        type="text"
        ref="password"
```



```
        placeholder="Enter Password"
        label="Password"
        value={this.state.password}
        onChange={this.onChange}
        onBlur={this.props.handleValidation("password")}
        getValidationMessages=
            {this.props.getValidationMessages}/>
        <button className="btn btn-success" type="submit">
            Submit
        </button>
    </form>
);
}
```

Here we are doing the following:

assigning `form.onSubmit` to `this.onSubmit` so that we can be notified when the form is submitted and execute our form level validations.

```
<form onSubmit={this.onSubmit}>
```

1. In `onSubmit` we are preventing the default HTML handling so the page doesn't post back and then we are able easily do form level validations as we have access to `this.state` and can execute whatever logic we like here. When a validation rule fails we are just showing an alert box but you can and should do something more appropriate in your project code. Here we are first creating an `onValidate` function that takes an error and then performs field level validations using the properties of the error argument that was passed in. We will pass this `onValidate` function into the `this.props.validate` method that is part of the `react-validation-mixin`. The `this.props.validate` method it will then run the validation rules we configured above and pass any errors to `onValidate` in the first argument which is the error argument in our code. Additionally we are calling `bind(this)` on `onValidate` so that our `this` context is correctly set to our component instance and not the React runtime. Now when our `onValidate` callback function is called we can easily access `this.props` and `this.state` to do our complex validations.

```
onSubmit(event) {
    event.preventDefault();

    // Handle field level validations
    var onValidate = function(error) {

        if (error) {
            if (error.userName) {
                alert(error.userName);
            }
        }
    };
    this.props.validate(onValidate, this.state);
}
```

```

    }

    if (error.password) {
        alert(error.password);
    }
}

// Handle form level validations
var passwordContainsUserName
    = this.state.password.indexOf(
        this.state.userName) > -1;

if (this.state.userName
    && passwordContainsUserName) {
    alert("Password cannot contain the user name.");
    return;
}

if (!error) {
    alert("Account created!");
}
};

this.props.validate(onValidate.bind(this));
},

```

2. Assigning the `ValidatedInput` value to properties of our state so that our inputs will be dynamic.

```

<ValidatedInput
    name="userName"
    type="text"
    ref="userName"
    placeholder="Enter User Name"
    label="User Name"
    value={this.state.userName}
    onChange={this.onChange}
    onBlur={this.props.
handleValidation('userName')}
    getValidationMessages=
        {this.props.getValidationMessages}/>

```

3. Assigning our `ValidatedInput` instances' `onChange` properties to `this.onChange` so that we can update the state when `onChange` fires. Once `this.onChange` is called we are then taking advantage of how JavaScript objects are dictionaries to dynamically update our state based on the `event.target.name`. Here we are indexing into our state object with `state[event.target.name]`. This allows us to keep our code generic and reduce boilerplate code by following the simple convention that we assign the name attribute of our inputs to the same name that we use for that input's data on our state object. This trick allows us to avoid having to write a function for each controlled component.

```
onChange: function(event) {  
  var state = {};  
  state[event.target.name] = event.target.value;  
  this.setState(state);  
},
```

4. Assigning `onBlur` to `this.props.handleValidation` so that when our text input's lose focus it's associated validation rules will fire. The `handleValidation` function was added to our component by the `react-validation-mixin` and provides a convenient way to validate a field via a key from an event handler. When the `handleValidation` function is called our form will re-render if there is a validation error allowing us to display the error as we did in the `ValidatedInput` component we saw earlier in the code.
5. Assigning `this.props.getValidationMessages` to our `ValidatedInput` instance's `getValidationMessages` property. The `getValidationMessages` function expects a call back that will be called to check for error messages that will be displayed as shown in the following code. We are simply delegating this call to `this.props.getValidationMessages` which is part of the `react-validation-mixin` which will use the configured label to create a standard user friendly error message.
6. Adding a submit button that will cause our form to submit shown as follows:

```
<button className="btn btn-success" type="submit">  
  Submit  
</button>
```

Now we have looked at doing simple field and complex form level validations using the `react-validation-mixin`. At the time of this writing there are several validation libraries available in the open source community to choose from and if you're feeling adventurous you could write your own which is a technique we will explore in the future chapters.

Summary

In this chapter we looked at dynamic components and saw how we could easily create repeated collections of components. We then looked at mixins and saw how we can decorate lifecycle events and share functionality with this extensibility point. Next we looked at forms and saw how when we set value we create a controlled component. We then discussed validation and looked at an example of how we could use the react-validation-mixin to handle both field and form level validations.

We have now covered most of the basic aspects of React and will now dive into some of the more advanced topics and look at some more substantial examples.

4

Anatomy of a React Application

In an application of any reasonable complexity, React plays a significant but limited role. React is a component rendering and composition system for views. This definition leaves out many facets of a complete application. In this chapter, we will explore complex web application design as three aspects. We will also identify the subcomponents of each aspect and develop a rationale for choosing particular tools to service each aspect.

In this chapter, we are going to cover the following:

- What is a Single Page Application (SPA)?
- Aspects of a SPA design
- Build systems
- CSS preprocessors
- Compiling modern JS syntax and JSX templates
- Front-end architecture components
- Application design

The goal of this chapter is to become familiar with the structure of web applications and the technologies involved. In the following chapters, 5 through 9, a full-featured multi-user blog application will be built. The preparation provided here will guide you not only with the configuration of the blog application, but will also introduce you to a few design procedures. The design procedures will define the components of the app and how they are interconnected. In the application design section of this chapter, an email application is used as an example in order to illustrate the design tasks for the first time. In *Chapter 5, Starting a React Application*, the same procedures are followed once more for the blog application. Then, in chapters 6-9, we'll flesh out the feature code for the multi-user blog application prototype.

What is a single-page application?

A **single-page application (SPA)** is a rich application—one that delivers the same features, functionality, and sophistication normally associated with a desktop or native application. In a SPA, only one main document, or page, is loaded into the browser. After the initial document load, other resources, such as scripts, stylesheets, data, and assets such as images, are loaded asynchronously, but the initially requested document does not change. In other words, throughout the lifecycle of the application, the content of the URL in front of the hash mark (#) typically does not change. As a result, the browser never requests any subsequent "page". The history API in modern browsers does allow changes to the URL before the hash without an entire page request, but most JavaScript frameworks and routing libraries use the portion after the hash exclusively for front-end routing. For simplicity, we'll operate using this clear separation of server-side and client-side routes.

http://example.com/app	#/primary View
SERVER ROUTE	CLIENT ROUTE

The portion of the URL before the hash mark is a server-bound (browser request) route.
The portion after the hash mark is the client-side route controlled by the SPA.

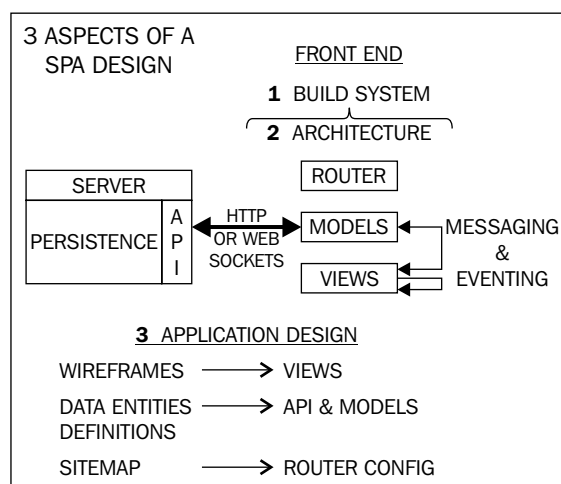
Navigation is handled on the front end through a router. The front-end router reacts to URL changes after the hash. This portion of the URL was historically used for contextual linking between headings within a web page via anchor tags referencing the hashed URLs, sometimes called jump links. When the application changes the contents of the URL after the hash, a view system morphs the DOM by composing and rendering different high-level views. The mapping between these URL changes and views is done via the router configuration. Any change before the hash belongs to the server route and would result in a new browser document request. By definition, a SPA exclusively drives navigation through front-end routes only.

At the highest level, a SPA still consists of a front end and back end. Historically, the back end performed most of the application logic and the front end merely handled the concern of presenting an interactive facade to the user in order to display data and gather input. With a SPA, the back end is typically just a persistence mechanism and an API whose design could still be largely defined by front-end modeling. In a SPA, the front end now handles all significant routing, DOM construction and composition, and view modeling. View models are projections or subsets of one or more of the models in the application or problem domain.

Three aspects of a SPA design

A SPA can be conceived in three parts: a **build system**, front-end **architecture components**, and application design. Application design is called out specifically because it is instrumental in defining the server API as well as moving forward with the actual implementation. The artifacts that result from the application design process bridge the gap between the various needed parts, and the interesting use of those parts in an actual implementation. We'll begin by identifying the various parts and end by exploring a few design procedures. This will carry us nicely into the next chapter, where we will begin to implement an actual application.

The following diagram illustrates the three major aspects of a SPA design and their relationships:



In the diagram, you can see a clear association between the core front-end models and the server API, as well as the somewhat nebulous relationship between the application design and everything else. Stay with me on the application design aspect. It will generate some meaningful artifacts and make both the relationships in the anatomy diagram and the implementation of our application clearer.

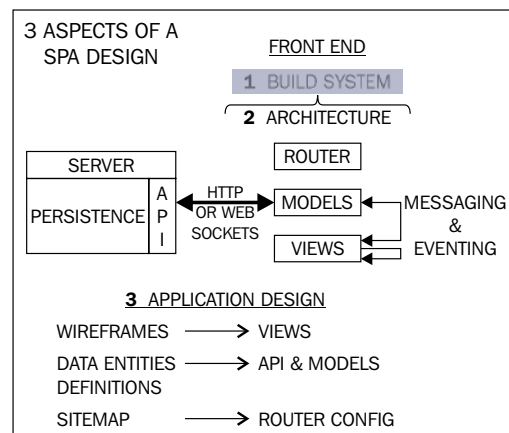
As we explore these aspects and their details, we'll also address how there are lots of options for each detail, offer some thoughts on trade-offs for each option, and ultimately describe a choice with some justification. There are so many ways to address each problem that, sooner or later, you just need to choose and move forward. How you make such choices should take into account the problem you are trying to solve, the other components you have already chosen, and obvious advantages each choice offers your particular application. Try to avoid naïve comparison analysis that is rampant on the Web (X versus Y) and focus on what each option actually **does**, which qualities of it **appeal to you personally** and how those qualities **apply to the problem at hand**. Popularity of an option should be considered in two regards:

- The ease of finding information on the option (documentation quality, help forums, thoughtful examples in blogs, etc)
- General acceptance of that option within the community of practitioners using the components you've already chosen

In other words, if you've already chosen React for its strengths at component composition and rendering performance, then you should probably lean a bit into other tools that are trending in the React community in order to have better support.

Build systems

Building a SPA (as opposed to an older style web application) means that many application concerns have migrated to the front end, making the client-side responsibilities necessarily more complex. Also, the nature of modern web development lends itself to an endless buffet of tools aimed at making HTML, JavaScript, and CSS more manageable.



The front-end build system

Here are some of the types of tools for managing code complexity:

- Module and code packaging/delivery systems
- CSS preprocessors
- Next-generation JavaScript syntax (ES6 and beyond)
- Templates and other syntax processing (needed for JSX)

In terms of managing complexity, the first item in this list is paramount. Being able to organize portions of code, inject it into other portions, and efficiently deliver it to the browser is essential for making web applications.

Choosing a build system

The build system will tie all of the builders and preprocessors together into a manageable pipeline of transformations for development and, ultimately, deployment.

For small experiments, using the in-browser JSX compiler is great. You can also use the ES6 (ECMAScript 6) Harmony syntax if you use `<script type="text/jsx;harmony=true">`. This feature was added in React 0.11.

For serious work, though, it is recommended that you use a build system for JSX compilation, CSS preprocessing (such as LESS and SASS), as well as for bundling your application into payloads of an efficiently small number of files. There are many solutions for this, and it seems that there's a new one every couple of months. An early and enduring favorite is **Grunt**, in which build tasks are specified in code but resemble a large configuration file. When streams caught on in the node community, a stream pipelining build task system was created called **Gulp**.

Typically, a module system is paired with a build system in order to manage dependencies. **CommonJS** and **Asynchronous Module Definition (AMD)** are the prominent modularity strategies. AMD is valued for its natural disposition toward asynchronous loading, and CommonJS is valued for its familiar looking syntax and use in NodeJS.

Alongside build tools are scaffolding tools that whip your filesystem structure into a ready-to-code state. These tools also download requisite libraries and perform configuration by asking you to answer a series of yes or no questions. One very popular scaffolding tool is **Yeoman**, which handles all these scaffolding tasks. While Gulp is used for general task running and sequencing, a Gulp-related answer to the scaffolding aspect exists called **Slush**. As previously stated, these types of tools (and JS frameworks for that matter) are released on a continual basis. Developing an aptitude for identifying the trade-offs and merits of new tools without getting overwhelmed or too complacent with a particular set is an essential skill. Shiny new ones are always on the horizon but, whichever you choose, you should spend enough time with them to complete a nontrivial project in order to know where their true power lies. Doing this will hone your technical judgment and help you to develop a personal style.

Within this swirl of options, there's a very versatile tool that the React JS community seems to have gravitated toward called **Webpack**. Webpack is substantial. It has a pluggable interface and a project build specification mechanism, which can intelligently split your code into chunks for efficient delivery to the browser. It uses a streaming pipeline style similar to Gulp but, unlike Gulp, which uses standard NodeJS imperative streaming code, Webpack configuration defines a build pipeline using a more succinct syntax. It also carries the burden of code chunk dependency calculation and dynamic loading. Further, it supports both CommonJS and AMD style dependency syntax. Chunk specification in Webpack is defined by syntax that closely resembles AMD-style **dependency injection (DI)** syntax. Webpack also has a server component which is used to dynamically generate necessary code chunks and hot load them into the browser environment during development. It's quite a full-featured and impressive tool.

In summary, a complete build system and pipeline includes a scaffolding aspect to kick-start your project organization, the means to optimize a dependency tree, a modularization component to express the dependency hierarchy, any precompilation processes (CSS preprocessors and ES6 transpilers), code minification steps for packaging, and active reload mechanisms for development. Take the time to try the latest tools, but for pragmatic purposes, the path of least resistance is to use what your community is using. For instance, you may find it easier to get help if you use Webpack while exploring React. So, that's what we'll use going forward. We are going to leave some of the details of Webpack aside for personal exploration, but we will examine an interesting project configuration that uses many features of Webpack for our example project.

Module systems

There are two dominant module system styles in the JavaScript community: AMD and CommonJS.

CommonJS

CommonJS is traditional in that a single assignment statement is used to specify a portion of code to be imported into the target code. The statement looks like this: `var someModule = require('path/to/module');` A key point to this structure is that, for the module referenced by the variable to be immediately used in a following statement, the module assignment must block until the module code can be loaded. So, it had better be loaded or your JS code would have to pause! Though, a more advanced parser, which understands both JavaScript and CommonJS syntax, can look ahead to require statements and build an intelligent dependency graph for preloading and concatenation. CommonJS syntax is the natively supported module system in NodeJS. It is also the style used in the JavaScript core language going forward from ES6.



Here's a quick note about CommonJS with Webpack. Typically, the way you call for a CommonJS module is by assigning the result of a `require` invocation to a `var` statement. This means that the `var` statement could potentially pollute the function scope in which it was defined, as `var` statements do. Webpack, though, wraps the module within another function scope. In fact, it rewrites your `require` invocation to a Webpack one that can intelligently load application chunks. So, your `var` statements within a Webpack module become effectively isolated. This is a nice extra isolation that brings CommonJS within Webpack closer to the isolation and DI style of AMD.

AMD

AMD (Asynchronous Module Definition) is a specification in which module code is encapsulated into the invocation of a function named `define`. The `define` signature is comprised of the following parameters:

- First, an optional module name (the module is referred to by filename if this parameter is omitted)
- Next, an optional list (array) of dependencies by name or filesystem location
- Finally, and most importantly, the definition of the module itself as an **IIFE (immediately invoking function expression)**

The IIFE (also the module definition function) signature contains positional parameters that directly map in arity (number of parameters) and parameter order to the dependencies specified in the previous define parameter, the dependency array. This allows assignment and symbol renaming upon invocation of the target module, our IIFE. Another way to say this is that the dependencies you specify in the dependency array parameter will directly be mapped to the function parameters of the module being defined, allowing you to name dependencies whatever you want within the module. This syntax lends itself to a more natural-feeling asynchronous loading pattern for JavaScript. It is considered more natural for asynchronous loading for two reasons.

First, it is a common pattern in JavaScript to make the final parameter to a function signature a callback for continuation of execution. It is also generally expected that the callee will invoke the function supplied in the final parameter asynchronously. This is the callback pattern for asynchronous JS.

Second, the define invocation itself establishes a registry of modules and their dependencies. This lends an AMD system to all sorts of optimization opportunities where modules can be loaded and executed more efficiently. For instance, as define invocations occur, the AMD system could begin loading the modules in turn. Logical branches within code can call define for subsequent on-demand loading. Here's another example of a possible optimization: the AMD system could lazily load modules only when required and fetch them from the server. A final reason AMD may be considered "natural" is that the syntax not only resembles a historically familiar pattern used in JavaScript, but also closely mirrors other DI systems.

Our module choice

I personally bought into the syntactical style and more "natural" asynchronicity of AMD during the duration of the great module debate. However, in the following chapters, we'll use CommonJS for pragmatic reasons. ES6 uses this style and has a lot of other great features that we want to use. The moment of judgment on this debate has largely passed as NodeJS and JavaScript natively support CommonJS going forward. Finally, there are a lot of tools, including Webpack, that take the burden of asynchronous loading and packaging of code for the wire off of the developer, even when using the CommonJS syntax. As such, further mentions of JavaScript modules will loosely imply the ES6 CommonJS style. It is the path of least resistance.

CSS preprocessors

CSS preprocessors are a fantastic way to organize CSS. They help to simplify complicated selectors, handle vendor prefixing, establish variables for reuse in layout measurements and color calculations, and even roll up image references into the stylesheet, eliminating extra HTTP requests. If you are a serious web developer, you need to get comfortable with CSS preprocessors.

The two prominent options are **LESS** and **SASS**. While many CSS gurus seem to advocate SASS, we are going to use LESS. This is not a stance on preprocessors in general, but this book is about React and we are already using Node for tooling because of Webpack. LESS syntax, while perhaps a bit less powerful, more closely resembles actual CSS and runs a bit more easily just with JavaScript (specifically node), while SASS requires Ruby or a Node bridge to native bindings. So, the barrier to entry, both cognitively and environmentally, is lower overall for our project using LESS.

Compiling the modern JS syntax and JSX templates

ECMAScript 6 (ES6), a long overdue update to the specification upon which JavaScript is based, has many useful features and additional convenient syntax. It heralded a future of frequent updates to the language, so you should get familiar with what it has to offer.

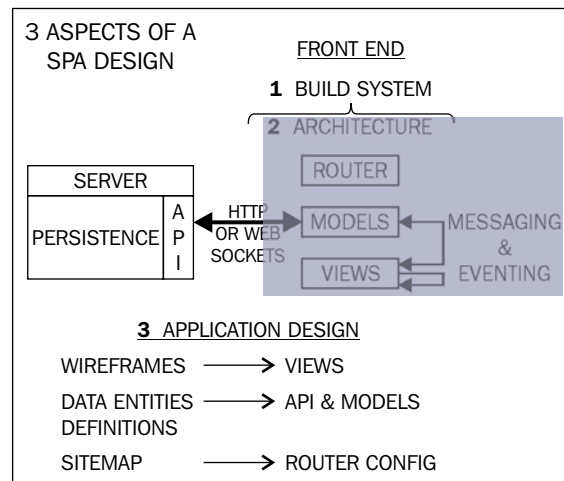
If we are going to inhabit React-land, we should use JSX. It's a very convenient way to compose React components, albeit a leaky abstraction of HTML (really XML). It looks like HTML, but it's really just shorthand for JS. Remembering this mantra at all times can keep you out of trouble: JSX is a dialect of JavaScript, not HTML.

In regard to language features, browsers don't really support a particular version of JavaScript. Adoption of features is more fluid. Browser updates roll out chunks or bursts of the latest features. Having many moving targets is annoying for a developer, but transpilers come to the rescue! It turns out, we can just write using our favorite features. Luckily, the hardworking people of the web development community have managed to make implementations of new features in runtime environments that don't explicitly support them via **transcompilation** (aka source-to-source compilers) and **polyfills** (runtime supported code that fills in missing features). A couple of past popular transpilers were Google Traceur compiler and 6to5. 6to5 was renamed to **Babel** because the maintainers wanted to be more forward looking than just compiling ES6 to ES5. Babel will support ES6 and beyond as new features are ratified. It's also generally easy to use and includes JSX support! One wonderful tool lets us use the latest and greatest of JavaScript and do JSX compilation.

Front-end architecture components

A user interface is all about views. The primary views tend to map directly to user goals. For instance, in an email application, you read and send emails. So, your primary views could be "inbox" and "create email". In most applications, defining the primary views typically consists of taking the system nouns (document, email, order, user, post, etc) and making a view for each associated verb, usually "find" and "create/edit".

The following diagram highlights the front-end architecture aspect of an SPA design:



Front-end architecture components

In a React app, your front-end architecture components are:

- The router
- Models
- Views (Layout and CSS)
- View models (compositions of system models)
- View controllers (logic and rules in the view)
- Messaging and eventing mechanisms

The front-end router

As mentioned at the beginning of this chapter, the front-end router is a piece of software that reacts to changes in the URL after the hash mark. The result of the routing is a transition between major views. This changes the user's context or workflow. An obvious choice for a front-end router in a React application is **React Router**. It handles all of the standard functions needed from a router. Router functions include the ability to map routes to views or compositions of views as well as to break apart the hashed part of the route into positional parameters and query parameters (everything after the "?" in the URL). The query parameters are packaged neatly into props for the components targeted for render by the router.

Front-end models

In the front end, models are often correlated one-to-one with the data entities of the application at large. Early on, there wasn't a particular solution for this concern targeted for React applications, although Backbone models were probably most often used for this purpose. Now there are several solutions specifically targeted at React applications.

Facebook, the creators and general maintainers of React, have devised a modeling pattern they call **Flux**. Flux embraces a one-way data flow model and works by supplying three types of entities: **stores**, **actions**, and the **dispatcher**. Actions are just verbs in the system, commands. When an action occurs, the dispatcher routes it to interested listeners, the stores. Stores are essentially bags of data that can be queried via actions and emit store change events which views respond to by updating internal state and re-rendering if needed.

Another model solution targeted at React with growing popularity is **Reflux**. Reflux is very similar to Flux but omits the dispatcher. Reflux was crafted with the notion that the dispatcher isn't particularly useful and is just extra work. Instead, in Reflux, actions are commands but are merely named commands that can be published or subscribed to directly. So, stores listen to actions directly instead of being mapped through the dispatcher and, just as in the Flux architecture, generate store events that can be listened to by views. Because of this simplification in Reflux, we'll use it in our prototype application in the application building chapters. Reflux also supplies some useful mixins for our views listening to stores. These mixins will automatically tie the payload of the store event emission to a state variable in our views and call `setState` whenever there is a store change.

Views, view models, and view controllers

These entities are the ones that will be taken care of by React. The view is an instance of a React component that was specified via `React.createClass`. Logic within that component definition is effectively the view controller. The internal component state or a portion of that state could be considered the view models in a React application. View models in our blog app will be the portions of internal state set by our Reflux stores via the convenience of the Reflux mixins. View models (component state) in the app may also be a composition of more than one store, if necessary.

Messaging and eventing

With everything in our app componentized, we'll need a way to coordinate reactions to changing data, user interactions, and possibly scheduled events. In React, a parent component can communicate to a child component via props. The child can react to prop changes using the `componentWillReceiveProps` lifecycle method. However, our data stores aren't part of this component hierarchy, and it's nice to have a general communication bus for communications besides those within the view hierarchy. For this, we can use actions in Reflux. Actions are merely publish-subscribe eventing mechanisms. In addition, actions can be specified as asynchronous and supply a promise interface. This is sufficient for just about any SPA. As a bonus, we can go to a single place, our action definitions object, to see a list of all the verbs in the system and whether or not they are async. Nice!

Other utility needs

At this point, we have everything related to view management and front-end communication covered but an important piece is still missing: the bold double-headed arrow in the diagram pointing between front-end models (now stores) and the server API. For this, we really just need a simple AJAX library. This is usually baked into frameworks and other toolkit libraries such as jQuery and Backbone, but such libraries also come with a lot of things that we don't really need. Superagent is a great example of a full-featured AJAX library. Its clean, chainable, interface handles different REST verbs and HTTP headers and supplies a promise. Consider React to be in the spirit of the Unix philosophy, which espouses the virtue of "doing only one thing, and doing it well". React does views, only views, and does them well. Likewise Superagent does HTTP requests, only HTTP requests, and it does them well.

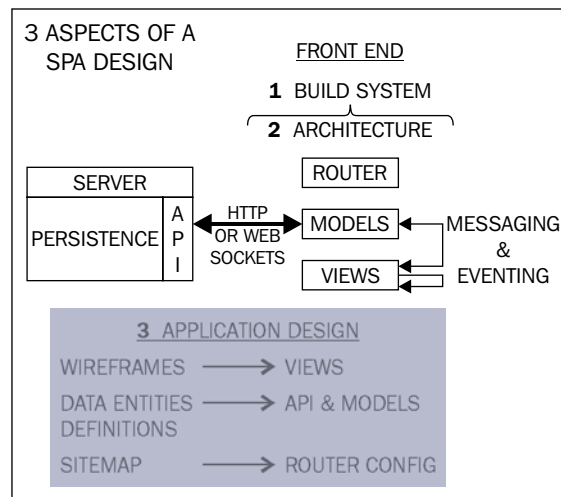
The last item of note in service of our blog application is some sort of rich text editor. After all, editing plain unformatted text would make a pure, albeit rather dull blogging experience. There are lots of options here. Among the ones I considered were Hallo and Quill. Both were simpler than TinyMCE or Aloha Editor, which I also evaluated. After some thought, Quill was chosen because unlike Hallo, which operates as a jQuery UI plugin, Quill does not require any additional libraries to operate.

The application design

First, a disclaimer: I am not a trained designer, but like most, I have some assumptions and notions in the department. Here are some tips that don't just apply to visual and interaction design, but also to software design in general. Think about the fewest number of things that identify what you are building. For starters, eschew all the features swirling in your brain. To begin a plan, focus is required. It will pay off when we start coding. Try to sum up the application using one word. For our impending blog app "posts" is an apt choice. For a lunar lander game "physics" comes to mind. For an adventure game, maybe that one word is "story". If you understand the number one thing that your app is supposed to get right and let it guide you throughout design and development, then it will at least do that one thing well.

In the next chapter, the following design tasks will be repeated for the blog application, which will be built over the course of chapters 5 through 9. In that application, the focus is on posts and people (bloggers). As such, the design should initially address text. A lot of color and other flourishes are probably okay, but in the spirit of simplicity, use of space and typography will be our primary concerns. So, we'll focus on placement, workflow (find posts, read posts, and follow an interesting author), and making things clear and easy to read.

The following diagram highlights the application design aspect of an SPA design:



The application design aspect.

The diagram shows how a few simple design procedures translate directly into implementations in our front-end application architecture. With a bit of upfront planning, we can save a lot of time. You don't have to have an eye for design to follow a process which will make ongoing code decisions a lot easier. In the next chapter, before we actually start writing the code, we'll make a few lists and diagrams. Right now, as a preview, let's explore these design procedures, what the output of each looks like, and how it will serve us when we actually begin to write code.

The following design procedures use an email application as an example. We'll repeat these procedures for the blog application in the next chapter.

Creating wireframes







This is a screen designing process. Sometimes it's good to just start sketching. Start drawing an interface that captures the placement of items and core user interactions in the application. There are a lot of tools that can be used for wireframing, but I prefer to dive in with a pen and paper. Color, fonts, and other final touches aren't necessary here. The aim is twofold:







- **Information hierarchy:** Spatial relationships and placement of data on screen
- **User interactions:** Navigation and user workflow (menus, links, and so on)

For the information hierarchy, think about the user goal in each main view, and how they would use each main component in sequence within each workflow. This will help you choose the correct size and placement in order to guide their eyes through their tasks linearly.

For interactions, stick with precedent. A chief principle of usability is "don't make me think". It turns out that *Don't Make Me Think* is also a very concise book on usability by Steve Krug, and is recommended reading for anyone making interfaces on the Web. Try to put primary and secondary navigation in a typical area. If you have primary action buttons, such as confirm buttons, give them a bit more visual weight and put them near the right edge of the screen (closer to a thumb on a mobile device).

The following diagram shows a wireframe for a familiar type of web application, an email web app:

FIND AND READ EMAIL				
 NEW EMAIL		<input type="text" value="Search"/> 		LOGO
FOLDERS		< FOLDER NAME OR SEARCH TERMS >		
▼ Inbox _____ _____ ► Sent Junk	 FROM NAME unread subject	SUBJECT _____ _____ _____	FROM DATE/TIME _____ _____ _____	
	 from name read subject			
	...			
CONTACTS				
 _____  _____			➔ FORWARD ↩ REPLY	
< CURRENT DATE/TIME > ☀ 76°				

CREATE NEW EMAIL				
 NEW EMAIL		<input type="text" value="Search"/> 		LOGO
FOLDERS		NEW MESSAGE		
▼ Inbox _____ _____ ► Sent Junk	Subject <input type="text"/>			
	To <input type="text"/>			
	Message _____ _____ _____			
CONTACTS				
 _____  _____			 SAVE  SEND	
8:38 PM 6/14/2015 ☀ 76°				

In the email application wireframe example, you can see how using a simple format such as a pen and paper can establish focus on component placement, relative size, and user workflow without the distractions of color, font choice, specific graphics, and the like. For your applications, you'll want to make a handful of these to capture the primary views that service main user goals. For email, this could be as few as two: "find mail" and "create new mail".

Examining the wireframes also forecasts reusable components. In the example image these would be: the header, contacts control, icon button, email list, and so on.

Main data entities and the API

Make a list of the main data entities in your app. This list should be quite short. One of the entities is probably the main focus of the app. For example, in an email app, the most important data item by far is the email! Alongside that, you'd probably have contacts and folders. If this list starts getting long, perhaps you are getting a little too detailed. To get an initial list, think about what objects a user would expect to see on a main page or screen of your app if you were to ask them conversationally.

Next, you can define an API for data persistence by writing out each entity and the standard list of verbs for data augmentation: **C.R.U.D.** (**create**, **read**, **update**, and **delete**). In the case of a RESTful API for a web application, the verbs would be **POST**, **GET**, **PUT**, and **DELETE**, respectively. Here's an example for an email application:

Entity name	Data members	Operations
Mails	<ul style="list-style-type: none">• mail uid• folder uid• from email• from name• to• subject• body• new	<ul style="list-style-type: none">• create• read• delete
Folders	<ul style="list-style-type: none">• folder uid• folder name	<ul style="list-style-type: none">• create• read• update• delete
Contacts	<ul style="list-style-type: none">• contact uid• contact name• contact email• contact photo	<ul style="list-style-type: none">• create• read• update• delete

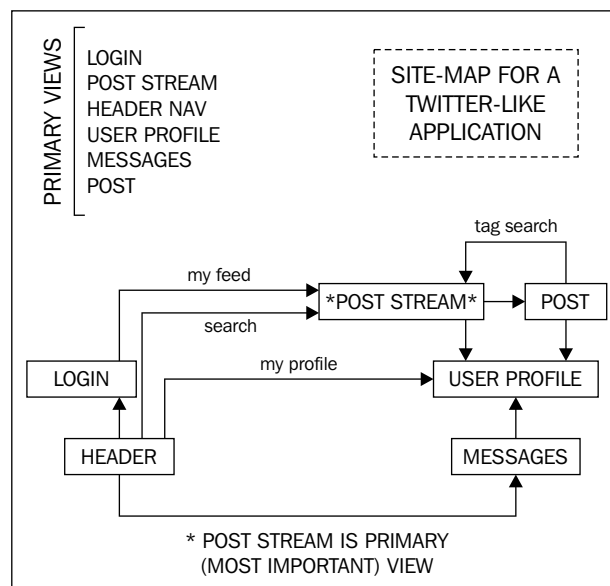
Most main entities will have the full complement of data operations. Often, they will have a read operation to get one instance and one or more read operations to get a collection of instances. Also, expect every main data entity to have a unique identifier (uid).

Main views, site map, and routes

Identifying the main views in the app is simple. They are usually just the user goals in regard to the main subject of the app. If you decide to wireframe most of your app, you've probably already identified all of these views. In the email app, two obvious main views could be a list of emails (inbox) and "create/edit email". You'll probably want to add the primary navigation view (often a header) to this list even though it's likely embedded in many or all of the main views. It will help when making the sitemap.

Once you have the main views figured out, you can make a sitemap. Sitemap is a bit of a misnomer here. After all, this isn't a website; it's an application! Still, this term is somewhat apt since it's also used as the name of a real artifact that can be used to index your application for **Search Engine Optimization (SEO)**.

Make a box for each main view and draw lines for user navigation between them. Don't be surprised if your sitemap looks like a spider web instead of a tree. This is the difference between a sitemap for an application, which deals with views, and one for a website, which deals with a hierarchy of pages. The following is an example of a sitemap for a Twitter-like application:



A wireframe for a twitter-like application

Finally, we've reached the goal of this exercise that will allow us to begin coding in earnest. Armed with the main views and the sitemap, you can now easily produce your router configuration. Each main view will be able to be linked and bookmarked. This final step will be especially useful when we design the blog application in the next chapter. The sitemap will allow us to scaffold our application workflows using React Router before filling in the details for the view logic and data management.

Summary

After reviewing a list of application aspects and some of the tools, it's apparent that programming applications for the web browser has become quite complex! Of course, if your application is very small and limited to only a single workflow or two, you may like to omit some of these tools. Although, it's best to become comfortable with all of them. They are the tools of your trade. Indeed, the reason for reaching for a tool like React is because it reduces complexity and affords power and performance for very complex interactive applications such as Facebook, the impetus for React's existence.

Finally, it's important to understand that just knowing a list of problems and possible tools isn't enough to effectively compose them into something complete. Some forethought and a design procedure can greatly improve both the coding process and the end result. A planning process is as important a part of an application developer's repertoire as their ability to decompose the parts of the app and choose software.

In the next chapter, we'll reiterate some of the tool choices here and repeat the design procedures for a multi-user blog application. The artifacts produced by this process will then be used to guide the construction of the app.

5

Starting a React Application

The aims of this chapter are to formulate a plan, set up the environment, and scaffold the code. First, we'll work through the application design tasks described in the previous chapter. Then, we'll fetch development tools and configure the programming environment. In the end we'll have a running skeleton of our application and will have covered the following technical subjects:

- **Webpack:** This is the build automation and development server. We'll get a basic configuration working that will service ES6 and JSX compilation, code bundling, hot loading React components, and polyfilling.
- **React application structure:** Though there are many ways to arrange the parts of your application using scaffolding tools, such as **Yeoman**, they make a lot of assumptions during the process. In this chapter we'll arrange the structure ourselves.
- **React router:** The user experience starts at the address bar or a link to your application. There are many expectations that come with using an application or website hosted in a web browser, such as bookmarking and deep linking. Setting up the router early is prudent since this is where everything really begins.

Application design

In *Chapter 4, Anatomy of a React Application*, we looked at a few design tasks that help to establish intent before coding in earnest. We will repeat those tasks for our blog application. The ultimate goal is to support blog entry for multiple users.

Creating wireframes

Starting with pen and paper is an approachable way to define a problem. We know we'll need a blog post entry screen to author rich text, a means to sign up a new user, a means for that user to log in, and ways to view the posts and the users.

User-related views

The user-related views include not only the ones that manage the user entity, but also those that manage the login session. The following figure shows the log in view:

The diagram shows a rectangular window titled "LOG IN". Inside the window, there is a header bar containing the text "REACTION" on the left, a "search" input field in the center, and the text "JOIN" and "LOG IN" on the right. Below the header bar, the main content area is titled "LOG IN" and contains two input fields labeled "username" and "password", followed by a "LOG IN" button.

Log in view

The log in screen is the simplest form in the application: just one heading, two fields, and a submit button.

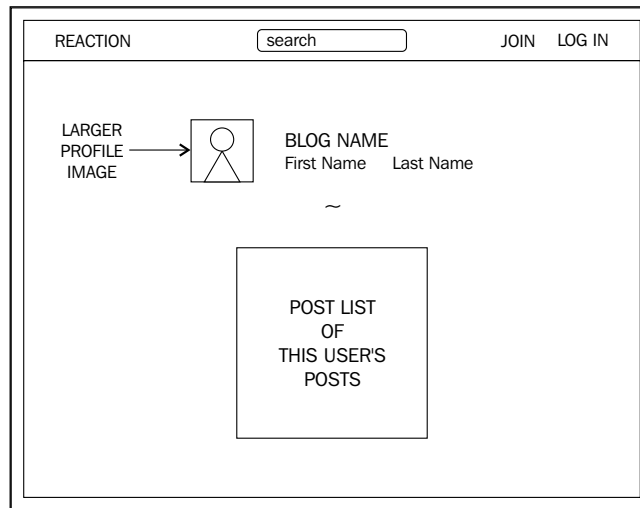
Here is the user sign-up screen. It is the largest and most complex input form in the application:

The diagram shows a rectangular window titled "SIGN UP". Inside the window, there is a header bar containing the text "REACTION" on the left, a "search" input field in the center, and the text "JOIN" and "LOG IN" on the right. Below the header bar, the main content area is titled "BECOME AN AUTHOR" and contains several input fields: "blog name", "username", "password", "profile image" (with a small person icon and a "CHOOSE" button), "first name", "last name", and "email". At the bottom right of the main content area is a button labeled "I'M READY!".

User create (sign-up) view

The sign-up screen could also be used to edit an existing user account. Using the same component for creation and editing is a common practice. When we implement this screen in the next chapter, we will use a trick to get a profile image from disk and persist it to the document database as text.

Finally, the user view displays a read-only form of the user profile and a filtered list of posts authored by the specific user:



User view

Our first representation of a user includes the profile image, the name of their blog, and their name. The user's posts will be displayed beneath the profile information.

Post-related views

The wireframes in this section constitute the screens for creating and viewing posts. First, let's look at the wireframe for the post creation view:

A wireframe for a post creation form. At the top, there is a header bar with the word "REACTION" on the left, a search input field in the center, and the text "HELLO NAME: WRITE" and "LOG OUT" on the right. Below the header, the main content area contains a "POST TITLE" label followed by a dotted line representing an input field. Below the title is a tilde symbol (~). Underneath is a large rectangular box labeled "<RICH TEXT CONTROLS>" containing the text "Hello World!". At the bottom right of the main content area is a button labeled "POST".

Create post view

This is where we'll put our rich text editor, Quill. The post title is a large, prominent, input field followed by a separator.

The next wireframe depicts the default view for the application, the post list view:

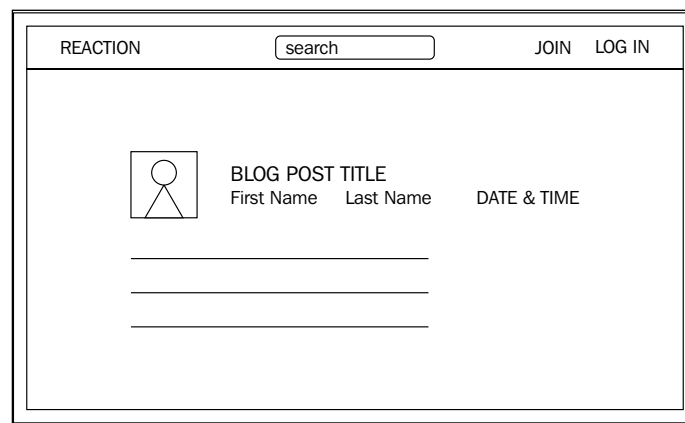
A wireframe for a post list view. The header bar at the top contains the text "SHOW ALL POSTS" in the center, "REACTION" on the left, a search input field, and "JOIN" and "LOG IN" on the right. The main content area displays a list of posts. The first post shows a "BLOG TITLE" with a user icon, "blogger name", and a timestamp "MM/DD/YY H:M:S". To the right of the title are two more user icons with labels "BLOG NAME username" and "BLOG NAME 2 username". Below the title is a "Summary text" followed by a dotted line and a "read more" link. An "EDIT POST" button is positioned to the right of the summary text, with an arrow pointing to it from the text "POST BELONGS TO LOGGED IN USER". Below the first post is a second post titled "BLOG TITLE 2" by "other blogger", followed by a vertical ellipsis. At the bottom left, there is a "LOAD ON SCROLL" label with an arrow pointing to the text "LOADING...", and an "All LOADED" label with an arrow pointing to the text "Showing X posts".

Post list view

This is the home view. It depicts the application header, the list of all posts, a loading message when the user scrolls, and a final **Showing X Posts** message with the total number of posts currently loaded. The user list appears on the right side of the screen. Clicking one of the users navigates to the user view screen shown in the previous figure.

There are two instances of obvious component repetition: a list of posts that now appears both on the user view and this post list view, and a representation of the user that has the photo, the name of their blog, and their name. When we set up the file structure, we'll account for these as reusable components.

Finally, the last post-related wireframe is for a single post entry:



View Post

The top portion of the post view is a mixture of user data and post information. The post title, date, and time are from the post data and the user photo and name are from the user data. The lines in the wireframe represent rendered markup created by the Quill editor from the create post view.

Data entities

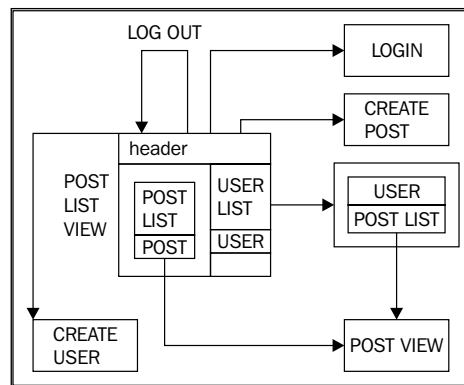
Users and posts are the primary data entities. Post deletion and user editing and deletion are left unimplemented for the sake of brevity, but could easily be added by you later. Other enhancement ideas are suggested at the end of the application tour in *Chapter 9, React Blog App Part 4 – Infinite Scroll and Search*.

Entity Name	Data Members	Operations
Posts	<ul style="list-style-type: none">• post id• user id• body• date• summary	<ul style="list-style-type: none">• create• edit
Users	<ul style="list-style-type: none">• user id• blog name• user name• password• profile image• first name• last name	<ul style="list-style-type: none">• create

Main views and the sitemap

Almost every main view is already sketched up from our wireframes. To capture the user workflows through the app, add the header bar component to that list. The header component is the first place an author will go to sign up, log in, or create a new post. It's the most explicit navigation in the application, resulting from direct user interaction. The navigation that occurs as a result of clicking on a post or user is subtler. The navigation that occurs after completing the create user or create post forms is automatic. The obvious destination choice is the read-only view version of the successfully submitted item.

All of the aforementioned navigation is represented in the next figure as an arrow in our final design task, the sitemap:



Sitemap

It is apparent from the wireframes that the header is in each main view, but it's only represented on the home view (all posts) of the sitemap in order to depict its navigation options just once.

Preparing the development environment

Before we start coding, we'll need to get our development tools installed and configured. This involves installing some Node modules and writing the Webpack configuration file.

Installing Node and its dependencies

Node.js is needed to run Webpack automation, the Webpack dev server, and the JSON mock server. Head over to [Nodejs.org](https://nodejs.org) and follow the installation instructions for your operating system. You'll also need a terminal to run commands and view the output of Webpack as it runs compilation steps. The Windows terminal is serviceable, but if you install Git for Windows it comes with a better shell called git-bash (which is part of MinGW, a minimalist GNU environment for Windows).

Initialize a Node project by running `npm init` and answering the prompts. The defaults will work for us, so you can just press *Enter* and accept each default. This will create the `package.json` file, which will contain a manifest of module versions for our development dependencies. If you haven't already, open a terminal, create a new directory, and execute the command:

```
npm init
```

Before installing and configuring Webpack, some Node packages must be fetched; these will be needed for the application code. If you need a refresher on why we've chosen these particular packages you can flip back to the previous chapter.

As mentioned before, Node Package Manager (npm) uses the configuration file named `package.json`. This file contains some metadata about your project, but the most important part is the manifests of modules and their respective versions. Node and npm use a pragmatic versioning scheme called **semver** (semantic versioning). Semver makes it clear when there's a breaking API change in a module revision. The version numbers consist of three components: major, minor, and patch. Different versions of the major component indicate API incompatibility. Different versions of the minor component denote backward-compatible API additions within the same major version. Changes to the patch component indicate defect fixes, and are always backward-compatible. The packages listed in `package.json` will each have a complete version, often with a tilde '~' in front of it. This means any non-breaking version near the stated version number will suffice when fetching updates or installing a fresh set of modules when there are none currently in the `node_modules` folder.

It can be tedious to manage the versions in `package.json`. Luckily, when you execute `npm install` you can ask npm to add version detail for a newly fetched module to the configuration in `package.json`. There are two kinds of dependencies: ones that are required for the app to run in production, and ones that are only needed for development. To make npm append the version detail to the file for application runtime dependencies, you can add the option `--save` to your `npm install` command. To get version information added to `package.json` for development dependencies, use `--save-dev`. Note that these web projects move very fast and change APIs from time to time. If you have problems with any of the code in this book, it's a good idea to reference the versions of modules referenced in the `package.json` file included in the respective chapter's code zip file.

To install our application dependencies, run the commands below. Alternatively you can just use the `package.json` file from one of the code zip files for this chapter.

```
npm install --save react
npm install --save react-dom
npm install --save react-addons-update
npm install --save react-router
npm install --save history
npm install --save reflux
npm install --save reflux-promise
npm install --save superagent
npm install --save classnames
npm install --save quill
npm install --save moment
```

Let's talk briefly here about Moment. If you code your own date manipulation math and formatting, you will have a bad time. It may seem simple on the surface, but it is definitely not. Moment is the premier date library for JS. Why aren't there loads of date libraries like everything else in JS? Because people don't want to (and usually shouldn't) code date and time math!

There is another item here that was not discussed in *Chapter 4, Anatomy of a React Application*. `Classnames` is a generic CSS class name string constructor module with semantics very similar to `ng-class` in Angular. It's a handy way to construct class names together based on the state of our React components. So handy, in fact, that it was part of the React addons for a while before it was broken out into a separate utility. You can find the documentation and source on GitHub under JedWatson's account.

The `react-dom` package is necessary for rendering and finding DOM elements. It was split from the main React package in version 0.14 in order to make things more modular. Similarly, React addons have also been split out. We'll use the `react-addons-update` module to create copies of objects.

The `history` module is used by `react-router` to manage browser history.

An update to the Reflux project split out the promise interface for asynchronous actions. So, `reflux-promise` is included here to add the promise interface back into those actions.

Installing and configuring Webpack

Install Webpack and the Webpack dev server. Here the Webpack dev server is installed globally so that the command is easily available from any command path.

```
npm install --save webpack
npm install -g webpack-dev-server
```

Within our Webpack configuration we are going to use the following: the Babel JS transpiler for ES6 and JSX, the React hot-loader so we'll have to refresh our browser less often, and the Webpack dev server to host our application locally. Reusable Webpack components are called **loaders**. You'll find that similar sorts of pluggable pieces are called tasks in **Gulp** or **Grunt**. They are transformations on source files performed in a pipeline fashion. Install each of these modules using the `npm` command:

```
npm install --save babel
npm install --save babel-core
npm install --save babel-polyfill
npm install --save babel-loader
```



```
npm install --save babel-preset-es2015
npm install --save babel-preset-react
npm install --save-dev react-hot-loader
```

A few more loaders and supporting modules are needed for Less CSS pre-processing:

```
npm install --save less
npm install --save less-loader
npm install --save style-loader
npm install --save css-loader
npm install --save autoprefixer-loader
```

Finally, install the mock dev server for our REST interface.

```
npm install -g json-server
```

We'll need to restart the `webpack-dev-server` from time to time. To make this simple, add a script to the `package.json` file. Inside that file, there's a member called `scripts` with a default `test` target. Alongside the `test` member, add one called `start` with a dev server start string like this.

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "webpack-dev-server --progress --colors --watch"
}
```

Don't forget to add that comma after the `test` value!

The Webpack configuration

The listing for the Webpack configuration file is a bit lengthy. It is broken into sections in the text, but it's really just one JS object. Once this configuration file is written, it still won't be ready to run until we start scaffolding the entry points. You can follow along with the listing by looking at the `webpack.config.js` file included in `ch5-1.zip`.

Webpack configuration starts by including the `path` module and the `Webpack` module. We'll be specifying a lot of directory locations in this file. The `path` module is used to make definitions of file paths simple and safe across operating systems. The `path` module handles operating system differences, such as different slashes used as path separators. The entire Webpack configuration is a single object exported in `webpack.config.js`.

```
var path      = require('path')
,    webpack = require('webpack')
;

module.exports = {
```

Entry and output sections

The Webpack configuration file starts with the `entry` and `output` sections. We only have one entry point and one output file, but you could define multiple ones if desired. The `entry` modules are loaded in order and the last one is exported. The first item in `entry` enables the client portion of our dev server. The second provides an avenue for our react-hot loader to push updated React components to our application without a refresh.

We can get all the non-transformable code-backed polyfills for ES6 by including the babel-polyfill runtime. This includes features such as generators, promises, array map, and many more. Further, this Babel runtime polyfill technique doesn't pollute the global namespace as traditional ones can.

Finally, our app entry point is listed. The `output` value is the file name that will go into the `index.html` file. This final output file will include all of the transformed code.

```
  entry: [
    // WebpackDevServer host and port
    'webpack-dev-server/client?http://localhost:8080',
    'webpack/hot/only-dev-server',
    'babel-polyfill',
    './js/app' // Your app's entry point
  ],
  output: {
    filename: "js/bundle.js"
  },
```

The plugins section

Next is the `plugins` section. `HotModuleReplacementPlugin` allows the server to push changed JS modules into the browser execution context without a page refresh. Next, `NoErrorsPlugin` will prevent erroneously built code from propagating into our hot-loaded browser environment, where it would certainly cause exceptions. Error output from the build will still appear on the console where we execute our dev server via `npm start`, but the `bundle.app.js` file will not be rebuilt and replaced until the compile errors are resolved.

I've left a small snippet in the plugins section for an array utilities polyfill. We won't need it since Babel will provide array utility functions (such as `map`, `forEach`, `reduce`, and so on) as part of the transpilation process. The commented out example is left in the listing to show how to include some global JavaScript in your Webpack bundle for any cases where global polyfilling is needed.

```
plugins: [
  new webpack.HotModuleReplacementPlugin(),
  new webpack.NoErrorsPlugin(),

  // example of polyfilling with webpack
  // alternatively, just include the babel runtime option below
  // and get Promises, Generators, Map, and much more!
  // You can even get forward looking proposed features
  // for ES7 and beyond with the
  // stage query parameter below
  // https://babeljs.io/docs/usage/experimental/
  // welcome to the future of JavaScript! :)
  //new webpack.ProvidePlugin({
  //  'arrayutils': 'imports?this=>global!exports?global.
arrayutils!arrayutils'
  //})
],
```

The resolve section

This section is for file location resolution. The `extensions` array is used when Webpack attempts to locate an imported code file in our application code. Alias is just what it sounds like, short names for paths during module search. Since Node has a handy `__dirname` variable we can make an alias for our application root. We'll use this throughout the app when we import modules instead of grappling with relative paths.

```
resolve: {
  // require files in app without specifying extensions
  extensions: ['', '.js', '.json', '.jsx', '.less'],
  alias: {
    // convenient anchor point for nested modules
    'appRoot': path.join(__dirname, 'js'),
    'vendor': 'appRoot/vendor'
  }
},
```

The module section

The module section contains Webpack loaders. The loaders look cryptic, but they are simply a transformation pipeline for text files. They are applied by testing filenames using regular expressions. We've added the Less autoprefixer plugin to the transformation pipeline for `.less` files in order to automatically inject CSS vendor prefixes for the browsers specified by the `browsers` parameter.

The most interesting loader section in our list is the one for `.js` and `.jsx` files. This loader pipeline runs Babel on our files; this does both the ES6 and JSX transformations. When a loader pipeline contains multiple items, as it does here, they are applied from right to left. The react hot loader is to the left of Babel so that it runs after it.

```

    module: {
      loaders: [
        {
          test: /\.less$/,
          loader: 'style-loader!css-
loader!autoprefixer?browsers=last 2 version!less-loader'
        },
        {
          test: /\.css$/,
          loader: 'style-loader!css-loader'
        },
        {
          test: /\. (png|jpg) $/,
          // inline base64 URLs for <=8k images,
          // use direct URLs for the rest
          loader: 'url-loader?limit=8192'
        },
        {
          test: /\.jsx?$/,
          include: [
            // files to apply this loader to
            path.join(__dirname, 'js')
          ],
          // loaders process from right to left
          loaders: [
            'react-hot',
            'babel?presets[]=react,presets[]=es2015',
            'reflux-wrap-loader'
          ]
        }
      ]
    } // end module
  };

```

Babel is separated completely into submodules as of version 6. This means that, without these other modules, Babel will do nothing to transform the files. The Babel submodules are called **presets** in Babel parlance. Here we have included the `react` preset, which handles the JSX compilation, and the `es2015` preset, which provides all of our ES6 goodness.

Ahead of the Babel and `react-hot` loaders is the `reflux-wrap-loader`. The source for this loader should be located in the file `web_modules/reflux-wrap-loader/index.js`. Go ahead and make this directory structure, and use the code listing below for the `index.js` file inside the `reflux-wrap-loader` directory. Webpack automatically searches the `node_modules` and `web_modules` directories for loaders. The `reflux-wrap-loader` is a simple example of a loader.

```
module.exports = function (source) {
  this.cacheable() && this.cacheable();
  var newSource;

  if (/reflux-core.*index.js$/.test(this.resourcePath)) {
    newSource = ";import RefluxPromise from 'reflux-promise';\n";
    newSource += source;
    newSource += "\nReflux.use(RefluxPromise(Promise));";
  }
  return newSource || source;
};
```

As mentioned before, a loader simply transforms text files. Since the `Reflux` project split out promises into a separate package, it's necessary to call `Reflux.use` on the `reflux-promise` package. There's not a great place to do this in the application modules, since every module would have to do it to be sure it was done without worrying about execution order. So, instead it's done here in a loader by wrapping the `Reflux` module and adding the invocation for the `import` and the `use` method. The loader matches on the `reflux-core` file and frames it with the invocation text to include the promise interface. This loader will be run before the Babel transformation, so we can use the ES6 `import` syntax without generating a build error. The loader will run on every file so you must be sure to return the original source if you want the loader to do nothing to the file. The `cacheable()` invocation instructs Webpack that, after the transformation, the result for the `reflux-core` file can be cached indefinitely.

Considerations before starting

Whew! I think we're ready to start, but before we dive into the code I want to impart how I believe you should think about the render step in the React component lifecycle as well as a way to approach browser support and form validation.

React and rendering

The way you should think about the React render function is much like the definition of a mathematical function: $f(\text{state}) = \text{UI}$. React treats the DOM as a rendering target, much like a computer program or game treats the **Graphics Processing Unit (GPU)**. State is kind of like the arrays of object data (vertices, and so on). That data is prepared for shipment to the GPU, and the rendering portion of the component lifecycle is kind of like the OpenGL rendering pipeline that consumes that data (state). The render function itself would be the shader code that processes the geometry and pixels in this analogy, and the virtual DOM would be the framebuffer. In the GPU analogy, object data and state go into the render pipeline and the result is an array of pixel data. In React, state goes into the render pipeline and a virtual DOM tree is the result.

This means that, when a potential render cascade begins (`shouldComponentUpdate` à `componentWillUpdate` à render), you shouldn't change the state. It's already too late for that sort of thing. That's worth saying again; the render function should be pure and never affect state or props. You are welcome to make intermediary variables in the render function to create projections (transformations) of state that are easier to consume by render logic, just don't change the state itself.

This application structure is quite powerful and is the reason that the React Native project can exist. The DOM isn't something that you typically interact with directly in React as you would with jQuery, for instance. Other native targets, such as iOS Cocoa, are just other rendering targets with their own specific render function. This one-way structure and focus on efficiency (for what is often the most expensive part of an application, getting pixels onto the screen) is what makes React special and, in some ways, more flexible for cross-platform development than other JavaScript libraries.

Browser support

For our application we want to use modern browsers: browsers that the vast majority of people use. This means being a bit choosy. Of course, in a real scenario you have to consider the target users who will actually use the application and support them, even if it means substantial additional effort. For example, if you make a government services website you may need to support an older IE browser if some users are citizens who use library computers that are often older and locked into running older browsers.

Conventional wisdom in the web community often asserts that we should start with the lowest common denominator (that is, the worst browser in terms of feature support) and work our way up to the fancier browsers. This is known as progressive enhancement. This is slightly at odds with a technique known as polyfilling in which newer features are back-ported to older browsers.

Polyfills are JavaScript code included to port features into browsers that do not yet support them natively. They enable you to write code as if the feature exists in an older browser. If it does already exist in the browser, the polyfill does nothing and the native code is used. They are often very cheap in terms of code size and performance. While not always on par with native speed, performance is acceptable (in some cases better!). A moment ago, I mentioned that this is slightly at odds with progressive enhancement because, with polyfills, you write code as if those features exist in browsers where they do not. However, it's not entirely at odds with the progressive enhancement strategy. One could argue that starting with the lowest supported browser set, then polyfilling, and then moving on to more complex features that will only be exhibited in newer browsers still fits the strategy. If you are supporting older browsers and following the progressive enhancement strategy, then significant user goals should still be achievable on those older browsers.

For our purposes (learning new stuff), we are focusing on interesting new technology and getting a prototype running easily. So, we'll start with our desired tech and fill it in using polyfills where we are compelled to do so, but we'll avoid writing two or more specific portions of code that have the same effect for two or more browsers.

Using the Babel runtime trick that you saw in the Webpack configuration section eliminates our immediate need for traditional polyfilling.

Form validation

In *Chapter 3, Dynamic Components, Mixins, Forms, and More JSX* there was some discussion about validation. In that discussion, three means of immediate (field-level) validation were explored: view level, view model level, and model level. A conclusion there stated that, for system consistency, a model containing the constraints and a shared mechanism to exercise constraints was an ideal architecture. While that is still the contention of this text, in this chapter we want to focus on application structure, mostly views. So, we are going to cheat a little and do minimal validation in the view via a small helper. The constraints will reside inside their respective view components. This is, in part, due to the fact that we are using JSON Server as a mock back-end. To build a model of validation constraints while using a simple document store would take time away from exploring our primary application architecture components (views, stores, and actions) and their interconnections.

Starting the app

We will begin exploring the construction of the application in detail in the next chapter. To finish up here, we'll lay the groundwork for file structure, and stub out the main views. We'll boot up our dev server and mock back-end, then add some linking between views using React Router.

The directory structure

Make a directory structure that looks like this:

```
.
├── css
│   ├── components
│   │   ├── posts
│   │   └── users
│   ├── vendor
│   └── views
│       ├── posts
│       └── users
├── db
└── js
    ├── components
    │   ├── posts
    │   └── users
    ├── mixins
    ├── stores
    ├── vendor
    │   └── polyfills
    └── views
        ├── posts
        └── users
```

If you are using a POSIX shell (such as Bash, the default shell on Mac OSX), here are a couple of commands to quickly create the directory structure:

```
mkdir -p db css/{components,vendor,views} js/{components,mixins,stores,vendor,views}
mkdir -p {css,js}/{components,views}/{users,posts} js/vendor/polyfills
```

Notice that the components and views subdirectories in `js` are mirrored in the `css` directory. Mirroring view and component structures in the style directories is an easy way to keep track of which styles belong to which JS constructs. Don't forget the directory made earlier for the `reflux-wrap-loader`.

The mock database

In the `db` directory, create a file called `db.json` with the following content:

```
{ "posts": [], "users": [] }
```

That's it! That's our database for `json-server`. If you want to try running it, open a new terminal and execute this command from the root of your app:

```
json-server db/db.json
```

index.html

The `index.html` file is merely a shell to include the application as bundled by Webpack.

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript">
      WebFontConfig = {
        google: { families: [ 'Open+Sans:300italic,400italic,600italic,700italic,800italic,400,300,600,700,800:latin' ] }
      };
      (function() {
        var wf = document.createElement('script');
        wf.src = ('https:' == document.location.protocol ? 'https' : 'http') +
          '://ajax.googleapis.com/ajax/libs/webfont/1/webfont.js';
        wf.type = 'text/javascript';
        wf.async = 'true';
        var s = document.getElementsByTagName('script')[0];
        s.parentNode.insertBefore(wf, s);
      })();
    </script>
  </head>
  <body>
    <div id="app"></div>
    <script src="js/bundle.js"></script>
  </body>
</html>
```

You may remember that our app bundle output file is `js/bundle.js` from the Webpack configuration. There is a `<div>` tag with the id `app` that we'll target to render the React app. The script tag at the top is font loading code taken from Google Fonts.

js/app.jsx

This is where it all starts. This main application file includes the React Router, primary views, and the router configuration.

```
import React      from 'react';
import ReactDOM   from 'react-dom';
import { Router, Route, IndexRoute } from 'react-router';
import CSS        from '../css/app.less';
import AppHeader  from 'appRoot/views/appHeader';
import Login      from 'appRoot/views/login';
import PostList   from 'appRoot/views/posts/list';
import PostView   from 'appRoot/views/posts/view';
import PostEdit   from 'appRoot/views/posts/edit';
import UserList   from 'appRoot/views/users/list';
import UserView   from 'appRoot/views/users/view';
import UserEdit   from 'appRoot/views/users/edit';

// Components must be uppercase - regular DOM is lowercase
let AppLayout = React.createClass({
  render: function () {
    return (
      <div className="app-container">
        <AppHeader />
        <main>
          {React.cloneElement(this.props.children, this.props)}
        </main>
      </div>
    );
  }
});

let routes = (
  <Route path="/" component={ AppLayout }>
    <IndexRoute component={ PostList } />
    <Route
      path="posts/:pageNum/?"
      component={ PostList }
      ignoreScrollBehavior
    />
    <Route
      path="/posts/create"
      component={ PostEdit }
    />
    <Route
```

```
        path="/posts/:postId/edit"
        component={ PostEdit }
      />
      <Route
        path="posts/:postId"
        component={ PostView }
      />
      <Route
        path="/users"
        component={ UserList }
      />
      <Route
        path="/users/create"
        component={ UserEdit }
      />
      <Route
        path="/users/:userId"
        component={ UserView }
      />
      <Route
        path="/users/:userId/edit"
        component={ UserEdit }
      />
      <Route
        path="/login"
        component={ Login }
      />
      <Route path="*" component={ PostList } />
    </Route>
  );

ReactDOM.render(<Router>{routes}</Router>, document.
  getElementById('app'));
```

React and React Router are imported because most of this file is router configuration. Next is a set of imports for our top-level views. Those are the views designed in the wireframes and sitemaps. After the imports, we see our first React component, which represents the app itself: `AppLayout`. The root of the router configuration will target this component.

After the application component, the JSX representing the routing table is defined as **routes**. This router configuration maps URL paths in a nested structure directly to the views that will be composed into the application component. These tags won't actually be rendered. They are used as configuration.

Each route has a path for the URL and a component, which is one of the top-level views imported at the top of the file. A special tag named `IndexRoute` could be considered the home view. Our home view is the `PostList` top-level view that shows all the blog posts. Finally, there's a `*` route that's like a 404 handler for the front-end. By putting `PostList` as the handler for this route we ensure that, if someone types in a random URL, the app will route to the home view.



Be sure to capitalize the names of your components. In React, lower-case component names are reserved for built-in components that correspond to DOM tags.

Last, the `Router` component is rendered. This causes the router to begin listening to URL changes. The top route `/` uses the `AppLayout` component. The components that should be rendered by the routes are supplied as children. Those children are included in the `AppLayout` and any relevant props supplied through the router are applied to them through the `React.cloneElement` method within the `AppLayout` source.

Main views

If we try to run the code now, it will emit errors because our imports in `app.jsx` won't resolve to files on disk. So, go ahead and make some components for the main views. After that, we'll wire them up to exercise the links shown as arrows in our sitemap.


The main views all reside in the `js/views` directory. For each of the main views, `login.jsx` and `appHeader.jsx`, as well as `edit.jsx`, `list.jsx`, and `view.jsx` in both the `posts/` and `users/` directories, make a simple React component container. We'll start each of these by importing React. This goes at the top of each file.

```
import React from 'react';
```

Then, export a React component with the minimum requirement, a render function.

```
export default React.createClass({
  render: function () {
    return ();
  }
});
```

The render function needs to return React DOM. Using the following table for each of the files, add a JSX tag inside the return parentheses with a `className` and some content that has the name of the component. Each of these files, once again, is in the `js/views` directory.

 Note: Your render function return value, and any other portion of JSX code you set on a variable, must always have only one root tag.

File	JSX
login.jsx	<code><form className="login-form">login form</form></code>
appHeader.jsx	<code><header className="app-header">app header</header></code>
posts/edit.jsx	<code><form className="post-edit">post edit</form></code>
posts/list.jsx	<code><div className="post-list-view">post list view</div></code>
posts/view.jsx	<code><div className="post-view-full">post view</div></code>
users/edit.jsx	<code><form className="user-edit">user edit</form></code>
users/list.jsx	<code><ul className="user-list">user list</code>
users/view.jsx	<code><div className="user-view">user view</div></code>

Before booting up the Webpack dev server, create the `app.less` file also referenced by an import in `app.jsx`. Just add an empty file for now.

Now, boot the dev server by executing `npm start` in another terminal. If you navigate to `localhost:8080` in your browser, you should see a pretty sparse page that says **app header** and **post list view**. You can also visit the routes we defined and see the stubbed views.

At this point your code should look like the code in `ch5-1.zip`.

Linking views with React Router

There are three main ways to transition views through React Router. The first simply changes the URL in the address bar of the browser. The second involves using the `Link` component supplied by the library. The third employs the `History` mixin, which will surface the `pushState` method on your component. We'll look at these usages for the app. The next few changes can be found in `ch5-2.zip`.

js/views/appHeader.jsx

The application header component needs a link to the login view. The `Link` component is imported from React Router. The `Log In` link is the simplest form of `Link` without any extra parameters. It is added to the `appHeader.jsx` file render function like this:

```
import React      from 'react';
import { Link } from 'react-router';

export default React.createClass({
  render: function () {
    return (
      <header className="app-header">
        app header
        <Link to="/login">Log In</Link>
      </header>
    );
  }
});
```

js/views/login.jsx

Now that we can get to the log in view, modify the `login.jsx` component with a Log In button.

```
import React      from 'react';
import { History } from 'react-router';

export default React.createClass({
  mixins: [ History ],
  logIn: function (e) {
    this.history.pushState('', '/');
  },
  render: function () {
    return (
      <form className="login-form" onSubmit={this.logIn}>
        <button type="submit">Log In</button>
      </form>
    );
  }
});
```

Here we've added the `History` mixin to get access to the `pushState` function. The component `logIn` function is triggered when the form is submitted. The `pushState` function forwards the user to the root route, `/`. In the next chapter, it will actually log the user in. If your dev server is still running, you should see your browser window update automatically as soon as you save the file.

Summary

In this chapter, we started the application not by jumping directly into the code, but by doing a bit of planning first. The design tasks we explored are a good way to get your priorities straight. We also laid the groundwork for the application by setting up the Webpack build pipeline and scaffolding out all of our main views.

In the next four chapters, the blog application will be built in earnest. This is done in four major parts:

Chapter 6, React Blog App Part 1 – Actions and Common Components

Chapter 7, React Blog App Part 2– Users

Chapter 8, React Blog App Part 3 – Posts

Chapter 9, React Blog App Part 4 – Infinite Scroll and Search

6

React Blog App Part 1 – Actions and Common Components

Armed with the groundwork laid in the last chapter (app design, development environment setup, and file structure), we'll press forward with writing the blog application in earnest. The next four chapters, including this one, are a tour through the blog application code. The application should allow log in and log out. It should also allow multiple users (bloggers) to post. It will eventually include a search feature and infinite scroll loading for post lists. At the end of this chapter, the following will have been covered:

- **React component composition:** Examining our views and sitemap will yield reusable components we can compose into many places in the application
- **Reflux Actions:** A simple messaging system for React

The construction of the application is split into the following four parts:

- **Part 1: Actions and common components**
- **Part 2:** User account management
- **Part 3:** Blog post operations
- **Part 4:** Infinite scroll and search

For now, we are going to examine Reflux Actions (the verbs of the application), some base CSS, and a few components that are used widely within the app: BasicInput, Loader, and the Application header.

Reflux actions

Actions are a simple messaging system. Action configuration has a flexible syntax and, like the rest of Reflux, many shortcut syntactical forms are provided for convenience.

It isn't necessary to go into all of the various forms of syntax here, as the documentation is clear and succinct. All actions have a name and are listenable. The important part is that there are action methods that are immediate and ones that are asynchronous. Action listeners can respond to both types. Asynchronous actions provide a promise interface when `reflux-promise` is used. This promise interface was set up in the last chapter via a Webpack loader, `reflux-wrap-loader`, which wrapped the Reflux module with the necessary invocations to use `reflux-promise`.

Here are the actions defined as a single module:

File: `js/actions.js`

```
import Reflux from 'reflux';

export default Reflux.createActions({
  'getPost': {
    asyncResult: true
  },
  'modifyPost': {
    asyncResult: true
  },
  'login': {
    asyncResult: true
  },
  'logout': {},
  'createUser': {
    asyncResult: true
  },
  'editUser': {
    asyncResult: true
  },
  'search': {},
  'getSessionContext': {}
});
```

It's apparent here which actions are asynchronous. Callers invoking an asynchronous action can use `promise` `then` and `catch` methods to consume the result or any errors. `login` is asynchronous because we'll fetch the users from our prototype back end to recognize an account, but `logout` will merely destroy a cookie on the front end.

Each action is a function object (functor) that can be invoked directly in order to trigger listeners. For example, if the actions module we just defined is imported into another code file as the variable `actions`, then the `logout` action can be invoked like this: `actions.logout()` ;.

We'll see more on how Reflux actions are used in the next chapter when the first action handlers are built. If you would like to familiarize yourself with the `Actions` interface beforehand, head over to the Reflux GitHub page at <https://github.com/reflux/refluxjs>. For now, just take a mental note that there are two ways to listen to actions:

- Using a Reflux Store's `listenTo` or `listenToMany` methods within a store itself
- Using the `listenables` shorthand property within a store

So, stores listen to actions. View components trigger actions and listen to stores. This is the one-way data flow of the Flux and Reflux architectures.

Reusable components and base styles

Grab the `ch6.zip` file to follow along. Much of the CSS is either self-explanatory or beyond our target scope for this text. The top-level application Less/CSS is covered here. It includes the base styles for commonly used tags and overall layout. Later, the `.less` files are called out in the file manifests but not in the chapter code listings. As the application building chapters progress, all of the code, including `.less` files, can be found in the `.zip` file for each respective chapter.

Base styles

The base styles, variables, mixins, and vendor styles, are comprised of these files: `app.less`, `colors.less`, `mixins.less`, `quill.snow.less`, and `normalize.less`. The main application `.less` file, `app.less`, contains the primary layout and some shared styles for the entire app. It's a bit long, but it contains all of the primary styles and layout for main components, such as the header. Here's the `app.less` file, we'll dive into the details after the listing:

File: `css/app.less`

```
@import "vendor/normalize.less";
@import "mixins.less";
@import "colors.less";

html, body {
```

```
    font-family: 'Open Sans' sans-serif;
    height: 100%;
    * {
      font-family: 'Open Sans' sans-serif;
    }
  }
  a {
    cursor: pointer;
  }
  // layout
  .app-container {
    height: 100%;

    .app-header {
      position: absolute;
      top: 0;
      height: 40px;
      width: 100%;
      min-width: 800px;
      z-index: 100;
    }
    main {
      width: 100%;
      min-width: 800px;
      height: ~'calc(100% - 40px)';
      position: absolute;
      top: 40px;
      //padding: 40px 0 0 0;

      &>* {
        overflow-y: auto;
        height: 100%;
        width: 100%;
      }
    }
  }
  // standard across application
  fieldset {
    border: none;

    legend {
      text-transform: uppercase;
      letter-spacing: 2px;
      text-align: center;
    }
  }
}
```

```
        margin: 10px 0;
    }

    .basic-input {
        width: 100%;
    }

    button[type=submit] {
        float: right;
        margin: 20px 0 0 0;
    }
}
hr {
    width: 50px;
    content: '';
    position: relative;

    border-width: 0 0 0 0;
    height: 20px;

    &:before {
        position: absolute;
        display: block;
        color: #aaa;
        content: '§';
        font-size: 18px;
        transform: scaleX(6) rotateZ(-90deg) ;
        height: 10px;
        font-weight: 100;
        line-height: 10px;
        width: 20px;
        text-align: center;
        left: 50%;
        margin-left: -10px;
        top: 5px;
    }
}
button {
    background-color: #bbb;
    color: black;

    line-height: 40px;
    text-transform: uppercase;
    font-size: 12px;
```

```
letter-spacing: 1px;
padding: 0 10px;
border: none;
outline: none;

&[type="submit"] {
  background-color: darken(@blue, 20%);
  color: white;

  &:focus,&:hover {
    background: #000099;
  }
}
box-shadow: 1px 3px 2px 0px #666;
transition: transform 100ms ease, box-shadow 100ms ease;

&:active {
  transform: translateY(2px);
  box-shadow: 1px 1px 2px 0px #666;
}
}
@import "vendor/quill.snow.less";
@import "views/appHeader.less";
```

Starting with the imports at the top, `normalize.less` is a style reset. Use style resets in your CSS to normalize the styles in all browsers so there's an equal starting point for your application-specific styles. The normalize reset is less aggressive than some other resets, which remove all native browser styling.

The `mixins.less` file is where we put reusable functions in for our Less code. Less functions are invocable portions of styles that generate a group of CSS rules in the final output at the point where they were invoked. For this app, the file has just one function called `slidelink`. The `slidelink` function will be used to animate a fancy underline on links in the app header bar upon hover.

The `colors.less` file contains reusable hex color values stored as Less variables.

The rest of the styles in the `app.less` file are used for the general layout of main application views. In this file, there's positioning for the header bar, but the styles for the bar itself will go into a separate file for that component. Most notable are the `fieldset` styles, because they are used in all of the forms. The universal `hr` style is a little squiggle separator used to visually break up fieldsets.

The file ends with an import used for styling the Quill rich text editor as well as an import for the `appHeader.less` file. That `.less` file can also be found in this chapter's code listing zip file, `ch6.zip`.

Inputs and loading indicator

These two components are used in many places throughout the application. Here is a manifest of the files for each component:

- **BasicInput:** `js/components/basicInput.jsx`, `css/components/basicInput.less`
- **Loader:** `js/components/loader.jsx`, `css/components/loader.less`

The BasicInput component

The BasicInput component is a container used to encapsulate an input coupled with help/hint text or an error message. A similar wrapper approach is often used to pair an input with its label.

File: `js/components/basicInput.js`

```
import React      from 'react';
import update     from 'react-addons-update';
import ClassNames from 'classnames';

let Types = React.PropTypes;

export default React.createClass({
  // this is how you enforce property types in React
  propTypes: {
    helpText: Types.string,
    error:    Types.string
  },
  render: function () {
    return (
      <div className={ClassNames({'basic-input': true, 'error': this.
props.error})} {...this.props} >
        <input
          className={this.props.error ? 'error' : ''}
          {...update(this.props, {children: {$set: null}})} />
        {this.props.children}
        <aside>{this.props helptext || this.props.error || ' '}</
aside>
      </div>
    );
  }
});
```

The first item to notice is the `propTypes` member. This React feature enforces type checking on the props passed into the component. It's also a nice way to define the interface to the React component in one place.

There are two interesting parts in the render function. The first is the use of the `Classnames` library. As mentioned in the last chapter, this is a class string builder similar to Angular's `ng-class` functionality. There's an optional "error" class applied to the top-level `div` in the component if the `error` prop is set. The "error" class turns the help text and field underline red. This red underline style, and the rest of the styles for the `BasicInput` component, can be found in `css/components/basicInput.less`.

The second interesting part in the render function stems from the fact that input tags should not have children. Since children are included in the `props` member, the `update` add-on function is used to produce a duplicate `props` member that omits the children from the rest of the props. This copy that omits any children props is used to cascade properties used on the `BasicInput` component tag down to the internal input tag. The children elements composed into a `BasicInput` instance are then rendered after the input tag.

required field

The `BasicInput` component with an inline validation error

Before we're finished with the `BasicInput` component, the import for the respective `.less` file needs to be added at the bottom of our `app.less` file:

```
@import "components/basicInput.less";
```

For future `.less` files in the file manifests, it is assumed that an import should be placed at the bottom of the `app.less` file. Any CSS files not explored in the chapter text reside in the chapter ZIP files.

The loader component

The loader component will be used anywhere we need to display a loading treatment while waiting for a server response. Here is the loader component source:

File: `js/components/loader.jsx`

```
import React      from 'react';
import ClassNames from 'classnames';

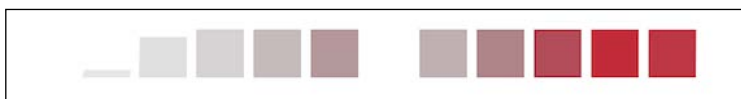
export default React.createClass({
```

```

render: function () {
  // like ng-class, but for React!
  var classes = ClassNames({
    'loader-container': true,
    'inline': this.props.inline
  });
  return (
    <div className="loader">
      <div className={classes}>
        <aside></aside>
        <aside></aside>
        <aside></aside>
        <aside></aside>
        <aside></aside>
      </div>
    </div>
  );
}
});

```

The loader component is made up of five elements (we chose `aside` elements here) wrapped in two containing `div`s. The asides are tiles that flip and fade in sequence. The first wrapper is a positioning container. There's an optional prop, "inline" consumed by this component that adds a class to make the loader flow with the content. This is used for post lists. Without it, the loader is positioned absolutely in the center of the app by default. The second, outermost, wrapper is used to define a perspective container. The flip animation looks more natural (not flat) when there's some perspective applied via this container. Here's what the loader animation looks like in action:



The loader animation

You'll find the animation CSS in `ch6.zip` in the `css/components/loader.less` file.

The application header

The application header will include our log in and log out links and eventually a link to compose a blog entry. It is ubiquitous across all main views. As such, it resides in our main application component and is just one JSX file along with its respective LESS file:

- **Application header**: `js/view/appHeader.jsx`, `css/views/appHeader.less`

File: `js/view/appHeader.jsx`

```
import React    from 'react';
import { Link } from 'react-router';

export default React.createClass({
  render: function () {
    return (
      <header className="app-header">
        <Link to="/"><h1>Reaction</h1></Link>
        <section className="account-ctrl">
          <Link to="/users/create">Join</Link>
          <Link to="/login">Log In</Link>
        </section>
      </header>
    );
  }
});
```

The application header component includes the name of the app, "Reaction", with a stylized "A", that links to the home (post list) view. There are just a couple of links here for now: a link to sign up and a link to log in. Later, we'll add a search box. Also, once the user is able to log in, we'll add a log out link and a link to the blog post entry page.

Summary

We're just getting started. With some base styles in place and our input wrapper component, `BasicInput`, we can start building out real features in the next chapter. We'll start with users so there's an identity that can be attached to each blog entry.

7

React Blog App Part 2 – Users

This chapter covers user session management as well as creating, viewing, and listing users (bloggers).

Our primary focus for the app is posts, but a post must be associated with a user identity. User account management is an often-underestimated and complex part of applications. One of the more difficult aspects of account management is security. Since we are making a mock application, we won't have much security. We are going to establish the user identity with a simple comparison during log in. If we moved beyond this prototype into a real, deployable, application, we'd replace most of this session management code with software that's suited specifically for user identity management.

This chapter comprises all of the code needed to get our user management in order. Since the application is already scaffolded, the code breakdown benefits from being organized by entity type (configurations, stores, and views). By the end of this chapter we'll be able to sign up, log in, log out, list, and view user accounts. The application throughout this chapter can be found in the `ch7.zip` code bundle.

The construction of the application is done in four parts:

- **Part 1:** Actions and common components
- **Part 2: User account management**
- **Part 3:** Blog post operations
- **Part 4:** Infinite scroll and search

Code manifest

Below is a manifest for all of the code in this chapter. You can follow along with code listings in the `ch7.zip` code bundle included with this book.

The user and session context stores are the first time the API endpoint will appear. The API URL root resides in the application configuration module:

- **Application configuration:** `js/appConfig.js`

Getting user accounts fully up-and-running will introduce these dependencies. One is for managing cookies' client-side. The other is for form validation. Here are those dependencies:

- **Cookie reader/writer:** `js/vendor/cookie.js`
- **Form utilities mixin:** `js/mixins/utility.js`

For our application, we are mocking user account management. So, there's a separate store for managing the session and another store for the user data:

- **Session context store:** `js/stores/sessionContext.js`
- **Users store:** `js/stores/users.js`

User-related views, and their respective styles, include:

- **Login view:** `js/views/login.jsx`
- **User edit view:** `js/views/users/edit.jsx`, `css/views/user/edit.less`
- **User view component:** `js/components/users/view.jsx`, `css/components/users/view.less`
- **User list view:** `js/views/users/list.jsx`, `css/views/user/list.less`
- **User view:** `js/views/users/view.jsx`, `css/views/user/view.less`

Another affected view, which we have already defined and will need to update, is the application header:

- **Application header:** `js/views/appHeader.jsx` (add session awareness, welcome, logout)

Application runtime configuration

The `appConfig` module keeps application-level configuration details all in one place.

File: `js/appConfig.js`

```
export default {
  pageSize: 10,
  apiRoot: '//localhost:3000',
  postSummaryLength: 512,
  loadTimeSimMs: 2000
};
```

At this point in development, only the `apiRoot` is needed. The other items you see in the source will be used later. The `pageSize` variable is for the infinite scroll feature we'll implement in *Chapter 9, React Blog App Part 4 – Infinite Scroll and Search*. The `postSummaryLength` member is for the summary post descriptions, which appear in post lists in the next chapter. Finally, the `loadTimeSimMs` is an artificial delay we'll use to get a sense of how the application would feel with non-trivial server communication latency.

Mixins and dependencies

The items in this section contain supporting code for the views. The cookie reader/writer will be used to mock session management. The form utility mixin will be used to validate the individual form elements in all of the forms.

Reading and writing cookies

Maintaining a user session can be complex. To make it as real as possible, the session detail is put into cookies. Reading and writing cookies is a simple parsing process, but there's no need to suffer the minutiae of it. So, we picked up a simple cookie reader/writer JavaScript utility from the **Mozilla Developer Network (MDN)** cookies documentation page (<https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie>). This code was put into the file `cookie.js` and placed in the `js/vendor` folder. In an application with real user session management, the cookies would be secure HTTP-only cookies and JavaScript would not be able to read them.

The form utilities mixin

This mixin will be used to validate form inputs. It is needed at this point for the user creation form and log in view.

File: js/mixins/utility.js

```
import ReactDOM from 'react-dom';

/**
 * returns the failed constraints { errors: [] } or true if valid
 * constraints are a map of supported constraint names and values
 * validators return true if valid, false otherwise
 */
export function validate (val, constraints) {
  var errors = [];
  var validators = {
    minlength: {
      fn: function (val, cVal) {
        return typeof val === 'string' && val.length >= cVal;
      },
      msg: function (val, cVal) {
        return 'minimum ' + cVal + ' characters';
      }
    },
    required: {
      fn: function (val) {
        return typeof val === 'string' ?
          !/^\\s*$/.test(val) : val !== undefined && val !== null;
      },
      msg: function () {
        return 'required field';
      }
    },
    exclusive: {
      fn: function (val, list) {
        if (!(list instanceof Array)) { return false; }
        return list.filter(function (v) {
          return v === val;
        }) < 1;
      },
      msg: function (val) {
        return val + ' is already taken';
      }
    }
  };
};
```

```

    if (!constraints || typeof constraints !== 'object') {
      return true;
    }

    // exercise each constraint
    for (let constraint in constraints) {
      let validator, currentConstraint;

      if (
        constraints.hasOwnProperty(constraint) &&
        validators.hasOwnProperty(constraint.toLowerCase())
      ) {
        validator = validators[constraint.toLowerCase()];
        currentConstraint = constraints[constraint];

        if (!validator.fn(val, currentConstraint)) {
          errors.push({
            constraint: constraint, // the failed constraint
            msg: validator.msg(val, currentConstraint)
          });
        }
      }
    }
    return errors.length > 0 ? {errors: errors} : true;
  } // end validate function

  // The Mixin
  export var formMixins = {
    getInputEle: function (ref) {
      if (!this.isMounted()) { return; }
      return this.refs[ref] ?
        ReactDOM.findDOMNode(this.refs[ref]).querySelector('input') :
        ReactDOM.findDOMNode(this).querySelector('[name='+ref+']
input');
    },
    validateField: function (fieldName, constraintOverride) {
      let fieldVal = this.getInputEle(fieldName).value
      ,    currentConstraint
      ,    errors
      ;

      if (fieldName in this.constraints) {
        currentConstraint = constraintOverride || this.
constraints[fieldName];

```

```
        errors = validate(fieldVal, currentConstraint);
        return !!errors.errors ? errors.errors : false;
    } else {
        return true;
    }
}
};
```

This utility module exports an object called `formMixins` that supplies two functions. The first function, `getInputEle`, is needed for the `BasicInput` wrapper component because it encapsulates the input elements in the app. Passing as a parameter either the `ref` attribute on the `BasicInput` or the field name attribute to the `getInputEle` function returns the input field wrapped by the `BasicInput` identified by the parameter. The `getInputEle` function is used primarily by the `validateField` function within this mixin in order to access the value of the input during validation. Another option would have been to supply an interface inside the `BasicInput` component itself to retrieve the input value.

The second function, `validateField`, is used by the views containing forms (user create view and post edit view) to evaluate a list of constraint objects. Since this is a mixin, the constraint objects will be defined directly on the component member `constraints` within each view component. This way, the mixin can reference the `constraints` member directly from `this`. Each constraint object contains sets of identifiers (constraint names) and values. One constraint object should be defined per field within the component that hosts the form fields to be validated. The identifiers are the name of the constraint and correlate with a member in the `validators` variable inside the `validate` function at the top of this mixin module. The value in the constraint is typically a boundary. For instance, the `minLength` identifier could have a value of 3 if we didn't want a form field to have fewer than three characters.

As stated before, the `validateField` mixin function obtains the `constraints` member from `this`, which is the mixing component instance. However, there are cases during validation where the caller may want to temporarily change those constraints. Our specific case involves checking for duplicate user names during creation of a user. Since the user store could change during runtime, we need a way to transfer the list of users to our validation engine at the moment the exclusive validator runs. This is the purpose of the `constraintOverride` parameter. You'll see it used later in the user create view.

The `validateField` function exercises the constraints against the respective form field value. Each `validator` defined in the mixin consists of a function that returns a Boolean and another function that returns an error message. After each constraint is checked, an array of error results is returned. Error results in the array contain the identifier (name) of the failed constraint and an error message for the calling component to display on the UI.

User-related stores

There are two stores that deal with users. The user store maintains a collection of users and responds to creation and modification actions. The session context store is used to mock user sessions in a nearly genuine fashion by managing a cookie. We'll start with the session context store.

The session context store will respond to the `login` and `logout` actions and set a login context cookie accordingly. Of course, if we were using real sessions the cookie would be set implicitly by an HTTP header via a server response to log in. Further, the cookies used in a real scenario would be the secure variety and not typically readable by JavaScript in the majority of browsers.

The session context store

This store represents the logged in state of the user. As such, its primary interface is its action handlers for `login` and `logout`. Here's the source for the session context store:

File: `js/stores/sessionContext.js`

```
import Reflux from 'reflux';
import Actions from 'appRoot/actions';
import Request from 'superagent';
import Config from 'appRoot/appConfig';
import Cookie from 'appRoot/vendor/cookie';

export default Reflux.createStore({
  listenables: Actions,
  endpoint: Config.apiRoot + '/users',
  context: { loggedIn: false },
  getInitialState: function () {
    this.context = JSON.parse(Cookie.getItem('session')) ||
  };
  this.context.loggedIn = this.context.loggedIn || false;
  return this.context;
},
getResponseResolver: function (action) {
  return function (err, res) {
    if (res.ok && res.body instanceof Array && res.body.length > 0)
    {
      this.context = res.body[0];
      this.context.loggedIn = true;
      this.context.profileImageData = null;

      this.trigger(this.context);
    }
  };
}
```



```
        action.completed();

        Cookie.setItem('session', JSON.stringify(this.context));
    } else {
        action.failed();
    }
    }.bind(this);
},
getSessionInfo: function () {
    return JSON.parse(Cookie.getItem('session'));
},
onLogin: function (name, pass) {
    Request
        .get(this.endpoint)
        .query({
            'username': name,
            'password': pass
        })
        .end(this.getResponseResolver(Actions.login))
    ;
},
onLogout: function () {
    Cookie.removeItem('session');
    this.context = { loggedIn: false };
    this.trigger(this.context);
    return true;
}
});
```

A convenient way for a component to listen to a group of actions in a Reflux Store is to assign the actions to a member called `listenables`. When using this mechanism, action handler names on the store are inferred by prepending the action name with `on` and camel-casing the result. This is what connects the `onLogin` and `onLogout` members as listeners to their respective actions. At the top of the store there are also assignments for an initial login context, called `context`, and the location of the API, called `endpoint`. The application configuration is imported here to obtain the root of the JSON Server endpoint.

This store responds to two actions, `login` and `logout`. `superagent` is used (imported as `Request`) to call HTTP `GET` on the `/users` endpoint. Additional query parameters for the `username` and `password` are supplied in the invocation of the `query` function. The response is resolved in a separate function supplied by `getResponseResolver` in order to keep the code clean and abstract response resolution for different scenarios.

If the query returned a result length greater than zero, then the local store member `context` is assigned the first returned item (there should only be one result if the username and password are unique). To determine the logged-in state between browser refreshes, a `loggedIn` Boolean is added to the context. Two interfaces are then serviced: the store listenable is triggered with the login context as a parameter, and the `login` action promise is resolved by the response resolver by calling `complete` on the action. Finally, and the part that makes the session stick, the login context is store in a cookie called `session`. Pay special attention to the fact that the profile image data is nulled out before attempting to store the cookie. Cookies have a limitation of 4 kilobytes, and the image data would cause the cookie to not save if the context value exceeded this limit.

If the user refreshes the browser, the store is initialized in `getInitialState` by parsing the cookie back out using the cookie parser library. Before returning the context, the `loggedIn` convenience Boolean is set. Returning a value in the `getInitialState` method of the store will set the initial state of any component using the `Reflux connect` mixin to connect to this store.

Logging out is a simple procedure. The cookie parser has a removal method, `removeItem`. This removal method is invoked to purge the cookie from browser cookie storage. As with the `login` action, the store listeners are triggered with what is now a single value in the context containing a single member: the convenience variable `loggedIn` set to `false`.

The user store

The user store will marshal user profile details to and from the JSON Server back end. Here's the source:

File: `js/stores/users.js`

```
import Reflux from 'reflux';
import Actions from 'appRoot/actions';
import Request from 'superagent';
import Config from 'appRoot/appConfig';

import SessionContext from 'appRoot/stores/sessionContext';

export default Reflux.createStore({
  listenables: Actions,
  users: [],
  endpoint: Config.apiRoot + '/users',
  init: function () {
    Request
      .get(this.endpoint)
```

```
.end(function (err, res) {
  if (res.ok) {
    this.users = res.body;
    this.trigger(this.users);
  } else {
  }
}.bind(this));
},
// called when mixin is used to init the component state
getInitialState: function () {
  return this.users;
},
modifyUser: function (method, details, action) {
  Request
    [method](this.endpoint)
    .send(details)
    .end(function (err, res) {
      if (res.ok) {
        Actions.login(res.body.username, res.password)
          .then(function () {
            action.completed(res.body);
          });
      } else {
        action.failed(err);
      }
    }).bind(this));
  ;
},
onCreateUser: function (details) {
  this.modifyUser('post', details, Actions.createUser);
},
onEditUser: function (details) {
  this.modifyUser('put', details, Actions.editUser);
}
});
```

The user store interacts with JSON Server to persist the user details gathered in the user creation view. This store is initialized using the same API endpoint used in the session store. Like the session store, this store is assigning handlers to actions through `listenables`. Two actions, `createUser` and `editUser`, have handlers here, but fully implementing `editUser` is left as a reader exercise. This would be achieved by adding some code to the user create view. If you would like to implement the user edit feature, doing so would involve reconstituting user data into the create/edit form and calling the edit action handled here.

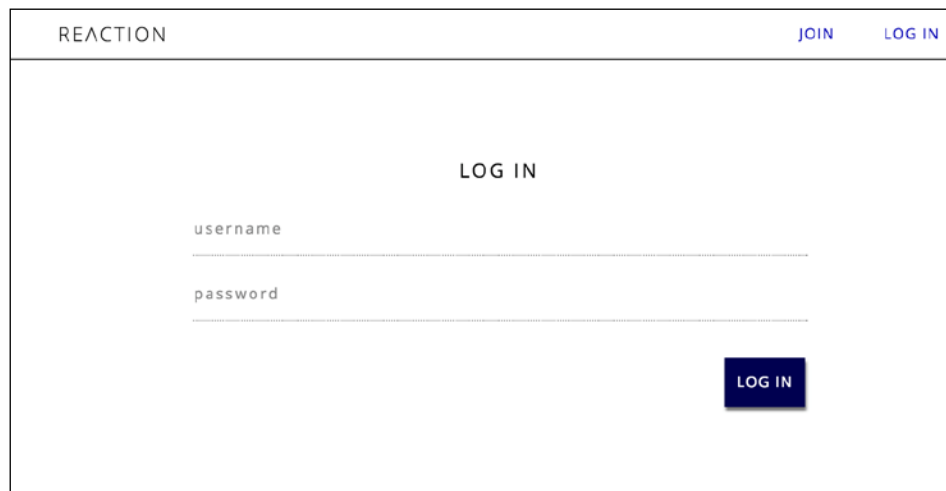
The `onEditUser` action handler was left here to show that the code for `create` and `edit` can be reused and differs only by HTTP method. For both the `createUser` and `editUser` actions, the HTTP method, details of the data to persist, and the `async` action that should be resolved are all supplied to the `modifyUser` function of this store. `superagent` is used once again, but in this case the `send` function is used for HTTP methods containing a request body, such as `PUT` and `POST`. In contrast, the previous session store code used `query` since it used the `GET` HTTP method. If the data persists correctly, then we've successfully created a user. The `login` action is then invoked to automatically log the new user in before resolving the `createUser` action by calling its `complete` method.

User views

Now that the session management and user store plumbing are in place, we turn our attention to user-related views and components.

The log in view

The log in view is our simplest form. Here's what it looks like in action:

The screenshot shows a web browser window with a header bar. The header bar has the word "REACTION" on the left and two links, "JOIN" and "LOG IN", on the right. The main content area of the browser displays a login form. At the top of the form is the text "LOG IN". Below this are two input fields: the first is labeled "username" and the second is labeled "password". At the bottom right of the form is a blue button with the text "LOG IN" in white.

The log in view

Here's the source for the log in view:

File: js/views/login.jsx

```
import React      from 'react';
import { History } from 'react-router';
import BasicInput  from 'appRoot/components/basicInput';
import Actions     from 'appRoot/actions';

export default React.createClass({
  mixins: [ History ],
  getInitialState: function () { return {}; },
  logIn: function (e) {
    var detail = {};

    Array.prototype.forEach.call(
      e.target.querySelectorAll('input'),
      function (v) {
        detail[v.getAttribute('name')] = v.value;
      });
    e.preventDefault();
    e.stopPropagation();

    Actions.login(detail.username, detail.password)
      .then(function () {
        this.history.pushState('', '/');
      }).bind(this)
      ['catch'](function () {
        this.setState({'loginError': 'bad username or password'});
      }).bind(this)
      ;
  },
  render: function () {
    return (
      <form className="login-form" onSubmit={this.logIn}>
        <fieldset>
          <legend>Log In</legend>
          <BasicInput name="username" type="text"
placeholder="username" />
          <BasicInput name="password" type="password"
placeholder="password" />
          { this.state.loginError && <aside className="error">{this.
state.loginError}</aside> }
          <button type="submit">Log In</button>
        </fieldset>
      </form>
    );
  }
});
```

The log in view is a garden-variety username and password form. `BasicInput` proxies type attributes through to the contained `input` tags. The `onSubmit` prop on the form is assigned the local `logIn` component function, which submits the form.

When the `logIn` function is triggered, the values for `username` and `password` are gathered up and used to invoke the login action. Before that, the submit event is prevented from doing a regular form submission by invoking the event object's `preventDefault` method. This is a common practice when using asynchronous HTTP requests and is used here to ensure the user has had a chance to complete the form before accidentally triggering validation. If able to log in, the user is navigated to the root route, the post list view. If log in fails due to the action being rejected in the user store, it is assumed that the `username` or `password` were bad and a form-level error state, `loginError`, is displayed.

The create user view

The following screenshot depicts the create user or join view. The blog name, username, and password are all required fields. The CSS specific to this view can be found inside the `ch7.zip` code bundle in the file `css/views/users/edit.less`.

The screenshot shows a web form titled "BECOME AN AUTHOR". At the top, there's a navigation bar with "REACTION" on the left and "JOIN" and "LOG IN" on the right. The form itself has several input fields: "blog name", "username", "password", "first name", "last name", and "email". There is also a "profile image" section with a placeholder icon and a "CHOOSE FILE" button. At the bottom right, there is a blue button labeled "I'M READY TO WRITE".

User create (edit) view

The user creation form is the most involved view of the application because of the volume and variety of form fields with inline validation. The listing here is lengthy, but we'll break it down afterwards by visiting the mixins and lifecycle methods, then the profile image feature, and finally the form validation and submission procedure.

Here's the source for the user creation form:

File: js/views/users/edit.jsx

```
import React      from 'react';
import { History } from 'react-router';
import Reflux      from 'reflux';
import update      from 'react-addons-update';
import BasicInput  from 'appRoot/components/basicInput';
import Actions     from 'appRoot/actions';
import UserStore   from 'appRoot/stores/users';
import {formMixins} from 'appRoot/mixins/utility';

export default React.createClass({
  mixins: [
    Reflux.connect(UserStore, 'users'),
    History,
    formMixins
  ],
  getInitialState: function () {
    return { validity: {} };
  },
  componentWillMount: function () {
    this.setPlaceholderImage();
  },
  constraints: {
    'username': {
      required: true,
      minlength: 3
    },
    'password': {
      required: true,
      minlength: 5
    },
    'blogName': {
      required: true,
      minlength: 5
    }
  }
},
```

```

    createUser: function (e) {
        var detail = {}
        ,   validationState = {}
        ,   hasErrors = false
        ;

        e.preventDefault();

        // node list isn't necessarily an array but can be iterable
        Array.prototype.forEach.call(
            this.refs.form.querySelectorAll('input'),
            function (v) {
                let fieldName = v.getAttribute('name')
                ,   errors

                ;

                detail[fieldName] = v.value;

                errors = fieldName === 'username' ?
                    this.validateField(fieldName, update(this.constraints.
username, {
                        exclusive: { $set: this.state.users.map(function (v) {
return v.username; }) }
                    ))) :
                    this.validateField(fieldName);

                !hasErrors && errors.length && v.focus(); // first encountered
error
                hasErrors = hasErrors || errors.length;
                validationState[fieldName] = { $set: errors.length ?
errors[0].msg : null };
                }.bind(this));

                if (this.state.profileImageData) {
                    detail.profileImageData = this.state.profileImageData;
                }

                this.setState(update(this.state, { validity: validationState }));
                if (!hasErrors) {
                    Actions.createUser(detail)
                        .then(function (result) {
                            // go to newly created entry
                            this.history.pushState('', `users/${result.id}`);
                        }).bind(this)
                ;
            }
        }
    }

```

```

chooseFile: function () {
  this.getInputEle('profileImage').click();
},
render: function () {

  // noValidate disables native validation
  // to avoid react collisions with native state
  return (
    <form ref="form"
      className="user-edit"
      name="useredit"
      onSubmit={function (e) { e.preventDefault(); }}
      noValidate>
    <fieldset>
      <legend>become an author</legend>

      <BasicInput
        type="text"
        name="blogName"
        placeholder="blog name"
        error={this.state.validity.blogName}
        autoFocus />
      <hr/>
      <BasicInput
        type="text"
        name="username"
        placeholder="username"
        minLength="3"
        error={this.state.validity.username}
        />
      <BasicInput
        type="password"
        name="password"
        minLength="6"
        placeholder="password"
        error={this.state.validity.password}
        required />
      <br/>

      <div className="profile-image-container">
        <label>profile image</label>
        <img className="profile-img" src={this.state.
profileImageData}/>
        <BasicInput name="profileImage" type="file"

```

```
    ref="profileImage" onChange={this.userImageUpload}
    helptext={this.state.sizeExceeded ? 'less than 1MB' : ''}>
      <button onClick={this.chooseFile}>choose file</button>
    </BasicInput>
  </div>

  <BasicInput type="text" name="firstName" placeholder="first
name" />
  <BasicInput type="text" name="lastName" placeholder="last
name" />
  <BasicInput type="email" name="email" placeholder="email"
/>

  <button type="submit" onClick={this.createUser}>I'm ready to
write</button>
</fieldset>
</form>
);
}
});
```

Mixins and lifecycle methods

The best way to browse the code of a React component is to visit the mixins, then the lifecycle methods, and finally the render function. Let's begin with the mixins and lifecycle methods.

Three mixins are being used here. The first is the `Reflux connect` mixin, which maintains a strong connection between the users store and a local component state member called `users`. The next mixin is the `React Router History` mixin, which will be used to navigate the user to the root route upon successful user creation. The last mixin is our `formMixins`, which supplies the `validateField` method.

There are two React lifecycle methods, `getInitialState` and `componentWillMount`. It's typical (and recommended) to scaffold component state members in `getInitialState` if the state is a complex object, as seen here. This is so that nested checks for object existence aren't needed in the render method. In this `componentWillMount` method, a local state member is set for a placeholder user profile image.

The user profile image

Let's jump into how the image code works. JSON Server stores a JSON document and, as such, it's easier to store the user profile image as a string in that file instead of having to implement binary file storage for our prototype application. When the component is preparing to mount (`componentWillMount`), `setPlaceholderImage` is used to set the Base64 image data to a hard-coded image string for a blank user avatar. Base64 is a wire-friendly text representation of binary data and also happens to be a valid resource identifier for an `img` tag `src` attribute.

Now, look at the `div` element with the class name `profile-image-container`. The first node within that `div` is an `img` tag that contains the Base64 of our image data bound to the `src` attribute. Just after the `img` tag is the proxy input component, `BasicInput`. The file input is validated when the `onChange` handler fires. During validation, if the image max size is exceeded, a `sizeExceeded` state is populated with an error message. This message is bound to the `helpText` prop of the file input, informing the user of a 1MB image size limit. The button inside the `BasicInput` uses a handler that simulates a click on the real file input. This is a common practice for styling file inputs, radio buttons, and other native elements that have shadow DOM elements (implicit elements that aren't easily styled or scripted across browsers). The native file element is moved off the screen with CSS and a surrogate styled set of elements forwards events to it. The synthetic click event triggers the native browser behavior of displaying a file selection dialog.

Finally comes the real trick. The `onChange` handler fires in response to file selection and calls `userImageUpload`, which creates a new `FileReader` instance. `FileReader` is a somewhat new means to read files from JavaScript and now has wide browser support. The `FileReader` instance hands control off to `imageLoadedHandler` once the file is read completely. The `imageLoadedHandler` function checks the boundaries of the image size. If the image fits, the state for the image data is set to a `url` data encoding from the image read from disk. If not, we revert back to the placeholder image source data.

Form validation and submission

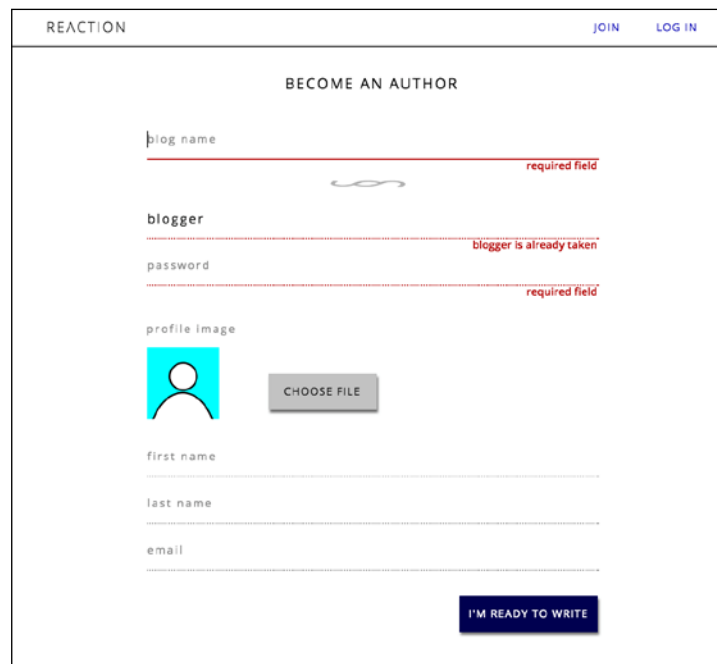
There are several actions within a form that can inadvertently cause submission, such as hitting the *Enter* key on the last field. This can cause a premature submission action that triggers our validation procedure. To prevent this, the `onSubmit` handler placed on the `form` element prevents these automatic sorts of submission by calling `preventDefault` on the event object. The real submit process is triggered by the form submit button, which calls `createUser`. By controlling this submit process completely, the validation routine is able to check the fields predictably and all at once.

In the `createUser` function all of the form inputs are gathered up. For each input, the `validateField` mixin is called. If you remember from the description of this mixin, there is a means to override the value of a validation constraint. This is done for the `username` `exclusive` constraint. Mapping over the user store collection produces a list of the current usernames to use in validation for uniqueness. This is done for every submission because the user store contents could potentially change during runtime.

Each field that could generate a validation error according to the `constraints` member on the component will receive a message from the `validateField` method. The messages are collected and used to populate a validity map state member that maps each field name to a list of its errors, if any. This is accomplished using the `update` function, an immutability helper supplied by React. `Update` is an object extension mechanism supplied by a React add-on. It provides the ability to describe only the changes necessary to generate a new object based on a previous one. It's like a fancy `Object.assign`. You can see how these errors in the state validity map are propagated into the `BasicInputs` using the `error` prop for those tags inside the render function.

If no errors are found, then the `createUser` action is invoked. If the user is created, then the `pushState` method supplied by React Router `History` mixin is used to navigate to the user view for the user that was just created.

The next screenshot shows inline validation in action:



The screenshot shows a web form titled "BECOME AN AUTHOR" within a header labeled "REACTION" with "JOIN" and "LOG IN" links. The form contains several input fields with red error messages:

- blog name**: A red line under the input with the text "required field" to its right.
- blogger**: A red line under the input with the text "blogger is already taken" to its right.
- password**: A red line under the input with the text "required field" to its right.
- profile image**: A placeholder image of a person with a "CHOOSE FILE" button next to it.
- first name**: An empty input field.
- last name**: An empty input field.
- email**: An empty input field.

At the bottom right of the form is a dark blue button labeled "I'M READY TO WRITE".

Validation in action

The user view component

There is a user view and a user view component. The user view is effectively a user profile page. It will eventually have both the user profile data and a collection of that user's posts. The user view component is just the user information part, without the posts. It's a separate component because the user information is used both in the user profile view and within the user list view. To reiterate, the user view component is a representation of the user that is used on the user profile page (the user view) and on the main post list view (our home view) to display a list of all bloggers.

Here's the source for the user view component:

File: `js/components/users/view.jsx`

```
import React      from 'react';
import Reflux      from 'reflux';
import Classnames from 'classnames';
import UserStore   from 'appRoot/stores/users';

export default React.createClass({
  mixins: [
    Reflux.connectFilter(UserStore, 'user', function (users) {
      // This syntax is necessary because babel runtime
      // polyfill statically analyzes code and cannot infer
      // the type of users and, by extension, the correct
      // "find" method
      return Array.find(users, function (user) {
        return user.id === parseInt(this.props.userId, 10);
      }).bind(this));
    })
  ],
  render: function () {
    var user = this.state.user;

    // you must have a root element!
    return user ? (
      <div className={Classnames({
        'user': true,
        'small': this.props.small
      })}>
        <img className={Classnames({
          'profile-img': true,
          'small': this.props.small
        })} src={user.profileImageData} />
        <div className="user-meta">
```

```
        <strong>{user.blogName}</strong>
        <small>
          {user.firstName}&nbsp;{user.lastName}
        </small>
      </div>
    </div>
  ) : <div className="user" />;
}
});
```

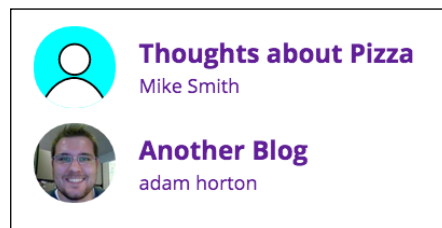
The first item of note here is the `connectFilter` mixin. The user store is a collection, but for this view we just want one user. The `connectFilter` mixin will run any time the user store changes via `trigger`. The mixin will set state using the filter function's return value, which in this case is the user being displayed in this component.

The user view component receives the `userId` as a prop. The `userId` value is used in the connect filter to ensure that this component instance is always connected to the correct user. Since we are using the Babel runtime, there are already many ES6 features included. However, to accomplish the `find` polyfill for arrays, Babel does a static code analysis. Since there's no way to know that the `find` method should be polyfilled from a generic symbol named `find`, the `find` method must be used on the `Array` class itself for Babel to know how to safely do the code replacement.

The render function is pretty basic. Multiple user properties from the user state set by the `connectFilter` mixin are peppered into the UI. An additional prop called `small` can be used to render a smaller profile image and lay out the component differently. This is for the smaller view, and is used later in the list of users on our home page, the post list view.

The user list view

This view simply iterates over user data items in the system and invokes the user view component for each user. The following screenshot shows how that looks:



User list view

Here is the source for the user list view:

File: js/views/users/list.jsx

```
import React      from 'react';
import Reflux     from 'reflux';
import { Link }   from 'react-router';
import UserStore  from 'appRoot/stores/users';
import UserView   from 'appRoot/components/users/view';

export default React.createClass({
  mixins: [
    Reflux.connect(UserStore, 'users')
  ],
  render: function () {
    return (
      <ul className="user-list">
        {this.state.users ?
          this.state.users.map(function (v) {
            return (
              <li key={v.id}>
                <Link to={`/${users}/${v.id}`}>
                  <UserView userId={v.id} small={true} />
                </Link>
              </li>
            );
          }) : []
        }
      </ul>
    );
  }
});
```

This is one of the simplest components. The primary purpose of this component is to later embed it into the post list view. It is bound to the user store via the Reflux connect mixin. The render function iterates over the users and invokes the user view component we just reviewed. Don't forget to key React DOM collections in the renderer. The `small` prop is used to render a more list-friendly size of the user view component. Each user view component is wrapped with a Link component supplied by React Router. This renders an anchor tag that navigates to the user view (profile), which is defined next.

The user view

The user view is the user profile page. Later this will include a list of the user's post. For now, it just invokes the user view component with the `userId` value supplied by the router through the `params` prop.

File: `js/views/users/view.jsx`

```
import React      from 'react';
import UserView   from 'appRoot/components/users/view';

export default React.createClass({
  render: function () {
    return (
      <div className="user-view">
        <UserView userId={this.props.params.userId} />
      </div>
    );
  }
});
```

Other affected views

The only previously defined view that needs modification for a logged-in user is the application header.

The app header

There's now a notion of the user being logged in. Adjustments need to be made to the application header to configure the links for login, logout, sign-up, and creating a new blog post. The application header component source now looks like this:

File: `js/views/appHeader.jsx`

```
import React      from 'react';
import Reflux     from 'reflux';
import { Link, History } from 'react-router';
import Actions    from 'appRoot/actions';
import SessionStore from 'appRoot/stores/sessionContext';

export default React.createClass({
  mixins: [
    Reflux.connect(SessionStore, 'session'),
    History
  ]
});
```

```

    ],
    logOut: function () {
      Actions.logOut();
      this.history.pushState('', '/');
    },
    render: function () {
      return (
        <header className="app-header">
          <Link to="/"><h1>Re&#923;ction</h1></Link>
          <section className="account-ctrl">
            {
              this.state.session.loggedIn ?
                (<Link to="/posts/create">
                  Hello {this.state.session.username}, write something!
                </Link>) :
                <Link to="/users/create">Join</Link>
            }
            {
              this.state.session.loggedIn ?
                <a onClick={this.logOut}>Log Out</a> :
                <Link to="/login">Log In</Link>
            }
          </section>
        </header>
      );
    }
  });

```

What's new here is switching the display based on the `loggedIn` state. The `loggedIn` state is garnered from the session store. The React Router `History.pushState` function is used for log out, and a rather plain `Link` component is used to navigate to the log in view. If the user is logged in, a greeting is displayed that links to the post creation view, which is defined in the next chapter.

Summary

Now we can sign up, log in, and log out. The sign-up form is the most complicated component in the system, because it is a large form with a substantial amount of validation. The next thing to do is the blogging experience... finally!

8

React Blog App Part 3 – Posts

This chapter contains all of the code necessary to create, edit, list, and view blog entries. It also includes integration with the Quill rich text editor. You can follow along with the code, including all of the necessary Less/CSS for this portion of the app, in `ch8.zip`.

The construction of the application is split into the following four parts:

- **Part 1:** Actions and common components
- **Part 2:** User account management
- **Part 3: Blog post operations**
- **Part 4:** Infinite scroll and search

Code manifest

The following is a manifest for all of the code in this chapter. You can follow along with code listings in `ch8.zip`.

There's just one store for posts.

- **Posts store:** `js/stores/posts.js`

Post-related views covered, and their respective styles, include:

- **Post create/edit view:** `js/views/posts/edit.jsx`, `css/views/posts/edit.less`
- **Post view:** `js/views/posts/view.jsx`, `css/views/posts/view.less`

- **Post list component:** `js/components/posts/list.jsx`, `css/components/posts/list.less`
- **Post list view:** `js/views/posts/list.jsx`, `css/views/posts/list.less`

Affected views:

- **User view:** `js/views/users/view.jsx` (add user posts)

The posts store

At this point, the posts store allows fetching a single post or all of the posts in the system. This will be revised to fetch batches of posts in *Chapter 9, React Blog App Part 4 – Infinite Scroll and Search*, for the infinite scroll loading feature.

Here's the posts store source:

File: `js/stores/posts.js`

```
import Reflux from 'reflux';
import Actions from 'appRoot/actions';
import Request from 'superagent';
import Config from 'appRoot/appConfig';

export default Reflux.createStore({
  listenables: Actions,
  endpoint: Config.apiRoot + '/posts',
  posts: [],
  // called when mixin is used to init the component state
  getInitialState: function () {
    return this.posts;
  },
  init: function () {
    Request
      .get(this.endpoint)
      .end(function (err, res) {
        if (res.ok) {
          this.posts = res.body;
          this.trigger(this.posts);
        } else {
        }
      }).bind(this));
  },
  //-- ACTION HANDLERS
  onGetPost: function (id) {
```

```

function req () {
  Request
    .get(this.endpoint)
    .query({
      id: id
    })
    .end(function (err, res) {
      if (res.ok) {
        if (res.body.length > 0) {
          Actions.getPost.completed(res.body[0]);
        } else {
          Actions.getPost.failed('Post (' + id + ') not found');
        }
      } else {
        Actions.getPost.failed(err);
      }
    });
}

Config.loadTimeSimMs ? setTimeout(req.bind(this), Config.
loadTimeSimMs) : req();
},
onModifyPost: function (post, id) {
  function req () {
    Request
      [id ? 'put' : 'post'](id ? this.endpoint+'/'+id : this.
endpoint)
      .send(post)
      .end(function (err, res) {
        if (res.ok) {
          Actions.modifyPost.completed(res);
          // if there's already a post in our local store we need to
modify it
          // if not, add this one
          var existingPostIdx = Array.findIndex(this.posts, function
(post) {
            return res.body.id == post.id;
          });

          if (existingPostIdx > -1) {
            this.posts[existingPostIdx] = res.body;
          } else {
            this.posts.push(res.body);
          }
        } else {
          Actions.modifyPost.completed();
        }
      });
    }
  }

```

```
        }
      }.bind(this));
    }
    Config.loadTimeSimMs ? setTimeout(req.bind(this), Config.
loadTimeSimMs) : req();
  }
});
```

The posts store is a simple collection for now, similar to the users store. As with the users store, `endpoint` and `posts` members are defined directly on the store.

When the store is initialized via Reflux, the `init` interface function is called and a request is made for all of the posts. Like the user store, the `getInitialState` method returns the local collection member, which in this case is `posts`.

Like the user store, there is one `modify` method that handles both creation and editing. This time, though, we decide whether this is an edit (PUT) based on a pre-existing id passed as a parameter.

Finally, the action handler, `onGetPost`, handles post fetching by id.

Post views

Now that we can persist posts to our JSON Server, we turn to the views.

Post create/edit

This is the view that is used to create and edit blog posts. The editor markup is omitted in the listing in the text, as it is very long and isn't needed to explain the component. You can see the complete source, including the Quill markup, in `ch8.zip`. As with the user creation screen in the previous chapter, we'll start with the mixins and React lifecycle methods, then go through the form submission sequence.

Here is what the post create/edit view will look like when we are done:

Post creation view with the Quill-rich text editor

Here is the post create/edit view source:

File: `js/views/posts/edit.jsx`

```
import React      from 'react';
import { History } from 'react-router';
import update     from 'react-addons-update';
import Reflux     from 'reflux';
import Quill      from 'quill';
import Moment     from 'moment';
import Config     from 'appRoot/appConfig';
import Actions    from 'appRoot/actions';
import BasicInput from 'appRoot/components/basicInput';
import Loader     from 'appRoot/components/loader';
import Session    from 'appRoot/stores/sessionContext';
import {formMixins} from 'appRoot/mixins/utility';

export default React.createClass({
  mixins: [
    Reflux.connect(Session, 'session'),
    History,
  ],
```



```
    formMixins
  ],
  getInitialState: function () {
    return { loading: true, validity: {}, post: {} };
  },
  constraints: {
    title: {
      required: true,
      minlength: 5
    }
  },
  componentWillMount: function () {
    this.editMode = this.props.params.hasOwnProperty('postId');
    this.createMode = !this.editMode;
    this.postId = this.editMode ? this.props.params.postId : null;

    this.setState({ loading: this.editMode ? true : false });

    if (this.editMode) {
      Actions.getPost(this.postId)
        .then(function (post) {
          setTimeout(function () {
            //console.log("POST", post);
            this.setState({ post: post, loading: false });
            this.initQuill(post.body);
          }.bind(this), 2000);
        }.bind(this))
        ['catch'](function (err) {
          this.setState({ error: err, loading: false });
        }.bind(this));
    }
  },
  componentDidMount: function () {
    var newPostTpl = '<div>Hello World!</div><div><b>This</b> is my  
story...</div><div><br/></div>';
    !this.editMode && this.initQuill(newPostTpl);
  },
  initQuill: function (html) {
    if (!this.quill) {
      this.quill = new Quill(this.refs.editor, {
        theme: 'snow',
        modules: {
          'link-tooltip': true,
          'image-tooltip': true,

```

```

        'toolbar': {
            container: this.refs.toolbar
        }
    }
    });
}
this.quill.setHTML(html);
},
submit: function (e) {
    var postBody = this.quill.getHTML().replace(/data-
reactid="[^"]+"/g, '')
    ,   fullText = this.quill.getText()
    ,   summary   = fullText.slice(0, Config.postSummaryLength)
    ,   errors    = this.validateField('title');
    ;

    e.preventDefault();
    if(errors.length > 0) {
        this.setState(update(this.state, { validity: { title: { $set:
errors[0].msg } } }));
        this.getInputEle('title').focus();
    } else {
        Actions.modifyPost({
            title: this.getInputEle('title').value,
            body: postBody,
            user: this.state.session.id,
            date: Moment().valueOf(), // unix UTC milliseconds
            summary: summary
        }, this.postId)
        .then(function (result) {
            // go to newly created entry
            this.history.pushState('', `posts/${result.body.id}`);
        }).bind(this)
    ;
    }
},
titleChange: function (e) {
    this.setState(update(this.state, {
        post: {
            title: { $set: e.target.value }
        }
    }));
},

```

```
// form parts of component is always the same so render won't diff
render: function () {
  return (
    <form
      className="post-edit"
      onSubmit={this.submit}
    >
      { this.state.loading ? <Loader /> : [] }
      <fieldset
        style={{ display: this.state.loading || this.state.error ?
'none' : 'block'}}
      >
        <BasicInput
          type="text"
          ref="title"
          name="title"
          value={this.state.post.title}
          error={this.state.validity.title}
          onChange={this.titleChange}
          placeholder="post title"
        />
        <hr/>
        <br/>
        <div className="rich-editor">
          { /* The quill markup goes here. It is quite long as it
includes all of the menus and options for the editor. Take a look at
the code in ch8.zip for this portion. */ }
        </div>
        <button type="submit">{this.editMode ? 'Edit Post' : 'Create
Post'}</button>
      </fieldset>
    </form>
  );
}
});
```

Mixins and lifecycle methods

As was the case with the user edit component in the last chapter, the best way to read a React component is to look at mixins, various lifecycle methods, and, finally, the render lifecycle method. First, though, let's get the Quill editor markup out of the way. The largest portion of this code is contained within the `<div>` tag with the class "rich-editor". It is omitted from this code listing because of its length. It is listed in full in the `ch8.zip` code bundle. This code inside the `<div>` tag with the class "rich-editor" is the markup required by Quill. It's dangerous to render a portion of DOM that is managed by another library. Execution can get tangled between React and the library because React uses the DOM diffing and point modification technique, while the alien component erroneously assumes that it's in full control of the DOM. In this case, it's somewhat safe to include it, though, as long as we invoke the `Quill` constructor on the target DOM at the right stage in the component lifecycle. For us, the right stage is the `componentDidMount` method, or any time after the first render. Also, don't put any bindings or interpolation, or use any other React DOM manipulation mechanisms in the Quill markup so that React will leave this portion of the DOM untouched after the initial render.

In the mixins, a state `session` member is tied to the session store in order to correlate this post submission with the logged-in user. As in the user create view, the `Router History` mixin is included to redirect the user after successful post submission. The `formMixins` utility mixin is included to perform form validation.

The bootstrapping lifecycle methods are more complex here because of the Quill editor. First, the `getInitialState` method stubs out pieces of the state that will be used in render. Having the state object fully structured before render helps avoid having to use superfluous object existence checks in the render method. The `getInitialState` method also sets a loading variable if this is a post edit scenario, during which we need to fetch the blog post information from the server before allowing edits.

As stated before, we should wait until the first time the DOM is rendered to initialize the Quill editor. The DOM is ready when the `componentDidMount` method is invoked, but we need to delay the initialization of the editor if we are editing an existing post and have to fetch the data. So, if the view wasn't passed a post id via props, then it's assumed that we are creating a new post. For a new post, the Quill editor is initialized with some "hello world" placeholder content in `componentDidMount`. If a post id was passed, and edit mode is assumed, then the post is fetched using the `getPost` action serviced by the post store. When the post returns, its data is set into the component state and the Quill editor is initialized with the editable content from the server.

Form submission

Form submission is attached to the `onSubmit` prop in this component, which invokes the local `submit` method. The only constraints for this component are the `required` and `minlength` constraints for the blog title. The constraints are defined in the `constraints` member of the component. The interesting part of the `submit` method is how the content from the rich editor is handled. The Quill editor can supply both plain text and HTML markup. The plain text is retrieved and truncated to a summary value based on a length in the `appConfig.js` file. This summary text is what is used in the post list component as a teaser for the contents of each blog entry. The date and time are formatted as UTC milliseconds for persistence. This is to ensure that they are normalized against a single time zone and easy to compare for sorting using JSON Server's sort capability. Always store datetime values with UTC offsets to ensure accurate comparability.

The HTML markup for the post entry is also obtained from the Quill editor. A regular expression is used to trim off some of the decorative attributes React has put onto the Quill markup. This decoration could have been avoided via the `dangerouslySetInnerHTML` attribute in React, but then we would have had to put the entire Quill markup into a separate text template instead of inline in the component. The `dangerouslySetInnerHTML` mechanism is another way to avoid having React accidentally manage portions of DOM and is often used when jQuery plugins are included in a React app.

In `submit`, if all goes well with the blog title validation, the `modifyPost` action is invoked to save the blog entry. Another helper method supplied by `formMixins`, called `getInputEle`, is used to dig out the blog title input HTML element from its `BasicInput` component wrapper. Once the post submission returns successfully, the `History` mixin method, `pushState`, is used to forward the user to view the newly created blog post.

The post view

The post view is both a primary view and a standalone component. Because it's both, it could have resided in the components folder, but we've opted for the view folder in order to more easily create a mental map of our top-level views. It has two view modes that are switched via a "mode" prop. The summary view mode is used when this component is rendered inside the post list component. The full view is for when this component is used as a primary view to display an entire blog post. Both modes, full and summary, contain the blogger profile image, the title of the blog post, the blogger name, and the date and time of the post. This information is styled slightly differently for each mode. The primary difference between the modes is that the summary mode shows the shortened teaser plain text that we saved in the post creation process as well as an edit button (if the blogger matches the logged-in user). The full view shows the entire post rendered as HTML.

Here is sneak peek at the "summary" mode for the post view that appears in the post list component:



Saturday in the Park

adam horton

08/20/2015 21:37:47

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer quis nibh pellentesque, dapibus dolor vel, convallis tortor. Nulla tincidunt consectetur auctor. Ut lobortis neque vitae laoreet maximus. Curabitur non ex eleifend, accumsan leo cursus, malesuada lectus. Sed congue dignissim nibh eu luctus. Duis varius leo id ligula sollicitudin, non tincidunt quam posuere. In pharetra vitae arcu eget eleifend. Etiam egestas diam id risus sollicitudin, eu maximus elit semper. Sed eu placerat magna, quis ullamc

[read more](#)

EDIT POST

The summary mode

Now, here is the post view in "full" mode with the complete, formatted blog entry:

REACTION

HELLO ADHORTON, WRITE SOMETHING! LOG OUT



Saturday in the Park

adam horton

08/20/2015 21:37:47

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer quis nibh pellentesque, dapibus dolor vel, convallis tortor. Nulla tincidunt consectetur auctor. **Ut lobortis neque** vitae laoreet maximus. Curabitur non ex eleifend, accumsan leo cursus, malesuada lectus. Sed congue dignissim nibh eu luctus. Duis varius leo id ligula sollicitudin, non tincidunt quam posuere. In pharetra vitae arcu eget eleifend. Etiam egestas diam id risus sollicitudin, eu maximus elit semper. Sed eu placerat magna, quis ullamcorper sem. Pellentesque sagittis rhoncus magna a faucibus. In vel egestas tortor, id cursus risus. Nullam **quis magna eget ligula blandit** vulputate. Donec sed ipsum ligula. Nunc arcu nulla, scelerisque non imperdiet placerat, sodales et sem.

Cras cursus dui mi, id scelerisque purus laoreet quis. Phasellus vitae magna mattis diam sagittis hendrerit. Etiam imperdiet id massa eu commodo. Vestibulum congue sit amet odio vel tincidunt. Curabitur maximus, augue et commodo commodo, libero sem congue urna, eu accumsan lacus nibh sed orci. Pellentesque in nunc dolor. Integer varius nibh sed sodales lacinia. Duis eu accumsan orci, ut lacinia diam. Morbi non nisi accumsan, ultricies lectus in, elementum urna. Aenean pellentesque ut turpis eget varius. Suspendisse commodo elit ut facilis tempor. Ut augue lectus, tempor ac fermentum tincidunt, dignissim sit amet quam. Sed ultrices volutpat congue. Vivamus iaculis enim et turpis facilisis, id varius turpis posuere. Proin tristique sit amet libero quis molestie.

The full view mode

Here's the source for the post view:

File: `js/views/posts/view.jsx`

```
import React      from 'react';
import Reflux      from 'reflux';
import { Link }    from 'react-router';
import ClassNames  from 'classnames';
import Moment      from 'moment';
import Actions     from 'appRoot/actions';
import PostStore   from 'appRoot/stores/posts';
import UserStore   from 'appRoot/stores/users';
import Session     from 'appRoot/stores/sessionContext';
import Loader      from 'appRoot/components/loader';

let dateFormat     = 'MM/DD/YYYY HH:mm:ss';

export default React.createClass({
  mixins: [
    Reflux.connect(Session, 'session'),
    Reflux.connect(UserStore, 'users')
  ],
```

Here, we see two uses of the Reflux "connect" mixin. One is used to tie local state to the current session, and the other is used to connect local state to the users collection. The users collection is used in the component to correlate the post information in the view to the user data associated with the post:

```
  getInitialState: function () {
    return {
      post: this.props.post
    };
  },
  componentWillMount: function () {
    if (this.state.post) {
    } else {
      // get post from query params
      this.getPost();
    }
  },
```

There are two ways that the post data can be set inside the component. The first is by passing the post record directly as a prop. This mechanism is used when this component is included inside the post list component to display summary views of each post. The `getInitialState` lifecycle method handles this case and sets the local post state as soon as the bootstrapping process for the component begins. In the `componentWillMount` method, the state is checked to determine if the post information is already there. If it isn't present at this stage, the second way to get post data is used.

The second way to source the post data uses the `postId` from the router params supplied by React Router to fetch the post from the server. The post fetch is done in the `getPost` method. A local loading state is set, but we are careful to check the mounted state of the component. In this particular sequence, `componentWillMount` has triggered the `getPost` method. React will complain if `setState` is called during the bootstrap sequence, but we can simply assign the member directly during this phase.

```

    getUserFromPost: function (post) {
      return Array.find(this.state.users, function (user) {
        return user.id === post.user;
      });
    },
    getPost: function () {
      if (this.isMounted()) {
        this.setState({loading: true});
      } else {
        this.state.loading = true;
      }

      Actions.getPost(this.props.params.postId)
        .then(function (data) {
          //this.state.posts = this.state.posts.concat(data);
          this.setState({
            loading: false,
            post: data
          });
        }).bind(this);
    },
    render: function () {
      if (this.state.loading) { return <Loader />; }
      var post = this.state.post
      ,   user = this.getUserFromPost(post)
      ,   name = user.firstName && user.lastName ?
        user.firstName + ' ' + user.lastName :
        user.firstName ?
        user.firstName :
        user.username
      ;

```



```
    return this.props.mode === 'summary' ? (
      // SUMMARY / LIST VIEW
      <li className="post-view-summary">
        <aside>
          <img className="profile-img small" src={user.
profileImageData} />
          <div className="post-metadata">
            <strong>{post.title}</strong>
            <span className="user-name">{name}</span>
            <em>{Moment(post.date, 'x').format(dateFormat)}</em>
          </div>
        </aside>
        <summary>{post.summary}</summary>
        &nbsp;
        <Link to={`/${posts}/${post.id}`}>read more</Link>
        {
          user.id === this.state.session.id ? (
            <div>
              <Link to={`/${posts}/${post.id}/edit`}>
                <button>edit post</button>
              </Link>
            </div>
          ) : ''
        }
      </li>
    ) : (
      // FULL POST VIEW
      <div className="post-view-full">

        <div className="post-view-container">
          <h2>
            <img className="profile-img" src={user.profileImageData}
/>

            <div className="post-metadata">
              <strong>{post.title}</strong>
              <span className="user-name">{name}</span>
              <em>{Moment(post.date, 'x').format(dateFormat)}</em>
            </div>
          </h2>
          <section className="post-body" dangerouslySetInnerHTML={{__
html: post.body}}>
          </section>
        </div>
      </div>
    );
  }
});
```

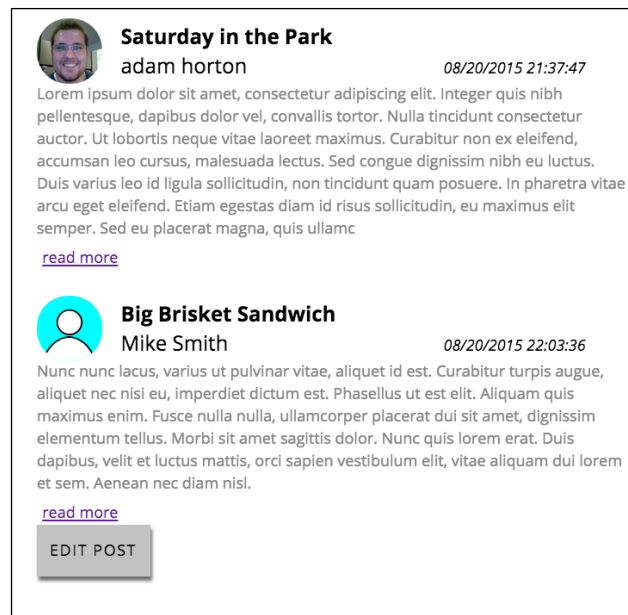
Now turn your attention to the render function. If the loading state is set, we only render the loader component. If the post is already loaded, the user information is retrieved with the `getUserFromPost` method. This method searches the user collection bound to the user store via the `Reflux connect` mixin. Since the first and last name aren't required at signup, some formatting is done on the user name parts with a fallback on the user's login name.

The user information portions of both the summary and full mode are essentially the same. There are two important differences, however. First, in the summary mode, an edit button is displayed if the logged-in user owns the post. The button is wrapped with a `router Link`, which navigates to the create/edit post view. The second key difference is that, while the post summary text is bound directly via an expression in the summary mode, the post body in the full view uses the `dangerouslySetInnerHTML` attribute to inject the richly formatted post DOM into the component. As mentioned before, this mechanism adds objects to the DOM in a way that makes React avoid them during the execution of its DOM diffing algorithm.

The post list component

The post list component is separated from the post list view because it will also be used in the user view to display a list of a specific user's posts.

Here's what the post list component looks like when rendered:



The post list component

Here's the source for the post list component:

File: `js/components/posts/list.jsx`

```
import React      from 'react';
import Reflux     from 'reflux';
import PostStore  from 'appRoot/stores/posts';
import PostView   from 'appRoot/views/posts/view';

export default React.createClass({
  mixins: [
    Reflux.connect(PostStore, 'posts')
  ],
  render: function () {
    var posts = this.props.user ? this.state.posts.filter(function
(post) {
      return post.user == this.props.user;
    }.bind(this)) : this.state.posts;

    var postsUI = posts.map(function (post) {
      return <PostView key={post.id} post={post} mode="summary"/>;
    });

    return (
      <div className="post-list">
        <ul>
          {postsUI}
        </ul>
      </div>
    );
  }
});
```

At this stage in the development of the app, the post list component attaches a local "posts" state member directly to the post store via the Reflux connect mixin. During render, if the user prop is supplied, the posts in this list are filtered by user id. Any time a list of items is rendered in a React component, a unique key must be supplied. This is required for the DOM diffing process. React can intelligently reuse portions of DOM if it knows which portions belong to specific collection members.

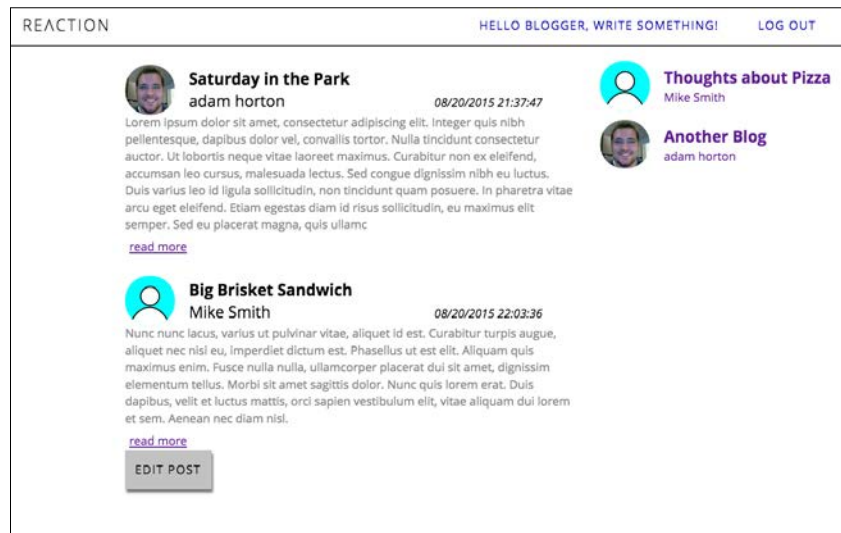


Don't forget that, any time a list of items is rendered in a React component, a unique key must be supplied. React won't let you forget by warning you in the console.

The post list view

The post list view, our home view, is a very simple composition of the post list component without any parameter and the user list component. The post list component handles the aforementioned React collection keying and will eventually also handle infinite loading.

Here's what the post list view will look like in action:



The post list view

Here is the post list view source:

File: `js/views/posts/list.jsx`

```
import React      from 'react';
import UserList   from 'appRoot/views/users/list';
import PostList   from 'appRoot/components/posts/list';

export default React.createClass({
  render: function () {
    return (
      <div className="post-list-view">
        <PostList />
        <div className="users-list">
          <UserList />
        </div>
      </div>
    );
  }
});
```

We know we are building things right. A top-level view that is a simple composition of reusable components is a sign of good abstraction.

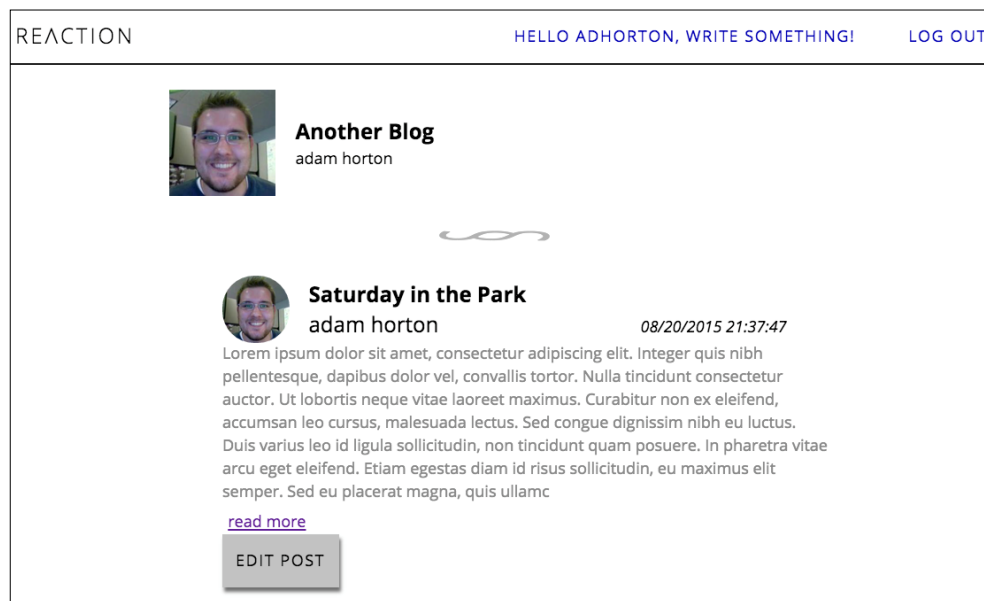
Other affected views

Now that the post list component is prepared, it needs to be added to the user view (profile).

The user view

The post list component is added to the user view to display the user's posts on their profile page. This fleshes out the user view into a more interesting destination. The user view is reached when someone clicks on a user profile in other parts of the app.

Here's what the user view looks like with their posts included:



The user view (profile page)

Here is the user view source with a list of posts added:

File: js/views/users/view.jsx

```
import React      from 'react';
import UserView   from 'appRoot/components/users/view';
import PostList   from 'appRoot/components/posts/list';
```

```
export default React.createClass({
  render: function () {
    return (
      <div className="user-view">
        <UserView userId={this.props.params.userId} />
        <hr />
        <PostList user={this.props.params.userId} />
      </div>
    );
  }
});
```

It's a very simple modification. All that has been added is a separator `hr` tag and the post list component with the user id attribute supplied from the route parameters.

Summary

The application is in a relatively complete state at this point. We have our users and posts and almost all of the desired operations on each. However, querying every post in both the main post list page and in the user profile is terribly inefficient. In the next chapter, we'll add two features to remedy this inefficiency: infinite scroll loading and a search feature.

9

React Blog App Part 4 – Infinite Scroll and Search

In this chapter, two enhancements are made to the application: pagination through infinite scroll loading, and a posts search. All modern blogs and micro blogs, such as Tumblr and Twitter, use an infinite scroll feature in place of explicit pagination to load blog entries in chunks. Since this is now a standard user experience, we'll implement it here. Infinite scroll isn't quite enough, though. Another reasonable user expectation for an app that manages large collections is the presence of a search feature. Luckily, our prototype backend software, JSON Server, has a full-text search capability.

The construction of the application is split into the following four parts:

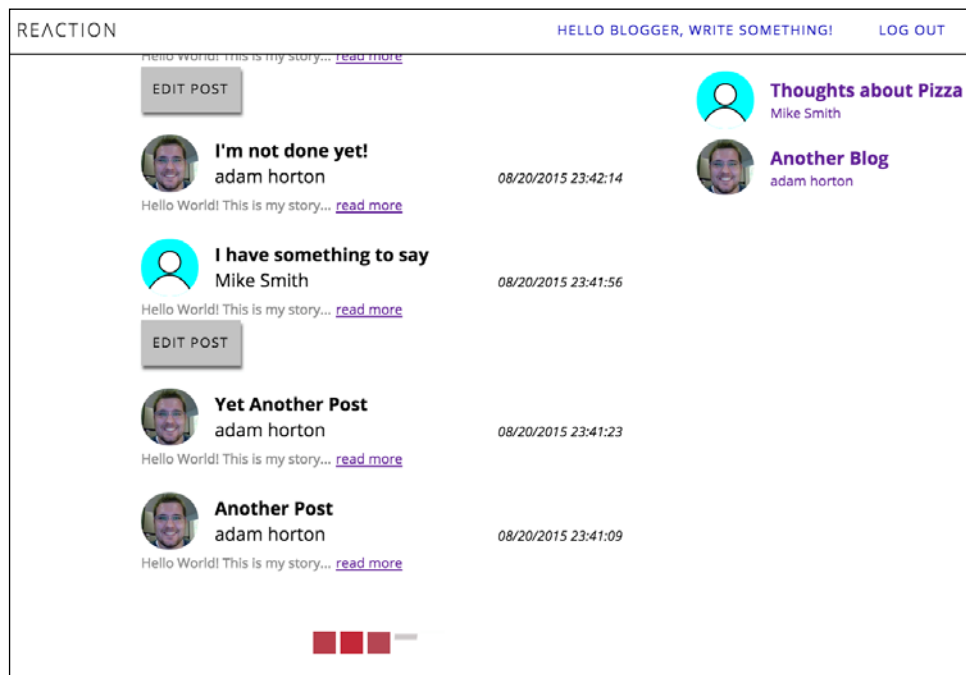
- **Part I:** Actions and common components
- **Part II:** User account management
- **Part III:** Blog post operations
- **Part IV:** Infinite scroll and search

The two features introduced in this chapter are split into two code bundles. The code from the previous several chapters all the way through the infinite scroll feature can be found in `ch9-1.zip`. All of the final code for the blog application, including both the infinite scroll feature and the search feature, is found in `ch9-2.zip`.

Infinite scroll loading

So far, we can create users and create posts. When the number of posts gets lengthy it's useful to load the list in chunks. The visual we'll need, the loader animation component, is already complete. So, the logic parts are all that need to be updated. The user view and the post list view both include the post list component. The shared post list component will handle the pagination. So, each of those views will benefit from this enhancement without modification. We are now going to use the posts store more as a service interface than a store. Ostensibly, its role as a store could retain more of its classic definition if we decided to cache certain posts or pages as a later enhancement. All of the code for this feature section can be found in `ch9-1.zip`. Changes and additions in the source are highlighted here in the text.

The following screenshot shows what infinite scroll loading will look like when we're finished:



Infinite scroll in action

Infinite scroll code manifest

The following is a manifest of all of the files involved in the infinite scroll feature.

The posts store will need to be modified to request posts in page chunks.

- **Posts store:** `js/stores/posts.js`

The post list component will drive the pagination process by requesting chunks.

- **Post list component:** `js/components/posts/list.jsx`

Modifying the posts store

To achieve pagination, we've simply surfaced a page request method, `getPostsByPage`, in the posts store. The pagination support from the server is achieved through JSON Server's ability to slice collections with `_start` and `_end` query parameters. It also supports sorting, which is used here to sort by date via the `_sort` and `_order` query parameters. It's left up to the caller to supply the page number, but page size is stored in the application configuration.

Since we are no longer retaining a structure for the posts, the `init` method, `getInitialState`, the `posts` member, as well as the insertion or replacement within the posts structure on `modify`, are all gone.

Here's the new source for the posts store. Changes are substantial and reside almost entirely in the new `getPostsByPage` function, so just the function signature is highlighted rather than the entire function.

File: `js/stores/posts.js`

```
import Reflux from 'reflux';
import Actions from 'appRoot/actions';
import Request from 'superagent';
import Config from 'appRoot/appConfig';

export default Reflux.createStore({
  listenables: Actions,
  endpoint: Config.apiRoot + '/posts',
  // posts, init, and getInitialState are removed. getPostsByPage
  handles list requests
  getPostsByPage: function (page = 1, params) {
    var start = Config.pageSize * (page-1)
    , end = start + Config.pageSize
    , query = {
      // newest to oldest
```

```
        '_sort': 'date',
        '_order': 'DESC',
        '_start': Config.pageSize * (page-1),
        '_end': Config.pageSize * (page-1) + Config.pageSize
    }
    , us = this
    ;

    if (typeof params === 'object') {
        // ES6 extend object
        Object.assign(query, params);
    }

    if (this.currentRequest) {
        this.currentRequest.abort();
        this.currentRequest = null;
    }

    return new Promise(function (resolve, reject) {
        us.currentRequest = Request.get(us.endpoint);
        us.currentRequest
            .query(query)
            .end(function (err, res) {
                var results = res.body;
                function complete () {
                    // unfortunately if multiple request had been made
                    // They would all get resolved on the first
                    // invocation of this function
                    // Undesireable, when we are rapid firing searches
                    // Actions.getPostsByPage.completed({ start: query._start,
end: query._end, results: results });
                    resolve({
                        start: query._start,
                        end: query._end,
                        results: results
                    });
                }
                if (res.ok) {
                    Config.loadTimeSimMs ? setTimeout(complete, Config.
loadTimeSimMs) : complete();
                } else {
                    reject(Error(err));
                    // same outcome as above
                    // Actions.getPostsByPage.failed(err);
                }
            });
    });
}
```

```

        }
        this.currentRequest = null;
    }.bind(us));
    });
},
//-- ACTION HANDLERS
onGetPost: function (id) {
    function req () {
        Request
        .get(this.endpoint)
        .query({
            id: id
        })
        .end(function (err, res) {
            // Here we no longer insert into the local posts member
            if (res.ok) {
                if (res.body.length > 0) {
                    Actions.getPost.completed(res.body[0]);
                } else {
                    Actions.getPost.failed('Post (' + id + ') not found');
                }
            } else {
                Actions.getPost.failed(err);
            }
        });
    }
    Config.loadTimeSimMs ? setTimeout(req.bind(this), Config.
loadTimeSimMs) : req();
},
onModifyPost: function (post, id) {
    function req () {
        Request
        [id ? 'put' : 'post'](id ? this.endpoint+'/'+id : this.
endpoint)
        .send(post)
        .end(function (err, res) {
            if (res.ok) {
                Actions.modifyPost.completed(res);
            } else {
                Actions.modifyPost.completed();
            }
        });
    }
    Config.loadTimeSimMs ?

```

```
      setTimeout(req.bind(this), Config.loadTimeSimMs) : req();  
    }  
  });  
};
```

We used to fetch all of the posts in the `init` method and then use the `connect` mixin in components to wire component state directly to the store. The difference here is the use of the `getPostsByPage` method. The `modify post` and `get post` action handlers are the same as before. Note that the signature for this method has a parameter default of 1 for page number. This syntax is another ES6 treat.

To achieve the pagination, a query is formed to perform an HTTP `GET` against the JSON Server endpoint. `_sort`, `_order`, `_start`, and `_end` are parameters supplied by JSON Server to manage collection pagination. Notice that there's an additional parameter to the method that is just called `params`. This is for any case where callers want to augment the AJAX call with any additional query parameters. Folding in any additional query parameters is achieved by extending the object before transport using the ES6 `Object.assign` method.

Another notable aspect of the `getPostsByPage` method is that it uses its own ES6 promise interface instead of relying on the `async` mechanism supplied by `Reflux` actions. In fact, it's just a method on the store and not really an `async` action handler. This is because of a wrinkle in the way `async` action handlers in `Reflux` operate. In `Reflux`, when an action is resolved using the `completed` method, all of the listeners on that action are fired at once. This means that all `then` handlers will be called simultaneously for each resolution despite invocations originating from different components with different parameters. For now this isn't an issue because, in the post list component, we'll avoid firing multiple requests at once. However, soon we'll add a search feature that needs to be able to deal with rapid requests since requests will occur in quick succession as the user types. To make resolution more explicit and manage each request promise individually for the search scenario, the `async` action mechanism has been bypassed.

Along with the post result set, the promise is resolved with the start and end pointers used in the query so that the results can always be accurately spliced into the consuming component's local copy.

Modifying the post list component

The list component needs to track its current page as well as make a new page request when the user scrolls. Almost all of this is new code. Every method in the following source except for `render` is new, so we've just highlighted the new method names.

File: js/components/posts/list.jsx

```
import React      from 'react';
import ReactDOM   from 'react-dom';
import Config     from 'appRoot/appConfig';
import PostStore  from 'appRoot/stores/posts';
import PostView   from 'appRoot/views/posts/view';
import Loader     from 'appRoot/components/loader';

export default React.createClass({
  getInitialState: function () {
    return {
      page: 1,
      posts: []
    };
  },
  componentWillMount: function () {
    this.getNextPage();
  },
  componentDidMount: function () {
    var ele = ReactDOM.findDOMNode(this).parentNode
    , style
    ;
    while (ele) {
      style = window.getComputedStyle(ele);

      if (style.overflow.length ||
          style.overflowY.length ||
          /body/i.test(ele.nodeName)
        ) {
        this.scrollParent = ele;
        break;
      } else {
        ele = ele.parentNode;
      }
    }
    this.scrollParent.addEventListener('scroll', this.onScroll);
  },
  componentWillUnmount: function () {
    this.scrollParent
      .removeEventListener('scroll', this.onScroll);
  },
  onScroll: function (e) {
    var scrollEle = this.scrollParent
```

```
,    scrollDiff = Math.abs(scrollEle.scrollHeight - (scrollEle.
scrollTop + scrollEle.clientHeight))
;

    if (!this.state.loading &&
        !this.state.hitmax &&
        scrollDiff < 100
    ) {
        this.getNextPage();
    }
},
getNextPage: function () {
    this.setState({
        loading: true
    });

    PostStore.getPostsByPage(
        this.state.page,
        this.props
    ).then(function (results) {
        var data = results.results;

        // Make sure we put the data in the correct
        // location in the array.
        // If many results are resolved at once
        // trust the request data for start and end
        // instead of some internal state
        Array.prototype.splice.apply(this.state.posts, [results.start,
results.end].concat(data));

        // user may navigate away -
        // changing state would cause a warning
        // So, check if we're mounted when this promise resolves
        this.isMounted() && this.setState({
            loading: false,
            hitmax: data.length === 0 || data.length < Config.pageSize,
            page: this.state.page+1
        });
    }).bind(this), function (err) {});
},
render: function () {
    var postsUI = this.state.posts.map(function (post) {
        return <PostView key={post.id} post={post} mode="summary"/>;
    });

    return (
        <div className="post-list">
            <ul>
```

```

        {postsUI}
      </ul>
      {this.state.hitmax && !this.state.loading ?
        (
          <div className="total-posts-msg">
            showing { this.state.posts.length } posts
          </div>
        ) : ''
      }
      {this.state.loading ? <Loader inline={true} /> : ''}
    </div>
  );
}
});

```

The component itself is going to maintain a local list of posts and a current page number. These are defaulted during the bootstrap process in the `getInitialState` method. When the component is about to mount, the `componentWillMount` method fetches the first page. The `getNextPage` method manages the loading state and fetches from the store via the `getPostsByPage` method. First, though, let's look at the `componentDidMount` method, which attaches the scroll event handlers.

To attach the scroll behavior, the scroll parent must be found. First, the DOM element for this component is obtained, then the code traverses up the DOM in a while loop. This traversal continues until it hits the first element that has an `overflow` CSS property set to `auto`, indicating a scrollable container. This is a tricky way to find the scroll parent in both the post list view and the user view by simply styling the container that we want to scroll as we usually would. Once the scroll parent is located, the scroll handler is attached. In the scroll handler, `onScroll`, some boundaries are checked on every scroll event to determine if the scroll parent is within 100 pixels from the bottom of its scroll height. If it is near the bottom, we aren't currently loading a page, and we haven't already hit the max number of available posts, then the `getNextPage` method is called.

Turning our attention to the `getNextPage` method, you can see that the `getPostsByPage` method is called on the store with the component props used as the additional query parameters. The props are passed through so that the user ID prop, `user`, in the user view flows through to the HTTP request. This prop goes all the way to the request to JSON Server to filter by user. When the promise resolves, the results are returned and spliced into the local collection. Before `setState` is called, a check is made to be sure the component is mounted in case the user has navigated away while the request was in flight. React will generate a warning if a `setState` is invoked on the context of a destroyed component. The `hitmax` member is a Boolean used to determine if there aren't any more posts to fetch. It is actively calculated each time a post payload is returned.

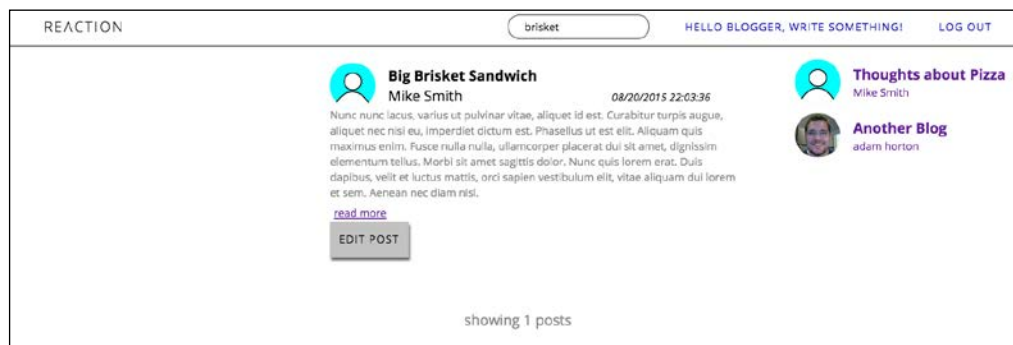
Finally, the loader is added to the `render` method. Also, if the maximum number of posts has been reached, a message is displayed reporting the number of posts that have loaded. You may remember this message from our wireframes way back in *Chapter 5, Starting a React Application*.

Searching posts

Remember that little search box we slipped into the header in our wireframes? We also made the post list a separate component since it appears both on our home view as well as in the user view.

Now, we'll wire that search field to the post list component. The search box is always there, while the post list comes and goes. This is an opportunity to use a store for a purely front-end concern. The search information in the header bar is just a piece of data that various ephemeral components in the application will potentially need. It's like any other application model managed by a store, except that it's not backed by a server request.

The final application code, including the search feature, can be found in the `ch9-2` zip code bundle. The listings are long, but the additions are minimal. Since they are more surgical changes, the differences between the last listing are highlighted here as they were in the **Infinite Scroll Loading** section.



The search feature in action

Search feature code manifest

Here are all the files involved in the search feature.

The search store is a new store. It is a front-end only store to dispatch the value of the search query to subscribers.

- **Search store:** `js/stores/search.js`

The posts store is the only store affected by the introduction of the search feature.

- **Posts store:** `js/stores/posts.js`

Views affected by the introduction of the search feature include:

- **Application header:** `js/views/appHeader.jsx`, `css/views/appHeader.less` (add search input)
- **Post list component:** `js/components/posts/list.jsx`

The search store

The search store is the simplest store you'll ever encounter. Here's the quite terse source code:

File: `js/stores/search.js`

```
import Reflux from 'reflux';
import Actions from 'appRoot/actions';

export default Reflux.createStore({
  listenables: Actions,
  // called when mixin is used to init the component state
  getInitialState: function () {
    return this.query;
  },
  onSearch: function (search) {
    this.query = search;
    this.trigger(search);
  }
});
```

The search store listens to the search action, sets a local query member, and then emits the query text to listeners using `trigger`. It's always a good idea to implement the `getInitialState` method so the Reflux connect family of mixins can set the initial state during component Bootstrap.

Modifying the posts store

The only change here is to remedy a defect with the way that JSON Server handles queries. The `q` parameter does a full text search on resources, but seems to override the other user filter parameter. So, we've added an extra filter to the store to handle this scenario in which both are needed. That small change is highlighted in the following code:

File: js/stores/posts.js

```
import Reflux from 'reflux';
import Actions from 'appRoot/actions';
import Request from 'superagent';
import Config from 'appRoot/appConfig';

export default Reflux.createStore({
  listenables: Actions,
  endpoint: Config.apiRoot + '/posts',
  getPostsByPage: function (page = 1, params) {
    var start = Config.pageSize * (page-1)
    , end = start + Config.pageSize
    , query = {
      // newest to oldest
      '_sort': 'date',
      '_order': 'DESC',
      '_start': Config.pageSize * (page-1),
      '_end': Config.pageSize * (page-1) + Config.pageSize
    }
    , us = this
    ;

    if (typeof params === 'object') {
      // ES6 extend object
      Object.assign(query, params);
    }

    if (this.currentRequest) {
      this.currentRequest.abort();
      this.currentRequest = null;
    }

    return new Promise(function (resolve, reject) {
      us.currentRequest = Request.get(us.endpoint);
      us.currentRequest
        .query(query)
        .end(function (err, res) {
          var results = res.body;
          function complete () {
            // unfortunately if multiple request had been made
            // They would all get resolved on the first
            // invocation of this function
            // This is undesirable, especially
```

```

        // when we are rapid firing searches
        // Actions.getPostsByPage.completed({ start: query._start,
end: query._end, results: results });
        resolve({
            start: query._start,
            end: query._end,
            results: results
        });
    }
    if (res.ok) {
        // if using q param (search),
        // filter by other params,
        // cause JSON server doesn't
        // This is a problem with json-server
        // realistically we'd fix this on our real server
        if (params.q) {
            results = results.filter(function (post) {
                return params.user ?
                    post.user == params.user : true;
            });
        }
        Config.loadTimeSimMs ?
            setTimeout(complete, Config.loadTimeSimMs) :
            complete();
    } else {
        reject(Error(err));
        // same outcome as above
        // Actions.getPostsByPage.failed(err);
    }
    this.currentRequest = null;
    }.bind(us));
    });
},
//-- ACTION HANDLERS
onGetPost: function (id) {
    function req () {
        Request
            .get(this.endpoint)
            .query({
                id: id
            })
            .end(function (err, res) {
                if (res.ok) {
                    if (res.body.length > 0) {

```

```
        Actions.getPost.completed(res.body[0]);
      } else {
        Actions.getPost.failed('Post ('+id+') not found');
      }
    } else {
      Actions.getPost.failed(err);
    }
  });
}
Config.loadTimeSimMs ?
  setTimeout(req.bind(this), Config.loadTimeSimMs) :
  req();
},
onModifyPost: function (post, id) {
  function req () {
    Request
      [id ? 'put' : 'post'](id ? this.endpoint+'/'+id : this.
endpoint)
      .send(post)
      .end(function (err, res) {
        if (res.ok) {
          Actions.modifyPost.completed(res);
        } else {
          Actions.modifyPost.completed();
        }
      });
  }
  Config.loadTimeSimMs ?
    setTimeout(req.bind(this), Config.loadTimeSimMs) :
    req();
}
});
```

Modifying the application header

In the application header, a search input is added to the render method. The search handler takes the text value from the box and invokes the `search` action handled by the search store, which then emits the search query to listeners of the store. Here's the source to the application header including the new search code:

File: js/views/appHeader.jsx

```
import React          from 'react';
import Reflux         from 'reflux';
import { Link, History } from 'react-router';
import Actions        from 'appRoot/actions';
import SessionStore   from 'appRoot/stores/sessionContext';

export default React.createClass({
  mixins: [
    Reflux.connect(SessionStore, 'session'),
    History
  ],
  logOut: function () {
    Actions.logOut();
    this.history.pushState('', '/');
  },
  search: function () {
    var searchVal = this.refs.search.value;
    Actions.search(searchVal);
  },
  render: function () {
    return (
      <header className="app-header">
        <Link to="/"><h1>Re&#923;ction</h1></Link>
        <section className="account-ctrl">
          <input
            ref="search"
            type="search"
            placeholder="search"
            defaultValue={this.state.initialQuery}
            onChange={this.search} />
          {
            this.state.session.loggedIn ?
              (<Link to="/posts/create">
                Hello {this.state.session.username}, write
something!
              </Link>) :
              <Link to="/users/create">Join</Link>
          }
          {
            this.state.session.loggedIn ?
              <a onClick={this.logOut}>Log Out</a> :
              <Link to="/login">Log In</Link>
          }
        </section>
      </header>
    );
  }
});
```

```
        }
      </section>
    </header>
  );
}
});
```

Modifying the post list component

The post list component will listen to the search store and reset its scroll pagination to load a fresh, matched, set of posts. Here are the changes:

File: js/components/posts/list.jsx

```
import React      from 'react';
import ReactDOM   from 'react-dom';
import Config     from 'appRoot/appConfig';
import PostStore  from 'appRoot/stores/posts';
import SearchStore from 'appRoot/stores/search';
import PostView   from 'appRoot/views/posts/view';
import Loader     from 'appRoot/components/loader';

export default React.createClass({
  getInitialState: function () {
    return {
      page: 1,
      posts: []
    };
  },
  componentWillMount: function () {
    this.searchUnsubscribe = SearchStore.listen(this.onSearch);
    this.getNextPage();
  },
  componentDidMount: function () {
    var ele = ReactDOM.findDOMNode(this).parentNode
    , style
    ;
    while (ele) {
      style = window.getComputedStyle(ele);

      if (style.overflow.length ||
          style.overflowY.length ||
          /body/i.test(ele.nodeName)
        ) {

```

```

        this.scrollParent = ele;
        break;
      } else {
        ele = ele.parentNode;
      }
    }
    this.scrollParent.addEventListener('scroll', this.onScroll);
  },
  componentWillUnmount: function () {
    this.searchUnsubscribe();
    this.scrollParent.removeEventListener('scroll', this.onScroll);
  },
  onSearch: function (search) {
    this.setState({
      page: 1,
      posts: [],
      search: search
    });
    this.getNextPage();
  },
  onScroll: function (e) {
    var scrollEle = this.scrollParent
    , scrollDiff = Math.abs(scrollEle.scrollHeight - (scrollEle.
scrollTop + scrollEle.clientHeight))
    ;

    if (!this.state.loading &&
        !this.state.hitmax &&
        scrollDiff < 100
    ) {
      this.getNextPage();
    }
  },
  getNextPage: function () {
    this.setState({
      loading: true
    });

    PostStore.getPostsByPage(
      this.state.page,
      Object.assign({}, this.state.search ? {q: this.state.search} :
{}, this.props)
    ).then(function (results) {
      var data = results.results;

      // make sure we put the data in the correct

```

```
        // location in the array
        // if many results resolved at once,
        // trust the request data for start and end
        // from the results instead of some internal state
        Array.prototype.splice.apply(this.state.posts, [results.start,
results.end].concat(data));

        // user may navigate away -
        // changing state would cause a warning
        // so, check if we're mounted when this promise resolves
        this.isMounted() && this.setState({
            loading: false,
            hitmax: data.length === 0 || data.length < Config.pageSize,
            page: this.state.page+1
        });
    }.bind(this), function (err) {});
},
render: function () {
    var postsUI = this.state.posts.map(function (post) {
        return <PostView key={post.id} post={post} mode="summary"/>;
    });

    return (
        <div className="post-list">
            <ul>
                {postsUI}
            </ul>
            {this.state.hitmax && !this.state.loading ?
                (
                    <div className="total-posts-msg">
                        showing { this.state.posts.length } posts
                    </div>
                ) : ''
            }
            {this.state.loading ? <Loader inline={true} /> : ''}
        </div>
    );
}
});
```

The post list component now listens to the search store via the `onSearch` method, which is attached to the search store in the `componentWillMount` lifecycle method.

When a search is triggered, `onSearch` resets the page to 1, the first page of the search results. It also clears out the locally held posts, and sets a local state member to the search query contents. Last, `onSearch` invokes the `getNextPage` method to fetch the first page of results.

Only one small modification in `getNextPage` remains to complete the search feature. The extra query parameters for the `getPostsByPage` service call may include the user parameter for the user view. So, `Object.assign`, an ES6 mechanism used for object extension, is used to layer in the `q` parameter needed for the JSON Server full text search.

Our infinite scroll works as usual, but now using this extra search parameter, `q`, on top of the rest of the query details. When the search content changes, the pagination details are reset and infinite scroll loading proceeds as before.

Final thoughts

Making a web application is very involved, but if you start with a plan, the structure of your React components, and the design of the data flow, things become manageable. Even after all of this effort, there are several things to be done if we want the app to be production ready. The following are a few suggestions for interesting enhancements.

Suggested improvements

That was a lot of work, but this application is still pretty basic. Here are some things you could try to make the application even more interesting:

- Implement delete post
- Implement edit user (profile)
- Implement delete user
- Add a comment stream for posts
- Add a post tagging feature and search or filter by tag

Level up the blog app

The following enhancements would make the application production ready:

- Wire up cloud deployment
- Implement real user accounts, or ...
- Introduce a social media login capability
- Add more social media integration - publish links to new entries on Twitter or Facebook

Moving forward

In the next chapter, several methods for implementing animation in React applications are explored.

10

Animation in React

Animation in React is the same as any web animation. Web animation techniques typically involve setting CSS class names on elements, or setting CSS properties directly on elements via the `style` attribute. Animation is achieved when CSS classes or attributes change CSS properties, which are either the target of the transition easing declaration, or directly manipulated frame-by-frame via JavaScript. Animation can also be achieved on SVG elements by changing path properties directly as well as through SVG `animate` elements and related animation properties that are specific to SVG. In this chapter, we won't cover SVG animation, but the animation techniques are very similar to animating other DOM elements.

Common instances of web animation include elements being added or removed from the DOM, or an application workflow state change. Examples of a workflow state change include a menu being opened or closed, or photo gallery navigation. Some animations are subtle and used mostly for a stylish effect, like a color change or shifting shadow on button hover.

In this chapter we'll cover:

- Animating by changing CSS class names as the result of a component state change
- Animating DOM addition and removal using `ReactCSSTransitionGroup`
- More complex animation via `requestAnimationFrame` in conjunction with Cheng Lou's `React-Motion` library

All of the code examples are available as GitHub gists, and can be viewed live on JSFiddle in the same fashion as the first chapters of this book. For each example, a ZIP file of the gist code, which is designed to work with JSFiddle, is also supplied. We'll look at the CSS and JavaScript for each example. First, let's review some animation terms.


Animation terms

In animation, a term that appears frequently is **tweening**. In animation, even in the hand-drawn medium, the most important or expressive frames are called **keyframes**. For smooth animation, all of the keyframes and frames in between the keyframes need to be carefully crafted or automatically calculated. This process is called **tweening**. Another term that crops up in animation is **easing**. Easing refers to mathematical functions of time that determine the state of an item being animated between two points. CSS has some built-in easing functions, which can be invoked as values of the `transition-timing-function` CSS property. Built-in easing function include values such as `linear`, `ease-in`, `ease-out`, and so on. *Linear* is what you would expect: an item tweened linearly moves at a constant speed between two endpoints. Another example: `ease-in` means that the value will start changing speedily and slow as it approaches its destination, similar to someone pressing on the brakes of a car at a stoplight.

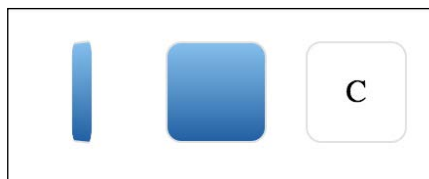
CSS transitions using class switching

Simple animations can be done with CSS class changes (the `className` attribute in React). The `classNames` library used in our prototype blog app, which closely resembles `ng-class` (for those familiar with Angular JS), is a handy way to construct CSS class name sets based on component state.

In this example, component state is used to drive class names on a tag. This is one of the simplest ways to achieve animation effects using React. An interesting UI pattern for hiding configuration controls, or extra information, is to make a card component. When triggered, the card flips over to reveal another view with the extra info.

 The source code of the `cardflipanimation.zip` file can be found at <http://bit.ly/Mastering-React-10-flipanim-gist> and a live example can be found at <http://j.mp/Mastering-React-10-flipanim-fiddle>.

Here's what the card flip looks like in action.



JavaScript code

Here's the code for an individual card component.

```
var Card = React.createClass({
  getInitialState: function () { return {}; },
  flip: function () {
    this.setState({flipped: !this.state.flipped});
  },
  render: function () {
    return (
      <div
        onClick={this.flip}
        className={classNames('card-component', {'flipped': this.state.
flipped})}>
        <div className="front">
          <div className="inner">{this.props.children}</div>
        </div>
        <div className="back">&nbsp;</div>
      </div>
    );
  }
});
```

The Card component is self-contained and tracks its own `flipped` state. The `flip` method toggles the `flipped` state when the card is clicked. The `Classnames` library is used to apply the "flipped" class or remove the class based on the value of the `flipped` state each time the component renders. The identity of the card is conveyed through any children that are nested inside an instance of the component. This will let us put content inside the face of the card as shown in the following Deck code.

```
var Deck = React.createClass({
  cards: ['A', 'B', 'C'],
  render: function () {
    var cards = this.cards.map(function (cardIdentity) {
      return <Card key={cardIdentity}>{cardIdentity}</Card>;
    }).bind(this);
    return <div className="deck-component">{cards}</div>;
  }
});
ReactDOM.render(<Deck />, document.getElementById('app'));
```

The `Deck` component is a simple collection of `Card` instances. Strings for card content are used as children in each respective rendered card. Arbitrary React DOM could also have been used within the card component. When creating a collection of React components, always supply a unique key so that React will intelligently calculate DOM differences and render efficiently. We are reusing the card identity as the key since it's simple, comparable, and unique.

CSS source

```
.deck-component {
  perspective: 1000px;
}
.card-component {
  position: relative;
  display: inline-block;
  cursor: pointer;
  width: 50px;
  height: 50px;
  margin: 10px;
  transition: transform 300ms ease;
  transform-style: preserve-3d;
}
.card-component.flipped {
  transform: rotateY(180deg);
}
.card-component > * {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  display: flex;
  align-items: center;
  justify-content: center;
  border: 1px solid #ddd;
  border-radius: 8px;
  backface-visibility: hidden;
}
.card-component .front {
  background-color: white;
  transform: rotateY(0deg);
  z-index: 1;
}
```

```
.card-component .back {
  background-color: #1e5799;
  background: linear-gradient(to top, #1e5799 0%, #7db9e8 100%); /* W3C
*/
  transform: rotateY(180deg);
}
```

This is as simple as CSS animation gets. The `card-component` class defines a transition against the `transform` property with an animation duration of 300ms and a `transition-timing-function` value of `ease`. These details could be defined as separate `transition-*` CSS declarations but can also be combined into one transition declaration, as they are here. The `preserve-3d` `transform-style` declaration is essential to use on the top-level wrapper for the card component in order to give the appearance of depth when the flip animation occurs.

Within the card component wrapper are front and back elements positioned using the direct child selector `> *`. To make the front and back overlap they are absolutely positioned with the same coordinates and size. It's always important to also include `backface-visibility` set to `hidden` when creating a flip animation so that one side doesn't "shine" through the other when the outer element is being flipped.


To transform the Card into the flipped state, the entire back of the card is first rotated 180 degrees. When the Card click handler, `flip`, is triggered, the "flipped" class is added to the container element. The element, including its front and back children, are flipped via the CSS `transform` property. Since the `transform` property is targeted by the `transition` property, the browser automatically animates the rotation according to the aforementioned transformation details.

Animating DOM enter and exit

Being able to introduce UI elements and dismiss them in a way that's fluid and not abrupt is an important part of a nice user experience. Having elements pop in and out of the DOM can be jarring. To animate components entering and leaving the DOM, React supplies a pair of interfaces: `ReactTransitionGroup` and `ReactCSSTransitionGroup`. These interfaces provide hooks to component mounting and unmounting lifecycle events. For `ReactCSSTransitionGroup`, the hooks are the automatic addition and removal of CSS class names using a documented naming convention. In these examples, we'll use the `ReactCSSTransitionGroup` interface.

Popover menus

The first example of DOM enter and exit animation is a popover menu. You've probably seen these sorts of menu boxes that pop up over everything when invoked and disappear when dismissed by clicking outside the box or selecting a menu item. The full code listing is in the following information box. Going forward we'll intersperse the code with commentary on how it works. This is because the listings get a bit longer as the complexity of the code increases.

 The source code of the ZIP file named `popoveranimation.zip`, can be found at <http://bit.ly/Mastering-React-10-popoveranim-gist> and a live example can be found at <http://j.mp/Mastering-React-10-popoveranim-fiddle>.

Here's what the menu popover DOM enter and exit animation looks like in action.



JavaScript source

```
var ReactCSSTransitionGroup = React.addons.CSSTransitionGroup;
var Popover = React.createClass({
  render: function () {
    return (
      <div className="popover-component">
        {this.props.children}
      </div>
    );
  }
});
```

The popover component renders its contents into a `<div>` tag with the class name `popover-component`. We'll use that `<div>` tag to style the look and positioning of the popover container.

```
var App = React.createClass({
  getInitialState: function() { return {}; },
  toggleMenu: function (id) {
    this.setState({
      'activeMenu': this.state.activeMenu === id ? null : id
    });
  }
});
```

```

    });
  },

```

The App component serves to provide a small amount of layout to contain the two popovers. It will also drive the animation. The `toggleMenu` function will be the target of menu click events and track a unique ID of the currently active menu. This way we can hide any open menu when another is opened.

```

render: function () {
  return (
    <div className="application">
      <header>
        <h1>My Rad App</h1>
        <nav>
          <ul>
            <li onClick={this.toggleMenu.bind(this, 1)}>
              <label>Menu 1</label>
              <ReactCSSTransitionGroup
                transitionName="popoveranim"
                transitionEnterTimeout={350}
                transitionLeaveTimeout={350}>
                {this.state.activeMenu === 1 ?
                  <Popover key={1}>
                    <strong>Menu 1 Content</strong><br/>
                    <a href="http://www.google.com">Goto Google</a>
                  </Popover>
                  : []
                }
              </ReactCSSTransitionGroup>
            </li>
            <li onClick={this.toggleMenu.bind(this, 2)}>
              <label>Menu 2</label>
              <ReactCSSTransitionGroup
                transitionName="popoveranim"
                transitionEnterTimeout={350}
                transitionLeaveTimeout={350}>
                {this.state.activeMenu === 2 ?
                  <Popover key={2}>
                    <strong>Menu 2 Content</strong>
                    <nav>
                      <ul>
                        <li>Menu 2 item</li>
                        <li>another menu 2 item</li>
                      </ul>
                    </nav>
                  </Popover>
                  : []
                }
              </ReactCSSTransitionGroup>
            </li>
          </ul>
        </nav>
      </header>
    </div>
  );
}

```

```
        </Popover>
      : []
    }
  </ReactCSSTransitionGroup>
</li>
```

By wrapping the two popovers in the `ReactCSSTransitionGroup` with a `transitionName` attribute, it is ensured that a sequence of class names will be applied to a popover element in succession. They are applied when React determines that the child popover component should be added or purged from the DOM. This determination is driven by the `activeMenu` component state. The sequence of classes is used to style transition animations. In this case, the sequence of class names will be `popoveranim-enter`, `popoveranim-enter-active`, `popover-leave`, and `popover-leave-active`. The `transitionEnterTimeout` and `transitionLeaveTimeout` attributes ensure that the desired states will be reached even if there's a failure with the animation due to a mistake in the CSS.

```
    </ul>
  </nav>
</header>
<main>
  Lorem Ipsum
</main>
</div>
);
}
});
ReactDOM.render(<App />, document.getElementById('app'));
```

CSS source

```
html *, body * {
  margin: 0;
  padding: 0;
  font-family: Verdana, arial;
}
header {
  background-color: #dcdcdc;
  box-shadow: 0 1px 4px #666;
  height: 40px;
  line-height: 40px;
}
header h1 {
  margin: 0;
```

```

padding: 0 0 0 50px;
font-size: 16px;
display: inline;
}
header > nav {
display: inline-block;
float: right;
margin: 0 80px 0 0;
}
header > nav ul {
list-style-type: none;
padding: 0;
}
header > nav li {
display: inline-block;
position: relative;
}
header > nav li > label {
display: block;
color: #44e;
cursor: pointer;
padding: 0 10px;
/* prevent text selection */
-webkit-touch-callout: none;
-webkit-user-select: none;
-moz-user-select: none;
-ms-user-select: none;
user-select: none;
}
header > nav li > label:hover {
background-color: #eee;
}
main {
padding: 50px;
}

```

All of the preceding styles serve as layout to create a header bar. The header bar contains a small menu, which is the stage for our popover elements and their animations. The `li` elements in the `<nav>` tag have a `relative` position so that we can adjust the location of the popovers to be next to the menu items that invoke them. Let's dive into the code for the popovers and their animations.

```

.popover-component {
position: absolute;
top: 40px;

```

```
    left: 50%;
    margin-left: -80px;
    font-size: 12px;
    width: 160px;
    background-color: white;
    border-radius: 8px;
    border: 1px solid gray;
    box-shadow: 1px 2px 4px gray;
    line-height: 12px;
    padding: 10px;
  }
```

This code above has positioning for our popover, as well as some framing for the visual box of the popover. Next is a small CSS pseudo element that creates a small caret pointing up at the menu item associated with the popover.

```
.popover-component:before, .popover-component::before {
  content: '';
  width: 15px;
  height: 15px;
  position: absolute;
  transform: rotateZ(-45deg);
  top: -9px;
  left: 50%;
  margin-left: -15px;

  background-color: white;
  border-style: solid;
  border-width: 1px 1px 0 0;
  border-color: gray;
}
```

The `before` pseudo element above is a triangle using the same border and background as the popover container. You can see this little triangle pointing up in the screenshot of the popover animation shown in the introduction of this section. The triangle is actually a rotated square with borders on only two sides to make a little triangle.

```
.popover-component ul {
  list-style: square inside;
  padding: 5px 10px;
}
.popover-component li {
  display: list-item;
  white-space: nowrap;
}
```

```
.popover-component strong {  
  white-space: nowrap;  
}
```

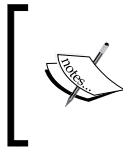
These few styles are just a bit of code to adjust the visuals of the list items inside our popover menus.

```
/* ReactCSSTransitionGroup Animation styles*/  
.popoveranim-enter {  
  opacity: 0.01;  
  transform: translateY(10px);  
}  
  
.popoveranim-enter.popoveranim-enter-active {  
  opacity: 1;  
  transform: translateY(0px);  
  transition: opacity .3s ease, transform .3s ease;  
}  
  
.popoveranim-leave {  
  opacity: 1;  
}  
  
.popoveranim-leave.popoveranim-leave-active {  
  opacity: 0.01;  
  transition: opacity .3s ease;  
}
```

Now that we are finally at the animation code, it's quite short! Both the `opacity` and `transform` properties are animated to give fade in and rise effects. The `ReactCSSTransitionGroup` mechanism that applies these classes first applies the appropriate enter or leave style based on if the component is entering or leaving the DOM. It then layers on the associated active style.

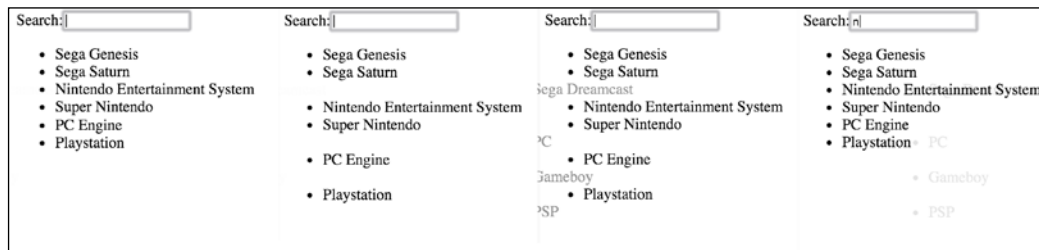
List filtering

This example uses a combination of animations facilitated by `ReactCSSTransitionGroup`. Here, DOM enter/exit is transitioned as well as the height CSS property on the list container. The example also uses measurement of incoming list items to give an extra effect which makes room when items are added back to the display.



The source code of the file named `listfilteranimation.zip`, can be found at <http://bit.ly/Mastering-React-10-listanim-gist> and a live example can be found at <http://j.mp/Mastering-React-10-listanim-fiddle>.

Here's what the list filter animation looks like in action. Items fly out when removed from the DOM. When items are added to the DOM, the height is tweened to make room for the new items flying back in.



JavaScript source

```
var ReactCSSTransitionGroup = React.addons.CSSTransitionGroup;
var gameSystems = [
  'Sega Genesis',
  'Sega Saturn',
  'Sega Dreamcast',
  'Nintendo Entertainment System',
  'Super Nintendo',
  'PC',
  'PC Engine',
  'Gameboy',
  'Playstation',
  'PSP'
];
```

This `gameSystems` member is our test list.

```
var SuperFlyList = React.createClass({
  propTypes: {
    filter: React.PropTypes.string
  },
  getInitialState: function () { return {}; },
  componentDidMount: function () {
    this.setState({
      eleHeight: ReactDOM.findDOMNode(this).querySelector('li').
```

```

    lientHeight
    });
  },

```

We are going to tween the height when new items are added. To do this we need to know how tall each list item will end up being. We wait for `componentDidMount` so items are in the DOM, and then measure the height. The height value is stored as a state member. We could have also stored it directly on the component by setting `this.eleHeight`, instead of the state member.

```

render: function () {
  var itemsToDisplay = this.props.list
    .map(function (item, idx) {
      return { name: item, key: idx };
    })
    .filter(function (item) {
      return this.props.filter ? item.name.toLowerCase()
        .indexOf(this.props.filter) !== -1 : true;
    }.bind(this))
    .map(function (item, idx) {
      return (
        <li
          key={item.key}
          style={{top: this.state.eleHeight*idx+'px'}}>
          {item.name}
        </li>
      );
    }.bind(this));

```

In the first part of the render function, we gather up the items and filter them based on the `filter` prop. This is how the outer component can push a filter string into the list component. First, we map them to create an artificial key. For the key, we used the original array index from the unfiltered list. After mapping our example list to get a key and filtering the list, we map the resulting items to React `li` components. Don't forget to put the unique key on each element. The original array index is used as the key, because the filtered set indices will change as items are purged. This is also the reason the original filter converts the simple strings in the array to objects containing the artificial key before the subsequent filtering, so that we can carry the values and the manufactured keys through the filter.

```

    var totalHeight = itemsToDisplay.length * this.state.eleHeight;

```

The total height for the list is calculated based on the original measurement in `componentDidMount` multiplied by the number of items that pass the filter during this render cycle. We need to adjust the height because any new arrivals to the DOM will be relatively positioned within this container.


```
    return (
      <ReactCSSTransitionGroup
        className="super-fly-list-component"
        style={{height: totalHeight + 'px'}}
        component="ul"
        transitionName="superfly"
        transitionEnterTimeout={300}
        transitionLeaveTimeout={300}>
        <li className="measure" key="measure">&nbsp;</li>
        {itemsToDisplay}
      </ReactCSSTransitionGroup>
    );
  }
});
```

Just like the popover example, `ReactCSSTransitionGroup` is used to strategically apply enter and leave classes with names based on the `transitionName` property of that JSX tag. Note that we always render the measurement `li` element into the DOM. This is needed to measure our `li` height reliably in `componentDidMount`, especially when we are transitioning from zero matches to one or more. There are a couple of extra tricks here. The first trick is that this is the first time we are seeing a state calculation being used directly to change a CSS property on an element, the `ReactCSSTransitionGroup` tag itself. The second trick is that we are using the `ReactCSSTransitionGroup` not only to drive the animation of its children, but rendering it as the appropriate `ul` tag by using the `component` attribute.

```
var App = React.createClass({
  getInitialState: function () { return {}; },
  search: function () {
    this.setState({query: ReactDOM.findDOMNode(this.refs.search).
value});
  },
  render: function () {
    return (
      <main>
        <label>
          Search:
          <input ref="search" type="text" onChange={this.search} />
        </label>
        <SuperFlyList list={gameSystems} filter={this.state.query} />
      </main>
    );
  }
});
```

The `App` component is a wrapper with a search input. It drives the search query into the list component. A change handler sets the `query` state, which is bound to a prop of `SuperFlyList`.

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

This final line renders the `App` component into the DOM.

CSS source

```
.super-fly-list-component .measure {
  position: absolute;
  left: -9999px;
}
```

This preceding rule pushes the measurement `li` off the screen.

```
.super-fly-list-component {
  position: relative;
}
.super-fly-list-component li {
  position: absolute;
  transition: opacity .3s ease, transform .3s ease, top .1s ease;
}
```

Each list item is absolutely positioned so that we can animate its `top` position. This is done automatically by the browser's calculation for `top` because `top` isn't explicitly declared. When the new list is rendered, all of the others will naturally recalculate their `top` position to make room for the new items. The transition will animate this aspect so that the items appear to make room for the new ones coming in. The opacity and transform properties are also animated to perform the fade and fly in and out effect.

```
/* ReactCSSTransitionGroup Animation styles*/
.superfly-enter {
  opacity: 0.01;
  transform: translateX(-100px);
}
.superfly-enter.superfly-enter-active {
  opacity: 1;
  transform: translateX(0px);
  transition-delay: 0.25s; /* wait a bit for space to be made */
}
.superfly-leave {
  opacity: 1;
  transform: translateX(0px);
}
```

```
.superfly-leave.superfly-leave-active {  
  opacity: 0.01;  
  transform: translateX(100px);  
}
```

The animation code is similar to the last example. On enter, we translate in from the left and fade in. There's a small delay placed on enter to emphasize the height transition for some time while room is being made for items. On leave, a list item is translated out to the right and faded out.

Using the React-Motion animation library

React-Motion is a very nice physics based animation library created by Cheng Lou, an avid contributor to many things React. In some ways, React-Motion is similar to jQuery animate. Specifically, it's a fancy interface to tween numbers. In previous examples, animation was achieved by strategically changing CSS classes attached to elements. When classes change CSS properties, which are the target of a transition property, the browser automatically uses the declared transition property details to handle animation frame-by-frame. Another, more direct, way to animate is to change CSS declarations on the style attribute of the virtual DOM elements themselves.

How React-Motion works

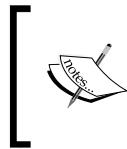
Animation on the web is a process of interpolating intermediate values of properties, such as position, over the course of some time between starting and ending values. To calculate these in-between values, or to tween them using a library, typically involves specifying the start and end values, the duration over which you want the animation to take place, and the easing function or a name for the way the intermediate values are calculated. React-Motion does this tweening calculation a bit differently than using easing function by offering a very basic and flexible interface. The interface is called a **spring**, and it uses stiffness and damping values to calculate the intermediate steps. It also integrates nicely into the React rendering process and is JSX friendly.

There's one quick item to note if you read the GitHub documentation for React-Motion. The documentation says that the Motion component parameters `defaultStyle` and `style` should have the same shape. This is referring to data structures. React-Motion springs are very flexible, allowing these values to be objects, arrays, or objects containing arrays or other objects. Just be sure that both parameters, `defaultStyle` and `style`, are the same shape or structure so that the library knows how to correlate the numbers being tweened or interpolated. This allows React-Motion to tween many properties at once.

For this example we are using version 0.3.1 of React-Motion.

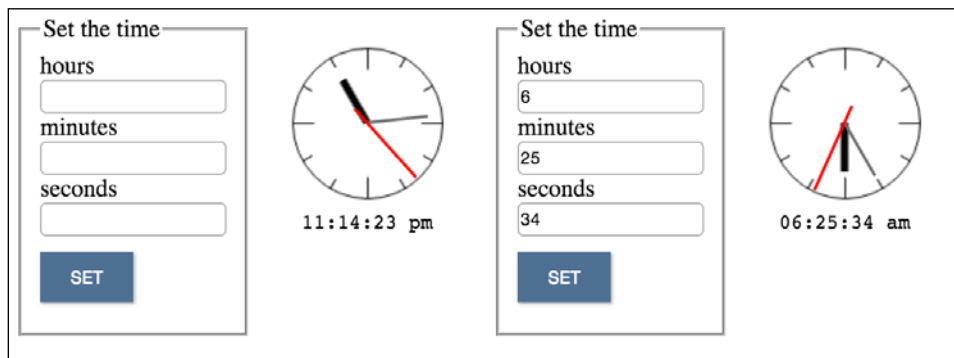
Clock animation

This animation example uses React-Motion to animate the hands of a clock. The clock is a set of overlapping canvas elements. The clock is continuously animating and the time can be set using input fields for hours, minutes, and seconds.



The source code of the file `clockanimation.zip` can be found at <http://bit.ly/Mastering-React-10-clockanim-gist> and a live example can be found at <http://j.mp/Mastering-React-10-clockanim-fiddle>.

Here are some static shots of the clock animation demo in action.



JavaScript source

```
var Motion = ReactMotion.Motion
,   spring = ReactMotion.spring
;
```

Motion is the simplest React-Motion animation component and the one that we'll use here. There's also a `spring` function, which handles easing functions via a simple interface using physics semantics of a spring: stiffness and damping.

```
var Clock = React.createClass({
  getInitialState: function () {
    return { baseDate: new Date(), hours: 0, mins: 0, secs: 0 };
  },

```

As the time counts forward, it does so from a base date. This is initially the time when the component is instantiated but will be replaced if props for hours, minutes, and seconds are passed in. More on this is discussed later when the `componentWillReceiveProps` method is explained. The initial values for hours, minutes, and seconds are also set in that method. These values will be in degrees for the rotation of the clock hands.

```
componentDidMount: function () {
  var node      = ReactDOM.findDOMNode(this)
  ,   get       = node.querySelector.bind(node)
  ,   parts     = node.querySelectorAll('canvas')
  ,   faceCtx   = get('.clockface').getContext('2d')
  ,   hourCtx   = get('.hourhand').getContext('2d')
  ,   minCtx    = get('.minutehand').getContext('2d')
  ,   secondCtx = get('.secondhand').getContext('2d')
  ,   width     = node.clientWidth
  ,   height    = node.clientHeight
  ;
```

Here we are just getting references to our canvas elements and their canvas drawing context in order to set a baseline clock drawing to 12:00:00. We are about to draw into the canvas elements, so this is done in `componentDidMount` after the canvas elements are first placed into the DOM.

```
Array.prototype.forEach.call(parts, function (canvas) {
  canvas.setAttribute('width', width);
  canvas.setAttribute('height', height);
});
```

Set all of the canvas dimensions equal to the container dimensions.

```
// render off pixel boundaries for bolder lines
faceCtx.translate(.5, .5);
```

This is a common trick to make a canvas draw appropriately bold lines. If you draw on the lines of the pixel grid, then the adjacent pixels will get anti-aliased to a lighter color. Think of it like pouring a finite amount of water onto the ridge of an ice cube tray. Each adjacent tray cell would get about half of the water. This context translation aligns strokes to the middle of the cell/pixel.

```
// create the clock face
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].forEach(
function (mult) {
  faceCtx.save();
  faceCtx.translate(width>>1,height>>1);
  faceCtx.rotate((360 / 12 * mult) * (Math.PI / 180));
```

```

        faceCtx.translate(0, -(width>>1));
        faceCtx.beginPath();
        // longer ticks every 3 hours
        faceCtx.moveTo(0, (mult)%3 ? 8 : 15);
        faceCtx.lineTo(0, 1);
        faceCtx.stroke();
        faceCtx.restore();
    });

```

This code draws the tick marks for the hours on the clock face. It does this by drawing one tick, then rotating the context and repeating the draw commands. A longer tick is drawn every three hours as is typical on round clock faces.

```

        faceCtx.beginPath();
        faceCtx.arc(width>>1, height>>1, (width>>1)-1, 0, 2 * Math.PI,
false);
        faceCtx.stroke();

```

This part draws the clock face circle using the `arc` method. You may notice a somewhat cryptic trick here that was also in the previous code chunk. The double greater-than symbol is a bit shift to the right. This is a divide by two, but don't do it unless you know your operand is divisible by two or you'll lose precision.

```

// create the clock hands
hourCtx.translate(width>>1, width>>1); // center
hourCtx.lineWidth = 5;
hourCtx.moveTo(0,0);
hourCtx.lineTo(0, -(width>>1) +18);
hourCtx.stroke();

minCtx.translate(width>>1, width>>1);
minCtx.strokeStyle = "#666";
minCtx.lineWidth = 2;
minCtx.moveTo(0,0);
minCtx.lineTo(0, -(width>>1) + 10);
minCtx.stroke();

secondCtx.translate(width>>1, width>>1);
secondCtx.strokeStyle = "#f00"; // red second hand
secondCtx.lineWidth = 2;
secondCtx.moveTo(0,12);
secondCtx.lineTo(0, -(width>>1) + 2);
secondCtx.stroke();

```

The preceding code draws the hands of our clock into each respective canvas.

```
// start with initial state base date
this.setBaseDate(this.state.baseDate);
// begin aggressively calculating updates
window.requestAnimationFrame(this.tick);
},
```

This kicks off time calculation. The call to `setBaseDate` will initialize our starting point for time measurement. Then `requestAnimationFrame` begins collating time passage.

```
componentWillReceiveProps: function (nextProps) {
  var newBaseDate = new Date();
  // if props aren't parseable set date to current
  if (!isNaN(parseInt(nextProps.hours,10) + parseInt(nextProps.
mins,10) + parseInt(nextProps.secs,10))) {
    newBaseDate.setHours(nextProps.hours%24);
    newBaseDate.setMinutes(nextProps.mins%60);
    newBaseDate.setSeconds(nextProps.secs%60);
    this.setBaseDate(newBaseDate);
  } else {
    this.setBaseDate(new Date());
  }
},
```

The `componentWillReceiveProps` lifecycle event is the interface used to set time from outside the component. There are props for hours (`hours`), minutes (`mins`), and seconds (`secs`). If the props are changed, they are checked for validity and a new base date is established. If they change and aren't valid, then we reset base date to the current time.

```
setBaseDate: function (date) {
  this.setState({ baseDate: date });
  this.startTick = new Date(); // re-establish starting point
},
format: function (num) {
  return num > 9 ? num : '0'+num;
},
```

This `format` function is used during render to display the numeric form of the time. It prepends a zero to the time component if it's a single digit.

```
tick: function () {
  var nextTick = new Date()
  ,   diff = nextTick.valueOf() - this.startTick.valueOf()
  ;

  // Here we use a logical OR to clamp
```

```

    // the values to whole numbers
    // This allows us to render just once per second
    // while aggressively updating our time data
    var clockState = {
      hoursDisp: ((this.state.baseDate.
getHours()+diff/1000/3600)|0),
      minsDisp: ((this.state.baseDate.getMinutes()+diff/1000/60)|0),
      secsDisp: ((this.state.baseDate.getSeconds()+diff/1000)|0),
    };
    clockState.amPm = clockState.hoursDisp%24 > 12 ? 'pm':'am';

    // degrees
    clockState.hours = clockState.hoursDisp*30;
    clockState.mins = clockState.minsDisp*6;
    clockState.secs = clockState.secsDisp*6;

    this.setState(clockState);
    // resume updates at 60fps
    window.requestAnimationFrame(this.tick);
  },

```

The `tick` function is probably the most interesting part of this demo aside from the usage of `React-Motion`. Every time `tick` is called, it calculates how much time has passed since our base date (`startTick`). Invoking `valueOf` on a `Date` object in JavaScript returns the UTC milliseconds, or time since the Unix Epoch (January 1, 1970 00:00:00). If you are curious about why it's that date, search for `unix time` or `epoch time`. There's a storied past which includes technical reasoning for that date and time. For our purposes, it gives a common point from which we can calculate our time difference.

After the time difference from the base date is calculated, some state is prepared and set. There's another trick here. Each of the `clockState` components has `|0` after it. This logical `or` truncates a floating-point number in JavaScript to the whole part of the number (without rounding) very efficiently. This is important for only triggering a render when appropriate, as you will soon see.

Notice that `requestAnimationFrame` is used to calculate the time passage continuously. Browsers attempt to render at 60 frames per second. When they re-render the page they attempt to reconcile DOM changes and navigate the specificity of CSS in order to finally lay out the actual pixels onto the screen. This can be an expensive process and, if you change the DOM with abandon, it can cause a lot of recalculation thrash. Think of `requestAnimationFrame` as a means to say, "hey browser, next time you decide to re-calc and re-render, please run this function first".

This means that our clock calculation will run at roughly 60 frames per second or every 16.667 milliseconds. That's pretty fast, and we don't want the React render pipeline to run that fast. This is what the next lifecycle method is for.

```
// only allow render when there's a value change
shouldComponentUpdate: function (nextProps, nextState) {
  return (
    nextState.hours !== this.state.hours ||
    nextState.mins !== this.state.mins ||
    nextState.secs !== this.state.secs
  );
},
```

The `shouldComponentUpdate` method is how we can aggressively track the time with `requestAnimationFrame`, but only render the component when a full hour, minute, or second changes. This is why earlier the values were truncated with a logical `or`, so that this lifecycle method only returns true every second or so. This means that the clock time will remain accurate. It will not accidentally skip a second because of a browser hiccup, but it will also not render inefficiently. However, we don't want the clock hands to move instantaneously between whole number values. This is where React-Motion finally comes into the picture. It will handle the tweening of the clock hands between the one-second renders of the greater clock component.

```
render: function () {
  return (
    <div className="clock-component">
      <canvas ref="clockface" className="clockface"></canvas>
      <Motion style={{
        hours: spring(this.state.hours),
        mins: spring(this.state.mins),
        secs: spring(this.state.secs)
      }}>
```

The next animation endpoint for our `Motion` component is an object, `style`, and is managed using the `spring` function. The `spring` function calculates the interpolated values using an easing function implied by the stiffness and damping configuration. Here, we let `spring` use the defaults for those values. If we wanted to we could pass a second parameter (array) to each `spring` invocation where the first array value was the stiffness and the second was the damping.

```
{({hours,mins,secs}) =>
  <div className="hands">
    <canvas ref="hourhand" className="hourhand" style={{
      WebkitTransform: `rotate(${hours}deg)`,
      transform: `rotate(${hours}deg)`
```

```

    }}></canvas>
    <canvas ref="minutehand" className="minutehand" style={{
      WebkitTransform: `rotate(${mins}deg)`,
      transform: `rotate(${mins}deg)`
    }}></canvas>
    <canvas ref="secondhand" className="secondhand" style={{
      WebkitTransform: `rotate(${secs}deg)`,
      transform: `rotate(${secs}deg)`
    }}></canvas>
  </div>
}
</Motion>

```

These components within the `Motion` component will be rapidly rendered as a result of the React-Motion spring tweening operation. The current value for each internal frame calculation will be available in the `hours`, `mins`, and `secs` parameters. As we did before with the list animation example height, the interpolated values are placed directly into the `style` attribute of each respective virtual DOM clock hand canvas element.

```

    <pre className="digital">
      {this.format(this.state.hoursDisp%12)}:{this.format(this.
state.minsDisp%60)}:{this.format(this.state.secsDisp%60)} {this.state.
amPm}
    </pre>
  </div>
);
}
});

```

The preceding code is just a small digital display to go below our animated clock. It will only render once a second as allowed by `shouldComponentUpdate`.

```

var ClockExample = React.createClass({
  getInitialState: function () { return {}; },
  getVal: function (name) {
    return ReactDOM.findDOMNode(this.refs[name]).value;
  },
  setTime: function () {
    this.setState({
      hours: this.getVal('hours'),
      mins: this.getVal('mins'),
      secs: this.getVal('secs')
    });
  },
});

```

The wrapper interface component, `ClockExample`, keeps its own state for hours, minutes, and seconds, and manages them with interactive inputs and a set button. The set button calls the `setTime` handler here and sets the state for each time component.

```
render: function () {
  return (
    <div className="clock-example">
      <fieldset>
        <legend>Set the time</legend>
        <label>hours   <input maxLength="2" ref="hours" /></label>
        <label>minutes <input maxLength="2" ref="mins" /></label>
        <label>seconds <input maxLength="2" ref="secs" /></label>
        <button onClick={this.setTime}>SET</button>
      </fieldset>
      <Clock hours={this.state.hours} mins={this.state.mins}
secs={this.state.secs}/>
    </div>
  );
}
```

When `setTime` is called, the values from the inputs are set on the local state of this outer component, triggering a render. When this component renders, the state for hours, minutes, and seconds is passed to our animated clock via props.

```
ReactDOM.render(<ClockExample />, document.getElementById('app'));
```

Don't forget to render the top-level component.

CSS source

```
* {
  box-sizing: border-box;
}

.clock-example {
  display: -webkit-flex; /*safari*/
  display: flex;
  align-items: flex-start;
  width: 300px;
  justify-content: space-between;
}
```

Flexbox is a wonderful layout tool. It is used here to put the clock controls and the clock component side by side.

```
.clock-example fieldset {
  width: 150px;
}
.clock-example input {
  display: block;
  line-height: 18px;
  border: 1px solid #aaa;
  border-radius: 4px;
  width: 100%;
}
.clock-example button {
  border: none;
  color: white;
  background-color: #446688;
  margin: 10px 0;
  padding: 10px 20px;
  cursor: pointer;
  outline: none;
  box-shadow: 1px 1px 2px #aaa;
}
.clock-example button:active {
  transform: translateY(2px);
  transition: transform .1s ease;
  box-shadow: 0 0 2px #aaa;
}
```

The preceding styles are visual treatment for the input form used to set the time into the clock component.

```
.clock-component {
  position: relative;
  width: 100px;
  height: 100px;
  margin: 20px;
}
.clock-component canvas {
  position: absolute;
  left: 0;
  top: 0;
}
.clock-component .digital {
  position: absolute;
  bottom: -35px;
  width: 100%;
  text-align: center;
}:
```

The actual `clock` component styles place all the `canvas` components (clock hands) on top of one another so they overlap correctly. The small digital display is positioned under the animated clock. Notice that, unlike the popover and menu animation examples, there aren't any animation styles for the `clock` component. The React-Motion component and `spring` handles all of that!

Summary

Animation on the web exists in a few fundamental forms that mostly involve changing out CSS classes, directly manipulating the `style` attribute, or a combination of the two. There's more sophistication and flexibility to discover when using the lower level `ReactTransitionGroup` interface, but the `ReactCSSTransitionGroup` mechanism, simple class swapping, or React-Motion will get you just about anywhere you need to be. There are also loads of other great examples on the React-Motion GitHub page (<https://github.com/chenglou/react-motion>), such as animated list reordering and follow the leader style cursor animation. They are definitely worth checking out.

Index

A

application design, email application

- about 90
- API 92
- data entities 92
- main views 93, 94
- routes 93, 94
- site map 93, 94
- wireframes, creating 90, 91

application design, blog application

- data entities 99, 100
- main views 100, 101
- sitemap 100, 101
- wireframes, creating 95

application header 128

Asynchronous Module Definition (AMD) 81-83

B

Babel

- about 85
- URL 8

base styles 121-124

BasicInput component 125, 126

blog application

- application design 95
- development environment, preparing 101
- directory structure 111
- enhancements 193, 194
- improvements 193
- index.html file 112
- js/app.jsx 113, 114

main views 115, 116

mock database 112

starting 111

views, linking with React Router 116

blog application, considerations

- about 109
- browser support 109
- form validation 110
- React render function, defining 109

Bootstrap

URL 28

build system

- about 79-81
- module systems 83
- selecting 81, 82

C

cardflipanimation.zip file

URL 196

clockanimation.zip file

URL 211

CommonJS 81-83

component composition

- about 19
- children, accessing 27-31
- simple components, composing 19-21
- with behavior 21-27

component lifecycle

- about 32
- events, updating 35
- mounting 32
- unmounting 32-34
- working 38-41

controlled components

- best practices 58
- one-way data flow model 54, 55
- used, for creating simple form 55-58
- with read and write input 52-54
- with read-only input 51, 52

cookies

- reading 131
- writing 131

create user view

- about 141, 142
- form submission 147
- form validation 147
- lifecycle methods 146
- mixins 146
- user profile image 147

create, read, update, and delete (C.R.U.D.) 92

CSS preprocessors

- about 85
- LESS 85
- SASS 85

CSS transitions

- CSS source 199
- JavaScript code 197, 198
- with class switching 196

D

Data Access Objects (DAO) 61

dependency injection (DI) 82

development environment, blog application

- dependencies, installing 101-103
- Node, installing 101-103
- preparing 101
- Webpack, installing 103, 104

Domain Specific Language (DSL) 21

DOM enter and exit

- animating 199
- CSS source 203-205, 209, 210
- JavaScript source 200-202, 206-209
- list, filtering 205, 206
- popover menu 200

Dumb components 59

dynamic components

- about 43-45
- UserList component 45, 46

- UserRow component 45, 46

E

easing 196

ECMAScript 6 (ES6) 85

F

Fiddle

- URL 21

Flexbox 218

Flux

- about 87
- actions 87
- dispatcher 87
- stores 87

forms

- about 51
- controlled components 51, 52
- creating, with controlled components 55-58
- refactoring 59, 60
- submission, handling 164

form utilities mixin 132-134

front-end architecture components

- about 79, 80, 86
- eventing 88
- front-end models 87
- front-end router 87
- messaging 88
- utilities, requisites 88
- view controllers 88
- view models 88
- views 88

G

GitHub repository

- URL 62

Graphics Processing Unit (GPU) 109

Grunt 81, 103

Gulp 81, 103

H

Hello React example

- about 1-5
- source code, URL 5

I

immediately invoking function expression (IIFE) 83

infinite scroll

code manifest 177

loading 176

post list component, modifying 180-183

posts store, modifying 177-180

J

JsFiddle

URL 7

JS syntax

compiling 85

JSX

about 6

decompiling 8

render result, structure 9-11

templates, compiling 85

working with 6, 8

K

keyframes 196

L

LASS 85

lifecycle methods 163

listfilteranimation.zip file

URL 206

loader component 125-127

loaders 103

log in view 139-141

M

mixins

about 47-49, 131, 163

cookies, reading 131

cookies, writing 131

form utilities mixin 132-134

implementing 49-51

module systems

about 83

Asynchronous Module Definition

(AMD) 83

CommonJS 83

selecting 84

Mozilla Developer Network (MDN) cookies

URL 131

N

Node

installing 101-103

URL 101

Node Package Manager (NPM) 62

P

polyfills 85, 110

popoveranimation.zip file

URL 200

popover menus

creating 200

post create/edit view

about 158, 159

form submission 164

lifecycle methods 163

mixins 163

post list component

about 169, 170

adding, to user view 172, 173

posts store 156-158

post views

about 158

full view mode 164-169

post create/edit 158, 159

post list component 169, 170

post list view 171, 172

summary view mode 164-169

presets 108

props

about 11, 12

getDefaultProps method, defining 15

propTypes, using 13, 14

working with 12, 13

R

React-Motion animation library

- CSS source 218-220
- JavaScript source 211-218
- used, for animating clock 211
- using 210
- working with 210

React Router

- about 87, 95
- views, linking with 116

react-validation-mixin example

- code, obtaining 62-74
- code, running 63-67

Reflux 87

Reflux actions

- about 120, 121
- reference link 121

reusable components

- about 121
- application header 128
- BasicInput component 125, 126
- loader component 125-127

routes 114

S

SASS 85

Search Engine Optimization (SEO) 93

search feature

- application header, modifying 188
- code manifest 184
- post list component, modifying 190-193
- posts store, modifying 185
- posts, searching 184
- search store, handling 185

semver 102

session context store 135, 136

single-page application (SPA) 78

Slush 82

Smart components 59

SPA design

- application design 79, 80
- build system 79, 80
- front-end architecture components 79, 80

spring 210

state

- about 16
- working with 17, 18

T

tweening 196

two-way binding

- reference link 54

U

user management

- application runtime configuration 131
- code manifest 130
- with dependencies 131
- with mixins 131

user-related stores

- about 135
- session context store 135, 136
- user store 137, 138

user views

- about 139
- app header 152, 153
- create user view 141, 142
- log in view 139-141
- post list component, adding 172, 173
- user list view 150, 151
- user profile page 152
- user view component 149, 150

V

validation

- field-level validation 62
- form-level validation 62
- handling 60
- react-validation-mixin example 62
- types 61, 62

views, blog application

- js/views/appHeader.jsx 116
- js/views/login.jsx 117
- linking, with React Router 116

virtual DOM

- about 17
- URL 17

W

Webpack

- about 82, 95
- configuring 103, 104
- installing 103, 104

Webpack configuration file

- about 104
- entry section 105
- module section 107, 108
- output section 105
- plugins section 105
- resolve section 106

wireframes, blog application

- creating 95
- post-related views 98, 99
- user-related views 96, 97

Y

Yeoman 82, 95



Thank you for buying Mastering React

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

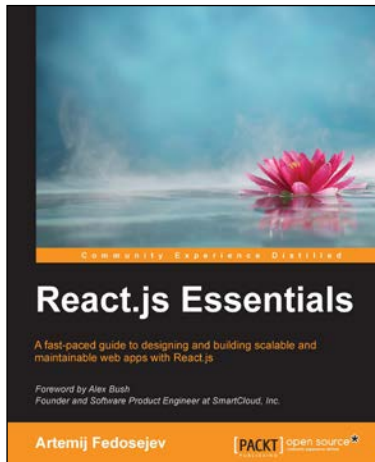
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



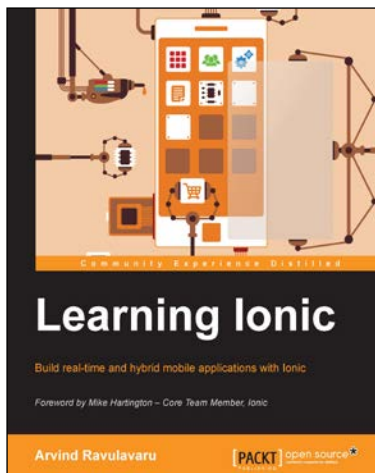
React.js Essentials

ISBN: 978-1-78355-162-0

Paperback: 208 pages

A fast-paced guide to designing and building scalable and maintainable web apps with React.js

1. Build maintainable and performant user interfaces for your web applications using React.js.
2. Create reusable React.js components to save time and effort in maintaining your user interfaces.
3. Learn how to build a ready-to-deploy React.js web application, following our step-by-step tutorial.



Learning Ionic

ISBN: 978-1-78355-260-3

Paperback: 388 pages

Build real-time and hybrid mobile applications with Ionic

1. Create hybrid mobile applications by combining the capabilities of Ionic, Cordova, and AngularJS.
2. Reduce the time to market your application using Ionic, that helps in rapid application development.
3. Detailed code examples and explanations, helping you get up and running with Ionic quickly and easily.

Please check www.PacktPub.com for information on our titles



Ext JS Application Development Blueprints

ISBN: 978-1-78439-530-8

Paperback: 340 pages

Develop robust and maintainable projects that exceed client expectations using Ext JS

1. Learn about the tools and ideas that support the architecture of an Ext JS 5 application.
2. Design and build rich real-world Ext JS 5 applications based on a set of client requirements.
3. Make strong architectural decisions based on project specifications with this practical guide.



Mastering AngularJS UI Development [Video]

ISBN: 978-1-78528-991-0

Duration: 1:27 hours

Master the art of creating amazing, reliable, and dynamic user interfaces for your AngularJS applications with the help of a real-world application

1. Comprehend the process of creating quality AngularJS UI from scratch to completion.
2. Understand key concepts related to building AngularJS UI, such as interacting with APIs, writing reusable components, and persisting user data.
3. Explore AngularJS UI Bootstrap and implement its key features into your applications.

Please check www.PacktPub.com for information on our titles