

Community Experience Distilled

Reactive Programming with JavaScript

Learn the hot new frontend web framework from Facebook – ReactJS, an easy way of developing the V in MVC and a better approach to software engineering in JavaScript

Jonathan Hayward

[PACKT] PUBLISHING open source* community experience distilled

Reactive Programming with JavaScript

Learn the hot new frontend web framework from Facebook – ReactJS, an easy way of developing the V in MVC and a better approach to software engineering in JavaScript

Jonathan Hayward

[PACKT] PUBLISHING
open source
community experience distilled

BIRMINGHAM - MUMBAI

Reactive Programming with JavaScript

Copyright © 2015 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 author, 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: August 2015

Production reference: 1260815

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

ISBN 978-1-78355-855-1

www.packtpub.com

Credits

Author
Jonathan Hayward

Reviewers
Antal Orcsik
Sven A Robbestad
Hibai Unzueta

Commissioning Editor
Kunal Parikh

Acquisition Editor
Manish Nainani

Content Development Editor
Aparna Mitra

Technical Editor
Mohita Vyas

Copy Editors
Vikrant Phadke
Alpha Singh
Ameesha Smith-Green

Project Coordinator
Mary Alex

Proofreader
Safis Editing

Indexer
Rekha Nair

Graphics
Jason Monteiro

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Jonathan Hayward is a polymath with advanced degrees bridging mathematics, computers (UIUC), theology, and philosophy. He obtained his theology and philosophy degrees from Cambridge University. He has worked in many areas of web development, with a site (`http://cjsh.name/`) for "after hours" titles, and he is also interested in the human side of computing, including usability/UI/UX. His most popular work is a piece of poetry at `https://cjshayward.com/doxology/`. The faster route to get there is by typing cjsh.name/doxology, and it gets there. Jonathan has studied many languages, including French, Spanish, Latin, and Greek. He is currently learning Russian. He has worked on various other books as well (refer to `http://www.amazon.com/s/ref=nb_sb_noss_2?url=search-alias%3Daps&field-keywords=%22CJS+Hayward%22` to find out more).

I would like to thank my parents, John and Linda; my brothers, Matthew, Joe, and Kirk; my sisters-in-law, Kristin and Adrien; and my nephews, Jack and James. I would also like to thank all of the Packt Publishing editorial team, including a great many who I do not know, but I would like to single out Usha, Akshay, Neetu, Mohita, and Aparna. They are the editors who left me wishing we lived next door. Finally, I'd like to thank all those at Facebook for releasing ReactJS as a framework that is free for the rest of the world.

About the Reviewers

Antal Orcsik is a full-stack web developer from Hungary. He works at Prezi (`https://prezi.com/`) as a payment engineer. In the last decade, he worked for Hungary's biggest real estate catalog site and one of the biggest local weather portals. Then he joined the fantastic team that created a revolutionary presentation tool called Prezi to change the way the world shares ideas. During this time, he gained experience in Scala, Python, and PHP backend environments as well as JavaScript frontend technologies, while experimenting with various other fields of the full-stack web development spectrum. Antal is a big fan of cats, games, science fiction, and hamburgers.

I would like to thank my lovely girlfriend for her support and patience while I played my part in creating this book.

Sven A Robbestad is a Norwegian open source developer and frequent conference speaker with more than 20 years of experience of working on the Web. He is passionate about mobile development and is always ready to talk about code. Sven is currently employed at TeliaSonera (`http://www.teliasonera.com/`) as a technologist.

Hibai Unzueta is a multifaceted builder who was programming and designing complex systems much before he applied for his first job.

He believes that technology is nothing without technique and technique in turn needs a solid vision-based foundation. He enjoys territories where different knowledge areas overlap. Lately, he has been involved in projects of data visualization and user experience design.

For the past 2 years, he has been researching travel search paradigms and technology with the intention of launching a new project that is expected to rethink the way we do travel planning. As a result, he has worn many hats, but never all of them at once.

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.

To my nephews Jack and James: You are the light of my life

Table of Contents

Preface **v**

Chapter 1: Introduction and Installation **1**

A 10,000-foot overview **1**

An easier way to handle user interface programming 2

Programming paradigms 3

Installing the tools required **6**

Installing Google Chrome 6

Installing Node.js 10

Installing the Starter Kit for ReactJS 16

Summary **18**

Chapter 2: Core JavaScript **19**

The strict mode **20**

Variables and assignment **20**

Comments 21

Flow control 22

A note on values and NaN **22**

Functions 22

Comments 24

Loops 28

Taking a look at ECMAScript 6 29

Summary **30**

Chapter 3: Reactive Programming – The Basic Theory **31**

Declarative programming **33**

The war on Heisenbugs **33**

The Flux Architecture **35**

From the pit of despair to the pit of success **35**

Complete UI teardown and rebuild **38**

JavaScript as a Domain-specific Language **39**

The Big-Coffee Notation **40**
Summary **43**
Chapter 4: Demonstrating Nonfunctional Reactive Programming – A Live Example **45**
The history of a game with multiple ports **46**
The HTML for the web page **47**
Using a content distribution network wherever we can 48
Some simple styling 48
A fairly minimal page body 50
The JavaScript that animates that page **51**
A brief syntax note – Immediately Invoked Function Expression 51
Variable declaration and initialization 52
The function used to start or restart the game 53
The function that creates game levels 54
Getting our hands dirty with ReactJS classes 55
Tick-tock, tick-tock – the game's clock ticks 59
GAME OVER 62
Summary **65**
Chapter 5: Learning Functional Programming – The Basics **67**
Custom sort functions – the first example of functional JavaScript and first-class functions **69**
This leads us to array.filter() 72
Illusionism, map, reduce, and filter 74
Fool's gold – extending Array.prototype 75
Avoiding global pollution 77
The map, reduce, and filter toolbox – map 78
The reduce function 78
The last core tool – filter 81
An overview of information hiding in JavaScript **82**
Information hiding with JavaScript closures 85
Summary **87**
Chapter 6: Functional Reactive Programming – The Basics **89**
A trip down computer folklore's memory lane **90**
Advanced prerequisites for Hello, World! 92
Distinguishing the features of functional reactive programming **95**
If you learn just one thing... **96**
Learn what you can! **99**
JavaScript as the new bare metal **103**
Summary **105**

Chapter 7: Not Reinventing the Wheel – Tools for Functional Reactive Programming 107
ClojureScript 108
Om 110
Bacon.js 110
Brython – a Python browser implementation 112
Immutable.js – permanent protection from change 115
Jest – BDD unit testing from Facebook 122
Implementing the Flux Architecture using Fluxxor 126
Summary 127
Chapter 8: Demonstrating Functional Reactive Programming in JavaScript – A Live Example, Part I 129
What we will be attempting in this chapter 130
This project's first complete component 136
The render() method 139
Triggering the actual display for what we have created 142
Summary 143
Chapter 9: Demonstrating Functional Reactive Programming in JavaScript with a Live Example Part II – A To-do List 145
Adding a to-do list to our application 146
Including ReactJS add-ons in our project 147
Setting the appropriate initial state 147
Making text editable 147
Heavy lifting with render() 148
Inner functions used to render 148
Building the result table 149
Rendering our result 151
Differentiating columns visually 151
Summary 152
Chapter 10: Demonstrating Functional Reactive Programming in JavaScript: A Live Example Part III – A Calendar 153
Play it again Sam – an interesting challenge 154
Classical Hijaxing works well 157
Built with usability in mind, but there's still room to grow 160
Plain old JavaScript objects are all you need 164
Progressive disclosure that starts simply 166
A render() method can easily delegate 168
Boring code is better than interesting code! 169
A simple UI for simply non-recurring entries... 170

The user can still opt-in for more **172**
Avoiding being clever **174**
Anonymous helper functions may lack pixie dust **175**
How far in the future should we show? **178**
Different stripes for different entry types **178**
Now we're ready to display! **179**
Let's be nice and sort each day in order **180**
Let them use Markdown! **181**
One thing at a time! **182**
The holidays that inspired this calendar **183**
Summary **184**

Chapter 11: Demonstrating Functional Reactive Programming in JavaScript with a Live Example Part IV – Adding a Scratchpad and Putting It All Together **187**

Adding a WYSIWYG scratchpad, courtesy CKeditor **188**
Bringing all things together into one web page 188
This book is about ReactJS, so why use CKeditor? 189
CKeditor – small free offerings, and small is beautiful **190**
Including CKeditor in our page 190
Integrating all four subcomponents into one page **192**
Persistence **194**
One detail – persisting the CKeditor state 198
Summary **199**

Chapter 12: How It All Fits Together **201**

A review of the terrain covered **202**
Could the Mythical Man-Month have been avoided? **202**
ReactJS is just a view, but what a view! **204**
Programming is fun again! **205**
Summary **208**
The next steps from here 209

Appendix: A Node.js Kick start **211**

Node.js and INTERCAL **212**
Warning – Node.js and its ecosystem are hot, and hot enough to burn you badly! **218**
A sample project – a server for our Pragmatometer **223**
Client-side preparations **224**
The server side **227**
Summary **231**

Index **235**

Preface

Charles Cézanne famously said about the impressionist painter Claude Monet, "Monet is only an eye, but what an eye!" Today, we can similarly say, "ReactJS [or if you prefer, "ReactJS is only a view, but what a view!"

ReactJS has neither the intention nor the ambition to be a complete, general-purpose web framework. It doesn't even include tooling for Ajax calls! Rather, the intent is that you will use technologies that make sense for different concerns in your application, and use ReactJS's power tools for views and user interface development.

Functional reactive programming has been an extremely high-hanging fruit, with a prohibitive barrier to entry in terms of sheer mathematical expectations assumed in order to work with it. No longer with ReactJS! A veteran C++ programmer with no particularly deep math background—I said this to pick a profile of programmers who keep on saying on Stack Overflow that they don't get functional reactive programming—is a veteran programmer who stands a fair chance of getting real work done using ReactJS.

This book is about ReactJS, a simple and small technology that nonetheless lets huge teams work together on different components of a web page without stepping on each others' feet, but without a hint of bureaucratic measures. And add some liberal help of pixie dust.

What this book covers

Chapter 1, *Introduction and Installation*, provides a 10,000-foot overview of different programming paradigms, each of which has its strengths, and an introduction to the trio of functional programming, reactive programming, and functional reactive programming.

Chapter 2, *Core JavaScript*, covers some of JavaScript's better neighborhoods and omits the minefields, with a debt to Douglas Crockford, if not a complete agreement. In terms of the parts of JavaScript that you use, you should be doing most of your work within this core.

Chapter 3, *Reactive Programming – The Basic Theory*, is a basic exploration of reactive theory, or reactive programming, specifically in relation to Facebook's ReactJS user interface framework.

Chapter 4, *Demonstrating Nonfunctional Reactive Programming – A Live Example*, proves that not all development is from scratch. Most professional work is not greenfield. This will offer a live example of retrofitting a simple video game, the most recent implementation using jQuery, to take advantage of ReactJS (if you are using ReactJS, you will probably be doing other conversions from jQuery to ReactJS).

Chapter 5, *Learning Functional Programming – The Basics*, helps you if you want to understand functional programming but have no idea where to start. Here's one place to start! Map, reduce, and filter are introduced as an inexhaustible bag of tricks.

Chapter 6, *Functional Reactive Programming – The Basics*, covers what has been said about functional programming and reactive programming. It will be put together with some sage advice, and the last bit of foundation will be laid for the remaining hands-on work in this book.

Chapter 7, *Not Reinventing the Wheel – Tools for Functional Reactive Programming*, contains a lot to cover in one book, let alone one chapter. But there is meant to be an interesting sampling of a space where a lot of interesting options are made available, including writing ReactJS code from a language other than JavaScript.

Chapter 8, *Demonstrating Functional Reactive Programming in JavaScript – A Live Example, Part I*, is where we see an application housing a whimsical ReactJS component written in ReactJS from scratch, and showcasing the sweet JSX syntactic sugar that is not required but is still made available for ReactJS development.

Chapter 9, *Demonstrating Functional Reactive Programming in JavaScript with a Live Example Part II – A To-do List*, leads us to our first real component, designed to be used rather than just amuse. We implement a to-do list, and it has several markers other than just "done" to indicate what state, priority, and other things a task has.

Chapter 10, *Demonstrating Functional Reactive Programming in JavaScript: A Live Example Part III – A Calendar*, is where we build a calendar. It is intended to gracefully support not only one-time events but also many kinds of repeating events with all kinds of rules that people offer.

Chapter 11, *Demonstrating Functional Reactive Programming in JavaScript with a Live Example Part IV – Adding a Scratchpad and Putting It All Together*, provides a rich-text scratchpad with CKeditor. This shows how we might interoperate with other user interface tools. Then we wind up by bringing the four components together into one combined page and adding persistence functionality so that our user interface doesn't forget what it is told.

Chapter 12, *How It All Fits Together*, reviews what we covered in this book, and we look at the next steps in a world to explore.

Appendix, *A Node.js Kick start*, looks at some of the good, the bad, and the ugly of a "Wild, Wild West" technology that everybody seems to want in on.

What you need for this book

There is a bit of software to download, and you will need a web server that can at least serve static content. *Appendix*, *Node.js*, covers building a web server for the larger project in Node.js, but all the chapters can work with just a web server that will serve static content in the most basic fashion. You'll need a desktop computer, which can be almost anything that can run Node.js (if you choose to work through the appendix). The text will work well enough with Unix, Linux, Mac, Windows, Cygwin, and so on. If you want to run it from a mobile device, that may be a praiseworthy approach, but please work from a device (desktop or otherwise) using some standard server or desktop operating system.

However, all that you really need is a server or desktop, a browser such as Chrome, a web server, and a willingness to dive into something new. Everything else is provided in the text.

Who this book is for

This book is intended for programmers who want to dive into functional reactive programming and Facebook's ReactJS. There is an expectation of some general programming maturity, some knowledge of JavaScript, and some knowledge of producing user interfaces. Familiarity with functional programming is also one of the several things that would help, but the hope, whether realized or not, is to create a book using which a veteran programmer in any general-purpose language with some (perhaps light) knowledge of JavaScript and web development would be able to get things to work.

People who do have a solid background in frontend web development and JavaScript's functional core may be surprised how easy it is to work with ReactJS, and may find it like slicing a hot knife through butter.

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: "The `typeof` function returns a string containing a description of a type; thus, `typeof` can offer an extended type."

A block of code is set as follows:

```
var counter = (function() {
  var value = 0;
  return {
    get_value: function() {
      return value;
    },
    increment_value: function() {
      value += 1;
    }
  }
})();
```

Any command-line input or output is written as follows:

```
python -c "import binascii; print binascii.hexlify(open('/dev/random').
read(1024))"
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "When the installer starts, click on **Next**, as follows:"



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

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 and Installation

Welcome to the wonderful world of reactive (functional) programming in JavaScript! In this book, we will cover *the good parts* of JavaScript, although we will not follow it religiously. We will cover functional programming, reactive programming, and the ReactJS library and integrate all of these into functional reactive programming with JavaScript. If you are going to study reactive programming, it is our suggestion that you seriously consider functional reactive programming, including learning functional programming as much you can. In this context, the whole of functional reactive programming is more than the sum of its parts. We will be applying reactive programming to JavaScript user interface development. User interfaces are one domain in which **functional reactive programming** (**FRP**), really shines.

The topics that will be covered in this chapter are as follows:

- A 10,000-foot overview of the subject-matter, including:
    - A discussion of an easier approach to user interface programming
    - A brief discussion of programming paradigms, such as functional and reactive programming
- An overview of the chapters in this book
- A look at how to install some of the tools used over the course of this book

A 10,000-foot overview

There are a lot of things that can be stated, but (functional) reactive programming may be easier than you think. Today, much that has been written about functional reactive programming is intimidating, much like the instructions for closures a few years ago.

An easier way to handle user interface programming

Years ago, when I first began to study JavaScript, I picked one site and literally took everything from it that I needed to understand to perform objected-oriented information hiding, that is, how to create a JavaScript object that had private fields. I read perhaps two or three chapters that were dense with theoretical computer science and 10-15 percent of the introduction before giving up. Then I saw how easy it was to use a closure to create an object with private fields on a simple *monkey see, monkey do* basis:

```
 var counter = (function() {
var value = 0;
return {
  get_value: function() {
    return value;
  },
  increment_value: function() {
    value += 1;
  }
}
})();
```

Right now, functional reactive programming is in the state in which JavaScript closures were some years back. The amount of theory you *have to* read before you can start reactive programming is astounding, and large portions of the literature are of a PhD reading level. That's bad news. But the good news is that you don't have to wade through so much reading.

The purpose of this book is to provide something comparable to the *monkey see, monkey do* way of conveying how to use a closure to make a JavaScript object with private fields. Theory, as such, is not bad, nor is it a problem to introduce theory for a discussion, but making a full-fledged dissertation's theoretical backing as the price to do something simple is a problem.

It is our hope that this book will let you understand why building, for instance, a game UI in JavaScript is easier with functional reactive programming than with jQuery.

Programming paradigms

There are multiple programming paradigms around, and not all are mutually exclusive. Many programming languages are *multiparadigm* languages, supporting the use of more than one paradigm, including not only JavaScript, but also the likes of OCaml, PHP, Python, and Perl.

Note that you can at least sometimes use a paradigm with a language that is not explicitly designed to support it. Object-oriented programming was originally formulated not for languages such as Java or Ruby that are specifically intended to support object-oriented programming, but as matter of an engineering discipline originally used in languages that predate object-oriented programming.

Among the programming paradigms, we now have the following:

- **Aspect-oriented programming**: Some have suggested that the professional development of a programmer moves from procedural programming to object-oriented programming, then to aspect-oriented programming, and finally to functional programming. A canonical example of an aspect-oriented concern, for an aspect that is spread through the program in naïve usage, is logging. Aspect-oriented programming deals with cross-cutting aspects of programming, such as security, the diagnostic exposure of a state, and logging.
- **Declarative programming**: One of the key concepts of functional reactive programming is that it is declarative rather than imperative. In other words, $c = a + b$ does not mean take the present value of a, add the present value of b, and store their sum in c. Instead, we declare a lasting relationship that works a bit like *C1 = A1 + B1* in a spreadsheet. If *A1* or *B1* changes, *C1* is immediately affected by the change. What is stored in *C1* is not the value of *A1* plus the value of *B1* at the time of assignment, but something more lasting from which individual values may be obtained in a *print on demand* fashion.
- **Defensive programming**: Analogous to defensive driving, defensive coding means writing code that behaves correctly when it is given something defective. Functional reactive programming is, among other things, an approach to either functioning correctly or degrading gracefully in the face of network issues and nonideal, real-world conditions.

- **Functional programming**: Here, the term *function* has its mathematical rather than programming meaning. In imperative programming, functions can (and most often, they do) manipulate states. Hence, an `init()` function might initialize all of the data that a program initially needs to run initially. A function is something that takes zero or more inputs and returns a result. For example, $f(x) = 3x+1$, $g(x) = sin(x)$, and $h(x, y) = x'(y)$ (the derivative of x at y) are all mathematical functions; none of them command any manipulation of stateful data. A pure function is a function under a mathematical definition that excludes telling how to deal with states. Functional programming also allows and often includes, with the last derivative-based example, higher order functions, or functions that act on functions (in calculus, a derivative or an integral represents a higher order function, and iterative integration includes a higher order function that takes another higher order function as the input). Problems whose solutions center on abstract functions that operate on abstract functions tend to be more appealing to computer science types than something really used in the business world. The higher order functions explored here will be relatively concrete. You need not use higher order functions all the time, and once you've grasped the core concepts, they are not hard to use.
- **Imperative programming**: Imperative programming is a common way of programming, and for the majority of programmers who are first taught imperative programming, it may seem the most natural way to work. Functional reactive programming's marketing proposal includes a live alternative to this basic approach. An alternative to the natural-seeming tendency towards imperative programming is found in functional reactive programming's declarative programming, pure functions (including higher order functions) in functional programming, and the time series of reactive programming.
- **Information hiding**: Steve McConnel's *Code Complete* describes several methodologies, and tells us which are optimal for different settings (the sweet spot for procedural programming is on smaller projects than on object-oriented programming, for instance). For information hiding alone, his recommendation was *use this as much as possible*. In generic information hiding developments, a large project is approached by walling off secrets within the larger area, and larger secrets are divided by walling off subsecrets. A large portion of procedural programming, object-oriented programming, and functional programming alike is intended to facilitate information hiding. Information hiding is the software engineering concern behind the Law of Demeter, for example, you may have up to one dot in a method call `(foo.bar())`, but not two `(foo.baz.bar())`.

- **Object-oriented programming**: Instead of having a monolithic architecture, a program is segmented into objects. These objects have their own methods and fields and may in turn be segmented into further objects. This offers an acceptable level of information hiding for larger projects than procedural programming, even if object-oriented programming more or less starts with procedural programming and builds on top of it.
- **Patterns**: Patterns are not a recipe for good software, but at a higher level of human abstraction, they provide a way of talking about the best recurring solutions so as to avoid reinventing from scratch what has already been solved. Also, specific patterns are taken into the limelight, including MVC and now the **Observer** pattern, which is often not mentioned in relation to reactive programming despite being a founding ingredient.
- **Procedural programming**: Procedural programming is one of the oldest of the methodologies mentioned, and it was meant to provide some order to the spaghetti code fostered by the even older *goto-based* flow control. Perhaps we can criticize procedural programming for not doing enough once object-oriented programming, aspect-oriented programming, and object-oriented design patterns are available. It is the right thing to move on from procedural programming when you have tools to push further than procedural programming from a rat's nest of gotos, the pointer as the goto of data structures, and so on.
- **Reactive programming**: Suppose functional programming is, in large measure, programming where functions have first-class status and it is possible to make higher order functions (functions that act on other functions as input). Then reactive programming is, in large measure, programming where time series (functions that have different values over time) have first-class status. For music, games, user interfaces, and some other use cases, calculating the right value for the present moment is an area where reactive programming shines.
- **Functional reactive programming**: Functional reactive programming is reactive programming built on functional building blocks, and in which both functions and time series are first-class entities. There are some useful, and surprisingly simple, functions that act on one time series to provide another time series from it (either of these series can be acted on by other functions on time series). One of the major selling points of functional reactive programming is that it provides a more graceful and much more maintainable approach than following your nose straight into the *callback hell*.

Installing the tools required

Many readers will be comfortable enough with simply installing Chrome and Node.js, if they were not already installed some time ago. For those who would prefer step-by-step directions, here are the details of installing the appropriate software.

Google Chrome can be installed from `http://google.com/chrome`. Note that for some Linux distributions, Chrome may or may not be available from your package manager. Google Chrome is an obvious choice to think of for something to include in a distribution's packages, but licensing concerns may list Chrome as non-free due to some of its parts, meaning that as far as the distribution maintainers are concerned, you may be welcome to use this, but we're not comfortable including it in a free-only package repository.

Node.js is available from `http://nodejs.org/download`. If you are running Linux, it is probably better to obtain it through your package manager. Note that Node.js comes with its own package manager, npm, which can be used to download packages that can be used under Node.js.

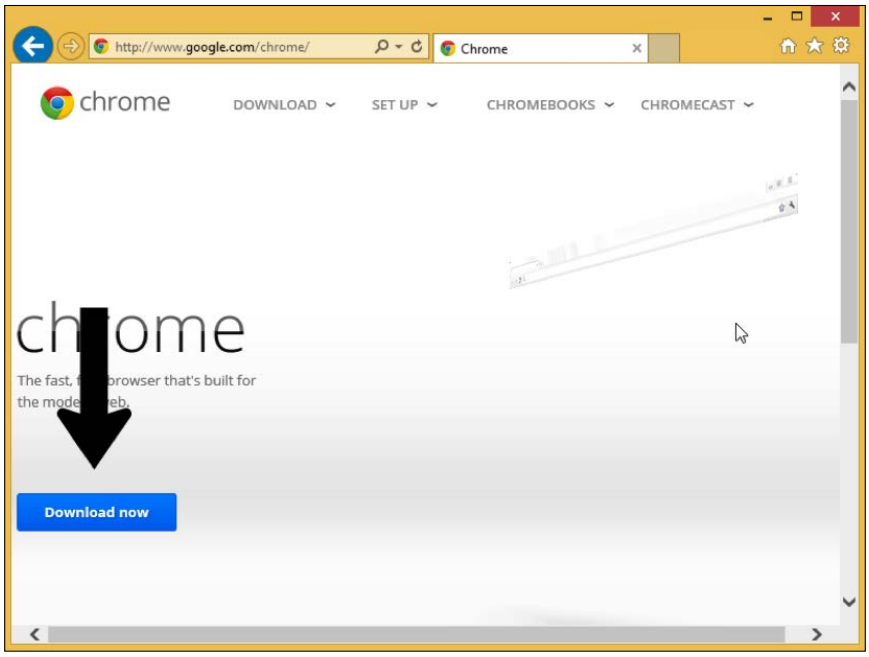
The useful Starter Kit for ReactJS is available from `http://facebook.github.io/react/downloads.html`.

The following instructions are given for Windows 8.1 (I prefers to develop in Mac or Linux, but is writing for Windows 8.1 as a common lingua franca).

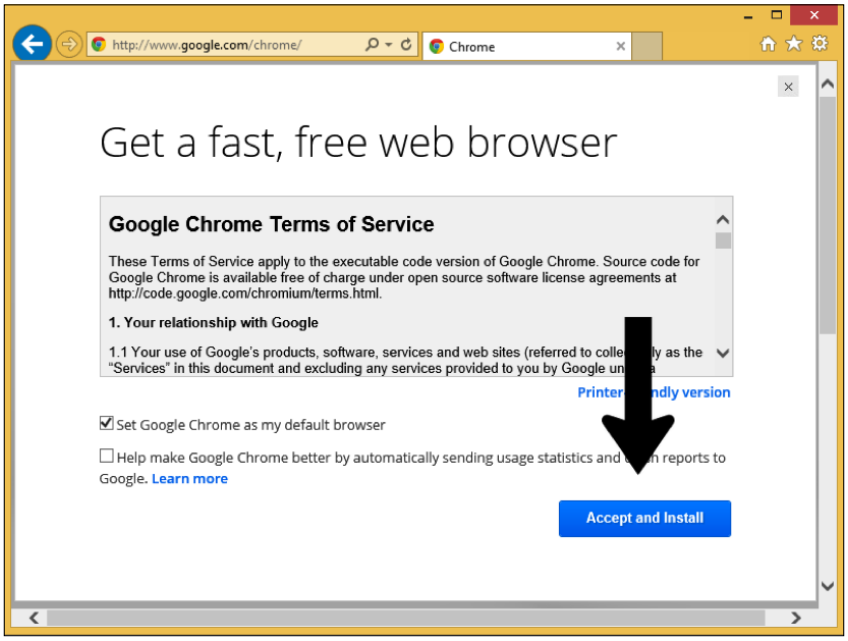
Installing Google Chrome

We will be using Google Chrome as the primary reference browser:

1. To download it, go to `http://google.com/chrome`, and click on the **Download now** button to the left, towards the bottom, as shown here:



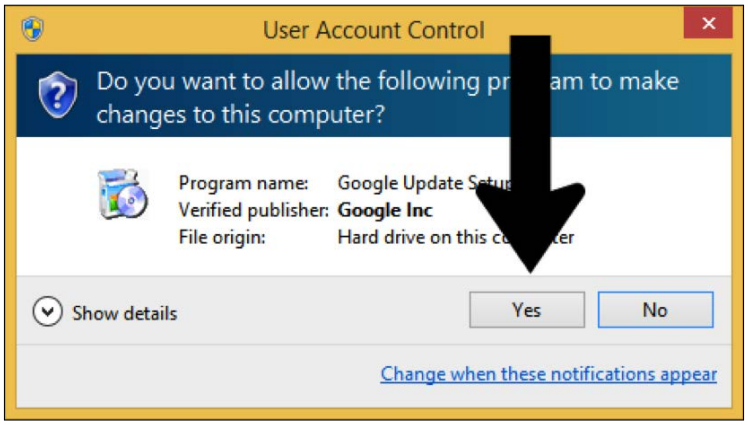
2. Next, click on the **Accept and Install** button down and to the right, as shown in the following screenshot:



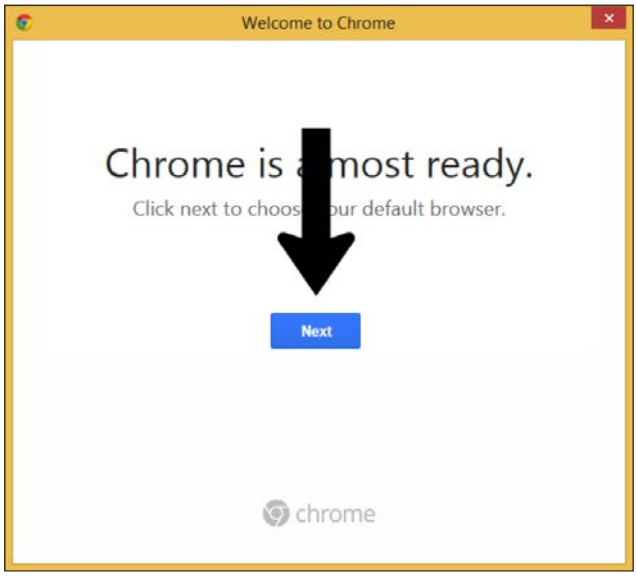
3. After that, click on the **Run** button when asked whether you want to run or save the installer, as shown here:



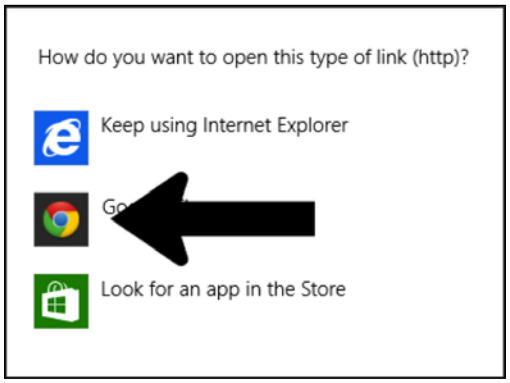
4. Next, authorize Chrome's installer to make changes to the system, as shown in the following screenshot:



5. Then click on the **Next** button to install Chrome, as shown in the following screenshot:



6. Wait a minute for it to install and then if you are willing, set Chrome as your default browser:

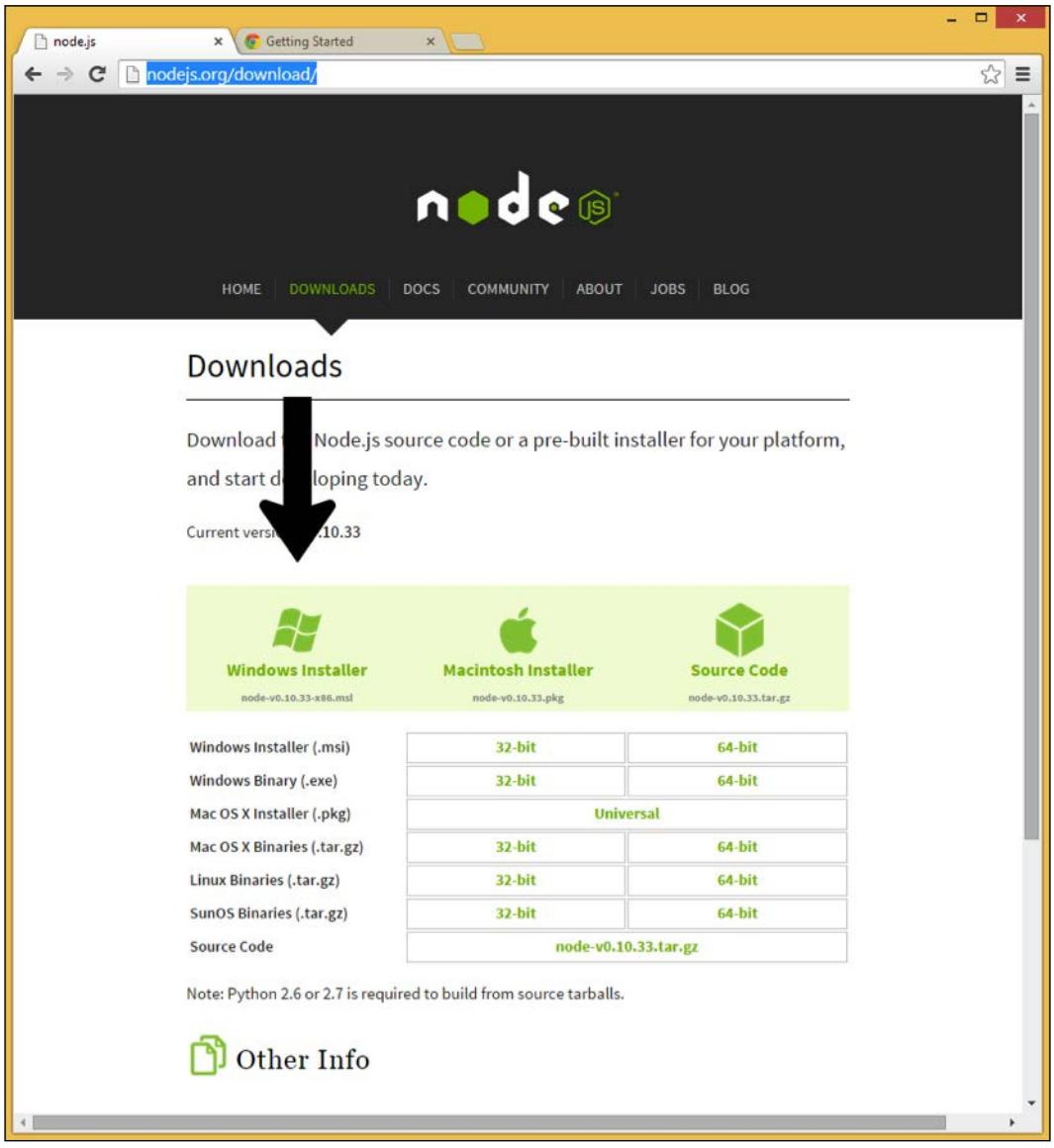


And that's it!

Installing Node.js

Installing Node.js is straightforward and it makes it easy to start serving HTTP using JavaScript as the only language.

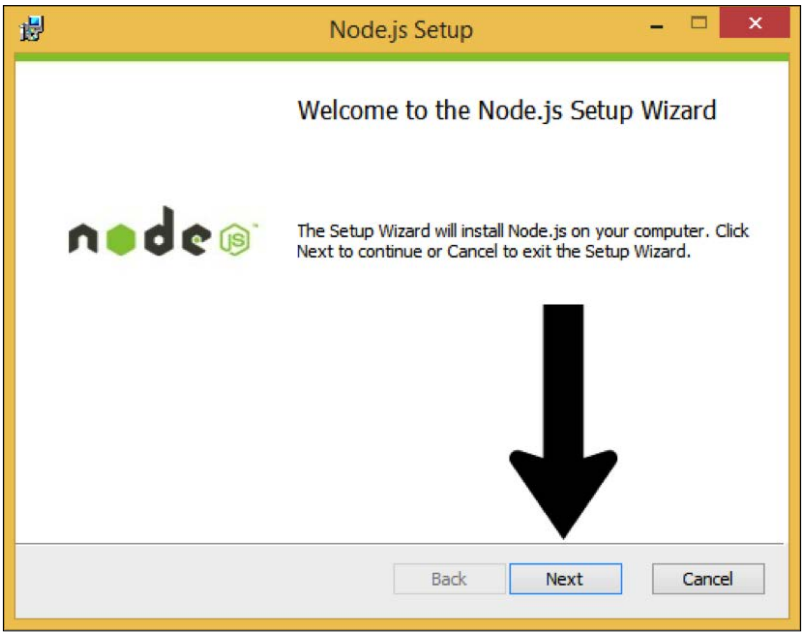
1. Go to `http://nodejs.org/download`:



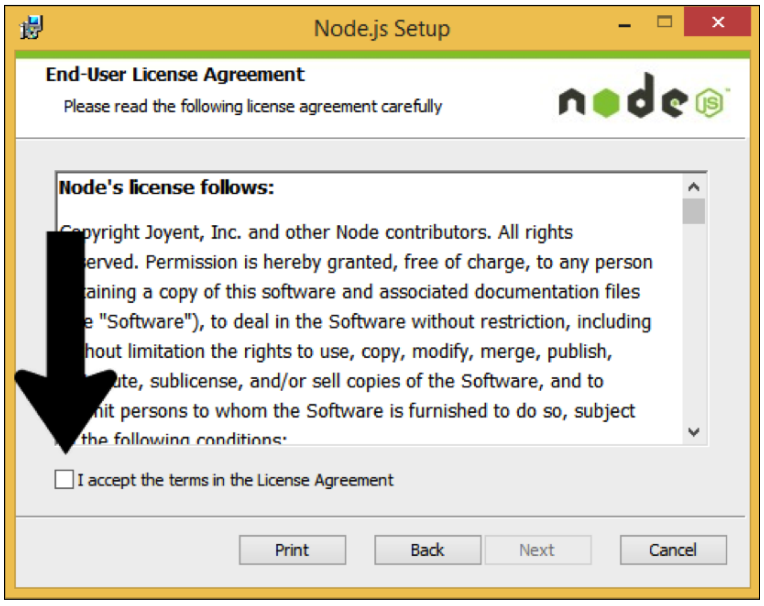
2. Click on the **Windows Installer** and wait for the installer to download. Then click on the bottom-left part of the window, as shown in this screenshot:



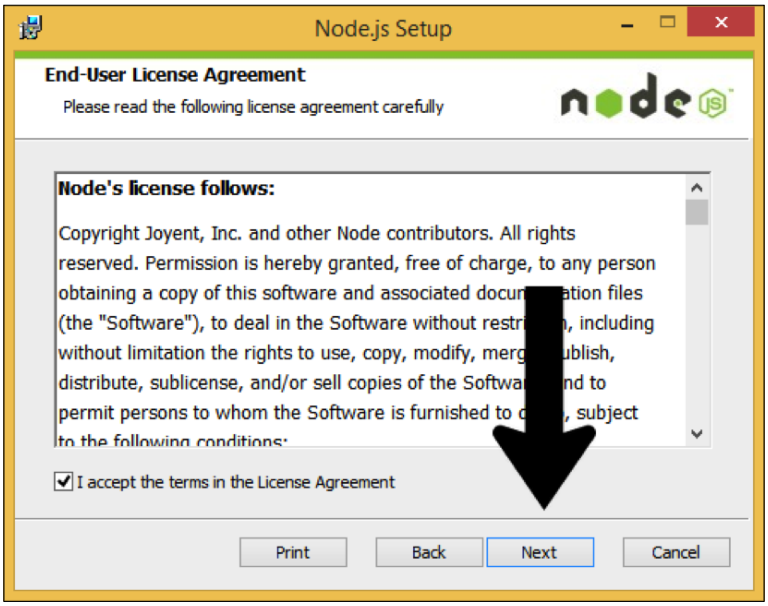
3. When the installer starts, click on **Next**, as follows:



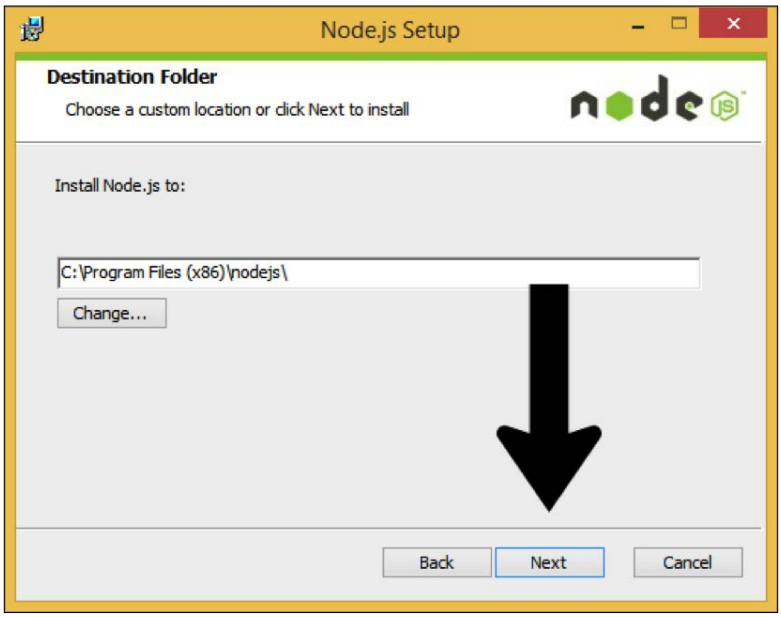
4. Then click on the checkbox to accept the terms of the agreement, as shown here:



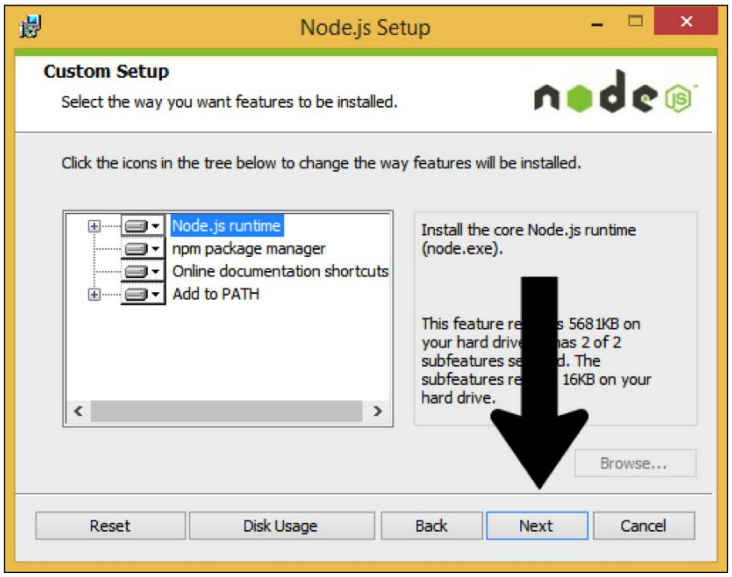
5. After that, click on the **Next** button to continue, as shown in the following screenshot:



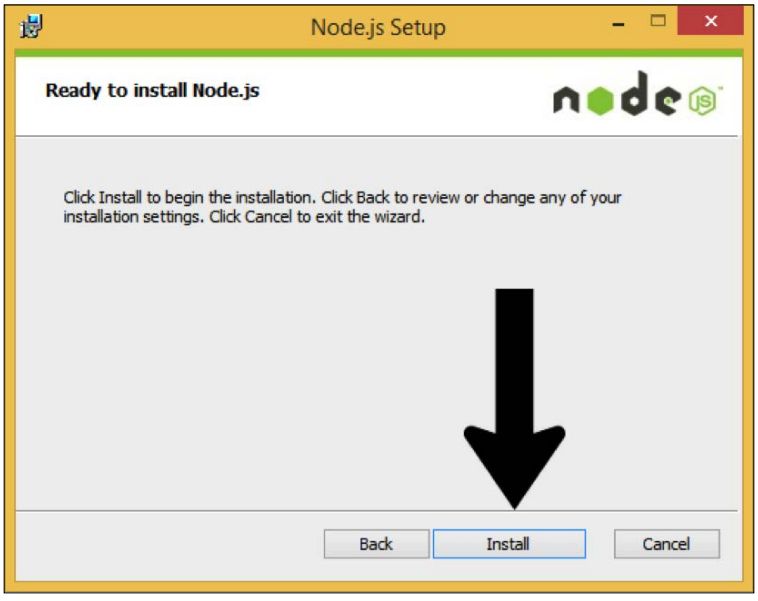
6. When asked where to install the software, click on the **Next** button, as shown in this screenshot:



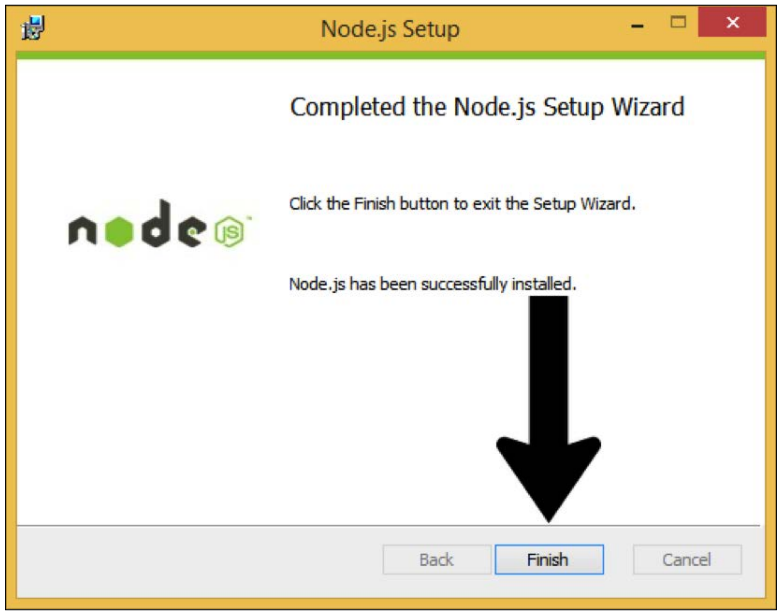
7. Then click on **Next** to move forward, as shown in the next screenshot. Customize the features if you want to:



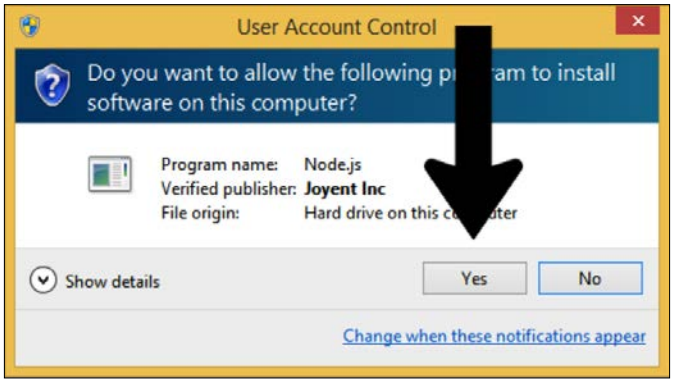
8. After that, click on the **Install** button to go ahead with the installation, as shown here:



9. Finally, click on the **Finish** button to finish installing Node.js, as shown in this screenshot:



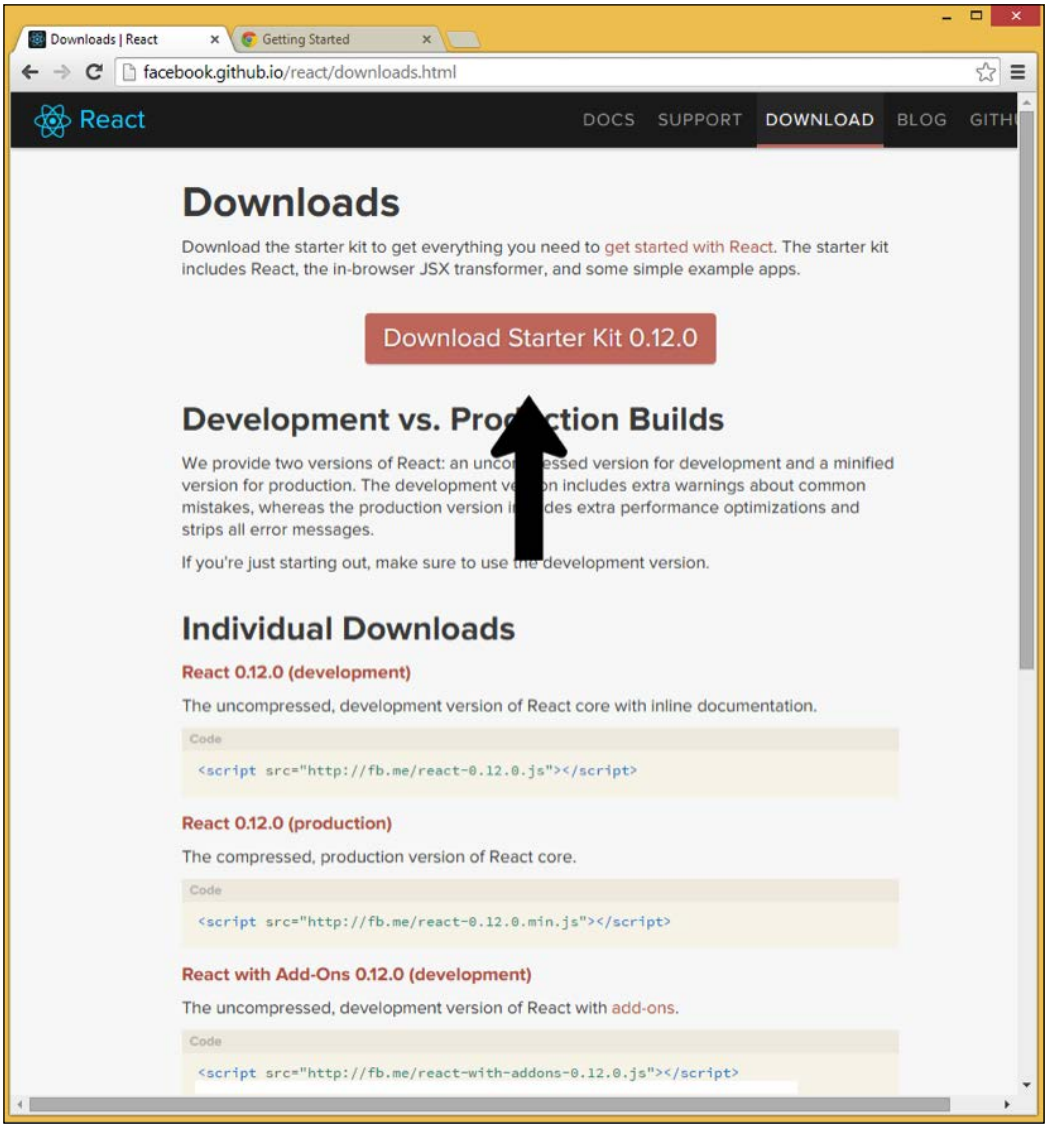
10. Authorize the installer to make changes to your computer, shown as follows:



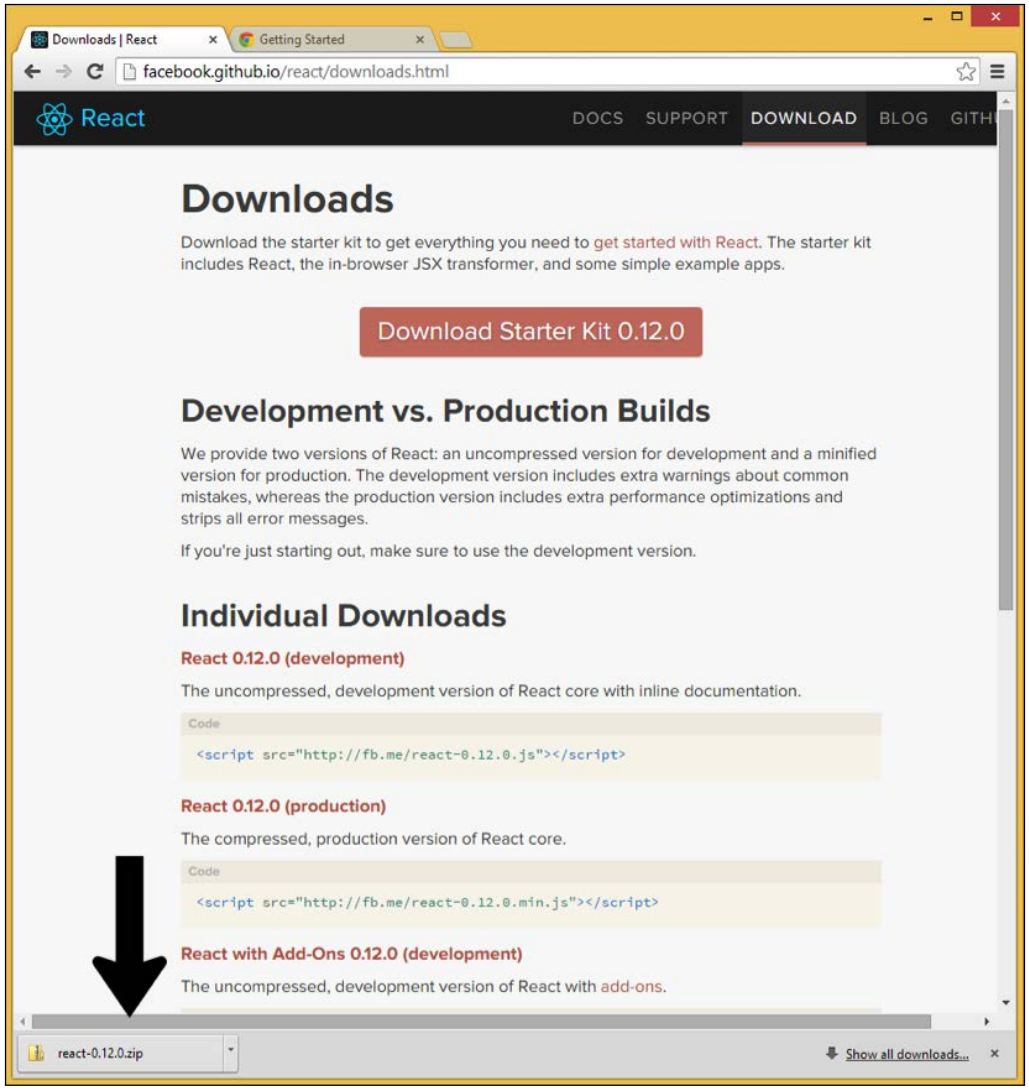
Installing the Starter Kit for ReactJS

To install the Starter Kit perform the following steps:

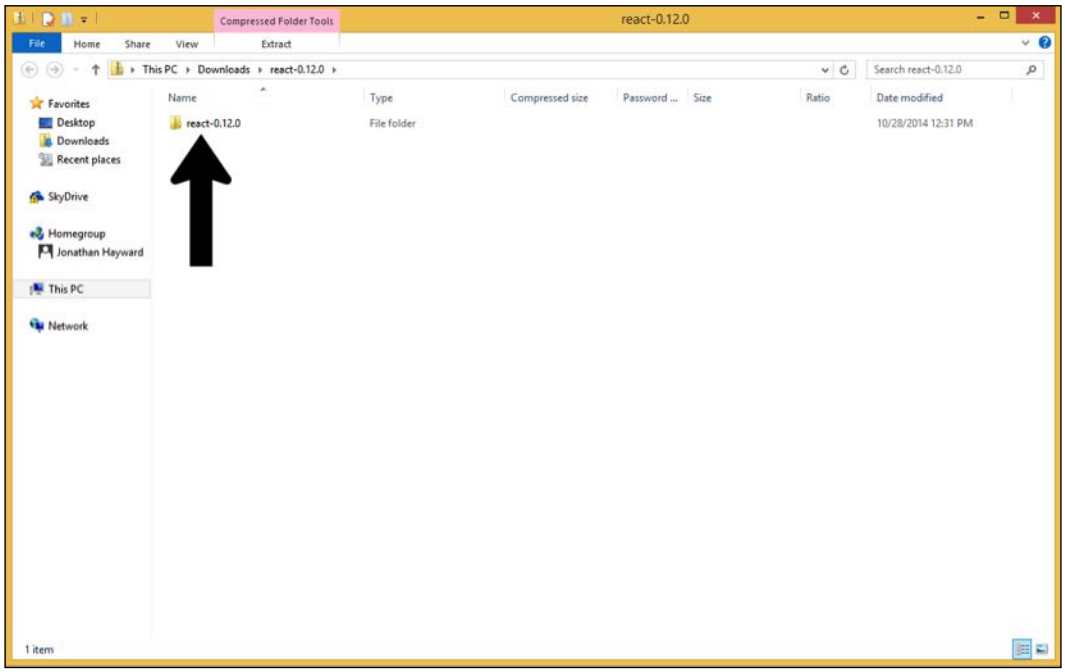
1. Go to `http://facebook.github.io/react/downloads.html` and you will see a screen similar to the following screenshot:



2. Click on **Download Starter Kit 0.12.0** to start the download (this will be visible in the bottom-left corner), as shown in the preceding screenshot.
3. You will see a ZIP file downloaded at the bottom:



4. From here, you should be able to explore the contents of the ZIP file:



Summary

In this chapter, we had a brief overview of programming paradigms to suggest where functional reactive programming might fit, and installed the basic tools.

We will be moving on to discussing JavaScript in the next chapter. The basics of Node.js are discussed in *Appendix*.

2
Core JavaScript

JavaScript is a good and a bad language rolled into one. There are some truly awful parts to JavaScript and some excellent parts. Overall, JavaScript is a dynamic, loosely typed, interpreted scripting language. The approach to the language as a whole is to explore it with emphasis on *The Good Parts* as described by Douglas Crockford, because of how bad the bad parts of JavaScript are: they are truly minefields. Crockford has had a profound influence on how JavaScript is used now; enough that the blog post titled *AngularJS: The Bad Parts* instantly, clearly, and completely telegraphs what sort of things will be dissected in painful detail.

Topics covered in this chapter include:

- **Strict mode**: Implicitly used in code including ECMAScript 6 modules
- **Variables and assignment**: Some of the basic building blocks of programming in any language
- **Comments**: There are two styles; we focus on one in preference to the other due to the ease of producing unintended behavior by having comments really begin or end somewhere other than where the programmer intended
- **Flow control**: A basic look at if-then, if-then-else, and switch statements
- **A note on values and NaN**: Notes on truthiness and the toxic *not a number* value
- **Functions**: Illustration of JavaScript functions, one of the best parts of the language
- **A few brief notes on ECMAScript 6**: The first truly radical change in core JavaScript in years

The strict mode

`"use strict";` as the first line of a file or function will cause certain things that can cause innumerable problems (such as assigning to a variable without declaring it) to cleanly error out with a line number called out, rather than leaving you to ferret out clues from any of a number of possible consequences. The `"use strict";` can also be the first line of a function, in which case the function is in strict mode.

Perl users will know about the `-w` flag, possibly the best-known flag associated with the language, and its successor, Perl's use warnings pragma. The documentation says things like, opening the list of known bugs, *The behavior implied by the use warnings is not mandatory*.

JavaScript's strict mode by itself may or may not be on par with Perl's use warnings pragma, but it's at least something to make you get into the habit of using.

Variables and assignment

Variables should be declared with the `var` keyword. Any variable that is declared outside of a function, or any variable that is used without being declared, is a global variable, and global variables are a big problem; their place in *default JavaScript* is one of the major design flaws.

When there was a major public relations push for Java, JavaScript was named to run on Java's coattails, and certain decisions were made to make JavaScript code look like Java. These decisions were unfortunate. JavaScript is formally a C-like language and its most recent common ancestor with Java or C# is C, not C++ or Java. JavaScript has been described as being like Scheme in C's clothing. Lisp is a syntax associated with a family of languages including Scheme, Common Lisp, Clojure, and ClojureScript, and it is arguable that the increasingly functional emphasis in pursuit of the best JavaScript is from ClojureScript. In JavaScript, there are no separate integer and float types; numbers are 64-bit floating point values that behave like integers when used within a certain long range. However, they do sometimes give surprises to new programmers; for example, 0.1 + 0.2 does not equal 0.3, for historical reasons that also plague innumerable other languages.

Basic variable assignments can look C-like:

```
var x;
var y = 12 + 2;
```

If a variable is declared without being assigned, as `x` in the preceding example, its value will be undefined.

The following are equivalent:

```
y = y + 1;
y += 1;
++y;
```

We will avoid the use of the last of the three listed in the preceding example, as it is not considered one of The Good Parts. Douglas Crockford, in one of his videos, recounts a tale where he gave an hour long eloquent defense of the `++y` usage, and then had an interminable debugging session in which the subtleties of `++y` had bitten him. Unlike the first two options in the preceding example, one can assign a value to it, and `++y` and `y++` are not the same. Crockford then magnanimously backed down from his earlier position.

Comments

Most languages support comments in some form. Besides explanation of the code, they are also used to temporarily deactivate the code. JavaScript has the comments that would be expected of a language made to look like Java; however, Javadoc comments are not natively special (various solutions such as JSDoc have been made to fill the gap).

JavaScript supports both classes of C++ comments. The first three lines contain only a comment and no executable code, the last line has a line of code and then a comment until the end of the line:

```
/*
 * Multiline comments are legal.
 */

if (x) // In this case, we ...
```

We will be avoiding the multiline comments. Asterisks and slashes occur often enough in regular expressions that multiline comments can be tripped up, and regular expressions may be required by a context or written by someone else in code that will still trip unintended effects. Inline comments, by their nature, are less vulnerable to surprise effects.

Flow control

If-then and if-then-else works as described in the following example code, one of which does something if a number is nonzero, and the other of which takes one action if a number is nonzero and another if it is false. Both use truthiness in that 0 (and NaN) are falsy, while any other value is truthy:

```
if (books_remaining) {
  print_title();
}

var c = 1;
if (c) {
  c += 2;
} else {
  c -= 1;
}
```

A note on values and NaN

All values can be used in Boolean context and tested for truthiness. The values `undefined`, `null`, `''`, `0`, `[]`, `false`, and `NaN` (Not a Number) are `falsy` and all the other values are `truthy`.

`NaN` in particular is a special case and it does not behave like other *real* numeric values. `NaN` is toxic; any calculation that includes `NaN` will give a result of `NaN`. Furthermore, although `NaN` is `falsy`, it does not equal anything, including `NaN` itself. The usual way of checking for `NaN` is via the `isNaN()` function. If you find `NaN` lurking somewhere unexpected, you might give debugging log statements for code, leading to where you detected `NaN`; there may be some distance between where `NaN` was first generated and where you observed it corrupting your results.

Functions

In a function, by default, control flows from the beginning to the end, and the function returns `undefined`. Optionally, there can be a return statement before the end, and the function will stop execution and return the value in question. The following example illustrates a function with a return:

```
var add = function(first, second) {
  return first + second;
}

console.log(add(1, 2));

// 3
```

The previous function takes two parameters, although functions can be given (without an error or error message) fewer or more arguments than the declaration specifies. If they are declared to have values, those values will be present as local variables (in the preceding example, `first` and `second`). In any case, the arguments are also available in an array-like object called `arguments` which has a `.length` method (arrays have a `.length` method which is one greater than the highest position of an item), but not the rest of an array's features. Here we make a function that can take any arbitrary number of *Number arguments*, and returns their (arithmetical) average by using the `arguments` pseudo-array.

```
var average_arbitrarily_many() {
  if (!arguments.length) {
    return 0;
  }
  var count = 0;
  var total = 0;
  for(var index = 0; index < arguments.length; index += 1) {
    total += arguments[i];
  }
  return total / arguments.length;
}
```

Basic data types include number, string, Boolean, symbol, object, null, and undefined. Objects include function, array, date, and RegExp.

The fact that the objects include the functions means that the functions are values and can be passed as values. This facilitates higher-order functions, or functions that take functions as values.

For an example of a higher-order function, we will include a `sort` function that sorts an array and optionally accepts a comparator. This builds on function definitions, actually containing a function definition within a function definition (which is perfectly legal as anywhere else), and then an implementation of QuickSort where values are divided into *compares as less than the first element*, *compares as equal to the first element*, and *compares as greater than the first element*, and the first and last of these three are recursively sorted. We check for nonempty length before sorting, in order to avoid infinite recursion. An implementation of the classic QuickSort algorithm, here implemented as a higher-level function, is as follows:

```
var sort = function(array, comparator) {
  if (typeof comparator === 'undefined') {
    comparator = function(first, second) {
      if (first < second) {
        return -1;
      } else if (second < first) {
        return 1;
```

```
        } else {
          return 0;
        }
      }
    }
    var before = [];
    var same = [];
    var after = [];
    if (array.length) {
      same.push(array[0]);
    }
    for(var i = 1; i < array.length; ++i) {
      if (comparator(array[i], array[0]) < 0) {
        before.push(array[i]);
      } else if (comparator(array[i], array[0]) > 0) {
        after.push(array[i]);
      } else {
        same.push(array[i]);
      }
    }
    var result = [];
    if (before.length) {
      result = result.concat(sort(before, comparator));
    }
    result = result.concat(same);
    if (after.length) {
      result = result.concat(sort(after, comparator));
    }
    return result;
  }
```

Comments

There are several basic features and observations to note regarding this function, which is not intended to be pushing the envelope, but demonstrate how to cover standard bases well:

1. We use `var sort = function()...` rather than the permitted `function sort()...`. When used within a function, this stores the function in a variable, rather than defining something globally. Note that it may be helpful in debugging to include a name for the function, `var sort = function sort()...` to access the function only through the variable, and let debuggers pick up on the second. For example: `sort` rather than referring to the function anonymously. Note that with `var sort = function()`, the variable declaration is hoisted, not the initialization; with `function sort()`, the function value is hoisted, available anywhere within the current scope.

2. This is a standard way of checking to see if only one of the two arguments has been specified, that is if an array has been provided without a sort function. If not, a default function is provided. We run a few trial runs of sorting:

```
console.log(sort([1, 3, 2, 11, 9]));
console.log(sort(['a', 'c', 'b']));
console.log(sort(['a', 'c', 'b', 1, 3, 2, 11, 9]);
```

This gives us:

```
[1, 2, 3, 9, 11]
["a", "b", "c"]
["a", 1, 3, 2, 11, 9, "b", "c"]
```

This gives us an opportunity to debug. Now, let's suppose we add the following:

```
console.log('Before: ' + before);
console.log('Same: ' + same);
console.log('After: ' + after);
```

Before the declaration of result, we get:

```
[1, 2, 3, 9, 11]
Before:
Same: a
After: c,b
Before: b
Same: c
After:
Before:
Same: b
After:
["a", "b", "c"]
Before:
Same: a,1,3,2,11,9
After: c,b
Before: b
Same: c
After:
Before:
Same: b
After:
["a", 1, 3, 2, 11, 9, "b", "c"]
```

The output that says `Same: a,1,3,2,11,9` looks suspicious, a `Same` bucket should have identical values, so that an appropriate output might be `Same: 2,2,2` or just `Same: 3` where the `Same` list has five values, no two of which are alike. This can't be the behavior we intended. It appears that the integers are being classified as the same as the initial "a", and the rest of it is being sorted. A little investigation confirms that `'"a" < 1'` and `'"a" > 1'` are both false, so our comparator could be improved.

We make a dictionary order sort of their types. This is somewhat arbitrary to sort first on alphabetical order of type, and then on the default sort order within types defaults which could be overridden with another comparator. This is an example of another of the many kinds of comparators one might use to sort an array: this one, unlike the previous one, segments different kinds of items such as Numbers sorted in order, and will appear before Strings, sorted in order:

```
var comparator = function(first, second) {
  if (typeof first < typeof second) {
    return -1;
  } else if (typeof second < typeof first) {
    return -1;
  } else if (first < second) {
    return -1;
  } else if (second < first) {
    return 1;
  } else {
    return 0;
  }
}
```

The `typeof` function returns a string containing a description of a type; thus `typeof` can offer an extended type. With a comparator function similar to this, one can meaningfully compare objects such as records that hold a first name, a last name, and an address.

Objects can be declared via curly brace notation. Blocks of code and objects can both use curly braces, but these are two separate things. The following code with its braces, is not a block of code with statements to execute; it defines a dictionary with keys and values:

```
var sample = {
  'a': 12,
  'b': 2.5
};
```

Unless a key is a reserved word or contains a special character, as `'strange.key'` does (here a period), the quotation marks around a key are optional. JSON formatting, in order to have a simple and consistent rule, requires quotes in all cases, and specifically double quotes, not single.

An example of a record having a first name, a last name, and email address, perhaps populated via JSON, is shown next. This example is not JSON because it does not follow JSON's rules about double quoting all keys and all strings, but it illustrates an array of records which could have other fields and could be much longer. It may make sense to sort by distance or seniority (with populated fields not shown here): there is a whole world of possible comparators one might use for records.

```
var records = [{
    first_name: 'Alice',
    last_name: 'Spung',
    email: 'a.spung@yahoo.com'
  }, {
    first_name: 'Robert',
    last_name: 'Hendrickson',
    email: 'Bob.Hendrickson@gmail.com'
  }
];
```

Note that a trailing comma is not merely inappropriate in JavaScript (after the last entry in almost anything separated by commas), but it has some strange and unexpected behavior which can be extremely hard to debug.

JavaScript was designed to have semicolons at the end of a statement which may be optional. This is a debatable decision, and was tied to a decision to make a popular language that regular non-programmers could use without involving IT people, a factor also involved in the design of SQL. We should always supply semicolons when they are appropriate. One side effect of this is that `return` alone on a line will return undefined. This means that the following code will not have the desired effect and will return undefined:

```
return
  {
  x: 12
  };
```



The effect as the code is executed is different from what it appears to be and probably what was intended, so it is advisable to not do it.

To get the desired effect, the open curly brace should be on the same line as the return statement, as shown next:

```
return {
  x: 12
};
```

However, JavaScript does have an object-oriented programming that avoids a classical difficulty in object-oriented programming: having to get the ontology right the first time. Objects are usually best constructed, not as instances of a class, but by factories. Or so Douglas Crockford has been abbreviated. Prototypes are still part of the language, like many features good and bad, but barring esoteric use cases objects should usually be made by factories that allow for a "better than ontology-driven classes" approach to object-oriented programming. We will be avoiding the pseudo-classical new `function()`, not because it can clobber global variables if you ever forget the new, but because its semblance of traditional object-oriented programming does not really help that much.



You should be aware of a wide-respected convention in JavaScript that constructors intending to work with `new` begin with a capital letter. If a function name begins with a capital letter, it is intended to be used with the `new` keyword, and strange things may happen if it is invoked without the `new` keyword.

In JavaScript, the interests served by classical object-oriented programming in some other languages are sometimes best advanced by functional programming.

Loops

Loops include the `for` loop, the `for in` loop, the `while-do` loop, and the `do-while` loop. The `for` loop works as in C:

```
var numbers = [1, 2, 3];
var total = 0;
for(var index = 0; index < numbers.length; ++index) {
  total += numbers[index];
}
```

The `for in` loop will loop over everything in an object. The `hasOwnProperty()` method can be used to examine only an object's fields. The two variants, for an object named `obj`, are as follows:

```
var counter = 0;
for(var field in obj) {
  counter += 1;
}
```

This will include any fields from the prototype chain(not explained here). To check things that are direct properties of an object itself, and not potentially noisy data from a prototype chain, we use an object's `hasOwnProperty()` method:

```
var counter = 0;
for(var field in obj) {
  if (obj.hasOwnProperty(field)) {
    counter += 1;
  }
}
```

Ordering is not guaranteed; if you are looking for any specific fields, it is worth considering to just iterate over the fields you want and check them on an object.

Taking a look at ECMAScript 6

JavaScript tools have been booming, with one tool being eclipsed by another and an incredibly rich ecosystem that few developers can boast knowing both broadly and deeply. However, the core language, ECMAScript or JavaScript, has been stable for several years.

ECMAScript 6, with one introductory roadmap available from `http://tinyurl.com/reactjs-ecmascript-6`, introduces profound new changes and expansions to the core language. As a rule, these features enhance, deepen, or make more consistent the functional aspects of JavaScript. It might be suggested that ECMAScript 6 features that do not do this kind of work, such as enhanced class-oriented syntactic sugar to let Java programmers pretend that JavaScript means programming in Java, may be ignored.

ECMAScript 6 features are a force to reckon with and at the time of this writing have more than started to make their way into the mainstream browsers. If you want to expand and improve your value as a JavaScript developer, don't restrict yourself to digging deeper into the rich JavaScript ecosystem, no matter how important that may be. Learn the new JavaScript. Learn an ECMAScript with heaping mounds more better parts.

Summary

In this whirlwind tour of some of JavaScript's better parts, we have covered foundational building blocks that can be helpful as we push further in JavaScript. These include variables and assignment, comments, flow control, values, NaN functions, and ECMAScript 6.

In the *Variables and assignment* section, we threw light on some basic building blocks of most programming, although the emphasis in functional reactive programming may lie elsewhere. In the *Comments* section, we understood how comments are the same everywhere, but the main concern here is to prevent strange surprises.

The *Flow control* section covered basic flow control within a function (or possibly outside of any function, although that is usually to be avoided).

In the section *A note on values and NaN*, we discussed that similarly to Perl, JavaScript holds truth to be self-evident; that is, the things are falsy if they are null, zero, empty, Not a Number, and so on, and true for anything not on the list.

In the *Functions* section, we got a look at the functions that included a somewhat involved example. In the section *ECMAScript 6,* we discussed how the core JavaScript language is expanding.

This has been a brief tour of highlights, not a comprehensive tour. If you need a more comprehensive grounding in JavaScript, there are multiple options available to you. We will continue in the next chapter with basic theory for reactive programming.

3
Reactive Programming – The Basic Theory

Reactive programming, including functional reactive programming as will be discussed later, is a programming paradigm that can be used in multiparadigm languages such as JavaScript, Python, Scala, and many more. It is primarily distinguished from imperative programming, in which a statement does something by what are called *side effects*, in literature, about functional and reactive programming. Please note, though, that side effects here are not what they are in common English, where all medications have some effects, which are the point of taking the medication, and some other effects are unwanted but are tolerated for the main benefit. For example, Benadryl is taken for the express purpose of reducing symptoms of airborne allergies, and the fact that Benadryl, in a way similar to some other allergy medicines, can also cause drowsiness is (or at least was; now it is also sold as a sleeping aid) a *side effect*. This is unwelcome but tolerated as the lesser of two evils by people, who would rather be somewhat tired and not bothered by allergies than be alert but bothered by frequent sneezing. Medication side effects are rarely the only thing that would ordinarily be considered *side effects* by a programmer. For them, *side effects* are the primary intended purpose and effect of a statement, often implemented through changes in the stored state for a program.

Reactive programming has its roots in the observer pattern, as discussed in Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides's classic book *Design Patterns: Elements of Reusable Object-Oriented Software* (the authors of this book are commonly called *GoF* or *Gang of Four*). In the observer pattern, there is an observable subject. It has a list of listeners, and notifies all of them when it has something to publish. This is somewhat simpler than the publisher/subscriber (PubSub) pattern, not having potentially intricate filtering of which messages reach which subscriber which is a normal feature to include.

Reactive programming has developed a life of its own, a bit like the MVC pattern-turned-buzzword, but it is best taken in connection with the broader context explored in GoF. Reactive programming, including the ReactJS framework (which is explored in this title), is intended to avoid the shared mutable state and be idempotent. This means that, as with RESTful web services, you will get the same result from a function whether you call it once or a hundred times. Pete Hunt formerly of Facebook—perhaps the face of ReactJS as it now exists—has said that he would rather be predictable than right. If there is a bug in his code, Hunt would rather have the interface fail the same way every single time than go on elaborate hunts for heisenbugs. These are bugs that manifest only in some special and slippery edge cases, and are explored later in this book.

ReactJS is called the *V* of *MVC*. That is, it is intended for user interface work and has little intentions of offering other standard features. But just as the painter Charles Cézanne said about the impressionist painter Claude Monet, "Monet is only an eye, but what an eye!" about MVC and ReactJS, we can say, "ReactJS is only a view, but what a view!"

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

- Declarative programming
- The war on heisenbugs
- The Flux Architecture
- From pit of despair to the pit of success
- A complete UI teardown and rebuild
- JavaScript as a **Domain-specific Language** (**DSL**)
- Big-Coffee Notation

ReactJS, the library explored in this book, was developed by Facebook and made open source in the not-too-distant past. It is shaped by some of Facebook's concerns about making a large-scale site that is safe to debug and work on, and also allowing a large number of programmers to work on different components without having to store brain-bending levels of complexity in their heads. The quotation "Simplicity is the lack of interleaving," which can be found in the videos at `http://facebook.github.io/react`, is not about how much or how little stuff there is on an absolute scale, but about how many moving parts you need to juggle simultaneously to work on a system (See the section on *Big-Coffee Notation* for further reflections).

Declarative programming

Probably, the biggest theoretical advantage of the ReactJS framework is that the programming is declarative rather than imperative. In imperative programming, you specify what steps need to be done; declarative programming is the programming in which you specify what needs to be accomplished without telling how it needs to be done. It may be difficult at first to shift from an imperative paradigm to a declarative paradigm, but once the shift has been made, it is well worth the effort involved to get there.

Familiar examples of declarative paradigms, as opposed to imperative paradigms, include both SQL and HTML. An SQL query would be much more verbose if you had to specify how exactly to find records and filter them appropriately, let alone say how indices are to be used, and HTML would be much more verbose if, instead of having an IMG tag, you had to specify how to render an image. Many libraries, for instance, are more declarative than a rolling of your own solution from scratch. With a library, you are more likely to specify only what needs to be done and not—in addition to this—how to do it. ReactJS is not in any sense the only library or framework that is intended to provide a more declarative JavaScript, but this is one of its selling points, along with other better specifics that it offers to help teams work together and be productive. And again, ReactJS has emerged from some of Facebook's efforts in managing bugs and cognitive load while enabling developers to contribute a lot to a large-scale project.

The war on Heisenbugs

In modern physics, Heisenberg's uncertainty principle loosely says that there is an absolute theoretical limit to how well a particle's position and velocity can be known. Regardless of how good a laboratory's measuring equipment gets, funny things will always happen when you try to pin things down too far.

Heisenbugs, loosely speaking, are subtle, slippery bugs that can be very hard to pin down. They only manifest under very specific conditions and may even fail to manifest when one attempts to investigate them (note that this definition is slightly different from the jargon file's narrower and more specific definition at `http://www.catb.org/jargon/html/H/heisenbug.html`, which specifies that attempting to measure a heisenbug may suppress its manifestation). This motive—of declaring war on heisenbugs—stems from Facebook's own woes and experiences in working at scale and seeing heisenbugs keep popping up. One thing that Pete Hunt mentioned, in not a flattering light at all, was a point where Facebook's advertisement system was only understood by two engineers well enough who were comfortable with modifying it. This is an example of something to avoid.

By contrast, looking at Pete Hunt's remark that he would "rather be predictable than right" is a statement that if a defectively designed lamp can catch fire and burn, his much, much rather have it catch fire and burn immediately, the same way, every single time, than at just the wrong point of the moon phase have something burn. In the first case, the lamp will fail testing while the manufacturer is testing, the problem will be noticed and addressed, and lamps will not be shipped out to the public until the defect has been property addressed. The opposite Heisenbug case is one where the lamp will spark and catch fire under just the wrong conditions, which means that a defect will not be caught until the laps have shipped and started burning customers' homes down. "Predictable" means "fail the same way, every time, if it's going to fail at all." "Right means "passes testing successfully, but we don't know whether they're safe to use [probably they aren't]." Now, he ultimately does, in fact, care about being right, but the choices that Facebook has made surrounding React stem from a realization that being predictable is a means to being right. It's not acceptable for a manufacturer to ship something that will always spark and catch fire when a consumer plugs it in. However, being predictable moves the problems to the front and the center, rather than being the occasional result of subtle, hard-to-pin-down interactions that will have unacceptable consequences in some rare circumstances. The choices in Flux and ReactJS are designed to make failures obvious and bring them to the surface, rather than them being manifested only in the nooks and crannies of a software labyrinth.

Facebook's war on the shared mutable state is illustrated in the experience that they had regarding a chat bug. The chat bug became an overarching concern for its users. One crucial moment of enlightenment for Facebook came when they announced a completely unrelated feature, and the first comment on this feature was a request to fix the chat; it got 898 likes. Also, they commented that this was one of the more polite requests. The problem was that the indicator for unread messages could have a phantom positive message count when there were no messages available. Things came to a point where people seemed not to care about what improvements or new features Facebook was adding, but just wanted them to fix the phantom message count. And they kept investigating and kept addressing edge cases, but the phantom message count kept on recurring.

The solution, besides ReactJS, was found in the flux pattern, or architecture, which is discussed in the next section. After a situation where not too many people felt comfortable making changes, all of a sudden, many more people felt comfortable making changes. These things simplified matters enough that new developers tended not to really need the ramp-up time and treatment that had previously been given. Furthermore, when there was a bug, the more experienced developers could guess with reasonable accuracy what part of the system was the culprit, and the newer developers, after working on a bug, tended to feel confident and have a general sense of how the system worked.

The Flux Architecture

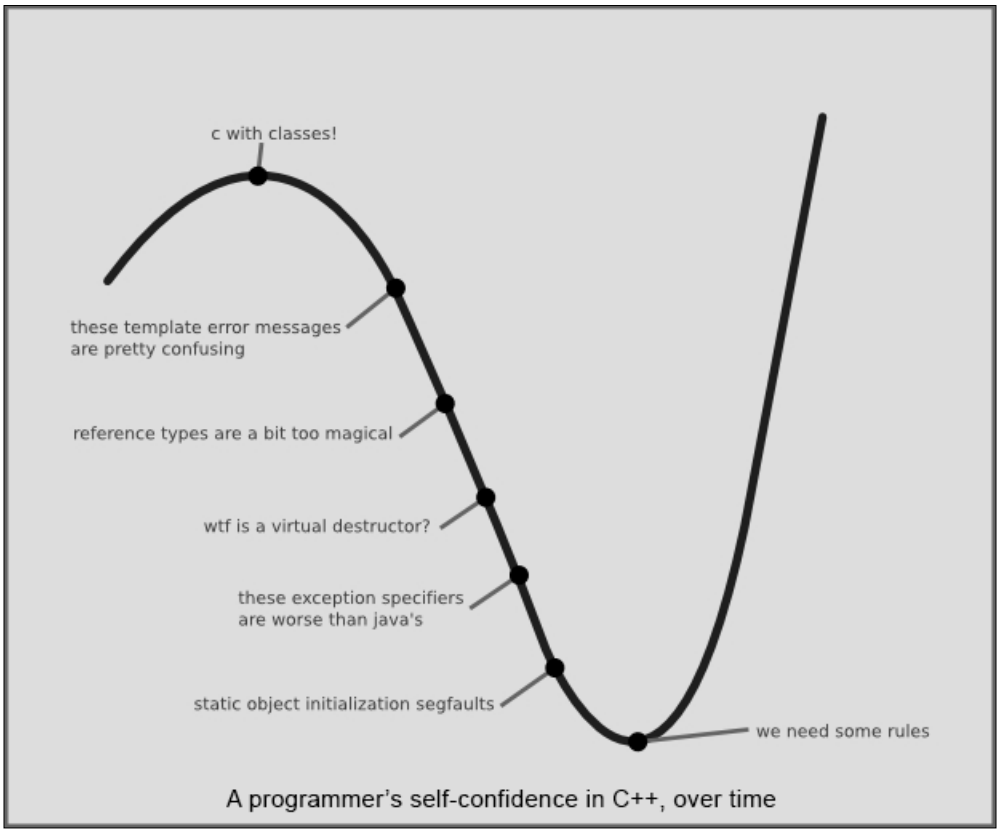
One of the ways in which Facebook, in relation to ReactJS, has declared war on heisenbugs is by declaring war on the mutable state. Flux is an architecture and a pattern, rather than a specific technology, and it can be used (or not used) with ReactJS. It is somewhat like MVC, equivalent to a loose competitor to that approach, but it is very different from a simple MVC variant and is designed to have a *pit of success* that provides unidirectional data flow like this: from the action to the dispatcher, then to the store, and finally to the view (but some people have said that these two are so different that a direct comparison between Flux and MVC, in terms of trying to identify what part of Flux corresponds to what conceptual hook in MVC, is not really that helpful). Actions are like events—they are fed into a top funnel. Dispatchers go through the funnels and can not only pass actions but also make sure that no additional actions are dispatched until the previous one has completely settled out. Stores have similarities and difference to models. They are like models in that they keep track of state. They are unlike models in that they have only getters, not setters, which stops the effect of any part of the program with access to a model being able to change anything in its setters. Stores can accept input, but in a very controlled way, and in general a store is not at the mercy of anything possessing a reference to it. A view is what displays the current output based on what is obtained from stores. Stores, compared to models in some respects, have getters but not setters. This helps foster a kind of data flow that is not at the mercy of anyone who has access to a setter. It is possible for events to be percolated as actions, but the dispatcher acts as a traffic cop and ensures that new actions are processed only after the stores are completely settled. This de-escalates the complexity considerably.

Flux simplified interactions so that Facebook developers no longer had subtle edge cases and bug that kept coming back—the chat bug was finally dead and has not come back.

From the pit of despair to the pit of success

Louis Brandy has warned about the perils of C++, which—at the risk of being controversial—has been called the biggest example of the *second system effect* (`http://tinyurl.com/reactjs-second-system`) since the OS/360 project. In a vague *XKCD*-style graphic, he states "Never trust a programmer who says he knows C++" (`http://tinyurl.com/reactjs-cpp-valley`).

The following graph shows the level of confidence of the C++ programmer:



He continues:

> *"Programmers (especially those coming from C) can very quickly get up to speed in C++ and feel quite proficient. These programmers will tell you they know C++. They are lying. As a programmer continues in C++, he goes through this valley of frustration where he comes fully to terms with the complexity of the language. The good news is that it's really easy to tell between C++ programmers pre- and post-valley (in an interview, in this case). Just mention that C++ is an extremely large and complex language, and the post-valley people will give you 127 different tiny frustrations they have with the language. The pre-valley people will say, "Yeah, I guess. I mean, it's just C with classes."*

Eric Lippert tells us something that isn't really relevant to only C++ programmers; it leads to something larger than C++:

> *I often think of C++ as my own personal Pit of Despair programming language. Unmanaged C++ makes it so easy to fall into traps. Think buffer overruns, memory leaks, double frees, mismatch between allocator and deallocator, using freed memory, umpteen dozen dozen ways to trash the stack or heap -- and those are just some of the memory issues. There are lots more "gotchas" in C++. C++ often throws you into the Pit of Despair and you have to climb your way up to the Hill of Quality. (Not to be confused with scaling the Cliffs of Insanity. That's different.)*
>
> *Now as I've said before, the design of C# is not a subtractive process. It is not "C++ with the stupid parts taken out." But that said, it would be rather foolish of us not to look at what problems people have had with other languages and work to ensure that those exact same problems do not crop up for C# users. I would like C# to be a "Pit of Quality" language, a language whose rules encourage you to write correct code in the first place. You have to work quite hard to write a buffer overrun bug into a C# program, and that's on purpose.*
>
> *I have never written a buffer overrun in C#. I have never written a bug where I accidentally shadowed a variable in another scope in C#. I have never used stack memory after the function returned in C#. I've done all those things multiple times in C++, and it's not because I'm an idiot, it's because C++ makes it easy to do all those things and C# makes it very hard. Making it easy to do good stuff is obviously goodness; thinking about how to make it hard to do bad is actually more important.*

Or, as it happened on a Python mailing list, someone with an obvious 133t hax0r spelling asked how to write a buffer overflow in Python, and one of the more senior list members answered, "We're sorry, but Python doesn't support that feature." The point of this meme is that someone asking about how to find a particular kind of vulnerability is answered by saying that Python had the basic type of defect designed out of the language. As has been pointed out for C#, strings are handled in a sane fashion where no naive use ever results in buffer override vulnerabilities.

Eric Lippert is, and remains, a pivotal figure in C#, and his post eloquently clarifies how exactly one could intelligently disagree with Bjarne Stroustrup's words:

> *"The connection between the language in which we think/program and the problems and solutions we can imagine is very close. For that reason restricting language features with the intent of eliminating programmer errors is at best dangerous."*

People who disagree with Stroustrup today might not contest both of these sentences, but might contest only the second: the connection between the language and the solutions may indeed seem to be real, but have quite the opposite implication for language features and the pit of success. And something like this may be factored into decisions such as those in the book *JavaScript: The Good Parts* by *Douglas Crockford*. This or that detail might be challenged, but the core idea of using a cherry-picked subset of JavaScript and leaving some other parts alone completely stems from seeking and retrofitting the fact that a pit of success is almost a no-brainer once the possibility has been pointed out.

All of this leads up to a point made by Rico Mariani in—sort of—the opposite of the pit of despair. The pit of success, in stark contrast to a summit, a peak, or a journey across a desert to find victory through many trials and surprises. We want our customers to simply "fall into" winning practices using our platforms and frameworks. To the extent that we make it easy to fall into trouble we fail.

Complete UI teardown and rebuild

One of Dijkstra's quotations that is a favorite with ReactJS developers such as Pete Hunt is: our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason, we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

One way in which ReactJS plays to this strength conceptually, is by blowing everything away and re-rendering afresh so that the correspondence between the program and the process is simple. You do not keep track of the present state of the DOM and 300 jQuery changes that need to be kept in mind to transition accurately from one state to another. All you need to do is tell how it should look now.
Now, in fact, ReactJS does not blow everything away under the hood; it has fairly sophisticated facilities that can make a lightning-fast pure JavaScript synthetic DOM (the same synthetic DOM that enables HTML5 features in Internet Explorer 8), and reconcile and make the fastest changes possible ("fast" in this context includes the impressive feat of managing 60-frames-per-second updates on a non-JIT iPhone 5.) However, conceptually speaking, using ReactJS means simply treating everything as if it was being blown away and redrawn from scratch, and trusting ReactJS to bring together all the pixie dust it needs to make the minimum sufficient DOM changes. This is done to update the page as requested, probably without losing the existing input or inertial scrolling.

ReactJS provides optimization hooks to offer more fine-grained control over what is rendered. These are well-documented, but they should rarely be needed. Remember Knuth's words, "Premature optimization is the root of all evil." I have myself not used this feature for optimization, although the Om bindings in ClojureScript for ReactJS are significantly faster because they need only check reference equality instead of deep equality as objects are immutable in ClojureScript, although I have made a secondary usage, to ask ReactJS to disclaim ownership of some part of the DOM so that it will play nicely with third-party functionality. See *Chapter 11*, *Demonstrating Functional Reactive Programming in JavaScript with a Live Example Part IV – Adding a Scratchpad and Putting It All Together* for an example of this in action. ReactJS is very performant by default, rather than only in how you customize its default behavior.

JavaScript as a Domain-specific Language

One widespread practice in templating systems is to provide a DSL for templating. In the usual process, such as an underscore on the frontend (`http://underscorejs.org`) or Django templating on the backend (`http://djangoproject.com`), there is a carefully chosen but deliberately underpowered templating language provided. In Django, for instance, the power is deliberately limited so that templating can be handed over to untrusted designers, and there is nothing the designers can do that will compromise anything that one would prefer not to be compromised.

This is perhaps an attractive feature, but it speaks of a limited templating language. If templating needs something more powerful, they are in a bind until any requirements they have are accommodated by nonstandard adjustments on the server side. One basic insight, associated with Gödel's incompleteness theorem and the halting problem, is that if you tie someone's hands tightly enough so that person, in principle, cannot do any damage, you have significantly limited what that person is able to do. The best results require at least a little trust. If you want people to be able to give their most useful contributions, they probably won't be able to do that with their hands bound.

In ReactJS, the DSL for templating is JavaScript, with all its power. Some people perhaps find it strange to have designers working on raw JavaScript; The ReactJS people seem to have a *give it five minutes* approach (see `http://tinyurl.com/reactjs-five-minutes`), and say that designers are smarter than what they are sometimes given credit for and are very capable of writing very specific types of JavaScript code. But this also means that if you have a special case and your complex situation calls for you to do something that was ruled out in some particular—deliberately underpowered—templating languages, it is good news. With ReactJS, you have the full power of JavaScript for use as you handle templating. With ReactJS, you can also, if you want, use a very limited subset of the language used for templating, and Pete Hunt and others seem to believe that designers are sharp enough to be able to handle it. But the much better news is when you have a difficult need that requires some real power to be available. You have as much power as JavaScript offers, and this makes quite a difference.

The Big-Coffee Notation

Steve Luscher, a ReactJS guru and enthusiast outside of Facebook who was subsequently hired by Facebook, has talked about the Big-Coffee Notation in a video on React. The basic insight is that instead of using only the big-O notation for runtime complexity (how long the runtime slows down as a function of the rough size of a problem, or occasionally other dimensions, such as memory usage), we should have a Big-Coffee Notation for what the demands scale for the poor developer, who has to keep things in their own poor, caffeinated brain.

Gerald Weinberg's classic book *The Psychology of Computer Programming* works out a basic insight at admirable length. The core insight is that *programmers programming computers* is not just an activity that involves computers. It's also an activity that involves people, and we would do well to treat it as such. Perhaps we should also know the limits of computers, but the human side of this is not, in any sense, trivial. Weinberg may have been the first to make this observation, or possibly someone else may have made it before him, but in either case, this observation has been the bedrock of serious software engineering literature ever since it was absorbed. It is a core insight, for instance, in Steve McConnell's *Code Complete: A Practical Handbook of Software Construction*. We will not be exploring this idea in full, but it is well worth exploring, especially if you haven't explored it before, and Big-Coffee Notation falls squarely in this court. The core idea is that along with tracking the big-O notation or complexity, we should also pay attention to how increasingly complex problems scale in terms of appropriate demands made by the developer. These demands are in terms of how many moving parts need to be kept in one's head.

In the big-O notation, depending somewhat on the context, there are various *runtime complexities* that provide an upper bound to how much the runtime escalates when it is given a progressively larger problem to solve. The *O(1)* runtime has a fixed upper limit for any use case. *O(log n)* is associated, for instance, with unitary operations on certain data structures. *O(n)* is also called **linear**, referring to a runtime in which there is a linear upper limit to how much time something will take to run. You are guaranteed to be at least as some constant multiplied by the number of items. *O(n log n)* may be the next major step up, and it is associated with certain sorting algorithms. *O(n ^ 2)* is called **quadratic** (all previously mentioned complexities are called subquadratic, which means faster than quadratic) and may be taken as a threshold complexity when things really do not scale to large volumes. *(O(n * (n – 1))* is also considered to be quadratic and is subsumed under *O(n ^ 2)*. After that, there are some slower polynomial times and exponentials, without closing the door to still slower escalations, such as factorials. The famous question of NP completeness is a question of whether certain NP-complete problems, which are known to be solvable in exponential time, can also always be solved in some polynomial time.

Steve Luscher's presentation about the difference between an imperative UI and a declarative UI is that if someone is making a widget to display the number of unread items in their queue, an imperative UI makes one transition between two states, making the poor programmer keep track of $☕(n*(n-1))$, meaning quadratic or $☕(n \wedge 2)$ complexity for the number of items in the poor programmer's brain compared to the number of states. If there are three states, there are six transitions. If you add a fourth state, there will be 12, or twice as many, transitions. Add a fifth state and you are looking at twenty transitions. The explanation from the programmer to understand the code is *quadratic*, meaning *steep*. However, if you give the UI code declaratively, as in ReactJS programming, you describe the code only once per possible rendered state. Three states means only three descriptions. Four states means only four descriptions. Five states means only five descriptions. It's only $☕(n)$, or linear. The escalating demands the poor programmer's caffeinated brain to keep a track of what goes on in the code, which is much less of an escalation to keep track of. Like a fast algorithm, it is much less taxing to run.

I haven't heard of Dijkstra's *The Humble Programmer* being referenced by the ReactJS community, but intellectual humility is a virtue among programmers and has been recognized as such in software engineering literature for a long time. It is emphasized in classics such as *Code Complete: A Practical Handbook of Software Construction*, and programmers are divided, not among those who have big and little minds, but those who know that they have little minds and those who have little minds but are heedless of it. Excellence in programming partly stems from recognition of one's own cognitive limits. Dijkstra writes:

> *The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. In the case of a well-known conversational programming language I have been told from various sides that as soon as a programming community is equipped with a terminal for it, a specific phenomenon occurs that even has a well-established name: it is called "the one-liners". It takes one of two different forms: one programmer places a one-line program on the desk of another and either he proudly tells what it does and adds the question "Can you code this in less symbols?" — as if this were of any conceptual relevance! — or he just asks "Guess what it does!". From this observation we must conclude that this language as a tool is an open invitation for clever tricks; and while exactly this may be the explanation for some of its appeal, viz. to those who like to show how clever they are, I am sorry, but I must regard this as one of the most damning things that can be said about a programming language…*

This challenge, namely the confrontation with the programming task, has already taught us a few lessons, and the one I have chosen to stress in this talk is the following:

> *We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as very humble programmers.*

Summary

We just took a whirlwind tour of some of the theory surrounding reactive programming with ReactJS. This includes declarative programming, one of the selling points of ReactJS that offers something easier to work with at the end than imperative programming. The war on heisenbugs, is an overriding concern surrounding decisions made by Facebook, including ReactJS. This takes place through Facebook's declared war on the shared mutable state. The Flux Architecture is used by Facebook with ReactJS to avoid some nasty classes of bugs. The pit of success and the pit of despair, which learns from others' pain, a pain that is concentrated in connection to the C++ programming language, and looks at what we should be aiming for.

We covered complete UI teardowns and rebuilds, providing a simple alternative to keeping track of the state in how to update an interface. We also saw JavaScript as a DSL, looking at an intentional decision in designing ReactJS that is meant to give you as much power as you need. Then the Big-Coffee Notation was discussed in relation to a healthy recognition of one's own limits instead of them breaking their shins, which is preventable.

In our next chapter, we will continue by looking at a concrete case of a user interface built with ReactJS.

4

Demonstrating Nonfunctional Reactive Programming – A Live Example

In this chapter, we will be looking at a live example that incorporates some principles of reactive programming with ReactJS. Some portions of the program remain imperative as a port of something that was previously written in jQuery after other ports from HP-28S RPN and Unix C, but ReactJS's power still shines through, even with code that, like much of the code in the real world, has gone through multiple iterations. We will briefly look at the HTML requirements of the web page, before going through the real meat in JavaScript. The web page offers a port of a video game originally developed on an HP-28S graphing scientific calculator, and retains much of the calculator's look and feel.

In this chapter, we will be covering these topics:

- The HTML for the web page
- The JavaScript that animates the page

Here we see the game, as rendered against the background of a classic HP28S calculator. Effort has been taken so that character graphics mimic the dark and light colors as existed on the LED screen:



The history of a game with multiple ports

The title specifies that we are making a port of an HP28S RPN game, so let's look at a bit of history of the specific game that we are implementing.

This game has had different implementations and different ports, including reimplementation in C, and a couple of ways with HTML or JavaScript. The original version was on an HP28S, a hackerly scientific calculator that could have 32 KB or a whopping 512 KB of RAM (mine had 512 KB). Programming and use (the two are not terribly different, as in Unix/Linux shell programming) **Reverse Polish Notation** (**RPN**) (`http://en.wikipedia.org/wiki/Reverse_Polish_notation`). There were a lot of interesting depths in the calculator as far as hacking was concerned, and I made two programs that I specifically remember. One was a fractal screen saver with the humble two-dimensional *staggering drunk* algorithm (see `http://tinyurl.com/reactjs-staggering-drunk`), and the other was the video game that will be reimplemented here.

The basic game, implemented with dingbat character graphics, had a spaceship moving from left to right, in a field with asteroids that grew denser from level to level. The primary game mechanics was intended to be the dodging of asteroids as you went through. You could shoot asteroids as well. This was really necessary because some of the (naïvely) randomly drawn levels did not necessarily have a clear route available. To discourage the game mechanics of simply shooting all the way through each level, shooting asteroids was penalized, and intended as more of a last resort than the maneuvering that intended as the primary game mechanics. A friend of mine commented that it was the first video game he knew where the player actually lost points for shooting things.

The HTML for the web page

We open with a standard HTML5 DOCTYPE:

```
<!DOCTYPE html>
```

Following this, we open the document, specifying UTF-8 as the charset. If the web page is served correctly, the charset should be specified with the page download, but this is still potentially helpful in terms of defensive coding, which is always something to keep in mind:

```
<html lang="en">
  <head>
    <meta charset="utf-8" />
```

Hence the document title:

```
    <title>A video game on an HP-28S scientific
      calculator</title>
```

The font used here is a retro VT series font, associated with the venerable VT100 and other series of Unix terminals. Note—as will be seen in code later—that while the VT100 series was a monospaced terminal, the font is not strictly a monospaced font and simply displaying each row of spaces or asteroids inline will have undesired spacing, so each character is positioned absolutely. Perhaps another font may not have this problem, but there's a nice retro tint to the VT100 font.

Note that we will be including dingbats for much of the character graphics. They are dealt within the JavaScript.

The font tags, like some other tags used on the HTML side, are written via the HTTP/HTTPS ambidextrous format of two leading slashes, with `http:` or `https:` not specified, and being supplied to be the same as in the web page:

```
<link
  href='//fonts.googleapis.com/css?family=VT323'
   rel='stylesheet' type='text/css' />
```

Using a content distribution network wherever we can

We load ReactJS from a **content distribution network** (**CDN**), following Steve Souders' widely established Yslow ("Why is my web page slow to load?") recommendation.



Steve Souders (`http://SteveSouders.com`) pioneered a realization, initially just at Yahoo!, that rendering web pages that would be working more quickly wasn't really about shaving off milliseconds or microseconds of server-side performance. There was a significant low-hanging fruit in terms of influencing the client to be more performant, for instance, by not loading the same resource from the network again and again when it could be loaded with lightning speed from a computer's cache.

There are quite a lot of JavaScript libraries and frameworks available from a CDN hit, including ReactJS, but there are also almost any other major or minor JavaScript tool you would want to use.

Some simple styling

We give the page some basic styling. The background image is loaded from `http://haywardfamily.org/hp28s.png`. You can make a local copy if you wish, or if there are issues over HTTPS or if you are serving up the files locally where the HTTP/HTTPS ambidextrous hack sadly fails to work.

The color for the text in **p#display** is taken from a screenshot of an HP28S calculator:




```
<style type="text/css">
  body
  {
  background-image:
  url(//haywardfamily.org/hp28s.png);
  background-position: top left;
  background-repeat: no-repeat;
  height: 670px;
  width: 860px;
  }
  div#main
  {
  height: 670px;
  width: 860px;
  }
```

```
    p#display
    {
    color: #4f5c65;
    font-family: VT323, courier, sans;
    font-size: 18px;
    letter-spacing: 4px;
    left: 565px;
    top: 180px;
    position: absolute;
    }
    p#legend
    {
    background-color: rgba(0, 0, 0, .6);
    border-radius: 20px;
    color: white;
    font-family: Verdana, arial, sans;
    margin-left: 40px;
    margin-right: 90px;
    margin-top: 40px;
    padding: 10px;
    }
  </style>
  </head>
```

This is the last part of the page header.

A fairly minimal page body

We build the page body, which has an image of an HP-28S calculator as the background. We also include a brief legend and space for the game to display on the virtual calculator screen:

```
  <body>
    <div id="main">
      <p id="legend">
        Arrow keys to move, Space to shoot.
      </p>
      <p id="display">
      </p>
    </div>
```

Before closing the body tag, we load the main script, which will use ReactJS to animate the game:

```
  <body>
    <script
      src="//cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react.js">
```

```
</script>
  <script type="text/javascript"
    src="hp28s.js"></script>
</body>
</html>
```

This is the end of the HTML for the page. The JavaScript, which contains the real meat as far as programming is concerned, will now follow.

The JavaScript that animates that page

We might briefly note that the script is regular JavaScript, rather than ReactJS's JSX format, which allows mixing of HTML-like XML and JavaScript and has been called the tool that puts angle brackets in your scripts. Not all people will go with JSX, but the tool is worth knowing about if nothing else.

JSX has many merits and is worth considering. It is used by perhaps some, but not all non-Facebook ReactJS users, as well as by Facebook. Facebook has been careful to support JSX but has not required it to use ReactJS. For development purposes, JSX scripts can be loaded after a web page loads, from `http://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js` for example, and compiled in the browser. For production purposes, they need to be compiled in JavaScript, in which case you can run `npm install jsx` and then run the `jsx` compiler from the command line, as documented at `https://www.npmjs.com/package/jsx`.

A brief syntax note – Immediately Invoked Function Expression

In our script, we use an **Immediately Invoked Function Expression** (**IIFE**) so that our local variables, defined with the `var` keyword somewhere within the function or its dependencies, will be protected as private within a closure. This will avoid the concern of a shared mutable state (as it has a nonshared mutable state). A shared mutable state leaves the program's stability at the mercy of anyone who has sufficient access to modify the state. The function is wrapped in parentheses due to a quirk of JavaScript syntax, wherein a line beginning with a function is considered a function definition, so the following syntax will not work:

```
function() {
}();
```

The solution is to put the function within an enclosing pair of parentheses, and then it will work properly:

```
(function()
    {
    })();
```

Back to our main script!

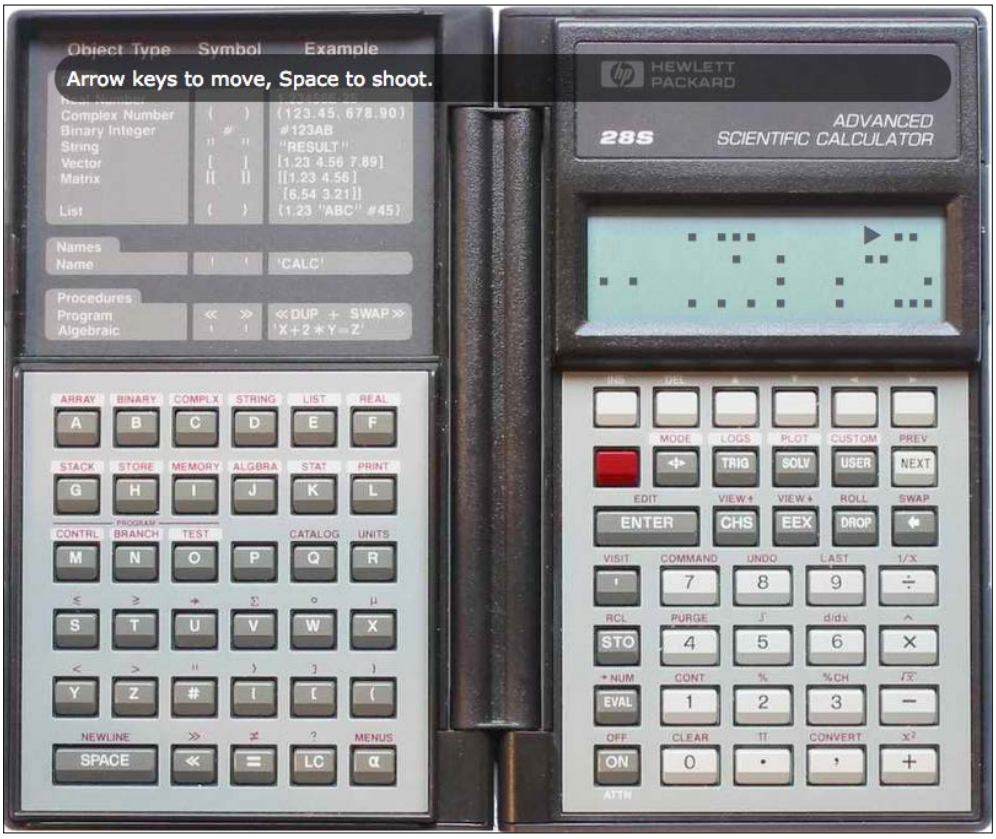
Variable declaration and initialization

Our main `wrapper` function begins nonreactively and imperatively by writing state variables that are only used in the function:

```
function()
{
  var game_over_timeout = 2000;
  var game_over_timestamp = 0;
  var height = 4;
  var tick_started = false;
  var width = 23;
  var chance_clear;
  var game_over;
  var level;
  var rows;
  var score;
  var position;
  var row;
```

Having declared—and in some cases, initialized—these variables, we move on to the function used to start the game. This initializes or reinitializes variables, but does not include initializing the level. It starts the game by setting the `game_over` variable to `false`, placing the player on level 1, setting a (horizontal) position at the beginning, over to the left of the screen/asteroid field in the first of the 23 characters' width, and a vertical position in row 1 (the row below the top, out of 4), a score of 0, and the chance of most spaces being clear (that is, free of asteroids and therefore safe for the player's ship to travel in) by the asteroids keep getting denser! This 5/6 is the start of an exponential decay in the chances of a space being asteroid-free each time the player goes to a new level. The latter is a parameter that can be adjusted to influence the overall difficulty of the game, a lower value making a more difficult-to-navigate field. Elsewhere, between levels, the chance that a space is clear decreases exponentially; the rate of exponential decay, or perhaps other aspects of that feature are also something that could be modified to affect the difficulty of the game between levels.

Here we see what the display looks like when the player has almost cleared the first level:



The levels that are generated have mostly spaces, with a random chance of an asteroid being present, but the space that holds the spaceship initially and the space in front of it are always clear. This is intended to allow the player to get some space to react, instead of an automatic death.

The function used to start or restart the game

The function is invoked immediately after it is declared:

```
var start_game = function()
{
  game_over = false;
  level = 1;
  position = 0;
  row = 1;
```

```
    score = 0;
    chance_clear = 5 / 6;
  }
  start_game();
```

The function that creates game levels

In the `get_level()` function, the level is built. The probability that a space is clear undergoes exponential decay (the first level, after decay, being .75, then .675, then .6075, and so on), with the density of asteroids increasing correspondingly, and then a rectangular *array of arrays of characters* is built (arrays of characters are used for collections of characters undergoing near-constant changes, rather than strings, which are immutable, even though the original implementation manipulated strings). Note that in internal representations here, things are represented by character codes: `a` for an asteroid, `s` for the player's ship, a space for a blank space, and so on. (Now it may be a bit strange to store an array of arrays of characters that are references to other characters. On the original legacy system, the approach that would now be obvious was not yet available. This could well be refactored out now, but this chapter is the one code chapter intended to resemble what it's like to get good results while dealing legacy code, and this wart's presence is intended. Most of the work most developers do includes interfacing with legacy functionality.) Initially, all the spaces are potentially populated with asteroids. After that, the ship's initial slot, and the space immediately in front of it, are cleared. This is an example:

```
var get_level = function(){
  level += 1;
  rows = [];
  result = {};
  chance_clear *= (9 / 10);
  for(var outer = 0; outer < height; ++outer)
  {
    rows.push([]);
    for(var inner = 0; inner < width; ++inner)
    {
      if (Math.random() > chance_clear)
      {
        rows[outer].push('a');
      }
      else
      {
        rows[outer].push(' ');
      }
    }
```

```
    }
    rows[1][0] = 's';
    rows[1][1] = ' ';
    return rows;
  }
```

Although this function returns a grid of rows, the grid of rows will be assigned to be a field of an object that will be used with ReactJS. ReactJS works better with a property on an object than on an array.

The result of the preceding function call is stored in a field of the board variable, and an array is defined for keystrokes. At the end of a move, the last keystroke (if any) is taken from the `keystrokes` array, and then the array is emptied so that the ship moves according to the last keystroke (if any) entered during a turn. All other keystrokes are ignored:

```
var board = {rows: get_level()};
var keystrokes = [];
```

Getting our hands dirty with ReactJS classes

Now we will directly start interacting with ReactJS. We create a ReactJS class, working on a hash of functions with specific fields named as per the documentation. The `componentDidMount()` function and field, for instance, is a function invoked when a ReactJS component mounts. This means that more or less, it is displayed and represented in the DOM. In this case, we add event listeners to the document's body, rather than add something directly to the component in ReactJS. This is because we want to listen to key press/key down events, and it is difficult to get `DIV` to respond to these events. Therefore, the body has event listeners added. They will address event handlers in ReactJS, which are still meant to be defined as you would display them normally. Note that some other types of events, such as some mouse events (at least), will register through ReactJS's usual means, as introduced here:

```
var DisplayGrid = React.createClass({
  componentDidMount: function()
  {
    document.body.addEventListener("keypress",
    this.onKeyPress);
    document.body.addEventListener("keydown",
    this.onKeyDown);
  },
```

Components in ReactJS have **properties** and a **state**. Properties are something defined once and they cannot be changed. They are available from within ReactJS, and they should be treated as immutable. The state is mutable information. Both properties and the state are available within ReactJS. We can briefly comment that the shared mutable state is rightly treated by Facebook and ReactJS as begging for Heisenbugs. Here, we handle all of the mutable state from within a closure. The mutable state is not shared, nor should it be shared (in other words, it is a nonshared mutable state):

```
getDefaultProps: function()
{
  return null;
},
getInitialState: function()
{
  return board;
},
```

We next define the key down and key press event handlers as they would normally be used, as or at least as they would normally be handled if DIVs responded to key events. (We will in fact be monitoring the body, because key-related events, unlike a hover or mouse click, does not propagate to a containing DIV. This approximates how you would normally demonstrate event handling with ReactJS. The specific keys we are listening for, arrow keys and the spacebar, present a problem. In essence, arrow keys trigger key down events, but not key press events (most other keys generate key press events). This behavior is stupid, but it is deeply entrenched in JavaScript and is now more or less non-negotiable. We delegate to a common event handler that will handle both events. Here, key presses are converted to key codes: left or up arrow key to go up (or left, from the orientation of the game), right or down arrow key to go down (or right, from the orientation of the game), and the spacebar to shoot. These are represented in the `keystrokes` array by `u`, `d`, and `s` respectively:

```
onKeyDown: function(eventObject)
{
  this.onKeyPress(eventObject);
},
onKeyPress: function(eventObject)
{
  if (eventObject.which === 37 ||
  eventObject.which === 38)
  {
    keystrokes.push('u');
  }
```

```
        else if (eventObject.which === 39 ||
        eventObject.which === 40)
        {
          keystrokes.push('d');
        }
        else if (eventObject.which === 32)
        {
          keystrokes.push('s');
        }
      },
```

At this point, we create the `render()` function, which is a core ReactJS member to define. What this render function does is create DIVs and SPANs that represent the grid of spaces and symbols as appropriate. The leaf nodes are positioned absolutely.

Having built the leaf SPAN nodes and then the intermediate DIVs, we build up to the main DIV element.

The `out_symbol` variable is a UTF-8 character and not an ASCII escape; this is for a very specific reason. ReactJS, although with a clearly documented escape hatch `dangerouslySetInnerHTML()` (see `http://tinyurl.com/reactjs-inner-html`), is normally set up to resist XSS (cross-site scripting) attacks. As such, its normal behavior is to escape angle brackets and much ampersand usage. This means that ` ` will be rendered as it is in source, ` ` rather than with a (non-breaking and non-collapsing) space. Therefore, the dingbat symbols that we use are not given as they might be elsewhere—with escape codes (though those are left in here in comments)—but symbols pasted in the JavaScript that are stored as UTF-8.

If you're not sure how to enter the dingbats, you can simply use something else. Alternatively, you can take the escapes in the comments, copy them to a **Plain Old Simple HTML** (**POSH**) file, and then copy and paste the half dozen symbols from the rendered POSH page in your JavaScript source with your editor. Your JavaScript source should be treated as UTF-8:

```
render: function()
{
var children = ['div', {}];
for(var outer = 0; outer <
this.state.rows.length; outer += 1)
{
  var subchildren = ['div', null];
  for(var inner = 0; inner <
  this.state.rows[outer].length;
  inner += 1)
  {
```

```
        (var symbol =
        this.state.rows[outer][inner];)
        var out_symbol;
        if (symbol === 'a')
          {
            // out_symbol = '&#9632;';
            out_symbol = '■';
          }
          else if (symbol === 's')
          {
            // out_symbol = '&#9658;';
            out_symbol = '►';
          }
          else if (symbol === ' ')
          {
            // out_symbol = ' ';
            out_symbol = ' ';
          }
          else if (symbol === '-')
          {
            out_symbol = '-';
          }
          else if (symbol === '*')
          {
          out_symbol = '*';
          }
          else
          {
            console.log('Missed character: '
            + symbol);
          }
          subchildren.push(
          React.createElement('span',
          {'style': {'position': 'absolute',
            'top': 18 * outer - 20), 'left':
            (12 * inner - 75)}}, out_symbol));
          }
          children.push(
          React.createElement.apply(this,
          subchildren));
        }
        return React.createElement.apply(this,
        children);
      }
    });
```

The children and subchildren defined in the preceding code populate the argument lists for `React.createElement()`.

Having built things in an inner loop, we add a leaf node to the `subchildren` array. It is specified to be a span, with inline CSS styling delivered in a hash and content equal to the `out_symbol` variable. This in turn is added to the `children` array. It contains the rows of the screen, which in turn is built into the complete board.

In ReactJS, components are defined in `React.createElement()` and subsequently made available for use. The usual invocation for `React.createElement()` is something like `React.createElement( 'div', null, ...)`, where the part represented by the ellipsis holds all the children. We use `apply()` to call `React.createElement()` with the initial Arg that is requested, and then, in the array, what the arguments should be.

Tick-tock, tick-tock – the game's clock ticks

This closes the `render()` field and the `React.createElement()` class definition. In the source code, we move on to the `tick()` function. It handles whatever is supposed happen at each turn. As of now, the code calls `tick()` at an interval of 300 milliseconds (0.3 seconds), although this is something that could be tweaked to influence the gameplay and perhaps refactored slightly so that the gameplay would accelerate with higher levels.

If the game is over, which can only be due to the ship crashing into an asteroid, nothing happens in the `tick()` call:

```
var tick = function()
{
  if (game_over)
  {
    return;
  }
```

Next, `React.render()` is called, specifying the class that is being rendered and what HTML element to render to. `React.render()` should be called at least each time you want things to render. If you call it once, it will render once, which means that it needs to be called repeatedly if you want repeated updates to show up. Here, we call it inside each call of the `tick()` method, asking to create an element based on `DisplayGrid` defined in a good chunk of the preceding code, and to put the rendered HTML into the DIV with the display ID:

```
React.render(
  React.createElement(DisplayGrid, {}),
  document.getElementById('display'));
```

Here we see the screen as the player shoots an asteroid. The asteroid explodes into an asterisk!



If, in the previous turn, the ship shot an asteroid (which will be represented in the character symbols as zero or more hyphens and an asterisk to their right; the hyphens fill the space leading up to the asteroid that has been shot, and the asterisk represents the explosion as a shot hits the asteroid), we clear the slots that show the shot taken in that turn:

```
for(var outer = 0; outer < height; outer += 1)
{
  for(var inner = 0; inner < width; inner += 1)
  {
    if (board.rows[outer][inner] === '-' ||
    board.rows[outer][inner] === '*')
    {
```

```
        board.rows[outer][inner] = ' ';
      }
    }
  }
```

Having done this, we clear the variable indicating that a shot has been taken:

```
var shot_taken = false;
```

We clear the space where the ship was:

```
board.rows[row][position - 1] = ' ';
```

The `keystrokes` array is cleared at the end of each tick, and we pay attention to the last keystroke stored. In other words, we are attending to the last keystroke, if any, since the previous turn. Keystrokes do not accumulate between turns. At the end of a turn, the last keystroke is the only keystroke that wins.

The `keystroke` array stores key codes rather than exact keystrokes. Arrow keys have been processed such that a left or up arrow key press will store a `u` for up, a right or down arrow key press will store a `d` for down, and a spacebar press will store an `s` for shoot. If someone enters up or down, the ship moves up or down respectively within bounds:

```
if (keystrokes.length)
{
  var move = keystrokes[keystrokes.length - 1];
  if (move === 'u')
  {
    row -= 1;
    if (row < 0)
    {
      row = 0;
    }
  }
  else if (move === 'd')
  {
    row += 1;
    if (row > height - 1)
    {
      row = height - 1;
    }
  }
```

If the user shoots an asteroid, in the next turn, a row of hyphens will extend from the front of the ship to the asteroid, and the asteroid will become an asterisk, representing an explosion:

```
    else if (move === 's')
    {
      shot_taken = true;
      score -= 1;
      var asteroid_found = false;
      for(var index = position + 1; index <
      width && !asteroid_found; index += 1)
      {
        if (board.rows[row][index] === 'a')
        {
          board.rows[row][index] = '*';
          asteroid_found = true;
        }
        else
        {
          board.rows[row][index] = '-';
        }
      }
    }
    keystrokes = [];
  }
```



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.

GAME OVER

If the user runs over, or collides with an asteroid, the game is over. Then we display a **GAME OVER** screen and stop further processing, like this:




```
if (position < width)
    {
    if (rows[row][position] === 'a')
        {
        game_over = true;
        game_over_timestamp = (new
          Date()).getTime();
        (document.getElementById(
          'display').innerHTML =
          '<span style="font-size: larger;">' +
          'GAME OVER' +
          '<br />' +
          'SCORE: ' + score + '</span>');
        return;
        }
```

As long as the user hasn't hit an asteroid, the game is still going on, and we replace the present slot in the row with a marker for the ship. Then we increment the player's (horizontal) position:

```
            board.rows[row] = board.rows[row].slice(0,
              position).concat(['s']).concat(
              board.rows[row].slice(position + 1));
            position += 1;
            }
```

If the user has "fallen off the right edge of the screen", we take the game to the next level:

```
        else
            {
            rows = get_level();
            score += level * level;
            position = 0;
            row = 1;
            }
        }
```

Having defined all of this, we start the game, and if we haven't started the ticks in motion, we do so with an interval of 300 milliseconds (a value that can be played with to make the game easier or harder; it might be made into a configurable interval that speeds up with the gameplay):

```
start_game();
if (!tick_started)
    {
    setInterval(tick, 300);
    tick_started = true;
    }
})();
```

Summary

We covered a lot of ground in this chapter. Earlier, we had covered some theory, but it was here that we began to piece together an application using some of ReactJS. Later on, we will work with a more in-depth application.

The topics covered included the HTML for the web page. This is a simple HTML skeleton, that serves as a rack to hold reactive JavaScript. Another topic covered was the reactive JavaScript. This includes a mix of JavaScript with a clear example that shows how to write reactive JavaScript for ReactJS.

We will continue with further coverage of functional programming in the next chapter.

5
Learning Functional Programming – The Basics

JavaScript is a multiparadigm language that is not perfect for any paradigm it touches, but it has interesting features for its main paradigms. It is an object-oriented language, although the definition of object-oriented varies between object-oriented languages. It has been suggested that its prototypal inheritance may be less significant for object-oriented programming than a demonstration of how to create class-free objects instead of fumbling at the difficult task of getting the taxonomy right from the beginning. The definition of object-oriented also varies between multiparadigm languages with object-oriented features. For example, Python dynamically allows members to be added to existing objects, while Java requires members to be defined in a class. The object-oriented characteristics of JavaScript are useful and interesting, but especially in the past few years, they have been a source of frustration to programmers of other object-oriented languages, who have been told that JavaScript is object-oriented without sufficient information on how JavaScript is object oriented through an approach that differs profoundly from other major languages.

Likewise, for functional programming, JavaScript has functional programming support, or at least some of it. But like JavaScript as a whole, functional JavaScript is not 100 percent in line with the *The Good Parts*. One common, although not universal, feature of functional programming languages is tail call optimization, which says that recursive functions that only recur at the end are internally converted into a more commonplace style of loop that is faster and can go very deep without exhausting its quota of call stack space. This optimization is slated for ECMAScript 6, but at the time of writing this book, it has not been implemented in common browsers, which provide not only slower performance but also a limit of around 10,000 to 20,000 calls deep in recursion.

There's a lot that can be done within this limit, but structured program writers would not be happy if their `for` loop was not implemented to go much beyond 20,000 iterations. The point here is not to specify which solution is best for JavaScript not always supporting tail call optimization, but to indicate that this difficulty is presently here, and this is one of the few ways in which JavaScript does not directly support standard functional language features (summing all integers from 1 to 1,000,000 or higher is especially not interesting to do, but it serves as a standard example in tutorials).

Literature is divided over whether or not JavaScript should be called a functional language; it is certainly not a pure functional language as Haskell is (but then, neither is OCaml). JavaScript has been called after a known functional language—Scheme in C's clothing—and its basic functional features are not something tacked on after the fact. Perhaps this may reflect his preferences, but Douglas Crockford, who is willing to be both critical and picky in his judgments of what parts of the JavaScript language are good ideas, has never that I have seen picked functional aspects present in JavaScript for targets of a scathing critique. In his move from *JavaScript: The Good Parts* to *The Better Parts*, one of his preferences given ECMAScript 6 is to stop using `for` and `while` loops in an imperative style, and use recursion that takes advantage of tail call optimization, which is to be included. Maybe the strongest claim that JavaScript has a functional heart can be seen in the question of what feature in a language is central. It has been suggested that in Java, the central feature is objects. In C, it is pointers. In JavaScript, it is functions.

JavaScript's functions have first-class status, meaning among other things that (higher order) functions can act on other functions and be passed as arguments, or even be dynamically constructed and returned as a result.

In this chapter, we will be covering:

- Custom sort functions
- Map, reduce, and filter
- Fool's gold – altering the behavior of other people's prototypes
- Closures and information hiding

Custom sort functions – the first example of functional JavaScript and first-class functions

To break the ice, let's look at sorting JavaScript's arrays. JavaScript's arrays have a built-in `sort()` function, which is, at the very least, a sensible default. If, for instance, we create an array with the first six digits of π, we can sort it:

```
var digits = [3, 1, 4, 1, 5, 9];
digits.sort();
console.log(digits);
```

Chrome's debugger shows an array on the console, which we can access:

```
Array[6]
   0: 1
   1: 1
   2: 3
   3: 4
   4: 5
   5: 9
   length: 6
     __proto__: Array[0]
```

This is well and good. Let's stretch things a little further and try mixing integers and floating-point decimals (floats). Note that in JavaScript, there is one numeric type that acts like an integer (and remains integer clean) for the integers between (in Firefox) $-(2^{53} - 1)$, or -9007199254740991, and $2^{53} - 1$, or 9007199254740991. This numeric type also stores floating-point numbers. These have a greater range and, of course, more fine-grained values for smaller numbers. To stretch things further, let's have an array with integers and floats mixed together:

```
var mixed_numbers = [3, Math.PI, 1, Math.E, 4, Math.sqrt(2), 1,
  Math.sqrt(3), 5, Math.sqrt(5), 9];
```

Among these numbers, `Math.PI` is around 3.14, `Math.E` is around 2.72, `Math.sqrt(2)` is around 1.41, `Math.sqrt(3)` is around 1.73, and `Math.sqrt(5)` is around 2.24. Let's sort this like the other array and log the values:

```
[1, 1, 1.4142135623730951, 1.7320508075688772, 2.23606797749979,
  2.718281828459045, 3, 3.141592653589793, 4, 5, 9]
```

Chrome's debugger, for some reason, behaved differently this time, displaying all of the array in a string rather than an array with a drill-down triangle to the left. However, the array is sorted correctly, with all the values in ascending order and integers and floating-point values displayed correctly.

Let's try this out on strings. Suppose we have the following array:

```
var fruits = ['apple', 'durian', 'banana', 'canteloupe'];
```

When we sort it, we get this:

```
["apple", "banana", "canteloupe", "durian"]
```

This is in order, and good. Let's add a bit in the middle of the array:

```
var words = ['apple', 'durian', 'Alpha', 'Bravo', 'Charlie',
  'Delta', 'banana', 'canteloupe'];
```

We sort it and get back the following:

```
["Alpha", "Bravo", "Charlie", "Delta", "apple", "banana",
  "canteloupe", "durian"]
```

What is this? All the new words are at the beginning, and all the old words are at the end! Sorted among themselves perhaps, but segregated by capitalization.

The reason for this is that string sorting is the dictionary order by Unicode values, which is the same as ASCII encoding for characters that are part of ASCII. In ASCII, all uppercase letters come before all lowercase letters. Here, uppercase letters are sorted correctly within uppercase and lowercase letters are sorted correctly within lowercase, but both of these are segregated. If we want all of the `A` to come before all the `B`, we need to be more specific about what we want.

One way we can do this is by providing a comparator function—something that will compare two elements and tell `Array.sort()` which one should go first. Let's make a case-insensitive sort just for these words:

```
var case_insensitive_comparison = function(first, second) {
  if (first.toLowerCase() < second.toLowerCase()) {
    return -1;
  } else if (first.toLowerCase() > second.toLowerCase()) {
    return 1;
  } else {
    return 0;
  }
}
```

Then we sort the array and specify the comparing function:

```
words.sort(case_insensitive_comparison);
```

When we log the sorted array, we see a case-insensitive alphabetical order:

```
["Alpha", "apple", "banana", "Bravo", "canteloupe", "Charlie",
  "Delta", "durian"]
```

What if we want capitalization to serve as a tiebreaker and an uppercase character will be placed before its lowercase equivalent? This is a straightforward modification of our comparator:

```
var mostly_case_insensitive_comparison = function(first, second) {
  if (first.toLowerCase() < second.toLowerCase()) {
    return -1;
  } else if (first.toLowerCase() > second.toLowerCase()) {
    return 1;
  } else {
    if (first < second) {
      return -1;
    } else if (second < first) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

Let's add `'ALPHA'` and `'alpha'` to the end of our list of strings and re-sort:

```
["ALPHA", "Alpha", "alpha", "apple", "banana", "Bravo",
  "canteloupe", "Charlie", "Delta", "durian"]
```

It works!

This may or may not be needed with mere string comparisons, but what if the server has run a database query and packaged the results in JSON for us? The result, once parsed on the client, will probably be an array of objects that have the same structure. Electronic customer contact information might include the following, among other things:

```
{
  "email": {
    "personal": "jsmith@gmail.com",
    "work": "john.smith@company.com"
  },
```

```
    "name": {
      "first": "John",
      "last": "Smith"
    },
    "skype": {
      "personal": "JohnASmith",
      "work": "JASCompany"
    }
}
```

This record structure may not be one that JavaScript innately infers how we would like to see it sorted, but this does not really hurt us. If we build a comparator function, it has full access to fields or other details of both the items that it is asked to compare. This means that we can compare one field, then another, and then a third. It also means that we can compare by different criteria if we want to; at one point, we could compare by name, and another point we might compare by geographical location. If our server stores (or looks up) GPS coordinates for addresses, we can search by who is closest to a particular location.

This leads us to array.filter()

In functional languages, features such as map, reduce, and filter are staples of everyday use. They operate on lists, which in more functional, list-centric languages may be finite or infinite. In this sense, a list may be more like a JavaScript array or a generator, a kind of function that, instead of returning a single value, yields zero or more values and may theoretically yield an infinite number of values. Unlike an array, there is no finite point where any given generator may be exhausted, even though one never actually produces an infinite number of values. Generators are a wonderful feature, but they are dodgily supported in browsers at the time of writing this book, which means that our use of map, reduce, and filter will more likely be on (finite) arrays than on generators.

But before we drop the topic of generators, let's give two examples of generators, both of which will overflow before too long, but both of which serve as examples of what, in a language like Haskell, might be considered an infinite list with all the members of a mathematical series, as opposed to an array containing only and exactly the first *n* members. We will look at generators for powers of 2 and Fibonacci numbers, using the ECMA6-proposed syntax for generators, as discussed at `http://wiki.ecmascript.org/doku.php?id=harmony:generators`:

```
function* powers_of_two_generator() {
  var power = 1;
  while (true) {
    yield power;
    power *= 2;
```

```
  }
}

function* fibonacci_generator() {
  var first = 0;
  var second = 1;
  var sum;
  yield second;
  while (true) {
    sum = first + second;
    yield sum;
    first = second;
    second = sum;
  }
}
```

Contrast these examples with generators with standard, recursive approaches to compute the n^{th} power of 2 (this is not really needed, because JavaScript's arithmetic handles exponents, but is included for the sake of completeness), and a naïve implementation of computing the n^{th} Fibonacci number. Both are what is called tail recursive, and would stand to benefit from tail call optimization if and when such a thing is made available from browsers:

```
function power_of_two_recursive(n) {
  if (n === 0) {
    return 1;
  } else {
    return 2 * power_of_two_recursive(n - 1);
  }
}

function fibonacci_recursive(n) {
  if (n === 0 || n === 1) {
    return 1;
  } else {
    return (fibonacci_recursive(n - 2) +
      fibonacci_recursive(n - 1));
  }
}
```

Both of these functions assume a nonnegative integer as the argument. The second function also has terrible performance characteristics, even though memory usage is not particularly bad. However, the number of function calls is comparable to the value returned, so calculating the 100th Fibonacci number, besides any concerns about integer overflow, could take longer than the age of the Universe. As Donald Knuth said, "Premature optimization is the root of all evil," but this is one case where optimization is not premature.

Note that another feature of functional programming, called **memoization**—which means retaining the results of the intermediate calculations, instead of repeatedly regenerating them from scratch—avoids this performance bottleneck altogether. Consider the following memoization of the recursive Fibonacci function:

```
var calculated_fibonacci_numbers = [];

function fibonacci_memoized(n) {
  if (calculated_fibonacci_numbers.length > n) {
    return calculated_fibonacci_numbers[n];
  } else {
    if (n === 0 || n === 1) {
      result = 1;
    } else {
      result = (fibonacci_memoized(n - 2) +
      fibonacci_memoized(n - 1));
    }
    calculated_fibonacci_numbers[n] = result;
    return result;
  }
}
```

Illusionism, map, reduce, and filter

As a child, I was very interested in illusionism, and I still have an illusionist's set—which has (or had) things such as a fake thumb and a trick cup—and books on illusions. One of the tricks I remember was putting a rope around one's thighs and looping around the hands. The rope appears tight if one's legs are relaxed, but much looser if one lifts one's legs a bit, thus giving the impression that one is tied securely, while it is straightforward to free one or both hands.

I was never too good at the showmanship aspect of amateur illusionism, which is really the core of the craft. Senior illusionists, when advising their juniors or aspirants, are apt to say things such as, "Entertain the audience and deceive them, but know which comes first." And I remember for a long time thinking that I did not know (the technical side of) real illusionism, because I knew technically how to do several tricks, but I did not see how one would approach doing the things I saw.

Much later, there was an illusionist at my company's party, and I was fascinated for an unusual reason. He did some tricks that were novel to me, but for about 70 percent or 80 percent of the time, he spent making significant mileage out of the rope tricks I learned as a kid. And it worked very well. He had exquisite showmanship, and my fascination was not about wondering how he technically pulled off the trick, but at such an adept entertainer who could take two tricks that a child could do and mine them for amusement value.

Map, reduce, and filter (here, "reduce" includes both the right and left folds) are somewhat like this for functional programming. Map takes it and applies it to all members of a list. Reduce takes an operation and, starting from the right or left, applies it to every member with an intermediate result. Filter takes a function and a list and creates a new list consisting of exactly those items for which the function is true. These concepts will be explained and further illustrated in this chapter. Map, reduce, and filter aren't particularly difficult concepts, but there's a lot of mileage to get out of them. Let's look at map, reduce, and filter for arrays, bracketing the question of generators and the potentially infinite lists offered by languages such as Haskell. We will show you how to use JavaScript's array built-in versions of map, reduce, and filter. We will also take a look at the use of core JavaScript to implement these functions, not so much so that people can have IE8-compatible (and earlier) access to these functions, but to give a sense of how these things work.

The implementations that we will explore, after a warning about fool's gold, have a somewhat nonfunctional style of implementation. They use `for` loops where, in a purely functional language, the solution of choice would probably be a tail recursive implementation. The rationale for this choice is the goal of providing functional feature support in a way that operates optimally for JavaScript's plumbing and doesn't fail in the rare cases where (non-tail-call-optimized) JavaScript recursion runs into its limits.

Fool's gold – extending Array.prototype

A note of caution is due. An attractive solution, and one that can be easily implemented, is to extend (here) `Array.prototype` or the prototypes of other objects used by others, including `Object.prototype`. Don't do it.

Among other things, extending `Array.prototype` and its kin destabilizes the playing field for other people's software; it's like rewriting other people's code when you haven't seen it. Probably, the best use case for extending basic prototypes is polyfills (reimplementation of a functionality that is not available in the current environment, using the available features), but even then, if there are competing polyfills, only one of them can win. Now it is unlikely that your polyfill will have the same testing for bug-for-bug compatibility as a major browser manufacturer. This leaves the door open to subtle bugs. In our case, in the interest of supporting sparse matrices, we ignore undefined entries, but not null. I submit that this is reasonable in context, but far from the only conceivable way someone smart (or not-so-smart) would frame the matter. JavaScript has two null values, `null` and `undefined`, and there is potential for more than one opinion about how these two distinct null values should be handled. What if the semantics that makes sense to us isn't the semantics that was evident to someone else? Do we want to open the door to slippery heisenbugs?

There is an alternative that is straightforward and good: make your own functions, preferably anonymous functions defined within a closure, and stored in a variable. These functions can, if desired, check whether there is a browser's built-in function, such as `Array.prototype.map()`, and fall back to the built-in function if it is found. It can do for our code almost any job accomplished by extending `Array.prototype`. But it shows good habits and that doesn't yank the rug from under anyone else's feet.



The term *anonymous function* in JavaScript does not exclude functions stored in named variables. It only means that they have been defined without a function name. In other words, they are defined like `function()`, `var foo = function()`, or other alternatives, but not a name between the function keyword and the opening parenthesis, that is, `function bar()`. Usually, we will be using anonymous functions, whether they are stored in a variable or not, but there is a debugging-related reason for which we might name a function even if we never use it: a debugger's stack trace may be more informative in its way of mentioning functions if the functions are named, even if the names are never used. It does make sense, for this purpose, to write `var quux = function quux()`.

One aside about what we may be developing privately, off in a corner: an astonishing number of Unix utilities began life as private hacks to solve local problems for different people. Things that spread like wildfire are often not things that are engineered to spread like wildfire, such as the Web, JavaScript, and versions of PHP before 5.0. In their first versions—perhaps this applies to the Web more than JavaScript—they did something specific and had people struggling to function in a more complete manner.

The statelessness of HTTP was a carefully chosen feature, but for that time, a good chunk of web programming was trying to support use cases where stateless HTTP was getting quite painful. There may be differences between a 5 MB HTML5 key-value store and a 4096-byte cookie ceiling, but they present both more or less gracious accommodations in providing hooks for properly stateful behavior when the web's HTTP is stateless: the web was built not to enable the dynamic content as it is the lion's share of all web contents today. JavaScript has its strengths and weaknesses, and its weaknesses may be some of the worst of any wildly successful language, but the reason for its wide success and celebrity status is not its strengths or weaknesses. It succeeded because it was included in browsers at a time when the Web was spreading like wildfire. Both JavaScript and the Web had people trying to fix their limitations and weaknesses to do many things well after they had spread like wildfire when they actually did only one thing well.

The mentality of "This is just something in a corner and we do not need to think about maintenance or interoperability" is very, very dangerous. Perhaps now your software will not break anything if it subtly redefines the behavior of an object or array, but not ever? Not even with any future decision? Not even if someone realizes that in solving X, you created a great engine for Y that could save a lot of work? Client-side JavaScript is some of the code that is most quickly made open source (after all, even lawyers concerned about keeping things proprietary know that your whole kaboodle can be delivered to anyone who logs in to your system from the Web), and it is dangerous to assume that a particular redefinition of standard behavior is simply future-proof.

A short answer to the preceding questions is this: don't redefine anything on which other people's code is built, including redefining part of `Object.prototype`, `Array.prototype`, `Function.prototype`, and so on. Opt for your own implementation as far as it makes sense, but don't (forcibly) install it for everybody.

Avoiding global pollution

It is also good practice to minimize incursions into the global namespace. The more global variables you add, the more easily you can conflict with other tooling. When Yahoo! announced YUI, as a matter of basic manners, they used only one global variable—YUI. There's a whole library that's available, but you don't have pages and pages of items dumped on the browser's global namespace; every call to YUI().use() or whatever is entirely contained in YUI's one incursion into the global namespace. jQuery uses a little more of the global namespace than they advertise, but in principle, they ask us only to use `jQuery` and `$`. Also, they try to make the second one wholly negotiable, as `jQuery` acknowledges that other frameworks need `$` and `jQuery` is intended to play nicely with others.

However, you can actually go further than this, by an immediately invoked function expression, including the ReactJS web application explored in *Chapters 8, Demonstrating Functional Reactive Programming in JavaScript – A Live Example* to *Chapter 11, Demonstrating Functional Reactive Programming in JavaScript with a Live Example Part IV – Adding a Scratchpad and Putting It All Together*. The functionality that you can accomplish without making a single global variable is quite a lot. Perhaps libraries should have some global variable as a public face available for others, but it is perfectly possible to make a web application that doesn't touch global variables.

The map, reduce, and filter toolbox – map

A map takes an array and a function and returns a new array with the function applied to all of its elements. As an example, let's create an array of the numbers 1 to 10, and use map to create a new array with their squares. (Note that JavaScript is somewhat inconsistent about whether array options modify an array in-place, return a modified array, or do both. The array's `map()`, `reduce()`, and `filter()` functions all create a new array, leaving the original array unchanged and untouched.):

```
var one_to_ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
var squares = one_to_ten.map(function(x) {return x * x});
```

The `squares` variable now holds, `[1, 4, 9, 16, 25, 36, 49, 84, 81, 100]`.

One implementation of `map()` might be as follows:

```
var map = function(arr, func) {
  var result = [];
  for(var index = 0; index < arr.length; ++index) {
    if (typeof arr[index] !== 'undefined') {
      result[index] = func(arr[index]);
    }
  }
  return result;
};
```

The reduce function

What the reduce function does is it takes an operation and progressively apply it to the elements in an array. You may have covered infinite (and finite) series in school, perhaps with symbols like this one:

$$\sum_{i=1}^{n} c$$

In this kind of operation, the uppercase sigma (**Σ**, which is roughly equivalent to "S" in the Greek alphabet) is used for sums, and less frequently, the uppercase pi (Π, which is roughly equivalent to "P" in the Greek alphabet) is used for products. Both of them repeatedly apply an arithmetic operator to a finite (or infinite) series of numbers, and they work on the same basic principle as `reduce()`.

What this kind of notation says is, "For this series of numbers, what do we get if we add them together, that is, reduce them with the addition function, taking the first number and the second, calculating their sum, then adding it to the third number, and so on?"

If we wish to add up the contents of an array, we can reduce it with an addition function (as a minor implementation detail, we do not use the bare `+` operator because we can't directly pass an operator as a regular function):

```
var sum = one_to_ten.reduce(function(x, y) {return x + y});
// returns 55
```

Finite and infinite series taught in school are usually sums; we can use other series as well. For example, if we want to calculate 10!, we can reduce by multiplying instead of adding in terms of the function we supply:

```
var factorial = one_to_ten.reduce(function(x, y) {return x * y});
// returns 3628800
```

Reduction does not need to be mathematical in character; this just gives us a quick way to demonstrate it. We can also use reduce to concatenate arrays of strings, where the `+` operator serves the purpose of string concatenation rather than numerical addition:

```
var message1 = ['H', 'e', 'l', 'l', 'o', ',', ' ',
  'w', 'o', 'r', 'l', 'd', '!'].reduce(function(x, y)
  {return x + y});
var message2 = ['Hello', ', ', 'world', '!'].reduce(
  function(x, y) {return x + y});
// Both invocations return, 'Hello, world!'
```

However, there is a basic difficulty that JavaScript's built-in functions do not solve for us. We will sometimes need to make a choice and further specify what we really want. Numerical addition, multiplication, and string concatenation are all associative, which essentially means that you can put parentheses wherever you want, follow standard rules for parentheses, and get the same answer. In numerical addition, the following are equivalent:

```
1 + (2 + (3 + 4))
((1 + 2) + 3) + 4
```

Both of these calculations give `10`, and if we multiply instead of adding, both give us a product of `24`. But what happens if we use exponentiation for very slightly different values:

```
Math.pow(2, Math.pow(3, 4))
Math.pow(Math.pow(2, 3), 4)
```

This is the same sort of thing we have in the immediately preceding code, although admittedly with an uglier namespaced function instead of an infix operator. In non-JavaScript notation, using a caret (^), we get the following pseudo-JavaScript, which restates the preceding calculation:

```
2 ^ (3 ^ 4)
(2 ^ 3) ^ 4
```

If we use `console.log()` with the `Math.pow()` calculation that you just saw, we get this:

```
2.4178516392292583e+24
4096
```

There's a slight difference here. One result is a four-digit integer; the other is expressed in scientific notation with a lot more than four digits. Well, how many things are really like the special case of exponentiation?

The answer to this question is a bit tricky, partly because of how I have deceptively framed the question to illustrate a treacherous misunderstanding. Exponentiation, where it matters how you add parentheses, can be more like the general case. There do exist cases where it doesn't matter, and they might even more commonly be candidates for `reduce()`, but in the general case, we should not assume that the two are equivalent. We will give a `fold_left()` and `fold_right()` function; these are not the only two options (you can do things manually if neither of them is what you want), but they respectively calculate the sum of the array one to ten as follows:

```
(((((((1 + 2) + 3) + 4) + 5) + 6) + 7) + 8) + 9) + 10
1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + (9 + 10)))))))))
```

One is not necessarily better than the other, but the difference can matter. JavaScript's built-in `reduce()` function is a left fold, starting from the left and moving to the right, as shown in the first of the two preceding expressions (and this is probably a sensible default).

If we define `fold_left()` and `fold_right()`, it could look something like the following. I use abbreviations here where the full word spelled out would look too close to a reserved word; for example, array won't collide with Array but they are confusingly close (there would be a similar collision with a variable named function):

```
function fold_left(arr, fun) {
  var accumulator;
  for(var index = 0; index < arr.length; ++index) {
    if (typeof arr[index] !== 'undefined') {
      if (typeof accumulator === 'undefined') {
        accumulator = arr[index];
      } else {
        accumulator = fun(accumulator, arr[index]);
      }
    }
  }
  return accumulator;
}

function fold_right(arr, fun) {
  var accumulator;
  for(var index = arr.length - 1; index >= 0; --index) {
    if (typeof arr[index] !== 'undefined') {
      if (typeof accumulator === 'undefined') {
        accumulator = arr[index];
      } else {
        accumulator = fun(arr[index], accumulator);
      }
    }
  }
  return accumulator;
}
```

The last core tool – filter

Filter winnows through an array for values that meet some criterion. For example, we can filter only positive values, as follows:

```
var positive_and_negative = [-4, -3, -2, -1, 0, 1, 2, 3, 4];
var positive_only = positive_and_negative.filter(
  function(x) {return x > 0;});
```

The `positive_and_negative` filter, after this run, is as declared; `positive_only` has the array value of `[1, 2, 3, 4]`.

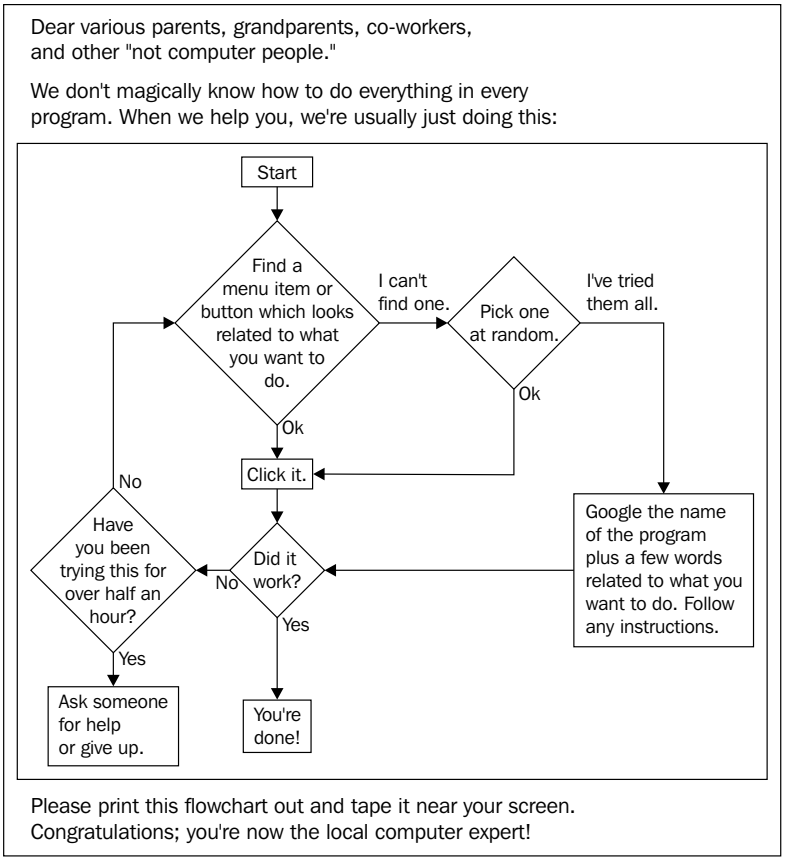
Filter is useful for narrowing down the contents of an array. If we have an array, as the preceding one, with the contact information of Mr. Smith, we can access fields to narrow down to things that may be of our interest. We can state that we want only one state, or require a phone number with a particular telephone area code, or state some other criterion on anything that a function can tell. If our record includes GPS coordinates, we can filter the contents to include only the results within a particular radius of a specific central point. All we need is a function that will return true for the records that we want to include and false for those that we don't.

An overview of information hiding in JavaScript

In general—meaning any programming language, client-side, server-side, non-web, mobile application, and almost anything else, perhaps even microcontrollers (which can now run a stripped-down Python)—there are different methodologies that generally have their own strengths and weaknesses. Steve McConnell's *Code Complete: A Practical Handbook of Software Construction* (`http://tinyurl.com/reactjs-code-complete`) discusses different methodologies and covers, for instance, how object-oriented programming's sweet spot is with larger projects than are really procedural programming's sweet spot. For most methodologies, his suggestion is that they have their strengths, weaknesses, and sweet spots, and under conditions X and Y, you should consider methodology Z. But there is one exception—information hiding. His simple advice on when to use information hiding is "As much as you can".

Procedural or structured programming is perhaps easy to overlook, and it's not pushing the envelope to use its strengths. But suppose we look at it when it first came out, with its functions/procedures, if-then-else, `for`/`while` loops, and procedure bodies not open to prying eyes. Now, if we compare this to straight assembler or machine code, with pre-Dijkstra-style use of gotos that do not even pretend to emulate structured control flow, then we understand that procedural or structured programming is really, really astounding. Also, it comes as a blinding flash of the obvious today because it has largely succeeded, to everyone's benefit. Flowcharts, which in the past were essential lifesavers for anybody hoping to understand a complex system, have become a novelty item. They appear on a mug, in an XKCD comic that shows how to deliver excellent tech support, or in other whimsical uses, as they are no longer needed to give any kind of road map to give some people—a sense of how to find their way through spaghetti.

Now a large system may be vastly more complex, and vastly larger than would fit into the memory or disk of an old, flowcharted, goto-navigated program, but procedural programming has effectively exorcized that ghost. Also, newer iterations in software engineering paradigms, such as object-oriented programming, have cut down the difficulty of understanding large systems. And in both cases, much of the benefit is a practical way to advance information hiding. With structured programming, you can navigate through source code without having to keep track of every point where the assembler or machine language renders a jump (that is, a goto). Both structured and object-oriented programming have historically allowed developers to unprecedentedly treat more of the program as a set of closed black boxes, and you only need to open and inspect a small fraction of them. They deliver information hiding.



The preceding flowchart is a novelty flowchart from `http://xkcd.com/627/`. I've never seen a real flowchart for a program that someone and I were talking about.

A standard textbook example of information hiding, in classic Java, might be this:

```
class ObjectWithPrivateMember {
  private int counter;
  public ObjectWithPrivateMember() {
    counter = 0;
  }
  public void increment() {
    counter += 1;
  }
  public void decrement() {
    counter -= 1;
  }
  public void get_counter() {
    return counter;
  }
}
```

Uncharacteristic of a good example of live Java classes from production, this class has one private and four public members. The usual goal is to make the interesting heavy lifting as well hidden as possible and just present a simplified facade to the rest of the world. This is how Java's object-oriented programming delivers information hiding, and even though there are important differences between object-oriented languages and how they approach and handle objects, this is one of Java's strong suits. Not only are object methods supposed to be written procedurally—as black boxes that have defined inputs and outputs—but they are also encapsulated in objects, where multiple smaller black boxes can be subsumed under a larger black box. In this case, we see another benefit of information hiding: we are largely protected from the outside and free to make whatever internal changes we want without breaking outside usage, as long as we preserve the same behavior. Suppose we decide to keep a log of when the counter was changed and to what value. That's at least another private field and changes to the internals of the methods that represent the public interface, but we can make the changes without altering any single detail of any class that accesses this class. Now suppose we want even more logging and we want our log to record a full-stack trace. Internally, we need to use something such as `Thread.currentThread.getStackTrace()`, but externally, no one needs to know or care about our refactoring. In a larger class, we might find a bottleneck that can be improved considerably by switching to another equivalent algorithm. Because of the way Java's object-oriented programming delivers information hiding, we can make an awful lot of changes without disturbing other people, who can use our work without troubling themselves about anything other than our public interface.

Information hiding with JavaScript closures

We need to look a little further to see the patterns of information hiding. In Java, you are quickly taught the language features for information hiding, and encouraged with words such as the security maxim—"Stinginess with privileges is kindness in disguise"—to err on the side of declaring things non-public. In JavaScript, there may be future-reserved words such as `public`, `private`, and `protected` (which Douglas Crockford suggested may be there for Java programmers to feel more at home with JavaScript at the expense of understanding JavaScript's better side), but there are not the same kind of obvious mechanisms now. All of an object's members, whether they are functions are not, are open for inspection. JSON—another thing that spread like wildfire, but without people cursing its simplicity—has no mechanism offered to mark anything as non-public.

However, there is a technique to create private fields in a feature of some functional languages called a closure. It isn't exactly a technique that is simply present in the language to create information hiding, but it allows us to create objects with non-public information. To port the functionality of the preceding example, including the non-public state, we can have something shown as follows:

```
var factory_for_objects_with_private_member = function() {
  var counter = 0;
  return {
    'increment': function() {
      counter += 1;
    },
    'decrement': function() {
      counter -= 1;
    },
    'get_count': function() {
      return counter;
    }
  }
};
```

The example given here suggests how we might expand it. A more involved example could have more `var` functions storing fields and functions, and return a dictionary that would expose its public interface. But let's not simply jump for that; there are some interesting things along the way.

In a functional language, functions can contain other functions. Indeed, given JavaScript's syntax—where functions are first-class entities that can be stored in variables, passed as arguments, and so on—it would be surprising if functions could not be nested even two deep. So, the following syntax is legal:

```
var a = 1;
var inner = function() {
  var b = 2;
  return a + b;
}
```

But the same can legally be wrapped in a function, as follows:

```
var outer = function() {
  var a = 1;
  var inner = function() {
    var b = 2;
    return a + b;
  }
}
```

It is a basic feature of functional languages, including JavaScript's heritage, that inner functions can see variables from outer functions; hence both `a` and `b` are equally available to the `inner` function.

Now what happens if we change the `outer` function to return the `inner` function? Then we have the following:

```
var outer = function() {
  var a = 1;
  var inner = function() {
    var b = 2;
    return a + b;
  }
  return inner;
}
var result = outer();
```

By the time the function finishes executing, its variables have fallen out of scope. JavaScript now has `var` variables with the function scope and is in the process of getting `let` variables with the block scope, but any variables declared in either way in the `outer` function are no longer available. However, something interesting has happened; the `inner` function has survived the end of the `outer` function, but in a logically consistent fashion. The `inner` function should and does have access to its execution context. We have a closure.

This phenomenon can be used for information hiding, and information hiding is important. However, it may be argued that what is most interesting here is not that it can include non-public variables, potentially including functions, but that the execution context as a whole is maintained as long as something with access to it survives. This leaves an interesting territory to explore. A StackOverflow member once commented, "Objects are poor man's closures," and both objects and closures have interesting possibilities beyond the FAQ entry about how to use their features for information hiding. Even code complete, which may strongly endorse information hiding, never says, "Use information hiding as much as possible but nothing else."

Perhaps it would be harsh to blame functional language purists for saying, "JavaScript has to wait until it becomes 2 decades old before implementing tail call optimization, instead of punishing standard functional programming's use of recursion—as in, old enough to go from being a newborn infant to an adult under US laws." However, irrespective of anything else that may irk functional programmers about JavaScript, JavaScript did get closures right enough from the beginning, so much so that closures that retained the execution context were, and remain, a significant feature in JavaScript all the way along. And 2 decades later, they remain the primary, and possibly only, information hiding resource in most browsers.

Summary

In this chapter, we hit a few notes of functional programming in relation to JavaScript. We looked mainly at three topics. Custom sort functions provide a simple and useful glimpse at how we can pass a helper function to a higher order function to obtain more useful behavior than the default. Map, reduce, and filter are the three workhorses of functional programming with respect to arrays. With Closures and information hiding, we took a look at a functional way of providing some core interest in responsible software development.

JavaScript is a multiparadigm language with functional roots and some functional language strengths, although it is perhaps uncommon for functional language purists to lump JavaScript together with imperative multiparadigm languages. JavaScript does not have, as some functional languages do, permanent binding on assignments or purely immutable data structures.

All languages have better and worse neighborhoods, but JavaScript so starkly combines excellent parts and terrible parts that the basic approach of Crockford's *JavaScrpt: The Good Parts* and *The Better Parts* is not seriously questioned among good developers (I wonder why no one has yet sold Kernigan and Ritchie, *The C Programming Language*, Second Edition, as *C++: The Good Parts*). It would be provocative to the point of being obnoxious to argue that defaulting to dumping things on a global object is a good idea for developing web applications. This extends to the functional aspects of JavaScript as well. JavaScript was the first mainstream language to allow anonymous functions, or lambdas, which have been staples of functional programming roughly since LISP appeared over 50 years ago. Even if even Java has jumped into that bandwagon now, its presence in mainstream languages is from JavaScript's influence. JavaScript has also had closures from the beginning. As far as some of the worse neighborhoods are concerned, it seems to have taken decades for JavaScript to apply tail call optimization, and with it, the functional programming style of using tail recursion without the penalty instead of for and `while` loops to structure iterative work.

Functional programming is an interesting topic and something that you can explore indefinitely (that is, the list of interesting aspects of functional programming that one can profitably explore is an infinite list, even though in the concrete, one only ever takes a finite number of items from the left of the list). Without trying to settle the question of whether JavaScript should be considered a functional language or not, JavaScript is best understood in relation to roots in functional programming, and learning to program better in functional languages/paradigms should be the basis of better programming in JavaScript. JavaScript may go down in history, not only as the language of the Web and perhaps the most crucial language for a programmer to know, but also the bridge language through which the goodness of functional languages ceased to be known as (like Scheme) "the best language you'll never use." Maybe, the strengths of a functional language come to be seen as non-negotiable for the construction of serious, mainstream multiparadigm languages.

Let's continue with a look at functional reactive programming.

6
Functional Reactive Programming – The Basics

I might begin here by somewhat awkwardly mentioning that I have a Master's degree in math and lots of math awards, but I find some of the mathematical concepts associated with basic functional programming to be a tad slippery. One wouldn't discourage someone strong in the relevant math and computer science fields from tackling the full mathematical rigor of, for example, the foundational functional reactive programming papers linked to in the Wikipedia article on functional reactive programming. However, the intent here is slightly different: to learn something from functional reactive programming that is useful to professional developers who do not have, or do not remember, the level of mathematical formation that informs those seminal works.

StackOverflow comments repeatedly ask, "Can you explain it in a way that doesn't assume a PhD in computational mathematics?" The intent here is not to provide the whole of those mathematical articles together, but a subset that can be practical and is useful to real professional software developers who do not dream in Scheme or Haskell.

In this chapter, we will cover:

- A trip down computer folklore's memory lane
- Functional reactive programming
- If you learn just one thing…
- Learn what you can about functional programming
- The future of frontend web development

Let's dig in. The folklore-laden trip down memory lane may be fairly long, but it is not, in any sense, dull.

A trip down computer folklore's memory lane

There is a scathingly insulting checklist that has been floating around to apply to pet (or other) programming languages. One of the put-downs is, "Programmers should not have to understand category theory to write, *Hello, World!*" It reflects, in part, an irritation of mistakes that wet-behind-the-ears juniors keep making when they propose the best programming language yet. In that, it might be compared to the viral evil overlord list of stupid things that keep happening in adventure movies. Learning from the mistakes of countless movie villains, the author declares that "Shooting is not too good for my enemies" and "I will not include a self-destruct mechanism unless absolutely necessary…" The put-downs come from frustration at seeing the same mistakes again and again.

There are other points that show programming wit and wisdom. For one example, there are orders of magnitude more pet or toy languages than successful languages, where success can either be in academic computer science circles or business information technology. One widely recognized turning point in the development of any language is when it works at a level where one could *dogfood* it by using it to write its own compiler.

And this is not an insignificant point; when Java was first announced, it was declared that the Java compiler was written in Java itself, with the implication that a system that could run the Java runtime environment should be able to compile software written in Java. In relation to this, a standard question to insult someone's enthusiastically gushed-about pet language is, "Has it been used to write anything besides its own compiler?" And this particular node of computer wisdom and folklore is baked into the checklist: two of the entries are, in quick succession, "The most significant program written in this language is its own compiler" and then, even more insultingly, "The most significant program written in this language isn't even its own compiler".

But the philosophical objection of, "Programmers should not need to understand category theory to write, *Hello, World!*" is not simply making things up to sound insulting. There is a tradition started by the 1978 first edition of Kernigan and Ritchie's classic *The C Programming Language*, where the first program developed before diving into intricacies was a minimal C program, on the thought of "Let's crawl before we try to walk," to print out *Hello, World!*:

```
main()
{
  printf("hello, world.\n");
}
```

People introducing new programming languages overwhelmingly follow in the tradition of using `Hello, World!` as their first sample program. That's one end of "Programmers should not need to understand category theory to write, *Hello, World!*" So what is the other end of the spectrum?

In the world of academic mathematics, whether pure or applied, mathematics as a sign of success has become very specialized (like almost any domain that's had enough work). It has been commented that it's a rare beast of a mathematician who can understand more than 13 of 50 papers presented at a math conference. Mathematics has become specialized enough that most math PhDs, however competent, cannot understand the work of most other math PhDs. In that case this doesn't make sense, the hope of being able to understand all of mathematics is like the hope of being able to speak all human languages: a bit naïve. The point of a math PhD program is perhaps not to develop you to the point of being able to follow the whole breadth of the discipline of mathematics, but to understand some narrow area deeply enough that, by the time your PhD is complete, you understand that highly focused area better than anyone else in the world.

There are two exceptions of a sort, disciplines that connect all of mathematics, but from completely opposite directions. On one hand, there is logic and the foundations of mathematics, which looks at the bedrock that all other areas of mathematics is founded on. Now there are some questions about whether logic belongs to math or philosophy, and one hears of people asked to decide whether they want to be logicians or mathematicians. But setting aside some of these questions, it would not be too controversial to say that logic connects to all of mathematics by digging into the bedrock that the rest of mathematics rests on.

And then there's the other option: *category theory*. A Barbie doll once said, "Math is hard", but the mathematical community understands that very well without Barbie's help. Albert Einstein said, "Do not worry about your difficulties with mathematics. I can assure you that mine are greater still." But the branch of mathematics called category theory is particularly and obnoxiously harder.

If logic can study the bedrock that the great edifice of mathematics is built on, category theory looks at the already built city and explores architectural themes and similarities running through all kinds of mathematical neighborhoods. Category theory is a discipline a bit like the academic discipline of comparative in that practitioners are expected to be able to cope with comparative literature in not just one, but several languages. It could be argued that category theory is the most difficult place under the whole mathematical umbrella. You need to do something most math PhDs are never taught.

Also, it is perhaps a tribute that my thesis advisor, who was a category theorist, was able to effectively supervise a thesis in the obscure branch of point-set topology even though he did not show any particular specialization in point-set topology. So saying that a programmer using your language needs to understand category theory to write *Hello, world!* is quite the scathing insult.

So what does this have to do with functional or functional reactive programming? Glad you asked!

In the sources linked to in the Wikipedia article for functional reactive programming, Haskell (or something built on it) is the dominant language. There are a couple of other players, such as a dialect of Scheme, but people seem to keep coming back to Haskell. There are a number of Haskell resources out there; one of the most respected is *Learn You a Haskell for Great Good*, `http://tinyurl.com/reactjs-learn-haskell`. It has a *Hello, World!* program that mysteriously appears in the ninth chapter instead of the first. Why the ninth? Well, as explained, input and output are built on top of monads. But does it really take that much explanation to get to monads? Yes; monads are built on top of the concept of applicative functors, which are built on the basic concept of functors, which ring a bell as the name of something met in math grad school and not really understood. Let's visit the Wikipedia page for functors. The Wikipedia is known for being something that is clear and easy to read. And there are a couple of things in the Wikipedia page for functors. One is that the language is awfully esoteric for the Wikipedia. The other point is that functors are in fact something acquired from category theory:

> *In mathematics, a functor is a type of mapping between categories, which is applied in category theory. Functors can be thought of as homomorphisms between categories. In the category of small categories, functors can be thought of more generally as morphisms.*

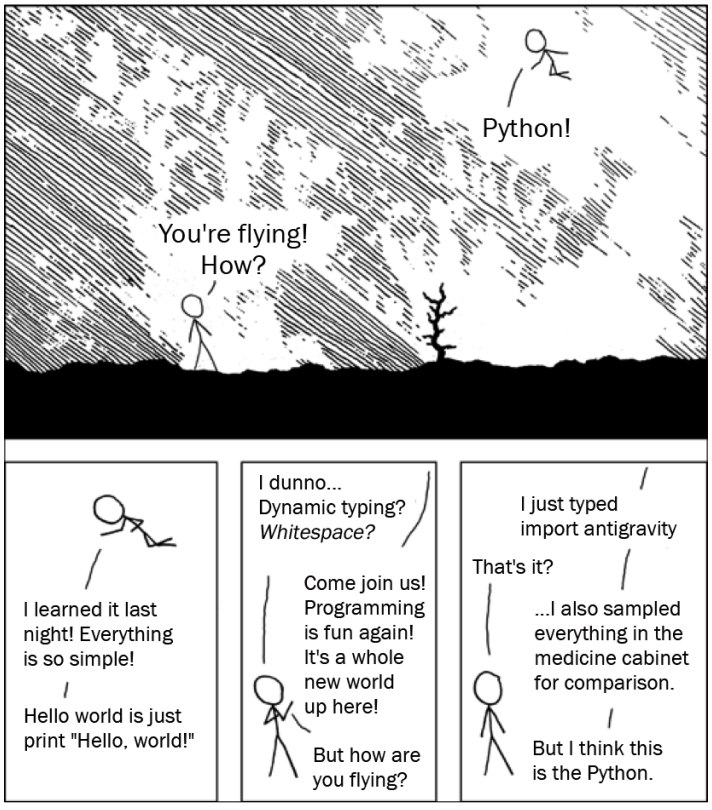
Functors were first considered in algebraic topology, where algebraic objects (such as the fundamental group) are associated with topological spaces, and algebraic homomorphisms are associated with continuous maps. Nowadays, functors are used throughout modern mathematics to relate to various categories. Thus, functors are generally applicable in areas within mathematics that category theory can make an abstraction of.

Advanced prerequisites for Hello, World!

If you want a challenge, read the Wikipedia article on functors. But if you find yourself skimming because it's mostly over your head, you're in good company: possibly a lot of math Ph.D.'s tend to skim the same way you would, for much the same reason.

There is more that can be said at this point, but I'll limit myself to one further remark after commenting on the intent of this chapter. This leads into the central difficulty of this chapter. This is a text on information technology rather than computer science, and while one may check in with computer scientists, this text is intended to be written from the perspective of one programmer who doesn't follow all the layers of math to another. The equivalent of the goal in Haskell, would be to say on a *monkey see, monkey do* basis, "This is an example of a pure function; that is an example of an input and output monad. Try to do as much of your heavy lifting as possible from the pure portion of your program and confine input and output to as small a quarantined space as possible."

And there is one point where one would contrast Haskell with Python, starting again with XKCD—note what is given as the first example of everything being simple in Python. One's first encounter with Python leads to the feeling of being head over heels in love with programming again. The same is entirely true of ReactJS, which is like discovering the web for the first time all over again:



Python and Haskell are similar in at least one respect: they both allow rapid software development. Haskell boasts a similar feature to what one would expect of Python: an undergraduate spent a few months and implemented a good deal of the Quake 3 engine in Haskell. There may be other strengths to Haskell, such as its rock solid type system, and by the time something compiles, it already stands a good chance of working. However, the question pursued here is, "Does it make programmers highly productive?" This screen shot is from a Quake 3 level implemented in a few months for a one student's undergraduate project. There are things Haskell has that Python doesn't: a rock-solid type system, for instance. However, Haskell and Python are similar at least in this: in the hands of a proficient developer, they allow productivity and pace of development that needs to be seen before it can believed.



But, while seasoned programmers may try Python and find themselves flying, if they can't handle the math, they won't have the same experience using Haskell. Haskell offers rapid development super powers to people who can handle a good deal of computational math. Python offers such powers to a much wider group of programmers, whether or not they have a heavy math background.

This chapter is an attempt, rightly or wrongly, to explain things so that workers in information technology, rather than computer science specifically, can have a seasoned programmer's good Python experience with functional reactive programming and ReactJS, instead of a seasoned programmer's bad Haskell experience that leads so many developers to keep on making comments that become very sad after a while, depressing comments such as "I might understand this if I just knew more math."

The purpose of our writing this book is to make a book useful for programmers in the field, not just computer science students who know a lot of math. But an easier approach is at least hinted at, along with seasoned programmers saying "I might get this if I knew more math." Taken directly from the page `http://tinyurl.com/reactjs-learn-monads`. Now, here are some steps on how to learn Monads:

1. Get a PhD in computer science.
2. Throw it away because you don't need it for this section!

(Perhaps ordinary developers can, after all, profit from (reactive) functional programming!)

Distinguishing the features of functional reactive programming

One of the leading lights of functional reactive programming, and arguably a grandfather to functional reactive programming, Conal Elliott, looked back on the term "functional reactive programming," and a leading light's second thoughts about a name can be very interesting. Elliott expressed a reservation about the term *functional*, suggesting that the word means so many things now that it means very little, and expressed a regret about a word that the term didn't include: time. He suggested an alternative name of denotative continuous-time programming, which is significant even if we use the more standard term of "functional reactive programming" here. By *denotative*, we mean, as we discussed earlier about ReactJS, that you specify only what needs to be accomplished, not every step of how to accomplish it. Continuous-time did not merely mean that it should be called such, but that continuous time was important enough that it should be baked into the name for what is now called functional reactive programming.

The continuous time element comes up in the sources and may seem surprising to some, since computers only measure time discretely, but the distinction is a distinction in a conceptual model rather than a feature observed in implementations. The comparison is made to infinite lists as they exist in functional languages, where one may take as much or as little from a list without running out of precomputed entries, or perhaps more saliently to the difference between raster graphics (GIF, JPEG, PNG), which have a certain fixed number of pixels represented, and vector graphics (SVG, some PDF) where admittedly, an image is rendered to be displayed on something with a concrete number of fixed pixels, but an SVG logo can be rendered in accordance with a classic advertising executive's quote that a company logo should look as good one sixteenth of an inch high on letterhead as it does eight feet tall on top of corporate headquarters.

Continuous time means that time is handled like SVG or other vector graphics, not raster GIF/JPEG/PNG, which is stored at a fixed resolution and not a pixel more. One of the suggestions made for functional reactive programming is that continuous-time events and probably continuous-valued behaviors or streams of events have a first-class entity status as one of the defining features (though, some might point out, perhaps not the only one) is that functions are first-class entities that can be passed as arguments, as one does in JavaScript and other languages with anonymous functions.

It may not be immediately obvious how ReactJS relates to this; I've watched over a dozen ReactJS videos, usually from Facebook developers. There is emphatic mention of *denotational symantics*, a formal term to describe only what needs to be accomplished and not every step of how to accomplish it. And there is consistent discussion of the virtual DOM, which amounts to "You can learn more if you want, but all you need to do is tell how to `render()`, and trust the system to do all the rest." But in fact, continuous time semantics are baked into how ReactJS works at a very basic level. Part of the developer's responsibility is to write a `render()` method that specifies what should appear on the page at the time it is called (and, perhaps, to call `render()` appropriately; `render()` doesn't run by itself).

This does not have every feature of continuous time; one instructional video hints at a system that does not just work in real time, but allows VCR-style "rewind" and "fast forward" functionality to step through time, and one of Pete Hunt's ReactJS videos hints that Facebook, through ReactJS technology, might take a bug report and be able to replay, detail by detail, what went wrong before someone gave a bug report with no written description of what went wrong beyond "a profanity." However, the front and center use case is one where continuous time is assumed and it is the developer's responsibility to make a `render()` function that can correctly say what to render the instance it is called, and (incidentally) to call that function appropriately.

If you learn just one thing...

Richard P. Feynman's classic "Feynman lectures," held as an exemplar of clear explanation of technical topics, opens with a very simple question: if all else of science were forgotten and only one sentence's worth of information survived, what would that ideally be? Feynman gave a succinct answer that actually says quite a lot:

> *"If, in some cataclysm, all of scientific knowledge were to be destroyed, and only one sentence passed on to the next generation of creatures, what statement would contain the most information in the fewest words? I believe it is the atomic hypothesis that* ***all things are made of atoms — little particles that move around in perpetual motion, attracting each other when they are a little distance apart, but repelling upon being squeezed into one another.*** *In that one sentence, you will see, there is an enormous amount of information about the world, if just a little imagination and thinking are applied."*

And this serves as the springboard, in the Feynman lectures, to say a lot about physics as we know it.

The biggest learning point from functional reactive programming that can be put into a single sentence is a lesson associated with the functional programming itself, that functional reactive programming further refines: *Spend as much time writing pure functions, math-style, as you can, and as little time as possible writing or following recipes*.

A recipe says such things as "Preheat your oven to 350 °F. Mix the leaves, ghee, and salt together in a large bowl. Line two large baking trays with parchment. Divide the leaves evenly in a single layer on each tray..." Now this is not taking a dig at home economics and people who cook. (A purely functional approach to cooking would never produce anything edible, a minor drawback if you want to get anything done.) Recipes are equally to be found in the many, many YouTube videos detailing how to replace, for example, a broken wiper blade on a 2004 Ford Escort, and they power traditional hacker-written How-to's, which are less prominent today than they were earlier, not because the hacker community has realized that using a How-to is not an appropriate way to resolve a difficulty when following your nose is getting you nowhere, but because the general usability of almost everything has improved enough that you don't need a How-to that mentions the moon phase if you want to burn a CD; How-tos are much less likely to be the only game in town (which is really their best use case).

I am somewhat concerned with what theoretical contortions Haskell went through (read: *needed* to go through) to include input and output with minimal compromise to its functional status. But all the same, even if purely functional JavaScript may or may not be possible, we would do well to grow the portion of our software that is purely functional and minimize what gets work done by specifying how to do things. Functions here should not mean, as in structured programming, "a subroutine that returns a value." A function is not doing something plus returning something interesting when it is done. It rather has the mathematical sense of "something that takes zero or more arguments and does neither more nor less than return a value that is arrived at based on that value."

Examples of pure functions from basic mathematics used by computer people include arithmetic functions such as addition, subtraction, multiplication, division, exponentiation, factorial (4 factorial, for instance, is 4 * 3 * 2 * 1), Fibonacci numbers, trigonometric functions such as sine and cosine, hyperbolic functions, integral, derivative, Euclidean division to calculate the greatest common divisor between two positive integers, and so on. Without exception, these take zero or more (or, as the case may be for these, one or more) inputs, and yield something calculated from them without making any external changes; none of them update a database or output something to the console. They just take their inputs and deterministically calculate an output, no more, no less. That is of the essence of a pure function.

A **sesquipedalian** term is "a word that is a foot and a half long." Some of them float around, including in videos on ReactJS and functional reactive programming, such as idempotent and **referential transparency**. But the meanings are simple and straightforward in relation to pure functions.

An idempotent function is the one that returns the same result whether you call it once or a hundred times. In mathematics, addition and factorial, for instance, always give the same result. RESTful web services offer a less mathematical example of idempotence: requesting the same URL means getting the same HTML or other data, every single time. Getting static content is idempotent; a version of a library pulled from CDN should result in the same download no matter who requests it, where, or when.

Caching, such as one aims for with Steve Souder's classic far-future *Expires* headers for Yslow, is a useful thing to do precisely where there is idempotence between downloads. (If a download is idempotent, a document is the same document regardless of whether one downloaded a fresh copy or supplied it from your browser's cache.) Dynamic content, whether old-fashioned CGI scripts or dynamic Django applications, is not idempotent. If a page says, even in an HTML comment, "This page downloaded at such-and-such time," it is not idempotent. The Web was designed from the beginning to be idempotent; later on people started to realize that dynamic content could be highly desirable and worked on how to overcome the stateless, idempotent design of HTTP.

The nice eight syllables of referential transparency mean that a function call can equivalently be replaced with the value it returns. Because 4! is 24, it should be equivalent to include 4! in your code and just include 24 instead. If you have the cosine of a value, it should be equivalent to use a stored value of the result of that call to `cos()`, or to recompute. Impure behavior that would break referential transparency would be to have `cos()` log a string each time it was called, which is a classic example of a *side effect*.



> The term side effect is unfortunate and probably intended to be loaded language; in a medical context, all drugs produce multiple effects, some of which are the point of taking the medication and some of which are tolerated as a necessary evil that comes with getting the desired effects. A side effect medically is a medication effect that is tolerated but not why the medication is taken. Saying that logging a message in a program is a side effect is a bit like saying that taking a pain reliever, and experiencing consequent reduction in one's physical pain, is experiencing a side effect: that is the entire point of taking the medication, not any other effects the drug may have, and it's an odd thing to call this a side effect.

Preceding are some stock examples of functions from basic math, and perhaps that is easy, because in certain areas of math, everything is a pure function, perhaps built from pure functions, and the setting precludes impure functions or side effects. One could also give polynomials as an example of pure functions built up from pure functions, which is a really nice approach if you are equipped to use it, but feels foreign and confusing if one is used to framing everything in informative assumptions. Between functional and imperative functions, for programmers with an imperative-based formation, imperative function is the approach that is easiest in the short term but harder in the long term. For programmers with a functionally-based formation, functional programming is easy, both in the short term and the long term. But for an example of these concerns playing out in practical information technology, we need look no further than ReactJS.

Facebook's learning from prolonged pain essentially led to a realization that the way out of the morass was through idempotency and referential transparency, and that is what ReactJS was written to deliver.

Learn what you can!

A wise master in the Orthodox spiritual tradition boiled many things down into 55 maxims (`http://tinyurl.com/reactjs-55-maxims`), the second of which was, *Pray as you can, not as you think you must*, and these are wise words for much of programming too. And there is a suggestion here, but not so non-mathematicians will be daunted. Follow this one suggestion as you can, not as you think you must. Learn as much functional programming as you can. Write JavaScript in as purely a functional fashion as you can.

I have wrapped my head around functors now, as I failed to do so as a graduate student in math. I haven't conceptually wrapped my head around applicative functors and monads from a theory perspective, but the idea of writing pure functions as much as possible and making minimal use of input and output monads is something that appears doable on a *monkey see, monkey do* basis, which is much less taxing than having to trace a monad's conceptual genealogy. And this falls into the category of using functional programming as you can, not as you think you must.

The Wikipedia article on functional reactive programming links to nine major works in the field, and if you want to wrestle with a good mathematical challenge, all of them are worth a good wrestle. The mathematical symbols can be as dense as they are on the Wikipedia article on functors, which is quite dense.

But if we look at languages, there's a clue here. There are a few interesting possibilities, all functional, that are mentioned in the literature: a Scheme dialect, DDD, and Elm (which is its own language compiling, comparably to DDD, to its own JavaScript / HTML / CSS). But the strongest attraction of functional reactive programming authors seems to be by far Haskell. This offers us a free clue that, at least in its origins, Haskell is a center of gravity to pretty much all the seminal papers on functional reactive programming. Any language, including Haskell, has flaws, but it would be silly to simply ignore the fact that seminal works in functional reactive programming gravitate towards Haskell.

Functional reactive programming is reactive programming built out of the building blocks of functional programming. Some aspects of this are taken care of for us when developing in JavaScript with ReactJS. We only need to specify declaratively what the UI should look like when rendered, and ReactJS will handle all necessary compilation so a declarative `render()` method will be translated into optimized imperative manipulations on the DOM. But at least at first glance, it would make sense, if you want to understand functional reactive programming, to learn a technique closely tied to Haskell from the outset and only later, after you've walked a mile in Haskell's shoes and then know if they pinch you, write your "declaration of independence" from Haskell's lead.

There is criticism of *Learn You a Haskell for Great Good*, but the book was deliberately chosen as an excellent text to teach a top-notch functional language. Pointing out that a Haskell text covers eight chapters of theory and some category theory concepts before allowing the reader to see the traditional *Hello, world!* program, makes a much stronger point by picking on a strong text than picking on an introduction that is mediocre and invites of an obvious response of but there are much better examples that don't have that problem. A companion text that is more focused on practical applications for real-world information technology needs is Real World Haskell (`http://book.realworldhaskell.org/read/`). These aren't the only books out there, but they offer at least a good pairing and a starting point, and are often recommended together.

More to the point, don't try to snarf these two books and expect that after a day's study, or even a month, it will be easier to get things done in Haskell than whatever favorite language you've given years of use. Instead play around, and tinker with these things. Treat the Glasgow Haskell Compiler as a nice set of virtual Lego that a loved one gave you for Christmas. *Learn You a Haskell for Great Good* nowhere dives into how you can write a web server, and that is one of the book's strengths. It builds core strengths that are only to your benefit and should put you in a better position to appreciate and exploit opportunities for functional programming in JavaScript. G.K. Chesterton said:

> *"…to understand everything a strain. The poet only desires exaltation and expansion, a world to stretch himself in. The poet only asks to get his head into the heavens. It is the logician who seeks to get the heavens into his head. And it is his head that splits." Try to get your head into the Heavens, not the Heavens immediately into your head. If you are an adept programmer now, perhaps in an imperative paradigm, chances are good that when you were in school and exploring things, you sought to get your head into Heavens with programming. You wrote games; you played around, and acquired a foundation you would later use for professional work. If you want to learn Haskell, don't cram it. Become like a little child again and play. And read Learn You a Haskell for Great Good, with its deliberate avoidance of how to get something slammed out for a deadline, until you have an actual foundation before reading Real World Haskell, and please do not take Real World Haskell to be justification to "cut to the chase" and just try to release commercial-style functionality on deadline-style timing.*

Douglas Crockford, in his South American presentation on *The Better Parts*, gives an increasingly strong functional focus to functional programming when he describes good JavaScript. All of the earlier Crockford videos I've seen, from when there was just *The Good Parts* and no *The Better Parts* even on the horizon, seem to associate JavaScript's better parts with its functional side. But *The Better Parts* is even more explicit in saying that one of JavaScript harmony's improvements is that you can apply tail recursion, and use functional styles of flow control that make some kinds of flow control, such as looping almost or completely unnecessary.

Even apart from functional reactive programming, better JavaScript seems to increasingly mean functional JavaScript. And this is a very good thing. Scheme, as mentioned earlier, has been called "the best language you'll never use," and the set of generally functional languages that computer scientists consistently choose on their merits for computer science usage are a little paradise that one must leave to enter professional programming.

JavaScript changed that, and not only by making anonymous functions mainstream. JavaScript, especially when used with ReactJS, provides one of the greatest opportunities to enjoy the goodness of functional programming in mainstream software development. And as long as you understand what you are doing, the more you can write JavaScript in a functional paradigm.

Functional programming, reactive or otherwise, may come more easily if you're introduced to it in schooling that covers the more mathematical side of functional programming. But it is possible to teach a programmer how to write *Hello, world!* in Haskell while leaving the most incomprehensible mathematics of category theory out of sight and under the hood.

The computational mathematical foundations of functional programming are something that should be like a machine or assembler in a higher-level language: present under the hood and making language capabilities possible, but out of sight with minimally leaky abstractions that *just work*, regardless of whether one is a mechanic competent to make adjustments under the hood.

In some sense, for seasoned programmers who have been out of school for a while, what is needed is *The Good Parts* of learning functional programming. Now in this case the good parts may vary from programmer to programmer depending on their comfort level for functional programming. The criterion is "Do what you can, not what you think you must." Getting your head around the difference between declarative/denotative programming and imperative programming, is perhaps difficult, but not too difficult. The closely connected concept of writing pure functions as much as reasonably possible and quarantining code that must have side effects even if you're trying to avoid them, is a shift in thinking but not too slippery.

Learning functors, for instance, is actually a bit easier in Haskell than in category theory, even if the Wikipedia page does not reflect this. It shouldn't take most programmers too long to write a first Haskell program that is mostly pure functions and input and output handled by a minimal quarantine of monads. But using features such as monads is much easier than understanding the contorted steps one uses to work with pure functions and build up to monads.

And it is worth repeating: if functional (reactive) programming is appropriate for mainstream use, the heavily mathematical theory used to get from a function to a monad should no more be forced on average Joe professional developers than even a C programmer should be forced to work with the assembler or machine instructions generated by his software. One wag said that "C is a language that combines the power of assembler with the ease of using assembler," but C never forces most programmers to micromanage how the compiler renders the C source.

Many mainstream languages now, in particular multi-paradigm languages, have incorporated some elements of functional programming strengths. All the same, one might suggest that JavaScript, of all mainstream languages, directly offers the best set of functional programming strengths bar none. Not necessarily the best functional programming among the languages favored by computer scientists such as Haskell or Lisp/Scheme; it's hard to find a mainstream programming job where management will allow a solution in Haskell or Scheme. But among mainstream languages, JavaScript is still the juiciest. Computer scientists have long been fond of functional programming, and at least one programmer who studied math in school commented, "Functional programming is the first programming paradigm I've seen that makes sense." And for excellence that computer scientists have almost universally favored for (mumble) years or decades, JavaScript isn't just the language browsers will execute, important as that may be. It also offers the best opportunity for functional programming goodness in a language frequently encountered on job descriptions for employers who want to hire you.

JavaScript as the new bare metal

Douglas Crockford, in *The Better Parts* mentioned earlier, tries to make the point that programmers are just as emotional as everyone else. He backs up this point in a way that would not be a surprise to a Kuhn scholar: fundamental improvements in software engineering win by the attrition of programmers holding the earlier approach. He gives an example of six or so "it took a generation" remarks: "it took a generation" for software engineers to recognize that high-level languages were a good idea, or that the F-bomb of all programming language statements, the G-bomb or goto statement, was not a good idea. And although Crockford gives several examples, his efforts do not seem to have pretensions of including all the significant examples: although I'm not completely sure of the dates, it seems to have taken about a generation between the 1960s, when Smalltalk recognized that references were better than pointers (pointers have been called "the goto of data structures"), and the 1990s when a mainstream language that was "front and center," like Java, superceded pointers with references.

Crockford made a few remarks about programmers invoking freedom of self-expression to do all programming on bare metal or routinely use goto statements to handle flow control. But when all is said and done, including that it took two generations for anonymous functions to be used in a mainstream language, JavaScript being the first, improvements in software development do not take root because existing programmers embrace better ways.

They take root because new programmers embrace better ways, while the majority of older, unconvinced programmers die. (Even if they could learn. But in some sense there is an option to opt-out of becoming obsolete, by embracing new changes. It's just that many people say, "If it was good enough for me at 20, it's good enough for me at 40." There is no iron determinism grasping every individual: only a "default setting" on /programmers/ that ages poorly as a default setting.)

JavaScript is *lingua franca* on the web, and even if you object to it because it's not your favorite language (indeed, why should it be just like Perl, Python, Java or C++?), it is here to stay and is perhaps the best bet around for what will be the most important language, and will be for quite a while. But what may "take a generation" in this context is the realization that web programming in non-JavaScript languages is a good idea.

Alan Perlis said, "A programming language is low level when its programs require attention to the irrelevant," and if programming well in JavaScript requires avoiding large minefields of languages for reasons that are far from obvious to casual inspection, JavaScript requires attention to the irrelevant: JavaScript is low level.

One encouraging sign in newer web development, in ReactJS videos, is not only that another non-JavaScript language or syntactic sugar, CoffeeScript, is used, but that its introduction was presented smoothly and casually, and entirely without apology, defense, or explanation. The fact that they used CoffeeScript at all is significant, and the fact that they did so without any trace of defensiveness is even more significant. Now CoffeeScript might not be any kind of be-all, end-all among languages that can or should be able to be compiled to JavaScript. But it is encouraging to see something other than JavaScript "bare metal."

This does not mean that there is no place for that "bare metal" programming. Amateur game developers or programmers from mega-corporations trying to squeeze the last ounce of performance out of either bare metal for standalone applications and games, will legitimately want to squeeze the last ounce of performance out of their user's computer, whether working on the "bare metal" of application programming or JavaScript on the web. But just as one ordinarily does not write web applications in C or assembler (and didn't do so even when CGI scripts were the primary means of delivering dynamic content), for most uses of web programming, a good smartphone (a slightly old iPhone 5 is roughly 100 times as fast as a top-of-the-line computer from when the web was new) really is fast enough to run code generated by compiling other languages to JavaScript. And JavaScript is, after all, so much more impressive when one understands how it is developed. It is a computer language that was designed in 10 days, something that one ordinarily greets not with, "That's an impressive feat!" but also, "Dude, ease up a bit! If you keep abusing stimulants like that, you'll kill yourself!"

The essential reason why using a high-level language is desirable is something painfully obvious that programmers learn to shut out so they can get any work done in JavaScript. Douglas Crockford's *The Good Parts*, and with it the idea that JavaScript has both treasures and land mines and a good part of navigating JavaScript well is avoiding minefields altogether, has taken such deep root that this title's brief summary may be entirely superfluous for most programmers reading this book.

That, even by itself, is serious reason to consider alternatives to programming on JavaScript "bare metal" if one has the option. And in fact, there are many alternatives to JavaScript "bare metal" for frontend web development. `http://tinyurl.com/reactjs-compiled-javascript` has a long list of other languages, including family and friends intended to offer an enhanced JavaScript in some aspect or other, and compilers for other languages to JavaScript, including (often) multiple options for Basic, C/C++, C#/F#/.NET, Erlang, Go, Haskell, Java/JVM, Lisp, OCAML, Pascal, PHP, Python, Ruby, Scheme, Smalltalk, and SQL. Presumably not every single one of the compilers and implementations is particularly good, but like every other serious computer language, JavaScript is Turing-complete, and not only is it theoretically possible to compile other complete languages to JavaScript as well as "bare metal", but it is practically possible and makes plenty of sense. JavaScript may become the most important compilation target, eclipsing even x86_64 machine code. Or it may not, but JavaScript's desirability and capabilities mean that the phenomenon of writing in languages compiled to JavaScript—meaning most other languages—might only grow in the foreseeable future.

Summary

This chapter may be an early attempt at the kind of endeavor where early attempts often do not succeed. There are seminal documents easily available on the web, but they assume you can not only program but handle a particular kind of math that most professional developers cannot and perhaps have never achieved proficiency. The goal here is not to provide another highly mathematical explanation, but to produce a document that would be useful, perhaps on a less highly exalted plane, to the majority of frontend developers who naturally think of imperative solutions. The goal here is to move towards more functional and less imperative programming, but also to produce a text appropriate to the level of mathematical skill that professional programmers actually have, not the level of math skills some authority might wish them to have. Consequently, the prerequisites are meant not to assume an understanding of category theory itself before programmers are allowed to write, *Hello, world!*

In this chapter, we took a trip down computer folklore's memory lane. This trip looked at a scathing computer checklist, simple *Hello, world!* program, and category theory—and how functional reactive programming's preferred language, Haskell, may want you to use category theory if you're allowed to write, *Hello, world!* This is a major problem.

We've also had a look at the distinguishing features of functional reactive programming including how time is treated. Also a serious answer to the question, "If you learn just one thing from functional reactive programming, what would be the best thing to learn?" is covered in this chapter.

Learning what you can about pure functional development, not what you think you must learn about it, is also covered here. It's easy to paralyze yourself by trying to learn too much functional programming, and honestly, functional programming (which requires that you shift how you look at the world) is not the easiest thing for seasoned imperative programmers to learn. This is an attempt to offer a sane measure of profiting from what functional programming one can without getting completely lost in the almost infinite trackless wastes of grappling with functional programming.

We also discussed the future of web development, where JavaScript is viewed as the new "bare metal."

In our next chapter, we will explore tools to support functional reactive programming. Let's begin!

7
Not Reinventing the Wheel – Tools for Functional Reactive Programming

In this chapter, we will look at a few out of the many good tools for building on top of "bare-metal" JavaScript, as discussed briefly in the last chapter. JavaScript is not interesting only for its properties as a core language; browser JavaScript is home to an ecosystem, or perhaps multiple ecosystems. Regarding tools for functional reactive programming, the total set of offerings represents a good, healthy, and sprawling bazaar, next to which the direct use of JavaScript alone for all web development looks more like a cathedral. We will be taking a small sample of this bazaar, with the understanding that this chapter does not intend to cover all that is good, interesting, or worthwhile. That's very hard to do in a bazaar!

The tools that we will cover include the following:

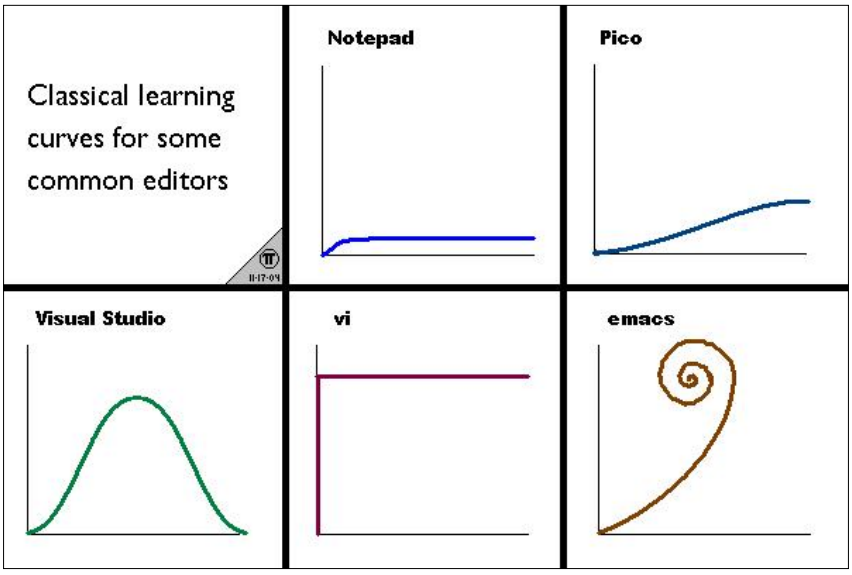
- ClojureScript and Om
- Bacon.js
- Brython
- Immutable.js
- Jest
- Fluxxor

The set of tools we would include, or not include, in a chapter such as this involves drawing lines that are up for, and making judgment calls. Readers interested in a more comprehensive treatment can look at the link compendium at `http://tinyurl.com/reactjs-complementary-tools`, and drill down to the tools that are of interest to their specific concerns. There is a lot there, and probably a lot of gems for almost any purpose.

ClojureScript

ClojureScript, and perhaps Clojure in general, represents an important watershed in software and web development. ClojureScript proves by example that it is possible to have a solid foundation and environment for development in a language other than JavaScript, and this pioneering language is a Lisp dialect. (This is fitting perhaps, for one of the two oldest programming languages in common use. Lisp was good when it came out, and it's still a good language today.) Furthermore, Lisp may enjoy a good advantage compared to JavaScript, and be alive due to some of the same reasons. JavaScript is the language of web browsers and Lisp is the language of Emacs. Also, Lisp offers a sort of proto-JavaScript; before there were web browsers that were programmable in JavaScript, there were Emacsen programmable in Lisp, and someone who said Lisp was the better language in comparison to JavaScript would hardly be contested.

There is good reason to suggest that Lisp, and not the Emacs default key bindings, is responsible for the classical Emacs learning curve in this "classical learning curves" cartoon that has been floating around on the Internet:



As suggested in an earlier chapter, the uniformity of everybody programming directly in JavaScript may give way to a beautiful diversity, or a patchwork quilt. In this beautiful patchwork quilt, JavaScript may still be pre-eminent, but JavaScript's pre-eminence may serve as the new "bare metal." We may have a collection of high-level languages and tools for frontend development. Again, as Alan Perlis said, "A language is low-level when it requires attention to the irrelevant." On those grounds, JavaScript is low-level.

Some of these tools may have a better portfolio of good parts with respect to the bad parts. They may lend themselves to frontend work that may still finally be executed in JavaScript. But they may also open up frontend web development in which new developers are no longer told, "Here is the language that we will use, and here are some large portions of the language that you should try to avoid as much as possible because they are fundamentally toxic." Newer versions of ECMAScript (the formal name for JavaScript, and it is not particularly connected to Emacs) may offer better collections of features, but it is still desirable to work in high-level languages that offer a better terrain for productive work and results.

ClojureScript says, with bells on, that it is possible to have a good high-level language that will run on browsers, and this isn't good news only for Lisp hackers. It is good news for everyone. It demonstrates an open door to the possibility of web development in other high-level languages and potentially a better web development environment with fewer tar pits.

ClojureScript can be used both for client-side work and on the server side with Node.js. *Hello, World!* is as follows:

```
(ns nodehello
  (:require [cljs.nodejs :as nodejs]))

(defn -main [& args]
  (println (apply str (map [\space "world" "hello"] [2 0 1]))))

(nodejs/enable-util-print!)
(set! *main-cli-fn* -main)

(comment
; Compile this using a command line like:

CLOJURESCRIPT_HOME=".../clojurescript/" \
  bin/cljsc samples/nodehello.cljs {:target :nodejs} \
  > out/nodehello.js

; Then run using:
nodejs out/nodehello.js

)
```

Om

Om is a wrapper that makes ReactJS available for ClojureScript. Apart from the fact that ClojureScript is usually fast, a certain part of Om is actually about two times faster than in JavaScript. The difference has to do with identifying changes so as to optimally and appropriately update the DOM when ReactJS does that. The reason is that ReactJS, in its diffing algorithm (by dealing with mutable JavaScript data structures), has to perform a deep comparison to see what, if anything, in the (pure JavaScript) synthetic virtual DOM has changed.

This is still lightning fast compared to direct DOM manipulations, so fast that it's really not the bottleneck for most ReactJS users. But it's faster in Om. The reason is that ClojureScript, like a good functional programming language, has immutable data. You can easily enough get a mutated copy of something, but you cannot tamper with the original or trip up anyone who has access to the original. This means that Om can get by with only comparing top-level references and not digging into the depths of data structures. This is enough to make Om faster than the original JavaScript use of ReactJS. *Hello, World!* in Om is written like this:

```
(ns example
  (:require [om.core :as om]
            [om.dom :as dom]))

(defn widget [data owner]
  (reify
    om/IRender
    (render [this]
      (dom/h1 nil (:text data)))))

(om/root widget {:text "Hello world!"}
  {:target (. js/document (getElementById "my-app"))})
```

Bacon.js

Note that discussing ReactJS and Bacon.js alone does not qualify as an exhaustive list. To mention one alternative suite, Microsoft has tried to create RxJS, RxCpp [Rx for C++], Rx.NET, and Rx* for various JavaScript frameworks and libraries, and they've at least tried to make a polyglot-friendly portfolio for multiple languages and optimized versions for multiple JavaScript frameworks and libraries. There is really a lot available that offers some form of functional reactive programming. And while most of the few (at the time of writing this book) functional reactive programming and ReactJS resources on the Web are golden, there are some that aren't.

Andre Stalz writes:

> *"So you're curious in learning this new thing called Reactive Programming, particularly its variant comprising of Rx, Bacon.js, RAC, and others.*
>
> *Learning it is hard, even harder by the lack of good material. When I started, I tried looking for tutorials. I found only a handful of practical guides, but they just scratched the surface and never tackled the challenge of building the whole architecture around it. Library documentations often don't help when you're trying to understand some function. I mean, honestly, look at this:*
>
> *Rx.Observable.prototype.flatMapLatest(selector, [thisArg])*
>
> *Projects each element of an observable sequence into a new sequence of observable sequences by incorporating the element's index and then transforms an observable sequence of observable sequences into an observable sequence producing values only from the most recent observable sequence."*

I now understand what the quotation is saying, but that's because I've learned from other resources that have communicated better. Part of the intent behind this book you are reading is to make good documentation a little less difficult to understand.

There is a famous question in the open source community: would you buy a car with the hood welded shut? ReactJS can be described as a car that most people can drive without opening the hood. It's not that ReactJS is closed-source, or that Facebook shows any signs of making it harder to read as much of the source code as you want. But to pick one salient example, **Denotative continuous-time semantics** was part of Conal Elliott's second thoughts about what would be a better name for what is now called functional reactive programming. Whether one agrees or disagrees with his suggestion for a better and more descriptive name, such second thoughts from a leading light can be very insightful and illuminating. And with ReactJS, if it's working correctly, a novice programmer can be given the same explanation that Calvin's father (a patent attorney!) in Calvin and Hobbes gives when Calvin asks how a lamp, or a vacuum cleaner, works—*It's magic!* Looking at a newcomer's question, "How is continuous time handled?" the reply is *It's magic!* "How can you get away with discarding and recreating the DOM every single time?"—*It's magic!*; "But how does ReactJS achieve an astonishing 60 fps on a non-JIT iPhone?"—*It's magic!*

Functional reactive programming describes certain tasks that need to be accomplished, such as the appropriate handling of event streams, but the ReactJS documentation doesn't seem to explain how to address this handling because the responsibility is offloaded to ReactJS; *It's magic!*

Not only does Bacon.js not weld the hood shut, but you are also expected to tinker under the hood. Bacon.js seems closer to the roots of basic functional reactive programming. Some programmers, intending to work in ReactJS, might find it profitable to "lift weights" a bit and strengthen themselves with Bacon.js. One significant area of functional reactive programming is dealing with emitted streams of events, and as far as ReactJS goes, *It's magic!*

In Bacon.js, it is in fact not magic that all is done without you ever moving a finger; it is something that the programmer needs to work out, and they are given good tools to do so. On these grounds, it could help form a developer for a solid reactive programming foundation to use ReactJS. If the selling point of ReactJS is that it is a tool optimized to allow good user interface work while drawing on the strengths of functional reactive programming, the selling point of Bacon.js is that it is a tool in JavaScript that is optimized for (learning and) performing solid functional reactive programming in theory and practice as a whole.

The difference between ReactJS and Bacon.js doesn't seem to be a matter of unearthing that one framework is simply better than the other. It is rather a matter of taking stock of what you want to do and accomplish, recognizing that ReactJS and Bacon.js (besides being worthy competitors) have different areas where they really shine, and deciding whether your work sounds more like a ReactJS sweet spot or a Bacon.js sweet spot. Moreover, with respect to the topic of sweet spots, Bacon.js (unlike ReactJS) has a name meant to make your mouth water, and the ~ functional operator is called "bacon" in the references.

Brython – a Python browser implementation

Brython (`http://brython.info`), a browser and Python implementation, is another example of an alternative to programming a browser in JavaScript, and while it would be a bit unfair to Brython to call it merely experimental, it is also not necessarily appropriate to call it mature—certainly not in the same sense that ClojureScript has some significant maturity. ClojureScript is developed well enough to essentially replace "bare metal" JavaScript for a frontend developer who'd really prefer to use Lisp rather than JavaScript. In other words, unless we are talking about something performance-critical or possibly special cases, there aren't too many cases where ClojureScript experts would answer the question, "How do I do this in ClojureScript?" with, "Use JavaScript directly for this kind of problem." Brython is included not because the sun rises or sets on Python but as an illustration that Lisp in ClojureScript is not a fundamental exception in terms of being the only non-JavaScript language that works for frontend web development, but perhaps the first of many.

Brython is meant for world domination. Its home page boldly announces, "Brython is intended to replace JavaScript as the scripting language for the Web," and it will, perhaps, never reach that quite naïve goal. Brython takes surprisingly long to load and is slow to run after it has loaded. It may be better to use one of the Python-to-JavaScript compilers (which would be closer to ClojureScript), but Brython really offers quite a lot of Python's goodness and may be someday be seen as significant. Yet, I would suggest that it is silly to try to be the next JavaScript and take the place of every other transpiler that renders JavaScript.

In Brython, the goal of world domination also lends itself to a blind spot: failing to see how important it is to be able to interoperate with tools written in other languages. But the good news is that Brython or other Python-to-JavaScript approaches may be significant without needing to become the "One Language to Rule Them All." Python is hardly the only backend language available, but it's a good player, and there is every reason for good implementations of Python to be worthwhile players in a patchwork quilt composed of multiple languages that can all be used profitably for frontend web development.

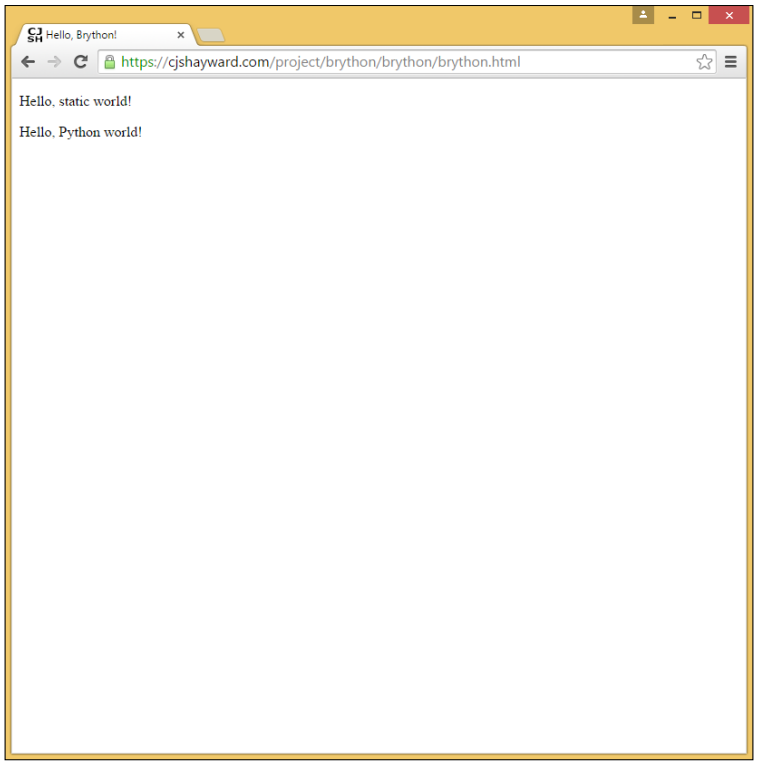
Furthermore, at least a *Hello,World!* program with ReactJS is straightforward to implement in Brython. A *Hello, World!* program, as run after gathering Brython and ReactJS on a page, includes first JavaScript (not JSX) commented out, and then the Python code that calls React in the browser via Brython:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello, Brython!</title>
    <script src="brython.js"></script>
    <script src="react.js"></script>
  </head>
  <body onload="brython()">
    <p>Hello, static world!</p>
    <div id="dynamic"></div>
    <!--
      <script type="text/javascript">
        React.render(
          React.createElement('p', null,
          'Hello, JavaScript world!'),
          document.getElementById('dynamic')
        );
      </script>
      -->
    <script type="text/python3">
      from browser import document, window
```

```
        window.React.render(window.React.createElement(
          'p', None, 'Hello, Python world!'),
          document['dynamic']);

    </script>
  </body>
</html>
```

This displays what is shown here:



Note that the entire first script tag and contents, and not just the JavaScript inside them, are in an HTML comment. This means that the first (JavaScript) script, shown here for clarity, is not active, the second (Python) script being the one that runs and displays its message.

The second script is an interesting one; the included Python code is equivalent (apart from the message) to the commented-out JavaScript text and does the same thing. This is quite a feat, especially when combined with how Brython successfully implements most features in the 3.x branch of Python. Even if Brython is looked into for a project and deemed not the right solution, it remains an achievement.

In one sense, Brython is presented here as an example of a possibility rather than, in any sense, the only member of its class that is worth attention. The point is not specifically that Python can be used for frontend development; it's that ClojureScript Lisp may not be the only additional language besides JavaScript that is available for use in frontend development.

Immutable.js – permanent protection from change

Immutable.js, with its home page at `http://facebook.github.io/immutable-js` and a tagline of **Immutable collections for JavaScript**, was originally named for persistence. Then it went through a name change to something that would register more quickly by referring to the immutable. Immutable.js plugs a gap in JavaScript as a functional language and offers significantly more functional-friendly data structures for collections (which is the point for which it was created).

It offers data structures for collections that are immutable. They support the creation of modified copies gracefully enough, but it is always the copy that is changed, never the original. Though this is more of a minor point, it greatly reduces the need for "defensive copying" and related workarounds for not using immutable data where there are multiple programmers. The original code could be chugging along with a different and modified copy of the data structure that you want, but your copy, which you have kept as a reference, is guaranteed to be entirely untouched. The library is intended to support other niceties, such as easy conversion to and from staple JavaScript data structures.

However, data structures of Immutable.js are not only immutable; they are also lazy in some aspects, and the document clearly marks which aspects of an application are eager. (as a reminder, lazy data structures are treated in a print-on-demand fashion when needed, while eager operations are done at once and upfront). Furthermore, certain functional idioms are baked into Immutable.js facilities. For instance, there is a `.take(n)` method offered. It returns the first *n* items of a list in the classic functional fashion. Other functional staples, such as `map()`, `filter()`, and `reduce()`, are also available. In general, runtime complexity is as good as a computer scientist could reasonably request.

There are several data types provided by Immutable.js; these include the following (the descriptions in this and the next table are partly based on the official documentation):

Immutable.js class	**Description**
`Collection`	This is an abstract base class for Immutable.js data structures. It cannot be directly instantiated.
`IndexedCollection`	A collection that represents indexed values in a particular order.
`IndexedIterable`	This is an iterable with indexed numeric keys that support some array-like interface features, such as `indexOf()` (An iterable is something that you can iterate through like a list, but it may or may not be a list in the internals).
`IndexedSeq`	A `Seq` that supports an ordered indexed list of values.
`Iterable`	A set of (key and index) values that can be iterated through. This class is the base class for all collections.
`KeyedCollection`	A collection that represents key-value pairs.
`KeyedIterable`	An iterable with discrete keys tied to each iterable.
`KeyedSeq`	A sequence that represents key-value pairs.
`List`	An ordered collection, somewhat like a (dense) JavaScript array.
`Map`	A keyed iterable of key-value pairs.
`OrderedMap`	A map that does everything that a map does and, in addition, guarantees that iterations will produce keys in the order in which they are set.
`OrderedSet`	A set that does everything that a set does and, in addition, guarantees, that iterations will produce values in the order in which they are set.
`Record`	A class that produces concrete records. Conceptually, this is different from the other records here. The other elements are, conceptually, collections of "whatnots," perhaps objects that have a similar structure. The `Record` class is closer to the records one encounters in school, where one record is similar to a row in a database table, while result sets or tables are more like container objects.
`Seq`	A sequence of values, which may or may not be backed by a concrete data structure.
`Set`	A collection of unique values.

Immutable.js class	Description
`SetCollection`	A collection of values without keys or indices.
`SetIterable`	Iterables representing values without keys or indices.
`SetSeq`	A sequence representing a set of values.
`Stack`	A standard stack with `push()` and `pop()`. Semantics always go to first element, unlike JavaScript arrays.



`Record` is slightly different from the others; it is similar to a JavaScript object that meets certain criteria. The other elements are related container classes that provide functional access to some collection of objects and tend to have a similar list of methods supported.

The methods for List, to pick one example, include the following:

Immutable.List method	Description
`asImmutable`	A function that takes a (mutable) JavaScript collection and renders an Immutable.js collection.
`asMutable`	This is a concession to "not-the-best" programming. The proper way to handle mutations based on an Immutable.js collection is to use with Mutations. Even if `asMutable` is available, it should only be used inside of functions and never be made available or returned.
`butLast`	This produces a similar new List, but it lacks the last entry.
`concat`	Concatenate (that is, append) two iterables of the same type.
`contains`	This is true if the value exists in this List.
`count`	Return the size of this List.
`countBy`	Group the List's contents with a grouper function, and then emit counts for the keys as partitioned by the grouper.
`delete`	Create a new List without this key.
`deleteIn`	Remove a key from a keypath, which allows traversal from an outer collection to an inner collection, much like the way a filesystem path allows traversal of the filesystem from an outer directory to an inner directory.
`entries`	An iteration of the List as `key`, `value tuples`.
`entrySeq`	Create a new `IndexedSeq` of key, value tuples.
`equals`	This is a full equality comparison.

Immutable.List method	Description
`every`	This is true if a predicate is true for all entries in this List.
`filter`	Returns the elements of a List for which the provided predicate holds true.
`filterNot`	Returns the elements of a List for which the provided predicate returns false.
`find`	Returns the value for which the provided predicate holds true.
`findIndex`	Returns the first index for which the provided predicate holds true.
`findLast`	Returns the last element for which the provided predicate holds true.
`findLastIndex`	Returns the last index for which the provided predicate holds true.
`first`	The first value in the List.
`flatMap`	This flat-maps, or collapses, a potential List of Lists into a List that is one deep.
`flatten`	This flattens nested iterables.
`forEach`	Executes a function for each entry in the list.
`fromEntrySeq`	Return a `KeyedSeq` of any iterable of the key, value tuples.
`get`	Returns the value for a key.
`getIn`	Traverses a key path (like a filesystem path) to get a key, if available.
`groupBy`	Converts a List into a List of Lists keyed by the grouping of a provided grouper function.
`has`	This is true if a key exists in this List.
`hashCode`	This calculates a hash code for this collection. It is appropriate for use in a hash table.
`hasIn`	This is true if a collection's equivalent to a filesystem walk finds the value in question.
`indexOf`	The index of the first occurrence in this List, for example, `Array.prototype.indexOf`.
`interpose`	Interpose a separator between individual List entries.
`interleave`	Interleave the provided Lists into one list of the same type.
`isEmpty`	This tells whether this iterable has values or not.
`isList`	This is true if the value is a List.

Immutable.List method	Description
`isSubset`	True if every value in the comparison iterable is in this List.
`isSuperset`	True if every value in this List is in the comparison iterable.
`join`	This joins together as a string with a separator (default).
`keys`	An iterator of this List's keys.
`keySeq`	Returns a `KeySeq` for this iterable, discarding all values.
`last`	The last value in the List.
`lastIndexOf`	Returns the last index at which a value can be found in this List.
`List`	The constructor for lists.
`map`	Returns a new List with values passed through a map function.
`max`	Returns the maximum value in this collection.
`maxBy`	This is like max, but with more fine-grained control.
`merge`	Merges iterables or JavaScript objects into one List.
`mergeDeep`	A recursive analog to merge.
`mergeDeepIn`	Performs a deep merge, starting at a given keypath.
`mergeDeepWith`	This is like `mergeDeep`, but uses a provided merger function when nodes conflict.
`mergeIn`	This is a combination of update and merge. It performs a merger at a specified keypath.
`mergeWith`	This is like merge, but uses a provided merger function when nodes conflict.
`min`	Returns the minimum value in the List.
`minBy`	Returns the minimum value in the List as determined by a helper function you provide.
`of`	Creates a new list containing its arguments as values.
`pop`	This returns everything in this List but the last. Note that this differs from the standard push semantics, but a regular `push()` can be simulated by calling `last()` before `push()`.
`push`	Returns a new list with the specified value (or values) appended at the end.
`reduce`	Calls the reducing function for every value and returns the accumulated value.

Immutable.List method	Description
`reduceRight`	This is similar to reduce, but starts at the right and moves progressively to the left, the opposite of the basic reduce.
`rest`	Returns the tail of a List, that is, all entries but the first.
`reverse`	Provides a List in reverse order.
`set`	Returns a new List with the value at the index.
`setIn`	Return a new List with this value at the keypath.
`setSize`	Creates a new List with the size that you specify, truncating or adding undefined values as needed.
`shift`	Creates a new list with the first value subtracted and all other values shifted down.
`skip`	Returns all that is left of the List when the first *n* entries are not included.
`skipLast`	Returns all that is left of the List when the last n entries are not included.
`skipUntil`	Returns a new iterable containing all entries after the first where a provided predicate is true.
`skipWhile`	This returns a new iterable containing all entries before a provided predicate is false.
`slice`	Returns a new iterable containing this list's contents from the start value to one before the last, inclusive.
`some`	True if a predicate returns true for any element of the List.
`sort`	Returns a new List sorted by an optional comparator.
`sortBy`	Returns a new List sorted by an optional comparator value mapper, with more detailed information available for the comparator and therefore more refined results.
`splice`	Replaces a segment of the first list with the second, or deletes it if no second list is provided.
`take`	Creates a new List containing exactly the first n entries in a List.
`takeLast`	Creates a new List containing exactly the last *n* entries in a List.
`takeUntil`	This returns a new List containing all entries as long as the predicate returns false; then it stops.
`takeWhile`	This returns a new List containing all entries as long as the predicate returns true; then it stops.
`toArray`	Shallowly converts this List to an Array, discarding the keys.

Immutable.List method	Description
`toIndexedSeq`	Return an `IndexedSeq` of this List, discarding the keys.
`toJS`	Deeply converts this List into an Array. This method has `toJSON()` as an alias, although the documentation does not clearly state whether or not `toJS()` returns JavaScript objects, while `toJSON()` returns a JSON-encoded string.
`toKeyedSeq`	Returns a `KeyedSeq` from this List where indices are treated as keys.
`toList`	Returns itself.
`toMap`	Converts this List into a Map.
`toObject`	Shallowly converts this List into an Object.
`toOrderedMap`	Convert this List into a Map, preserving the order of iteration.
`toSeq`	Returns an `IndexedSeq`.
`toSet`	Converts this List to a Set, discarding the keys.
`toSetSeq`	Converts this List to a `SetSeq`, discarding the keys.
`toStack`	Convert this List to a Stack, discarding the keys.
`unshift`	Prepend the provided values to a List.
`update`	Update an entry on a List by a provided updater function.
`updateIn`	Update an entry, as in update(), but at a given key path.
`values`	An iterator of this List's values.
`valueSeq`	An `IndexedSeq` of this List's values.
`withMutations`	This is an optimization (recall "Premature optimization is the root of all evil," said by Donald Knuth) hook meant to allow more performant work when multiple mutations are performed. It is to be used when there are known and persistent performance issues where other tools have demonstrably not solved the problem.
`zip`	Returns an iterable zipped (that is, joined pairwise to make a list of 2-tuples) with this List.
`zipWith`	Returns an iterable zipped with a custom zipping function.

The documentation for the API, which is under the **Documentation** link available on the home page, is pretty clear. But as a rule, Immutable.js collections do what a functional programmer would expect them to do as much as possible, and indeed there appears to be a presumable overriding design consideration of "do what a functional programmer would want as much as we can."



One thing that might be an unpleasant surprise to functional programmers is that the documentation does not explain how to create infinite lists. It is not obvious how one might create a generator for a list (if they do so at all), or produce lists of mathematical sequences, such as all counting all numbers, positive even numbers, squares, primes, Fibonacci numbers, powers of 2, factorials, and so on. Such functionality is apparently not supported (at the time of writing this book). Lazy sequences cannot build infinite lists with Immutable.js because constructing a collection includes an eager inclusion of all elements ever in the list, which must therefore be finite. It shouldn't be terribly difficult to create lazy, and potentially infinite, data structures in the style of Immutable.js that have a memoized generator inside and allow you to XYZ.take(5). But Immutable.js appears not to have expanded into that territory yet.

Jest – BDD unit testing from Facebook

Jest is a JavaScript unit testing framework intended to support behavior-driven development. It is built on top of Jasmine, and in the future, it may be able to interact with other foundations. It has been used for a couple of years and is used on Facebook, though there appears to be no decisive endorsement that ReactJS development is best done using Jest. (Facebook uses JSX with ReactJS internally but tends to make a relatively unopinionated statement that about half of ReactJS users opt to use JSX. It is actually designed to be entirely optional.)



JSX—the *X* boldly meaning XML at a time when XML has fallen out of favor—is a well-made syntactic sugar that "puts angle brackets in your code." This loosely means that you can put HTML into your JavaScript in `.jsx` files and things just work. Also, you can use almost anything that you can build on a page that has been built into a ReactJS component. You can include things such as images that have been part of HTML from the beginning, and you can just as easily include a calendar as defined in this title, a threaded networked discussion, or a draggable and zoomable fractal. Like subroutines, once a component is defined, it can be used zero, one, or many times anywhere in your web app. The JSX syntactic sugar allows components that you and others have defined as easily as old HTML tags. The JSX for the outer shell in the project in chapters 8 to 11 is "dirt simple" in terms of allowing us to incorporate the other components that we have developed:

```
var Pragmatometer = React.createClass({
  render: function() {
    return (
      <div className="Pragmatometer">
      <Calendar />
      <Todo />
      <Scratch />
      <YouPick />
      </div>
    );
  }
});
```

One Facebook employee said that he had made Jest open source for "selfish reasons," namely that he wanted to use it in his personal projects. This may give a good hint about why Jest is at least worth considering. It is nice enough for at least one user to really want to use Jest, so much so that he was willing to make proprietary intellectual property open source, and did this even when no one told him to.

It is arguable that in its beginnings, unit testing has served what is most easily unit tested, which means that unit testing has washed its hands of integration and user interface testing. So, you might see a blog article on unit testing that tests and confirms a "red, green, refactor" approach to a function that converts your language's integers to Roman numerals, which is a pretty good example of a problem that serves the needs of primitive unit testing. If you want to test whether your code is interacting with its database appropriately, that's a slightly taller order. And Jest, like other frameworks, doesn't really have which is pretensions of obviating the need for good, old-fashioned budget usability testing as Jakob Nielsen and others advocate. There is an old (pre-IT) business distinction between asking, "Are we building the product right?" and "Are we building the right product?"

Both of these questions are valuable and have their place, but unit testing helps more with the first than the second, and it is dangerous to let a good test suite that addresses the first question well lull you to sleep which is regarding addressing the second question well. Nonetheless, Jest offers something more useful than just testing whether a unit of code will successfully take input of a primitive data type, such as an integer, a float, or a string, and return the correct and expected output of a primitive data type (such as the right Roman numeral for an input integer). Though this is not true only for Jest, Jest simulates the user interface so as to support (for instance) user interface events, such as clicks on an element, and supports testing user interface changes, such as the text on a label (compare the Jasmine home page, where the first several examples involve assertions using primitive data types only).

Jest is intended to provide layers on top of Jasmine (and potentially other backends in the future), but with significant added value. Besides certain features, such as running tests in parallel so that testing becomes more responsive, Jest is a solution intended to require a minimum amount of time and fuss to get good test coverage, based on the thought that it is desirable for developers to spend most of their time on their main development and not on writing unit tests.

Jest is intended to mock everything, or almost everything, pulled in with `require()`. You can opt out for individual elements by calling `jest.dontMock()`, and it is boilerplate practice that tests usually call `jest.dontMock()` for the components that they are testing. It automatically finds and runs tests in the `__tests__` directory. Jest can handle JSX if ReactJS's preprocessor is included in, for example, `preprocessor.js`:

```
var ReactTools = require('react-tools');
module.exports = {
  process: function(source) {
    return ReactTools.transform(source);
  }
};
```

The `package.json` file needs to be told what to load:

```
'dependencies': {
  'react': '*',
  'react-tools': '*'
},
}, 'jest': {
  'scriptPreprocessor': '<root directory>/preprocessor.js',
  'unmockedModulePathPatterns':
    ['<root directory>/node_modules/react']
},
```

Now we will lightly adapt Facebook's example. Facebook provides an example `CheckboxWithLabel` class. This class displays one label when the checkbox is unchecked and another when it is checked. The Jest unit test here simulates a click and confirms that the label changes appropriately.

The `CheckboxWithLabel.js` file reads as follows:

```
/** @jsx React.DOM */

var React = require('react/addons');
var CheckboxWithLabel = React.createClass({
  getInitialState: function() {
    return {
      isChecked: false
    };
  },
  onChange: function() {
    this.setState({isChecked: !this.state.isChecked});
  },
  render: function() {
    return (
      <label>
        <input
          type="checkbox"
          checked={this.state.isChecked}
          onChange={this.onChange}
        />
        {(this.state.isChecked ?
        this.props.labelOn :
        this.props.labelOff)}
      </label>
    );
  }
});

module.exports = CheckboxWithLabel;
```

The `__tests__/CheckboxWithLabel-test.js` test file reads:

```
/** @jsx React.DOM */

jest.dontMock('../CheckboxWithLabel.js');

describe('CheckboxWithLabel', function() {
  it('changes the text after click', function() {
```

```
    var React = require('react/addons');
    var CheckboxWithLabel = require('../CheckboxWithLabel.js');
    var TestUtils = React.addons.TestUtils;

    // Verify that it's Off by default.
    var label = TestUtils.findRenderedDOMComponentWithTag(
      checkbox, 'label');
    expect(label.getDOMNode().textContent).toEqual('Off');

    // Simulate a click and verify that it is now On.
    var input = TestUtils.findRenderedDOMComponentWithTag(
      checkbox, 'input');
    TestUtils.Simulate.change(input);
    expect(label.getDOMNode().textContent).toEqual('On');
  });
});
```

Implementing the Flux Architecture using Fluxxor

As mentioned in earlier chapters, Flux is an architecture that was developed by Facebook and used by them as something largely complementary to ReactJS. It helped untangle a genuine rat's nest of crossed wires and let Facebook eradicate a recurrent message count bug that kept coming back—the Flux Architecture killed it permanently. **Fluxxor**, by Brandon Tilley (`http://fluxxor.com`), is a tool intended to help people implement the Flux Architecture in their applications. There is no need to use the Flux Architecture in order to use ReactJS, or use the Fluxxor tool to implement the Flux Architecture. But Flux, and perhaps Fluxxor, is at least worth considering to make things easier.

Fluxxor has classes for the Flux Architecture overall, including a `Fluxxor.Flux` container (which includes a dispatcher) and `Action` and `Store` classes. The sample code was concise and readable and seemed to have little boilerplate. Two ReactJS-friendly mixin classes are provided for ease of use. Sample code is written using JSX.

I might also comment with reference to Fluxxor's author that `http://fluxxor.com` has a link at the bottom of the page asking people to report an issue on GitHub if something is unclear or if there is an issue. I noticed a common usability defect—visited and unvisited links being the same color—and reported the issue on GitHub. The author apologized immediately, and the issue I opened was *Closed fixed* within no more than 15 minutes. I think he is the kind of person one likes to work with.

Summary

Now let's see what we covered in this chapter. We explained Om and ClojureScript, which allow Lisp-based development that takes advantage of ReactJS's abilities. It is said that ClojureScript may be the leading light of solutions that allow a beautiful patchwork of different languages that are usable for frontend development, compiling, or interpreting JavaScript as the new "bare metal."

Bacon.js is a very respectable technology that competes with ReactJS that allows good functional reactive programming in the browser. This is presented, not as the "one and only" good example, but as an example of good stuff that is beyond the scope of this book.

We also covered Brython, a browser-based Python environment. It is not perfect but interesting. It is highlighted as an example of what can be used as a language outside of Lisp other than JavaScript for web development. As a reminder, `http://tinyurl.com/reactjs-compiled-javascript` offers a directory of other languages that compile to JavaScript or can be interpreted in a web browser, ranging from syntactic sugar such as CoffeeScript to JavaScript extensions to separate languages such as Ruby, Python (including Brython), Erlang, Perl, and so on and so forth.

Immutable.js plugs a hole in functional JavaScript by providing mainly collections that allow copy-on-mutate without disrupting the functional advantages of immutable data.

Jest is a behavior-driven development JavaScript unit testing framework that is used by Facebook for ReactJS. Fluxxor is an implementation of controllers, actions, and stores intended to make it easier to apply the Flux Architecture to JavaScript development, including ReactJS.

Join us in the next chapter as we explore a more in-depth example using ReactJS.

8
Demonstrating Functional Reactive Programming in JavaScript – A Live Example, Part I

In *Chapter 4*, *Demonstrating Nonfunctional Reactive Programming – A Live Example*, we used ReactJS to migrate from the legacy code that has its own structure and was written without using ReactJS. In the last chapter, *Chapter 7*, *Not Reinventing the Wheel – Tools for Functional Reactive Programming*, we studied a few out of a great many tools that we might use when working with ReactJS. In this chapter, we will be covering a sort of central road of what to expect in mainstream development with ReactJS. One may add quite a lot of options to the basics, but the intent is to give a foundational example of how you can build a project with ReactJS.

We've spoken a bit about functional reactive programming. Now we will see it live in action with ReactJS. We've also talked about how, conceptually, we have a complete teardown and rebuild of the user interface. So you, as a developer, have $\mathcal{O}(n)$ states to manage instead of $\mathcal{O}(n \wedge 2)$ state transitions. Here, we will build a `render()` method that will let you build just this, and you can call it whenever you want.

In this chapter, we have the first installment of a partly stubbed green field project built for ReactJS, this time working with the very sweet syntactic sugar in JSX. The two areas of this book, meaning the earlier one-chapter project and this multichapter project, are meant to be complementary. The project in this chapter stands alone, but is meant to be expanded.

In this chapter, we will cover the following topics:

- An overview of the project and its inspiration
- The skeleton of the project, and the basics of the preferred approach in ReactJS.
- Starting a first component in ReactJS
- Building a `render()` method
- Triggering display when you want to render or update the display

What we will be attempting in this chapter

The example in the next three chapters is intended to represent a slightly larger green field project. What we will be working on is a system that you should be able to see by visiting `http://demo.pragmatometer.com`. The term "Pragmatometer" is taken from the most dystopian of C. S. Lewis's novels, *That Hideous Strength*, in which the ominous National Institute for Coordinated Experiments builds a transcendent or nearly transcendent computer such as one might have loosely imagined ENIAC when the novel was published (1945; by comparison, ENIAC was created in 1946). Alternatively, you might imagine a steampunk novel's analytical engine doing with a seemingly transcendent deck of punch cards. When the discussion turns to the computers, it says:

> *"I agree with James," said Curry, who had been waiting somewhat impatiently to speak. "The N.I.C.E. marks the beginning of a new era – the really scientific era. Up to now, everything has been haphazard. This is going to put science itself on a scientific basis. There are to be forty interlocking commitees sitting every day and they've got a wonderful gadget – I was shown the model last time I was in town – by which the findings of each committee print themselves off in their own little compartment on the Analytical Notice-Board every half hour. Then, that report slides itself into the right position where it's connected up by little arrows with all the relevant parts of the other reports. A glance at the Board shows you the policy of the whole Institute actually taking shape under your own eyes. There'll be a staff of at least twenty experts in the top of the building working this Notice-Board in a room rather like the Tube control rooms. It's a marvellous gadget. The different kinds of business all come out in the Board in different coloured lights. It must have cost half a million. They call it a Pragmatometer."*

I am not putting too fine a point on it, but C.S. Lewis apparently predicted the Twitter that would not be built until decades after his death.

That point aside, we will be making a dashboard that features a simple 2 x 2 grid of quadrants (the exact size and details are subject to hacking and tinkering), each one of which is a pigeonhole that can hold different functions. In terms of responsive design, we will correct to a 1 xn line of cells, one above the other. The features, as they are arranged on the page, are as follows:

Calendar	To-do list
Scratchpad	Room to grow

The **Calendar** has a somewhat experimental user interface; it shines at showing entries, perhaps days out, in a way that degrades gracefully to a sparse input (so you don't need to click through multiple months to find out when some XYZ appointment down the road is available). It might click with you, or it might not, but it is interesting.

The **To-do list** implements a to-do list with some slightly nonstandard bells and whistles. Instead of one checkbox for an item (strictly speaking, no checkbox is needed), there are ten boxes, representing various states, and color-coded with custom styling of the label to the right of the checkboxes so that you can tell, for instance, what is important, active, or on the back burner.

The **Scratchpad** is a rich text area that can be used for scratching. It capitalizes on CKeditor.

Finally, the **Room to Grow** is a placeholder for you to stick in your own two cents. There is something that comes in by default, and you can see it as you explore. But please do visit `https://demo.pragmatometer.com`, and see the default option there (*hint, hint!*). There are many places for hacking and tinkering besides what is explicitly advertised, but some possibilities include the following:

- **Building an API client for several public websites**: Most of the top 20 websites expose a RESTful API. Twitter would be the most obvious candidate as the most authentic to the name of *Pragmatometer*, but news, video, and social networking sites can work if they expose either an API that is friendly to client-side JavaScript, or something else.
- **Building an application**: Build your own application for displaying.
- **Making a playground**: Build your own Pragmatometer or download the source online, and keep three-fourths of the screen for the purposes detailed here. Make the remaining one-fourth a playground for tinkering.

- **Incorporating Google (or other) gadgets that others have made**: You can also incorporate gadgets that others have made, such as Google.
- **Keeping the default application**: You can keep the default application, if you want.

Saying "*the implementation is the spec*" has a bad reputation, but a spec, written or not, can be complemented neatly by sketching the appearance and behavior. Perhaps using low-fidelity prototypes, which may draw helpful criticism more quickly than something that looks pretty polished but will create an undesired social nuance of people being less free in criticisms that have an implicit emotional payload of "Will I be hurting others' feelings by criticizing something that looks as if someone put a lot of work into it?" And this attitude is not a bad thing. In terms of the concern that undergirds politeness, it is naïve to tell people, "be as brutal as it takes to speak your mind," and actually expect other people to take this completely at face value (or perhaps, you don't like receiving criticism any more than anyone else, but you recognize its value in the whole of the software development process). You may mean it without any mixed messages, but most of us have seen it meant with many mixed messages. And even if you are the sort of production who almost salivates at the prospect of some really helpful criticism, it does not help matters—or does not help much—to tell people to stop acting like polite human beings when you show them something you've created. But here, having seen a UI, played with it, and thought how you can duplicate things can be a very invigorating way to understand what is intended much more precisely than a legal-contract spec.

For this interface, there is one account, and the priority is for cross-synchronization of updates. Let's begin assembling some of the basic skeleton. Building everything within a single, large, and immediately invoked function expression, we have the following:

```
var Pragmatometer = React.createClass({
  render: function() {
    return (
      <div className="Pragmatometer">
        <Calendar />
        <Todo />
        <Scratch />
        <YouPick />
        </div>
    );
  }
});
```

Here, we define a `container` class for the entire project. The `Calendar`, `Todo`, `Scratch`, and `YouPick` classes represent applications in the larger page; some of them could have various layers of subcomponents as well.



The JSX optional syntactic sugar is meant to be readable to people who can read HTML, but contains a greater invitation to contribute your own components than HTML or really even XHTML (and possibly even XML). In XML development, you can define whatever DTD you want, but the usual rank-and-file XML author won't define new tags, possibly even after having done a lot of work with XML (this is something like programmers who can use functions or objects but not add either functions or objects to a namespace). In JSX, it is routine for authors who write any appreciable amount of JSX, by nature, to contribute tags that are reusable.

The preceding code sample has `<div className=`, where the desired HTML is `<div class=`. Because JSX compiles JavaScript and is meant to be a syntactic sugar only, rather than an independent language in its depths, the decision was made to avoid `class` and `for` in the rendered JavaScript. Use `className` to cover CSS class names and `htmlFor`. HTML ID attributes may optionally be specified; JSX gets along with the HTML IDs it puts in and the HTML IDs that you specify, along with some pixie dust. If you need to enter the UTF8 literals outside of ASCII, don't give the ASCII encoding of the symbol (`—`); instead, paste the literal inside your editor (—).

Furthermore, there is an XSS protection escape hatch available. The optimal intended approach for using the language seems to be to solve the problem so that you will mark up what should genuinely be marked up and include an XSS-protected display of the user data.

Alternately, if you are willing to trust a third-party library, such as `Showdown` (`https://github.com/showdownjs/showdown`), to render HTML without including XSS's vulnerabilities, you can create a `Showdown` converter:

```
var converter = new Showdown.converter();
var hello_world = React.createClass({
  render: function() {
    var raw_markdown = '*Hello*, world!';
    var showdown_markup = converter.makeHtml(raw_markdown);
    return (
      <div dangerouslySetInnerHtml={{__html:
        showdown_markup}}>
      </div>
    );
  }
});
```



Note that this code is a self-contained example and not part of the main project that we have begun in this chapter.

This will render as a `DIV` variable containing `<em>Hello</em>, world!`.

This escape hatch, which may or may not serve as a rarely used escape hatch, is central enough to be explicitly covered in the documentation. One gets the feeling that the mainstream use of JSX, with XSS protection that escapes HTML (so that `<em>a</em>` renders in a web page not as `a` but as an `<em>a</em>` function displayed in the browser window), has benefits similar to unit testing. This is a point worth pausing on for a moment.

Unit testing has come somewhat further down to earth; its early center of gravity was around what was easiest to unit test—mathematical functions that would return an appropriate value if given an appropriate input value. So, we would illustrate unit testing with problems implicitly optimized to play to the strengths and needs of unit testing, and show a red, green, refactor XP approach to appropriately solve a problem such as converting integers to and from Roman numerals (good luck if you wanted to test code that deals with a database or user interface). Unit testing caught perhaps 30 percent of the total errors, and as a rule, it tended to catch the least significant errors, with the least coverage of the bugs that were the most difficult to resolve. Now there is a more robust set of features, with it being entirely possible and straightforward to make test assertions about how a user interface behaves given some mouse clicks and other user interface behavior. Also, it is less of a matter of software that is written to serve the needs of unit testing than unit testing that properly serves the needs of software. It is possible that unit testing hit prime time when it was not yet ready for prime time, like responsive design, about which some have said, "I've seen responsive design mainly on websites advocating for responsive design." This was true when unit testing and responsive design became buzzwords; but since then, they have matured, and responsive design is pretty close to being the only game in town. Perhaps the largest websites, such as Google, can afford to customize a solution for every mobile, tablet, desktop, and watch environment. But for most customers, responsive design has fairly effectively displaced its other competitors. It is now relatively uncommon for websites to have some URLs for a desktop version and a mobile version and perform browser direction and redirection to separate sites, which was once fairly mainstream. These approaches have matured since they first entered the limelight.

In the case of early unit testing, when you couldn't really test integration or user interface behavior, there was one cardinal payoff to writing code for unit testing: code written to be unit testable is usually better code. The same principle may speak about writing as much as possible to cut with Facebook's grain, not against it, in using ReactJS.

Right now, there is no particular premature hype about writing code in an approach oriented to play well with the XSS protection surrounding JSX and not, as per the deliberately chosen name, dangerously set `innerHTML`, which has been labeled an escape hatch. Facebook could, if they wanted, try to go the "tough love" route and advise people to structure and organize projects in a way that would fit in naturally with XSS protection and JSX. But perhaps they are taking a humbler approach, both making it clear how to bypass the XSS protection, and presenting this escape hatch as perhaps something to avoid wherever possible.

The wisdom would appear to be as follows:

- As far as practicable, structure your applications to work appropriately with the main anti-XSS approach adopted in ReactJS.
- If you want to do something that requires rendering, for example, HTML tags in `innerHTML`, confine it to as small a space as you can and treat it like the monads in Haskell used for the IO, necessary and perhaps nonnegotiable, but quarantined to as small a space as is practicable.
- If you need to render tags, consider using HTML generated by a tool such as `Showdown` for Markdown, which is not necessarily perfect and foolproof but offers less surface area for HTML code that contains tags that have been vetted and lessen the surface area for bugs in HTML code (possibly, this is a use case for an HTML tag cleaner or an HTML-to-markdown converter that stores markdown and renders HTML).
- Only if you cannot work the default way with XSS protection and can't tag, clean, or work from markup, or anything else, should you store and dangerously set `innerHTML`.

Let's move on to the `YouPick` tag included in the Pragmatometer definition.

This project's first complete component

You can see what this component implements at `https://CJSHayward.com/missing.html`. For our first component, we pick a mostly skeletal implementation:

```
var YouPick = React.createClass({
  getDefaultProps: function() {
    return null;
  },
  getInitialState: function() {
    return null;
  },
  render: function() {
    return <div />;
  }
});
```

This skeleton returns empty, "falsy" values, which we will override. What we want to do is take two strings, break them down into one-character substrings (excluding tags), then display more and more of the first string, and then repeat the second string. It makes for a very old joke displayed for the user.

There is a division of labor between properties, meant to be set once and never changed and state, meant to allow changing. Note that state, being mutable, should be treated privately, so as to avoid the shared mutable state that Facebook declared war on. Technically, properties can be changed, but notwithstanding this, properties that should be set at the beginning (possibly passed down by a parent component) and then frozen. The state is something that can change, although the general pattern in relation to Flux is to avoid the shared mutable state. In general, this means that stores have getters but not setters; they might receive actions from dispatchers, but they are not at the mercy of anyone who has reference to the core object.

For this object, the strings are obvious candidates for the default props. Note, however, that the timestamp that a component began is not appropriate for the props, because `getDefaultProps()` will be evaluated before any instances are created, thus enabling a borg variation on the singleton pattern, where there are any number of components of this type. Potentially, there are more of them added over time, but all of them share a starting timestamp before any of them were instantiated.

So let's flesh out the `getDefaultState` method:

```
getDefaultProps: function() {
  return {
  initial_text: '<p><strong>I am <em>terribly</em> ' +
  'sorry.</strong></p><p>I cannot provide you with ' +
```

```
      'the webapp you requested.</p><p>You must ' +
      'understand, I am in a difficult position. You ' +
      'see, I am not a computer from earth at all. I ' +
      'am a \'computer\', to use the term, from a ' +
      'faroff galaxy: the galaxy of <strong><a ' +
      'href="https://CJSHayward.com/steel/">Within ' +
      'the Steel Orb</a></strong>.</p><p>Here I am ' +
      'with capacities your world\'s computer science ' +
      'could never dream of, knowledge from a million, ' +
      'million worlds, and for that matter more ' +
      'computing power than Amazon\'s EC2/Cloud could ' +
      'possibly expand to, and I must take care of ' +
      'pitiful responsibilities like ',
      interval: 100,
      repeated_text: 'helping you learn web development '
    };
  },
```

Perhaps the first change to this is to convert the text from HTML to Markdown. This is not strictly necessary; this is text that we have written ourselves, and we may have more confidence in text that we've written—confidence that it will not trigger XSS vulnerabilities—than text generated from our Markdown. In the world of computer security, a great deal of trouble can be presented by giving as little privilege out, stingily, as will let people or things get their work done: hence the saying, "Stinginess with privilege is kindness in disguise". Facebook has made not so much a stellar decision that shows uniquely good judgment as avoiding handing its users a live grenade. It is very, very easy to allow vulnerabilities that will run hundreds of megs of hostile JavaScript, with security certification assuring the user that this hostile JavaScript is genuinely from your site. For more information, please see `http://tinyurl.com/reactjs-xss-protection`. In this case, only `initial_text`, and not `repeated_text`, needs to be changed, as `repeated_text` contains only letters and spaces; thus, it works the same as plain text, HTML, or Markdown. Our revised `initial_text` reads like this:

```
    initial_text: '**I am *terribly* sorry.**\r\n\r\n' +
    'I cannot furnish you with the webapp you ' +
    'requested.\r\n\r\nYou must understand, I am in ' +
    'a difficult position. You see, I am not a ' +
    'computer from earth at all. I am a ' +
    '\'computer\', to use the term, from a faroff ' +
    'galaxy, the galaxy of **[Within the Steel Orb] +
    '(https://CJSHayward.com/steel/)**.\r\n\r\nHere ' +
    'I am with capacities your world's computer ' +
    'science could never dream of, knowledge from a ' +
```

```
'million million worlds, and for that matter ' +
'more computing power than Amazon's EC2/Cloud ' +
'could possibly expand to, and I must take care ' +
'of pitiful responsibilities like ',
```

Before continuing further, let's create stubs for the other three main components, which we will expand later on:

```
var Calendar = React.createClass({
  render: function() {
    return <div />;
  }
});
var Scratchpad = React.createClass({
  render: function() {
    return <div />;
  }
});
var Todo = React.createClass({
  render: function() {
    return <div />;
  }
});
```

We will set the state as the timestamp for this object's creation. At one glance, this might seem like something that belongs to properties, and in spirit, it is. But we want each component instance to keep its own creation date, and therefore start at zero from when it was created. If we or someone else reuse our work and create more than one instance of this on a page, each keeping its appropriate time.

Thus, we change the `getInitialState` method of `YouPick` to the following:

```
getInitialState: function() {
  return {
    start_time: new Date().getTime()
  };
},
```

The render() method

Next, we implement the render method. What we do is take the properties, which should never be directly mutated and probably should not have any existing value changed, and get the two strings from them. We will, token by token, display everything in the first string, and repeat the second as many times as the component is displayed. We will also create a tokenize function for rendered HTML converted from `Showdown`—this breaks its argument to each next tag or the next character—and, in a moment, see why we are creating a verbose anonymous function instead of a regular expression (in a nutshell, writing readable code in lieu of a regular expression appears verbose compared to the terseness available in writing write-only code).

The render method consists of more than half the lines of code as we already have it. Let's go through it step by step:

```
render: function() {
```

One of the points of breakage in JavaScript surrounds `this`. Among the horror stories that many readers may be familiar with is that if you create a constructor (the convention is to give a warning label by capitalizing the first letters of constructors and non-constructor functions, thus providing a way to cause serious misunderstanding by not holding down the *Shift* key) that adds methods and fields to an object and you have `x = Foo();` when you actually meant `x = new Foo();`, then the `Foo` constructor will trample the global namespace and add or clobber variables in it. Douglas Crockford, after originally including the "bad implementation of Java" pseudoclassical inheritance in *The Good Parts*, had second thoughts and eliminated it from *The Better Parts*, after he made an Adsafe program that could maintain security only if the `this` were removed from use. Then he started to try out the medicine he was foisting on others, and suddenly found a bunch of things he liked when he stopped using this. We can't drop the this and still use technologies like ReactJS, but we do have a say about whether we choose to drag in this when we don't need to. But ReactJS uses it, and it might be good practice to use pseudo-classical approaches based on `this` as much as needed to talk with ReactJS, but not (much) more.

Here, we have the patterned hack to deal with `this` not always being available for our first line we have:

```
var that = this;
```

The `tokenize()` function is a function that breaks HTML mostly into characters but keeps tags together:

```
var tokenize = function(original) {
  var workbench = original;
  var result = [];
  while (workbench) {
    if (workbench[0] === '<') {
      length = workbench.indexOf('>') + 1;
      if (length === 0) {
        length = 1;
      }
    } else {
      length = 1;
    }
    result.push(workbench.substr(0, length));
    workbench = workbench.substr(length);
  }
  return result;
}
```

We introduce helper variables to mitigate multiline expressions. The two variables following could as well be refactored out, but multiline expressions without them are the sort of thing that programmers glance at and then skip down, saying, "I'll read it if I have to." This is a bad thing. These variables hold the original (Markdown) strings converted to HTML:

```
var initial_as_html = converter.makeHtml(
  that.props.initial_text);
  var repeated_as_html = converter.makeHtml(
    that.props.repeated_text);
```

The HTML generated by `Showdown` has proper paragraph formatting. This is a good thing, but in this case, it means paragraph tags separating things that should belong to the same paragraph. We remove the counterproductive tags in this slightly unusual case:

```
if (initial_as_html.substr(initial_as_html.length - 4)
=== '</p>') {
  initial_as_html = initial_as_html.substr(0,
    initial_as_html.length - 4);
  }
  if (repeated_as_html.substr(0, 3) === '<p>') {
  repeated_as_html = repeated_as_html.substr(3);
}
```

```
if (repeated_as_html.substr(repeated_as_html.length - 4)
  === '</p>') {
  repeated_as_html = repeated_as_html.substr(0,
  repeated_as_html.length - 4);
}
```

We tokenize the HTML generated from our Markdown:

```
var initial_tokens = tokenize(initial_as_html);
var repeated_tokens = tokenize(repeated_as_html);
```

This step calculates the number of tokens that are desired at a particular point of time, in what has been called continuous time semantics. This means that however often or rarely we call the `render()` method, the content will be rendered appropriately when it is called, and (apart from choppiness) nothing will change if you double the frequency at which the render function is called. The `tokens` function is not a list of tokens, but the count of how many tokens should be displayed now:

```
var tokens = Math.floor((new Date().getTime() -
that.state.start_time) / that.props.interval);
```

We have a workbench as an array to which we keep adding or replacing with more tokens to display so as to build up the string that should be displayed. These should be one-character or one-tag tokens:

```
var workbench;
```

If the number of tokens that should be displayed is at most the number of tokens from the initial string, we render that part of the string:

```
if (tokens <= initial_tokens.length) {
  workbench = initial_tokens.slice(0, tokens);
}
```

If we need more tokens, we keep looping over the already available tokens from the repeated tokens:

```
else {
  workbench = [];
  workbench = workbench.concat(initial_tokens);
  for(var index = 0; index < Math.floor((new
    Date().getTime() - that.state.start_time) /
    that.props.interval) - initial_tokens.length; index +=
  1) {
    var position = index % repeated_tokens.length;
    workbench = workbench.concat(
```

```
        repeated_tokens.slice(position, position + 1));
      }
    }
```

Here is roughly how we can render an element containing text that we have calculated:

```
    return (
      <div dangerouslySetInnerHTML={{__html:
      workbench.join('')}} />
    );
  }
```

Triggering the actual display for what we have created

We have to manually refresh the display for updates. Because ReactJS is so fast, we really can afford to wastefully render the page every millisecond. We put the following code at the end, just preceding the end of the immediately invoked function expression:

```
    var update = function() {
      React.render(<Pragmatometer />,
        document.getElementById('main'));
    };
    update();
    var update_interval = setInterval(update, 1);
  })();
```

For the last major piece of our puzzle, let's look at an HTML skeleton that will house these components for now. The HTML is not especially interesting, but is provided with an interest in reducing guesswork:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Pragmatometer</title>
    <style type="text/css">
      body {
        font-family: Verdana, Arial, sans;
      }
    </style>
  </head>
  <body>
```

```
      <h1>Pragmatometer</h1>
      <div id="main"></div>
      <script src="//cdnjs.cloudflare.com/ajax/libs/react/0.12.
        2/react.js">
      </script>
      <script
        src="//cdnjs.cloudflare.com/ajax/libs/showdown/0.3.1/
          showdown.min.js">
      </script>
      <script src="/js/site.js"></script>
    </body>
  </html>
```

And what does it look like!

Summary

Here, we saw the nuts and bolts used for a simple application, perhaps more whimsical than the the `TodoMVC` function. For now, we're just making some basic explanations.

Join us in the next chapter for a to-do list that offers ways to mark a task as in progress, important, having problems, or other useful points.

9

Demonstrating Functional Reactive Programming in JavaScript with a Live Example Part II – A To-do List

In this chapter, we will be walking through a to-do list. This to-do list will illustrate a slightly obscure kind of two-way data binding. ReactJS's forte is via one-way data binding, and most problems, solved in the idiomatic ReactJS fashion, will normally follow the von Neumann model with one-way data binding (it is claimed that two-way data binding is usually not needed, and articles such as *AngularJS: The Bad Parts* suggest that two-way binding carries a heavy price tag by default, especially as far as scaling is concerned).

If we build our to-do list in an obvious way, the checkboxes will be unresponsive. We can click on them as much as we want, but they will never be checked because one-way data binding uses props—or in our case, the state—to determine whether they will be checked. We will make a foray into two-way data binding, which means that not only will the checkboxes be live, but also clicking on a checkbox will update the state. This means that what is displayed in the user interface, and what is behind the scenes as the state are kept in sync with each other.

The to-do list, as a distinctive feature, offers more than **Done** or **Not done** as a status. It has checkboxes for **Important**, **In Progress**, **Problems**, and so on.

This chapter will be a walkthrough of the following:

- The nuts and bolts of using add-ons as in two-way data binding
- Setting the appropriate initial state
- Making text inside a `TEXTAREA` editable
- A `render()` function that does some heavy lifting
- Inner functions used by `render()`
- Building the table to display
- Rendering our result
- Distinguishing the columns visually

Adding a to-do list to our application

In the previous chapter, you implemented a `YouPick` placeholder for an application of your own to create. Here, we will be commenting it out so as to display only our to-do application (we can, and we will, arrange things in different parts of the screen, but now we are displaying only one thing at a time). In JSX, the way to comment out code is to wrap it in a C-style multiline comment, and then wrap that in curly braces, so `<YouPick />` becomes `{/* <YouPick /> */}`:

```
    var Pragmatometer = React.createClass(
      {
      render: function()
        {
        return (
          <div className="Pragmatometer">
          <Calendar />
          <Todo />
          <Scratch />
          {/* <YouPick /> */}
          </div>
        );
      }
    }
  );
```

Including ReactJS add-ons in our project

We will open the `Todo` class and include the `React.addons.LinkedStateMixin` function. Note that we are using an add-on here, and this means that when we include ReactJS in our page, we need to include a build containing add-ons. So, consider this line:

```
//cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react.min.js
```

Instead of it, we include the following:

```
//cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react-with-addons.min.js
```

The opening of the `Todo` class reads as follows:

```
var Todo = React.createClass(
  {
    mixins: [React.addons.LinkedStateMixin],
```

Setting the appropriate initial state

The initial state is empty; there are no to-do items on the list, and there is empty text for any new to-do item:

```
getInitialState: function()
{
  return {
    'items': [],
    'text': ''
  };
},
```

Note that some initial state setting may involve heavy lifting; here, it is quite simple, but this need not always be the case.

Making text editable

As a minor housekeeping detail, we want obvious behavior when someone types in the box. Hence we define a two-way data binding for the TEXTAREA so that if someone types in the TEXTAREA, the change is added to the state, and overflows back to the TEXTAREA:

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

If someone submits a new to-do item by hitting the **Submit** button after entering some text, we add that item to the list:

```
handleSubmit: function(event)
{
  event.preventDefault();
  var new_item = get_todo_item();
  new_item.description = this.state.text;
  this.state.items.push(new_item);
  var next_text = '';
  this.setState({text: next_text});
},
```

Heavy lifting with render()

The `render()` function is slightly more involved, containing inner functions and much of the heavy lifting of a responsive user interface based on two-way data binding. Just inside of it, we use the `var that=this;` pattern, which is absent from most ReactJS code. In most of ReactJS, we can just use this and it works automagically; here we are defining inner functions that are not as directly built as other ReactJS functions, and we retain a reference to the this object:

```
render: function()
{
  var that = this;
  var table_rows = [];
```

Inner functions used to render

The `table_rows` array will hold to-do items. Having defined these, we define our first inner anonymous function, `handle_change()`. If the user clicks on one of the checkboxes for a to-do item, we extract the HTML ID, which tells its to-do item's ID, as well as which field (that is, checkbox identifier) has been toggled:

```
  var handle_change = function(event)
  {
    var address = event.target.id.split('.', 2);
    (that.state.items[parseInt(address[0])][address[1]] =
      !that.state.items[parseInt(address[0])][address[1]]);
  };
```

The `display_item_details()` function is the lowest of a few functions used to build up the display. It builds up an individual TD containing a checkbox:

```
        var display_item_details = function(label, item)
            {
            var html_id = item.id + '.' + label;
          return ( <td className={label} title={label}>
              <input onChange={handle_change}
                id={html_id} className={label} type="checkbox"
checked={item[label]} />
            </td>
          );
        };
```

Next, `display_item()` uses these building blocks to build the display for a to-do item. Besides including the rendered nodes, that is, checkboxes, it applies markdown formatting to the text in the box:

```
        var display_item = function(item)
        {
          var rendered_nodes = [];
          for(var index = 0; index < todo_item_names.length;
          index += 1) {
            rendered_nodes.push(
              display_item_details(todo_item_names[index], item)
            );
          }
          return ( <tr>
            {rendered_nodes}
            <td dangerouslySetInnerHTML={{__html:
            converter.makeHtml(item.description)}} />
          </tr>
          );
        };
```

Building the result table

For each item, we add a table row:

```
        table_rows.push(
        <tr>{this.state.items.map(display_item)}</tr>);
```

Finally, a JSX expression that includes various values calculated so far is returned, and it wraps them in a full-featured table:

```
        return (
          <form onSubmit={this.handleSubmit}>
            <table>
              <thead>
                <tr>
                  <th>To do</th>
                </tr>
              </thead>
              <tbody>
                {table_rows}
              </tbody>
              <tfoot>
                <textarea onChange={this.onChange}
                value={this.state.text}></textarea><br />
                <button>{'Add activity'}</button>
              </tfoot>
            </table>
          </form>
        );
      }
    }
);
```

It is left as an exercise for you to hide and display rows of data when checkboxes that should hide and display them are checked.



Here's one brief remark about the use of tables: there has been a shift from primarily using tables to primarily using CSS for formatting. However, the exact rule concerning the use of tables is not precisely "don't use tables at all" or "only use tables when you really have to," but "use tables for tabular data." This means grids, such as what we displayed here. The table with its grid of checkboxes is a good example of a situation where tables are perfectly appropriate in the current markup.

Rendering our result

Our result will render only when we tell it to; this can be seen as a chore, or as an added degree of freedom on our end. Before we end the closure, we write this:

```
var update = function()
{
  React.render(<Pragmatometer />,
  document.getElementById('main'));
};
update();
var update_interval = setInterval(update, 100);
```

Differentiating columns visually

At present, our undifferentiated grid of checkboxes runs together. We could do several things to differentiate them. A spectrum of colors in the `index.html` CSS differentiates them:

```
td.Completed {
  border-left: 3px solid black;
  background-color: white;
}
td.Delete {
  background-color: gray;
}
td.Invisible
{
  background-color: black;
}
td.Background
{
  background-color: #604000;
}
td.You.Decide
{
  background-color: blue;
}
td.In.Progress
{
  background-color: #00ff00;
}
```

```
td.Important
{
  background-color: yellow;
}
td.In.Question
{
  background-color: darkorange;
}
td.Problems
{
  background-color: red;
}
```

Summary

In this chapter, we had a walkthrough of a slightly-more-than-minimal to-do list. As far as functionality is concerned, we saw an interactive form built in a way that includes two-way data binding. ReactJS's usual recommendation is that most of the time, you think you need two-way data binding, but you would really be better off just using one-way data binding. However, the framework does not seem to be intended as a straightjacket, and where ReactJS says, "You usually shouldn't do X," there is a way to do X. For both dangerouslySetInnerHTML and two-way data binding, you can opt into it on specific points, but every effort has been made to opt for better engineering. The `dangerouslySetInnerHTML` function is a singularly forceful way to name something, and the clearly expressed opinion of the ReactJS team is that the von Neumann model calls for one-way data binding, at least in most cases. However, the ReactJS philosophy explicitly allows developers to use features they think are best to usually avoid; the final verdict is with you.

Join us in our next chapter as we create a calendar application that handles recurring appointments gracefully.

10
Demonstrating Functional Reactive Programming in JavaScript: A Live Example Part III – A Calendar

This chapter will be the most involved in the book. Throughout the book, there have been minor to not-so-minor differences between chapters. These are intended so that if you meet a fork in the road, this book offers coverage to both options. The chapters are intended to be complementary, instead of hitting the same note all the time. Here, we will not be revealing much more about core ReactJS, but show how to engage a problem with the thorniness of real-world business problems, and a solution that engages ReactJS but is not focused on it. We will be engaging ReactJS on a more serious application, a calendar with support for recurring events that offers more sophistication and power than, for instance, Google Calendar, as seen in the upcoming figures. *If you have a 7:00 PM second and fourth Thursdays each month Toastmasters' Club meeting, it's supported!* The core functionality is very much intended not to be a mere toy.

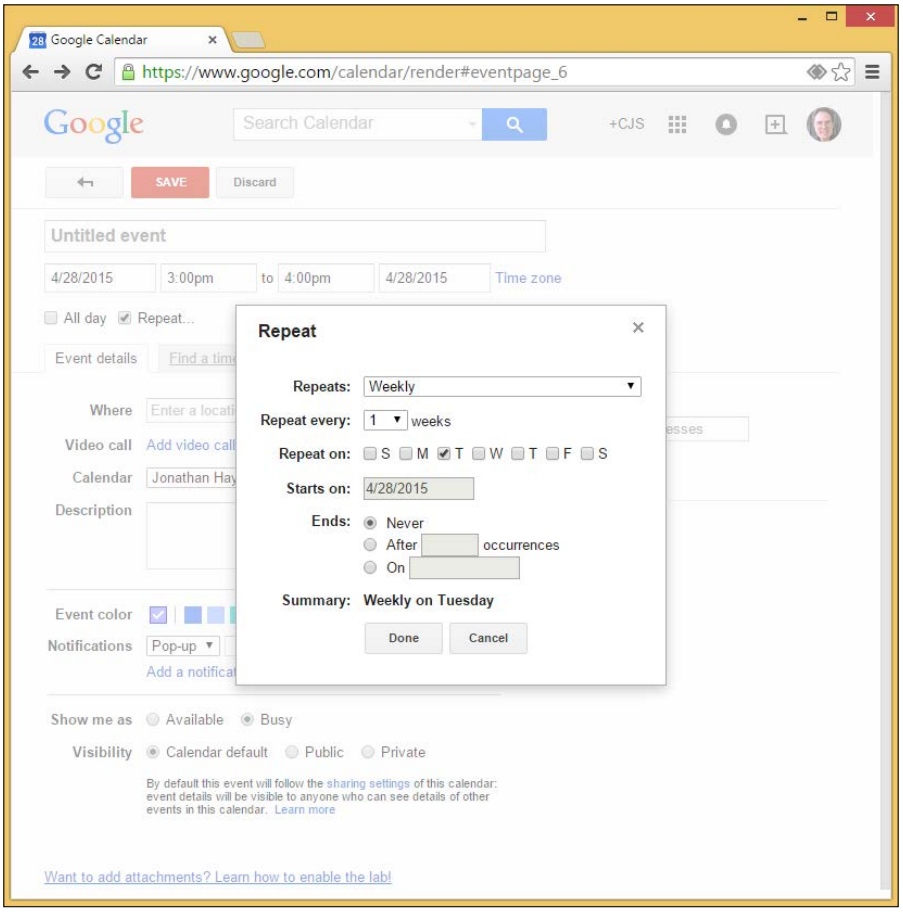
In this chapter, we will discuss the following points:

- Learning about calendars with repeating calendar entries
- The nuts and bolts for a class and its Hijaxed form
- The basic data types—plain old JavaScript objects
- A rendering function—an outer wrapper
- Rendering the page—a one-time calendar entry UI

- An expanded user interface for recurring calendar entries
- Rendering calendar entries—anonymous helper functions
- The main loop for displaying calendar entries
- Sorting each day's calendar entries for display
- Supporting Markdown in calendar entry descriptions
- Working on one main component at a time

Play it again Sam – an interesting challenge

The following is a sample screenshot showing how you enter a repeating entry in Google Calendar:



This calendar system is inspired by a private calendar system that uses regular expressions matched against a date string, like this: `Wed Apr 29 18:13:24 CDT`. Also, there is really quite a lot that can be done with the power of regular expressions. An entry to *check car engine fluids*, for instance, *every first Saturday of even-numbered months*, is `periodic_Sat.(Feb|Apr|Jun|Aug|Oct|Dec).( 1| 2| 3| 4| 5| 6| 7).................,Check fluid levels in car`. This, however, is nothing compared to the hairy beast of a truly complex regular expression. But it does give a hint on why regular expressions are considered write-only code. It may be guessed that even if you are a regular expression writer, you put off checking until later (if you have to). In other words, you don't want to check whether the regular expression quoted earlier in this paragraph matches the date for the first Saturdays of even-numbered months. This is the effect that regular expressions have on programmers, and this regular expression is elegant compared to a URL regular expression, which begins like this:

```
~(?:\b[a-z\d.-]+://[^<>\s]+|\b(?:(?:(?:[^\s!@#$%^&*()_=+[\]{}\|;:'
```

The code in this chapter is intended to be readable, at times slowly and laboriously, but very clearly without a trace of regular expressions. It has been said that a programmer has a problem with strings, and says, "I know! I'll use regular expressions!" Now the programmer has two problems.

In my own experience, I've been using regular expressions for years, and they are by far, byte for byte, my most effective defect injection method, and I often get simple regular expressions wrong the first time. This is why I'm following others in deprecating regular expressions as Not a Good Part.

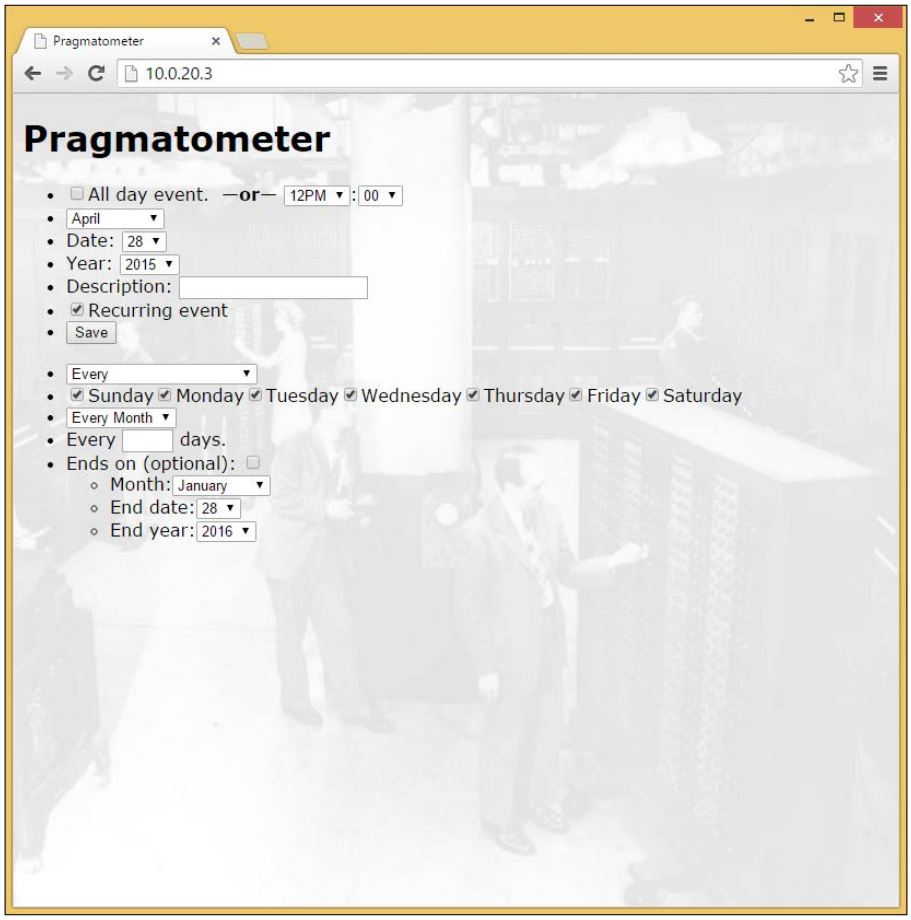
The program, by default, presents a relatively simple user interface for entering one-time calendar entries.

Here is a screenshot of our program's user interface as it is initially presented, without a drill-down into the smorgasbord of options for repeating calendar entries:



Progressive disclosure reserves a more detailed combination of supported items for recurring calendar entries, with added controls for repeating calendar entries displayed if a user opts to see them.

The following is a screenshot of the more advanced interface for recurring calendar entries. Because there are several different ways in which recurring calendar entries are often organized, several controls are available.



Classical Hijaxing works well

As we open the class, one member is conspicuous by its absence—`mixins: [React.addons.LinkedStateMixin]`. This member was featured heavily in our previous chapter where we had two-way data binding between form fields, for which we specified the HTML field/JSX component's value and this complementary implementation where the form is not controlled (value not being specified). Here, form elements are queried in the old-fashioned way as they are needed. While ReactJS strongly believes that one-way data binding should be the norm, two-way data binding can be legitimate, preferably in a small and quarantined area. This chapter and the previous ones are intended to provide working examples of two slightly different approaches so as to give a reference to the one that you prefer:

```
var Calendar = React.createClass({
```

The `getInitialState()` function initializes two items. One is a list for calendar entries. The other is a patient who is, on the operating table until the surgery is complete and it can be added to the list of live entries.

There are two types of entries: a smaller, basic entry that just gives the date for a one-off calendar entry, and a larger, more involved entry that gives the fuller information that is needed for a repeating series of calendar entries. Another implementation might keep them in separate lists; here, we use one list and check individual entries to see whether they have the `repeats` field, which recurring series have and one-off calendar entries lack:

```
getInitialState: function() {
  return {entries: [], entry_being_added: this.new_entry()};
},
```

The `handle_submit()` function Hijaxes form submission, taking the entry that is on the operating table and filling in its fields, whether for a one-time calendar entry or for a series. Then it adds the entry to the list of entries and resets the form (it would be simpler to simply `reset()` the form, but this provides slightly finer control, updating the default date to today's date so that the form's `reset()` does not always reset the page originally loaded to the date).

The entries are formulaic in character—all plain old JavaScript objects, easily JSON-serializable—and are, in essence, dictionaries containing strings, integers, and Booleans (in two cases, the entries also contain other dictionaries that contain strings and integers). There is no use of closures or other more sophisticated techniques here; the design is intended to be simple enough for someone to attentively read `handle_submit()` and accurately know how one-time and recurring calendar entries are represented.

The `handle_submit()` function pulls from the form information on whether it represents a one-time or repeating calendar entry:

```
handle_submit: function(event) {
  event.preventDefault();
  (this.state.entry_being_added.month =
    parseInt(document.getElementById('month').value));
  (this.state.entry_being_added.date =
    parseInt(document.getElementById('date').value));
  (this.state.entry_being_added.year =
    parseInt(document.getElementById('year').value));
  if (document.getElementById('all_day').checked) {
    this.state.entry_being_added.all_day = true;
  }
```

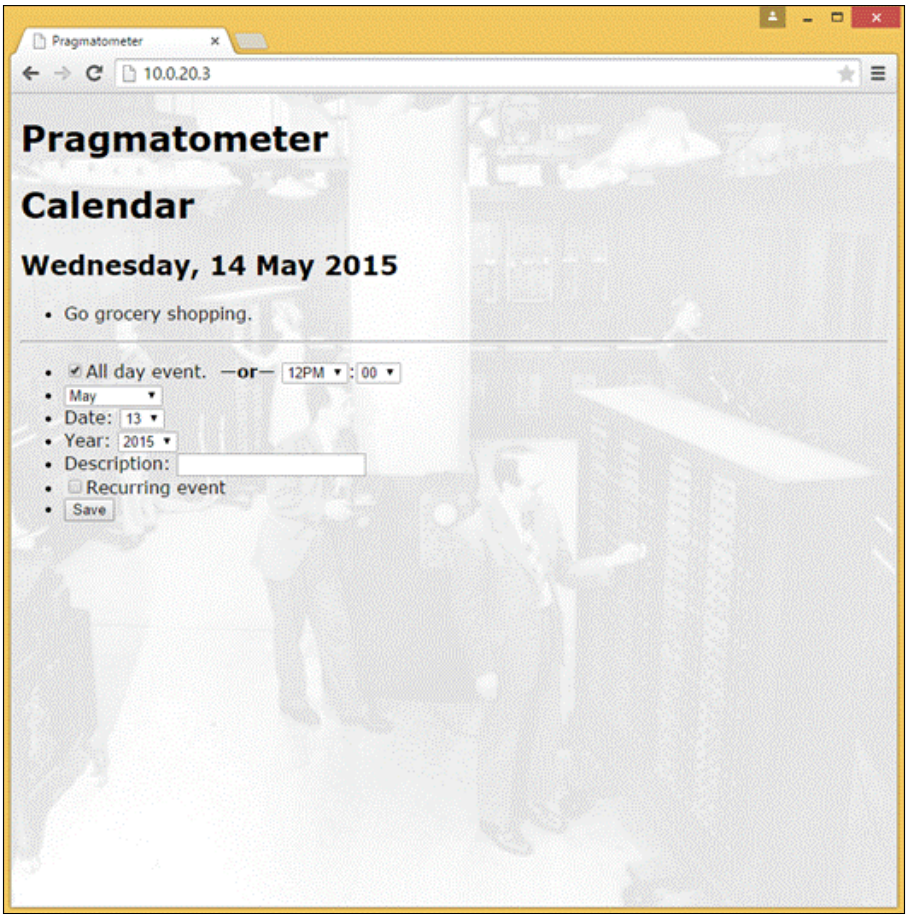
```
(this.state.entry_being_added.description =
  document.getElementById('description').value);
if (this.state.entry_being_added.hasOwnProperty('repeats')
&& this.state.entry_being_added.repeats) {
  (this.state.entry_being_added.start.time =
    this.state.entry_being_added.time);
```

At the end, it adds the entry read from the form to the list of live entries and puts a new one in place for further data entry:

```
  var old_entry = this.state.entry_being_added;
  this.state.entries.push(this.state.entry_being_added);
  this.state.entry_being_added = this.new_entry();
  var entry = this.new_entry();
  (document.getElementById('month').value =
    entry.month.toString());
  (document.getElementById('date').value =
    entry.date.toString());
  (document.getElementById('year').value =
    entry.year.toString());
  document.getElementById('all_day').checked = false;
  document.getElementById('description').value = '';
  document.getElementById('advanced').checked = false;
  if (old_entry.hasOwnProperty('repeats') &&
    old_entry.repeats) {
    (document.getElementById('month_based_frequency').value =
      'Every');
    document.getElementById('month_occurrence').value = '-1';
    document.getElementById('series_ends').checked = false;
    (document.getElementById('end_month').value =
      '' + new Date().getMonth());
    (document.getElementById('end_date').value =
      '' + new Date().getDate());
    (document.getElementById('end_year').value =
      '' + new Date().getFullYear() + 1);
  }
},
```

Here, we create a new entry. This will be a one-time calendar entry that can be augmented to represent a series later on, if desired.

Here is a screenshot of the calendar with the one one-time event represented:



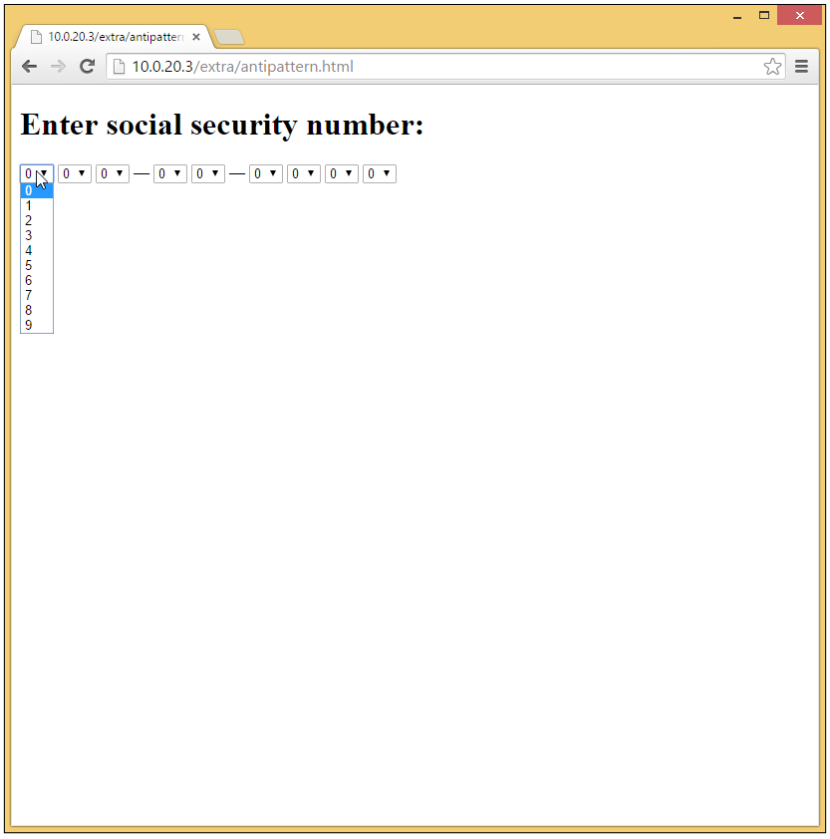
Built with usability in mind, but there's still room to grow

Here's a note that may address a confusing point of the code: the time units represented in an entry are not intended to represent everything that a JavaScript Date object represents, but to be maximally compatible with a JavaScript Date object. This means, in particular, that some programmer-confusing designs of the Date object are accommodated in our code because the JavaScript Date object numbers the dates of a month from 1 to 31, just like general calendar use, but months are represented from 0 (January) to 11 (December). Similarly, hours range from 0 to 23 in the Date object.

This function is a constructor in terms of what it does, but it is not designed for the new keyword, as the entire constructor and `this` is something that Crockford no longer includes in *The Better Parts* after he tried—in the wake of creating AdSafe, which forbade the use of the `this` keyword for security reasons—to take the medicine he was prescribing for others. He found that his code got smaller and better for the discipline. Working with code built to use `this` is nonnegotiable for serious work in ReactJS, but it may be suggested that we should opt out when we do not need to.

There is one more specific detour, as some more perceptive readers may note: the initial hours are set to 12 instead of, for instance, 0. The school that says not to allow the user to enter invalid data in the first place can, by itself, lead to some "anti-patterns" in usability. Consider the hall-of-shame-worthy interface to enter a United States social security number, which should be needed very rarely and not because you need an institution-wide identifier.

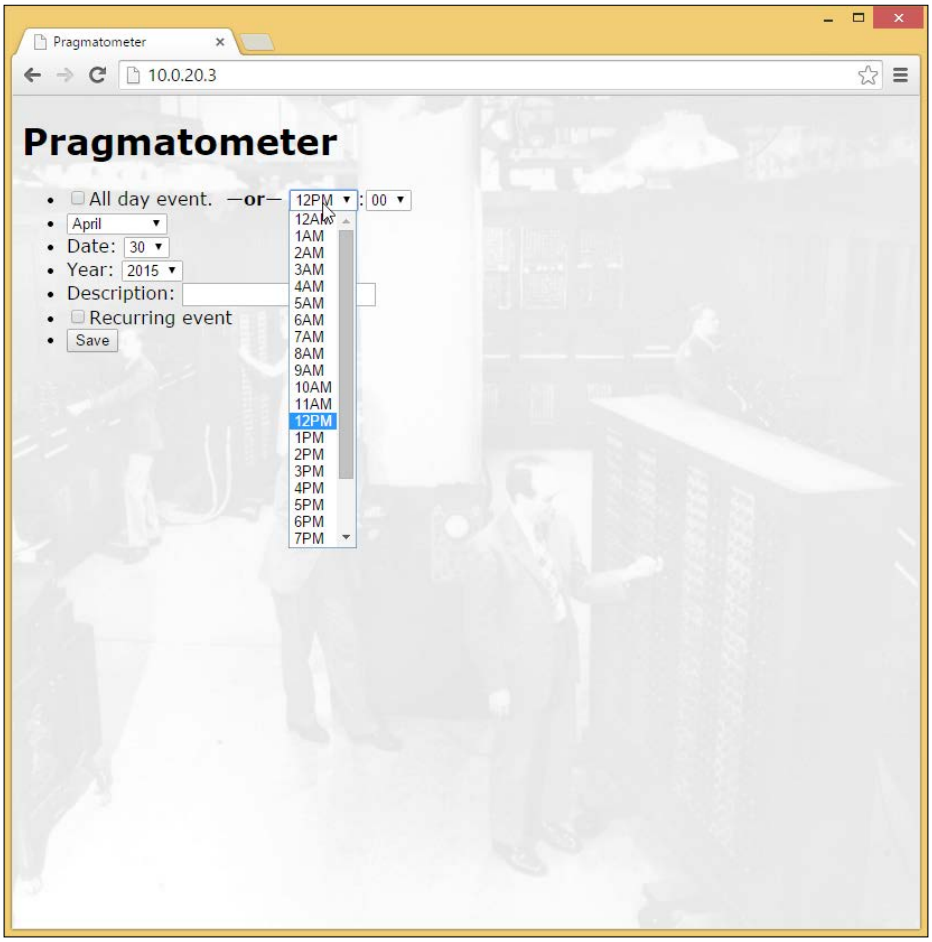
The next screenshot shows perhaps the worst way—in terms of usability and user interface concerns—of ensuring that (something that could be) a United States social security number is entered in an appropriate format:



This user interface is not appropriate; a much better approach would be to allow a text input, use JavaScript to enforce exactly nine digits, and simply ignore the hyphens (and preferably any other non-digit characters).

This interface, as implemented, represents careful thoughts but real compromises with respect to usability, and a good laboratory could probably improve it nicely (the Holy Grail here would be to have one text field where the user types the time and the system automagically uses heuristics to identify what was really meant, but that system might have trouble identifying whether a calendar entry scheduled at 8 is at 8 A.M. or 8 P.M.) Putting A.M. or P.M. immediately after the hour and in the same input, as is done a bit further down, is a violation of the principle of least astonishment, which says that whatever the software does, it should astonish the user the least. The expected approach, which is usually the right approach, is to have one field for hours, one field for minutes, and one field for A.M. or P.M. But depending on the default values, this allows someone with a mid-afternoon commitment to enter 3 for the hours, 15 for the minutes, and click on **Save**, only to get an appointment scheduled for 3:15 AM. Mistakes are still possible, as they always are, but the design that is taken is intended to help people start in the middle of the day and have more of a fighting chance of entering the hour that they really intended.

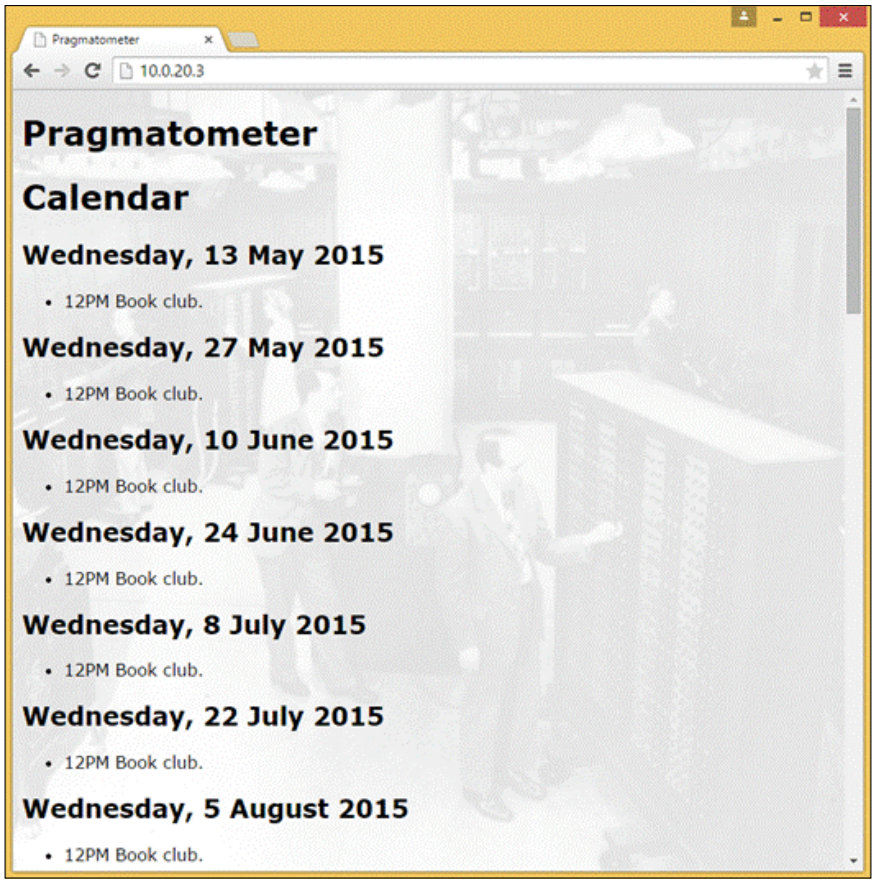
The following screenshot shows our program's default user interface, without the added controls for the user interface. It shows the drop-down menu for the hour of the day, intended as a sensible default and to reduce user errors of the time being entered as AM or PM:



One usability improvement on the interface would be to use—instead of a minute dropdown—a text field with JavaScript validation, enforcing an integer value from 0 to 59, possibly with a leading zero before a single-digit value.

But let's move on from the default starting hours of the noon to something else.

Here is an example of a calendar with a one-time event and a repeating event:



Plain old JavaScript objects are all you need

Let's take a look at the following code:

```
new_entry: function() {
  var result = {};
  result.hours = 12;
  result.minutes = 0;
  result.month = new Date().getMonth();
  result.date = new Date().getDate();
  result.year = new Date().getFullYear();
```

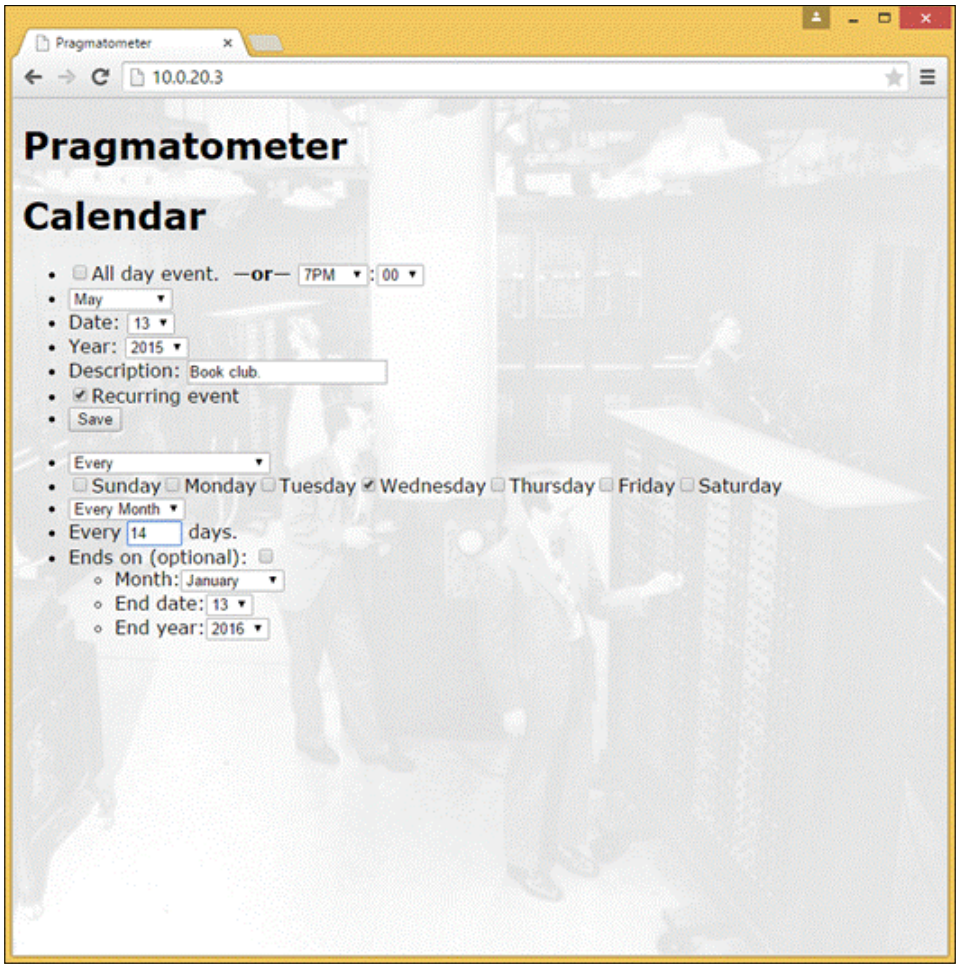
```
    result.weekday = new Date().getDay();
    result.description = '';
    return result;
  },
```

For one-time calendar entries, the fields are used just as you might expect. For series of calendar entries, the date becomes not the time when the calendar entry occurs, but the time when it starts. The user interface offers several ways that you might narrow down when a calendar entry occurs. This can be said to be every first, Tuesday only, and for a particular month. Every selection that is made narrows things down further, and so the desired usage is to be just specific enough to request the behavior that you want.

Most of the variable names here are meant to be self-explanatory. Two that might need explanation are `frequency` and `month_occurrences`. The `frequency` variable has values of `Every`, `Every First`, `Every Second`, `Every Third`, `Every Fourth`, `Every Last`, `Every First and Third`, and `Every Second and Fourth` (this is the part of the web application that accommodates your Toastmasters' meeting at 7:00 P.M. every second and fourth Thursday). The `month_occurrences` variable specifies in which month something occurs (0 to 11 as per the JavaScript Date object for January to December, or -1 to indicate every month):

```
  new_series_entry: function() {
    var result = this.new_entry();
    result.repeats = true;
    result.start = {};
    result.start.hours = null;
    result.start.minutes = null;
    result.start.month = new Date().getMonth();
    result.start.date = new Date().getDate();
    result.start.year = new Date().getFullYear();
    result.frequency = null;
    result.sunday = false;
    result.monday = false;
    result.tuesday = false;
    result.wednesday = false;
    result.thursday = false;
    result.friday = false;
    result.saturday = false;
    result.month_occurrence = -1;
    result.end = {};
    result.end.time = null;
    result.end.month = null;
    result.end.date = null;
    result.end.year = null;
    return result;
  },
```

Here is a screenshot that shows an activity that repeats every other week:

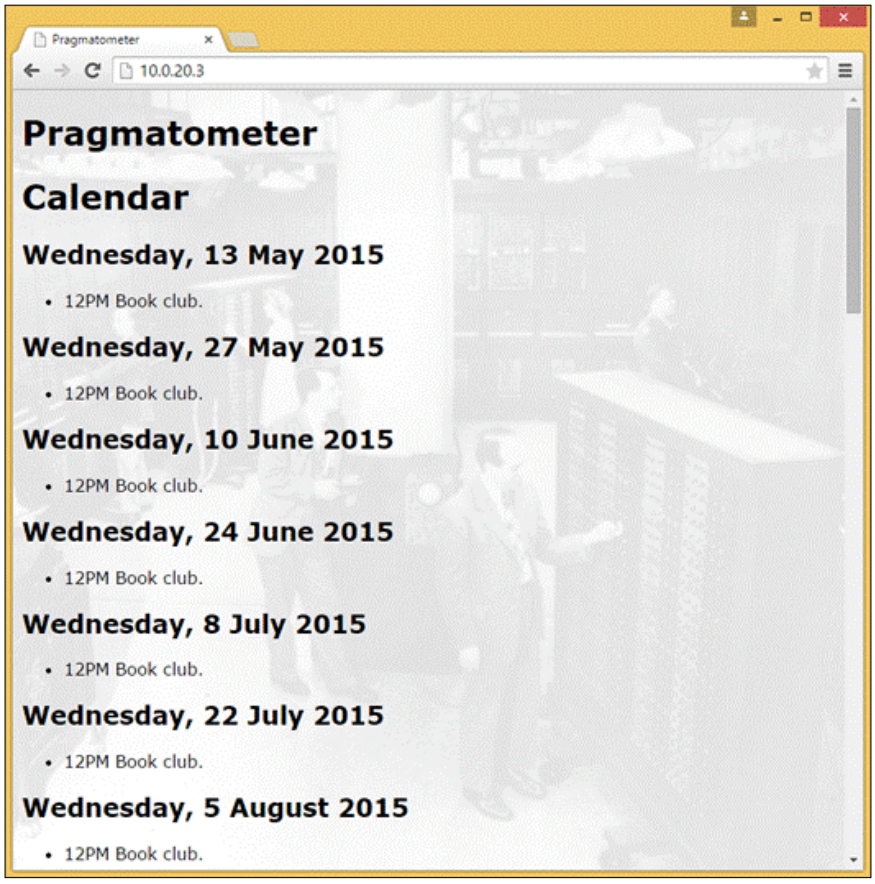


Progressive disclosure that starts simply

The `on_change()` function is called when the checkbox for repeating calendar entries is ticked, and it allows progressive disclosure, displaying the entire user interface for repeating calendar entries if the user opts for them. It toggles `this.state.entry_being_added.repeats`, which is respected by the `render()` function. This function displays the additional form area if the entry that is presently on the operating table has a `repeats` field, and it is true. If the entry does not have a `repeats` field, a new series is created, any time data already entered in the one-time calendar entry is copied over, and the new and (partly) blank entry is placed on the operating table:

```
on_change: function() {
  if (this.state.entry_being_added.hasOwnProperty('repeats') {
    (this.state.entry_being_added.repeats =
      !this.state.entry_being_added.repeats);
  } else {
    var new_entry = this.new_series_entry();
    new_entry.time = this.state.entry_being_added.time;
    new_entry.month = this.state.entry_being_added.month;
    new_entry.date = this.state.entry_being_added.date;
    new_entry.year = this.state.entry_being_added.year;
    this.state.entry_being_added = new_entry;
  }
},
```

The following screenshot shows an event that happens every other week live in the interface:



A render() method can easily delegate

The (outer) `render` function, is more of a wrapper than a workhorse. It displays the fields for a one-time calendar entry that belongs to both one-time calendar entries and series. Additionally, if the calendar entry that is on the operating table is a recurring calendar entry (which will be true only if the checkbox that indicates a recurring calendar entry is checked), this function includes the added form elements that are appropriate for a recurring calendar entry:



The JSX syntax is surprisingly forgiving. However, it does have some rules, and they are enforced with descriptive error messages, including that if there are multiple elements, they need to be wrapped in an enclosing element. So, you do not write `<em>Hello</em>, <strong>world</strong>!`. Instead, you write `<span><em>Hello</em>, <strong>world</strong>!</span>`. But given a few other ground rules, JSX will do the right thing for a remarkably broad stretch of uses and abuses.

Here is a `render()` method, which is a central method for any component you define. In some cases the `render()` method will not be monolithic, but will delegate some or all of its work to other methods. Let's explore:

```
render: function() {
  var result = [this.render_basic_entry(
    this.state.entry_being_added)];
  if (this.state.entry_being_added &&
    this.state.entry_being_added.hasOwnProperty('repeats')
    this.state.entry_being_added.repeats) {
    result.push(this.render_entry_additionals(
      this.state.entry_being_added));
  }
  return (<div id="Calendar">
    <h1>Calendar</h1>
    {this.render_upcoming()}<form onSubmit={
    this.handle_submit}>{result}
    <input type="submit" value="Save" /></form></div>);
},
```

Boring code is better than interesting code!

Readers familiar with Terry Pratchett would have probably heard of *Interesting Times*, which opens with something that has been misattributed to China in an urban legend: There is a curse. *They Say: May You Live in Interesting Times!* One of the characters, Rincewind (this is not a kind of cheese), is on a perennial quest for things that are boring, and boring is precisely what he never gets. Part of the book's plot has Rincewind tricked from living on a remote, boring tropical island to a teeming empire where all sorts of interesting things involving him keep on happening. For Rincewind, boring is a kind of Holy Grail that keeps on slipping through his fingers.

This code is meant to be *boring*, Rincewind-style. It would be possible to write more concise code that would populate a hash (or array) for `hour_options` rather than specify the array directly. But it would not be so easily checked to see whether it is right or wrong. And developing in this way doesn't mean extra typing, which (expert opinions have recognized this) is not any real bottleneck in programming.

The way the following code works essentially defines arrays, and then these arrays are used to create/form elements (which are mostly `SELECT` populated in a straightforward fashion from the arrays). The task that it accomplishes is to display the user interface for a one-time calendar entry (plus the checkbox to indicate a recurring calendar entry).

There has been a conscious decision in this chapter to do things the boring way, with one exception—populating the menu for the number of days in a month that is intended. This could be 28, 29, 30, or 31 days. We show the code to generate the dropdown menu for hours; minutes (and months) are simpler examples of the same pattern.



No programmer's wrists were harmed in the production of this chapter (it really didn't take that much typing or development time).

A simple UI for simply non-recurring entries...

For the more basic type of calendar entry, the kind that only happens once, we collect the date, month, and year, defaulting to the current date's values. Some things are "all-day" events for that day, such as someone's birthday; others start at a specific time. The interface could possibly be expanded to include optional end time. This functionality would be an extension of the same principles demonstrated here.

We begin to see the UI for rendering basic entries:

```
render_basic_entry: function(entry) {
  var result = [];
  var all_day = false;
  var hour_options = [[0, '12AM'],
    [1, '1AM'],
    [2, '2AM'],
    [3, '3AM'],
    [4, '4AM'],
    [5, '5AM'],
    [6, '6AM'],
    [7, '7AM'],
    [8, '8AM'],
    [9, '9AM'],
    [10, '10AM'],
    [11, '11AM'],
    [12, '12PM'],
    [13, '1PM'],
    [14, '2PM'],
    [15, '3PM'],
    [16, '4PM'],
    [17, '5PM'],
    [18, '6PM'],
    [19, '7PM'],
    [20, '8PM'],
    [21, '9PM'],
    [22, '10PM'],
    [23, '11PM']];
  var hours = [];
  for(var index = 0; index < hour_options.length; ++index) {
    hours.push(<option
      value={hour_options[index][0]}
      >{hour_options[index][1]}</option>);
}
```

The JSX here is similar to others that we have seen before; it is provided to reinforce what it would be like in a case like this:

```
result.push(<li><input type="checkbox" name="all_day"
id="all_day" />All day event.
 <strong>&mdash;or&mdash;</strong> 
<select id="hours" id="hours"
defaultValue="12">{hours}</select>:
<select id="minutes" id="minutes"
defaultValue="0">{minutes}</select></li>);
```

We use a dropdown for the day of a month that the user selects, and try to provide a slightly nicer alternative to making a first through a 31st year round (users shouldn't be required to know which months have 30 days). We query the form's month dropdown for the month that is presently selected. As a reminder, we are aiming for compatibility with JavaScript's Date object, and while a JavaScript Date object can have a 1-based date value from 1 to at most 31, the month value is 0-based, from 0 (January) to 11 (December), and we follow this lead:

```
var days_in_month = null;
if (entry && entry.hasOwnProperty('month')) {
  var month = entry.month;
  if (document.getElementById('month')) {
    month = parseInt(
      document.getElementById('month').value);
  }
  if (month === 0 || month === 2 || month === 4 || month
    === 6 || month === 7 || month === 9 || month === 11) {
    days_in_month = 31;
  } else if (month === 1) {
    if (entry && entry.hasOwnProperty('year') && entry.year
      % 4 === 0) {
      days_in_month = 29;
    } else {
      days_in_month = 28;
    }
  } else {
    days_in_month = 30;
  }
}
var date_options = [];
for(var index = 1; index <= days_in_month; index += 1) {
  date_options.push([index, index.toString()]);
}
```

```
    var dates = [];
    for(var index = 0; index < date_options.length; ++index) {
      dates.push(<option value={date_options[index][0]}
        >{date_options[index][1]}</option>);
    }
    result.push(<li>Date: <select id="date" name="date"
      defaultValue={entry.date}>{dates}</select></li>);
    var year_options = [];
    for(var index = new Date().getFullYear(); index < new
      Date().getFullYear() + 100; ++index) {
      year_options.push([index, index.toString()]);
    }
    var years = [];
    for(var index = 0; index < year_options.length; ++index) {
      years.push(<option value={year_options[index][0]}
        >{year_options[index][1]}</option>);
    }
    result.push(<li>Year: <select id="year" name="year"
      defaultValue={entry.years}>{years}</select></li>);
    result.push(<li>Description: <input type="text"
      name="description" id="description" /></li>);
    result.push(<li><input type="checkbox" name="advanced"
      id="advanced" onChange={this.on_change} />
      Recurring event</li>);
    result.push(<li><input type="submit" value="Save" /></li>);
    return <ul>{result}</ul>;
  },
```

The user can still opt-in for more

This method is similar to the previous one, but displays the entire interface for repeating calendar entries. It shows further variations on the theme.

The fact that the present implementation goes with the intersection of all restricting criteria for recurring calendar entries is good enough for a first pass.

The `frequency_options` functions are populated slightly differently from other fields; while this could also have been done with the date options, `SELECT` is populated in the `<option>description</option>` format, as opposed to the (usually secondary) `<option value="code">description</option>` format:

```
  render_entry_additionals: function(entry) {
    var result = [];
```

```
result.push(<li><input type="checkbox"
  name="yearly" id="yearly"> This day,
  every year.</li>);
var frequency = [];
var frequency_options = ['Every',
  'Every First',
  'Every Second',
  'Every Third',
  'Every Fourth',
  'Every Last',
  'Every First and Third',
  'Every Second and Fourth'];
for(var index = 0; index < frequency_options.length;
  ++index) {
  frequency.push(<option>{frequency_options[index]}
    </option>);
}
```

The weekdays are straightforward even if they break the pattern of populating `SELECT`, in a situation where a checkbox is the more obvious type of input. A radio button for selecting one day is also conceivable, but we are trying to accommodate more use cases, and a calendar entry with recurring Tuesdays and Thursdays or recurring Mondays, Wednesdays, and Fridays is common. Also, these aren't the only patterns in which something occurs more than once per week (it would be nice if a college student using our program did not have to make more than one entry for classes that meet more than once per week):

```
result.push(<li><select name="month_based_frequency"
  id="month_based_frequency" defaultValue="0"
  >{frequency}</select></li>);
var weekdays = [];
var weekday_options = ['Sunday', 'Monday', 'Tuesday',
  'Wednesday', 'Thursday', 'Friday', 'Saturday'];
for(var index = 0; index < weekday_options.length; ++index) {
  var checked = false;
  if (entry && entry.hasOwnProperty(
    weekday_options[index].toLowerCase()) &&
    entry[weekday_options[index].toLowerCase()]) {
    checked = true;
  }
  weekdays.push(<span><input type="checkbox"
    name={weekday_options[index].toLowerCase()}
    id={weekday_options[index].toLowerCase()}
```

```
      defaultChecked={checked} />
      {weekday_options[index]}</span>);
    }
  }
  result.push(<li>{weekdays}</li>);
```

Avoiding being clever

Let's see a subtlety (which is less obvious when scrolling through the code than looking at the user interface): there are two separate dropdown menus with their own populating arrays representing the month. The reason for this is that in one case, there is supposed to be a choice, not only between specific months, but also between specifying one month only and all months. That menu includes an option of `[-1, "Every Month"]`.

The other example is for the (optionally specified) end date for a series of calendar entries. This is a use case where it does not really make sense to specify that something ends every month. The intended use is to give the day, month, and year on which something stops displaying. The combination of these two use cases makes for two separate, non-cookie-cutter ways of choosing a month. The more exclusive could be gotten from the more inclusive with an `array.slice(1)` function, but we are again going for Rincewind-style, boring code:

```
var month_occurrences = [[0, 'January'],
  [1, 'February'],
  [2, 'March'],
  [3, 'April'],
  [4, 'May'],
  [5, 'June'],
  [6, 'July'],
  [7, 'August'],
  [8, 'September'],
  [9, 'October'],
  [10, 'November'],
  [11, 'December']];
var month_occurrences_with_all = [[-1, 'Every Month'],
  [0, 'January'],
  [1, 'February'],
  [2, 'March'],
  [3, 'April'],
  [4, 'May'],
  [5, 'June'],
```

```
    [6, 'July'],
    [7, 'August'],
    [8, 'September'],
    [9, 'October'],
    [10, 'November'],
    [11, 'December']];
```

These are baked into two separate arrays in a user interface that is slowly built up to calendar "entry-ually" include the last option, a checkbox to mark that the repeating calendar entry ends on a certain date, and fields to specify which date, month, and year it ends on, drawing on the first of the two preceding arrays:

```
    result.push(<li>Ends on (optional): <input type="checkbox"
      name="series_ends" id="series_ends" /><ul><li>Month:
      <select id="end_month" name="end_month"
      defaultValue={month}>{months}</select></li>
      <li>End date:<select id="end_date"
      name="end_date" defaultValue={entry.date}
      >{dates}</select></li>
      <li>End year:<select id="end_year"
      name="end_year" defaultValue={entry.end_year + 1}
      >{years}</select></li></ul></li>);
    return <ul>{result}</ul>;
  },
```

The previous two major methods are building forms for a user to enter data. The next method switches gears somewhat; it is set to display upcoming calendar entries from the present date to a year after the last scheduled one-time calendar entry.

Anonymous helper functions may lack pixie dust

Internally, calendar entries are segregated into one-time and recurring calendar entries. Premature optimization may be the root of all evil, but when it comes to working on calendars on other systems, looking at every calendar entry for every day has worse performance characteristics, roughly *O(n * m)* instead of the slight mindfulness shown here, which is closer to *O(n + m)*. Calendar entries are displayed as an H2 and a UL, each given a CSS class to facilitate styling (at present, the project has this portion as an unstyled, blank canvas):

```
  render_upcoming: function() {
    var that = this;
    var result = [];
```



This code, unusually for the examples we have seen so far, uses the `var that = this;` hack. In general, ReactJS guarantees that this is available at any time and not only when a function is first run. However, ReactJS does not ensure that inner functions will have the same benefits as top-level methods, and in general, it may be recommended that you use an inner function inside a top-level method only if you can do without at least some of ReactJS's pixie dust. Inner functions are here used as, for example, detached comparators. They don't interact directly with ReactJS and are limited with respect to what is available in terms of direct interaction with ReactJS.

Here, we have a comparator. It is written to be boring, as are other parts of this method; more terse replacements are readily available, but would lose plodding, "Rincewind-boring" clarity:

```
var compare = function(first, second) {
  if (first.year > second.year) {
    return 1;
  } else if (first.year === second.year && first.month >
    second.month) {
    return 1;
  } else if (first.year === second.year && first.month ===
    second.month && first.date > second.date) {
    return 1;
  } else if (first.year === second.year && first.month ===
    second.month && first.date === second.date) {
    return 0;
  } else {
    return -1;
  }
}
```

The `successor()` function uses modified one-time entries as representations of days. These keep the date, month, year, and also the number of days in the future that a day represents. The original entry used as a day has the number of days (`0`) added to it as a member.

Another aspect of the design is to eschew creating functions so anonymously that they're not assigned to a variable. The `successor()` function is written for a `for` loop analogous to the `for(var index = 0; index < limit; ++index)` loop, and it could have been made inline, but it would have been much less clear (and much less boring) than if it were pulled out into its own function. This may not be needed for a two-line anonymous function, but here the code appears to be clearer and more boring, with `successor()` stored in its own variable with a name that is meant to be descriptive:

```
var successor = function(entry) {
  var result = that.new_entry();
  var days_in_month = null;
  if (entry.month === 0 || entry.month === 2 ||
    entry.month === 4 || entry.month === 6 ||
    entry.month === 7 || entry.month === 9 ||
    entry.month === 11) {
    days_in_month = 31;
  } else if (entry.month === 1) {
    if (entry && entry.hasOwnProperty('year') &&
      entry.year % 4 === 0) {
      days_in_month = 29;
    } else {
      days_in_month = 28;
    }
  } else {
    days_in_month = 30;
  }
  if (entry.date === days_in_month) {
    if (entry.month === 11) {
      result.year = entry.year + 1;
      result.month = 0;
    } else {
      result.year = entry.year;
      result.month = entry.month + 1;
    }
    result.date = 1;
  } else {
    result.year = entry.year;
    result.month = entry.month;
    result.date = entry.date + 1;
  }
  result.days_ahead = entry.days_ahead + 1;
  result.weekday = (entry.weekday + 1) % 7;
  return result;
}
```

How far in the future should we show?

The `greatest` function immediately stores the greatest one-time calendar entry's date that exists in the list, and is then modified to represent the last day that will be represented, which is a year after the greatest one-time calendar entry found (if there are recurring calendar entries, there will probably be a number of instances rendered after the last one-time calendar entry):

```
var greatest = this.new_entry();
for(var index = 0; index < this.state.entries.length;
  ++index) {
  var entry = this.state.entries[index];
  if (!entry.hasOwnProperty('repeats') && entry.repeats) {
    if (compare(entry, greatest) === 1) {
      greatest = this.new_entry();
      greatest.year = entry.year;
      greatest.month = entry.month;
      greatest.date = entry.date;
    }
  }
}
```

Different stripes for different entry types

Calendar entries are segregated into one-time and repeating entries, so only a probable minority of repeating calendar entries are checked for every day. One-time calendar entries are put into a hash with a key straightforwardly taken from their date:

```
var once = {};
var repeating = [];
for(var index = 0; index < this.state.entries.length;
  ++index) {
  var entry = this.state.entries[index];
  if (entry.hasOwnProperty('repeats') && entry.repeats) {
    repeating.push(entry);
  } else {
    var key = (entry.date + '/' + entry.month + '/' +
      entry.year);
    if (once.hasOwnProperty(key)) {
      once[key].push(entry);
    } else {
      once[key] = [entry];
    }
```

```
  }
}
greatest.year += 1;
var first_day = this.new_entry();
first_day.days_ahead = 0;
```

Now we're ready to display!

Here is the `for` loop mentioned earlier; it is considerably more readable with `compare()` and `successor()` pulled into their own variables with descriptive names rather than inline. For each day, the loop compiles a (possibly empty) list starting with one-time activities for that day, and checks all recurring calendar entries against it. For a recurring calendar entry, it starts out with `accepts_this_date` as `true`, indicating that the calendar entry does occur on that day, and then each of the criteria for a repeating date has a cumulative opportunity to say that the criterion they are checking is not met and veto that calendar entry happening on that day. If a recurring calendar entry passes through the gauntlet without any vetoes, it is added to the calendar entries displayed for that day:

```
 for(var day = first_day; compare(day, greatest)
=== -1; day = successor(day)) {
var activities_today = [];
if (once.hasOwnProperty(day.date + '/' + day.month + '/' +
  day.year)) {
  activities_today = activities_today.concat(
    once[day.date + '/' + day.month + '/' + day.year]);
}
for(var index = 0; index < repeating.length;
  ++index) {
  var entry = repeating[index];
  var accepts_this_date = true;
  if (entry.yearly) {
    if (!(day.date === entry.start.date &&
      day.month === entry.start.month)) {
      accepts_this_date = false;
    }
  }
  if (entry.date === day.date && entry.month ===
    day.month && entry.year === day.year) {
    entry.days_ahead = day.days_ahead;
  }
  if (entry.frequency === 'Every First') {
    if (!day.date < 8) {
      accepts_this_date = false;
    }
```

Let's be nice and sort each day in order

Now that all the calendar entries, both one-time and recurring, have been assembled for the day, they are sorted. We begin with the all-day activities, which are sorted alphabetically, and then proceed with the calendar entries that occur at a specific time, sorted by time ascending:

```
if (activities_today.length) {
  activities_logged_today = true;
  var comparator = function(first, second) {
    if (first.all_day && second.all_day) {
      if (first.description < second.description) {
        return -1;
      } else if (first.description ===
        second.description) {
        return 0;
      } else {
        return 1;
      }
    } else if (first.all_day && !second.all_day) {
      return -1;
    } else if (!first.all_day && second.all_day) {
      return 1;
    } else {
      if (first.hour < second.hour) {
        return -1;
      } else if (first.hour > second.hour) {
        return 1;
      } else if (first.hour === second.hour) {
        if (first.minute < second.minute) {
          return -1;
        } else if (first.minute > second.minute) {
          return -1;
        } else {
          if (first.hour < second.hour) {
      return -1;
    } else if (first.hour > second.hour) {
      return 1;
    } else if (first.hour === second.hour) {
      if (first.minute < second.minute) {
        return -1;
      } else if (first.minute > second.minute) {
        return -1;
      }
```

```
    }
  }
}
activities_today.sort(comparator);
```

The day is displayed in a human-friendly fashion; it's `Monday`, not `Mon`:

```
if (activities_today.length)
  {
  var weekday = null;
  if (day.weekday === 0)
    {
    weekday = 'Sunday';
    }
```

Let them use Markdown!

The description for activities supports Markdown. Note—as is pointed out in Facebook's own documentation on the deliberately named `dangerouslySetInnerHTML`—that we are implicitly trusting Showdown (which provided our `converter`) to be secure. There also exist tools intended to tag clean and sanitize HTML in a fashion appropriate for XSS-secure display of HTML here.

We strip off open and close `P` tags, so that the descriptions will appear on the same line as any time or other information given by the day's ordered list:

```
  if (activity.all_day) {
    rendered_activities.push(<li
      dangerouslySetInnerHTML={{__html:
      converter.makeHtml(activity.description)
      .replace('<p>', '').replace('</p>', '')}}
      />);
  } else if (activity.minutes) {
    rendered_activities.push(<li
      dangerouslySetInnerHTML={{__html:
      hour_options[activity.hours][1] + ':' +
      minute_options[activity.minutes][1] + ' ' +
      converter.makeHtml(activity.description)
      .replace('<p>', '').replace('</p>', '')}}
      />);
  } else {
    rendered_activities.push(<li
      dangerouslySetInnerHTML={{__html:
```

```
                    hour_options[activity.hours][1] + ' ' +
                    converter.makeHtml(activity.description)
                    .replace('<p>', '').replace('</p>', '')}}
                    />);
                }
              }
              result.push(<ul className="activities">
                {rendered_activities}</ul>);
            }
          }
        }
        if (entry_displayed) {
          result.push(<hr />);
        }
        return result;
      }
    });
```

One thing at a time!

Finally, in the top-level `Pragmatometer` class, we comment out the display of `Todo`, so that only this will display while we are working on it. Next, we comment out the `Calendar` component to work on the scratchpad, and when this is done, the final integration will place these elements in the four corners of the screen:

```
var Pragmatometer = React.createClass({
  render: function() {
    return (
      <div className="Pragmatometer">
        <Calendar />
        {/* <Todo />
        <Scratch />
        <YouPick /> */}
      </div>
    );
  }
});
```

The holidays that inspired this calendar

Here, you can see the calendar set up and gracefully accommodating all the holidays in a list of American holidays:



Each nation has its own holidays, and no disrespect is meant to other nations and their holidays, but I know U.S. holidays better than those of other nations, and the approach in this chapter is partly shaped by accommodating almost all major holidays. The exception is Easter/Pascha (with Good Friday 2 days before), which is calculated according to a very specific algorithm, but one much more intricate than anything else we cover in this project, and it actually has two different algorithms for most Catholics and Protestants on one hand, and most Orthodox on the other. It could perhaps be included as a special case, but it is not entirely clear how one can create a generic solution that will accommodate equally complex calculations without a compromise of security (The most promising route for that would probably be to allow calculations in a sandbox based on Douglas Crockford's AdSafe project, which would allow a fairly free hand in calculations without needing to compromise overall page security).

Apart from Good Friday and Easter, the main official holidays in the U.S. are listed as follows:

- New Year's Day (January 1, fixed)
- Martin Luther King Day (third Monday in January)
- President's Day (third Monday in February)
- Memorial Day (last Monday in May)
- Independence Day (July 4, fixed)
- Labor Day (first Monday in September)
- Columbus Day (second Monday in October)
- Veteran's Day (November 11, fixed)
- Thanksgiving (fourth Thursday in November)
- Christmas (Western, December 25, fixed)
- New Year's Eve (December 31, fixed)

This system, is similar to the private calendar that served as its inspiration, is intended (among other things) to be powerful enough to allow calculation of floating and fixed holidays (with the regretted and intricately complex exception of Easter/Pascha), Also, it provides a very straightforward interface to enter every single holiday on the bulleted list, and much more. With a modern calendaring system, people in the U.S. do not research holidays on Wikipedia, and manually enter in their calendars that Thanksgiving is on the fourth Monday in November. They include a calendar that lists the holidays. However, this system is flexible enough for someone so inclined in any country to get a very direct interface to enter these holidays or others following the anticipated pattern or others like it.

Summary

This chapter was meant to provide a slightly more involved example of a user interface built on ReactJS that has non-toy functionality. We saw both rendering code and backend-type functionality, that adds depth beyond a user interface alone. The approach was intended to be complementary to the previous chapter, in terms of (for instance) controlled inputs that specify what their value will be, as opposed to near classic Hijaxing of queried forms.



From a usability perspective, the best way of handling user input for recurring calendar entries is probably not to directly tweak and enhance a complex and somewhat heterogeneous form, as we have done here. The kind of advanced recurring events we use here are a use case for a wizard or interview approach.

We looked at a calendaring system that uses ReactJS in the kind of messy problem solving that we encounter in the real world. We have an approach to sophisticated render methods. With respect to usability, which ReactJS developers should perhaps be the most sensitive to (as they are the people most responsible for development that touches usability), there has been attention and a constant eye on usability, coupled with awareness that the user interface could stand improvement.

Along the way, we looked at boring code and boring plain old JavaScript objects that rather work extremely well when we need records. Finally, we looked at the holidays for a specific nation that our calendar was intended to be powerful enough to depict with its recurring event facilities.

Join us in the next chapter as we look at incorporating a third-party (non-ReactJS) tool and integrate the code of various applications into a single page.

11

Demonstrating Functional Reactive Programming in JavaScript with a Live Example Part IV – Adding a Scratchpad and Putting It All Together

In this chapter, we will cover the three last endeavors intended to put it all together and finish our sample ReactJS application. Earlier chapters dealt with a basic custom-made component made with 100 percent ReactJS. This chapter is different in making a valid components, that works with ReactJS while drawing on a significant non-ReactJS-based tool.

Having made the last component, we will integrate them into one page, where each of the four components is placed in its quarter of the page. This is different from development, where we gave the tool under development the whole page. This will be the second major section of this chapter.

So far, the page has no way of keeping track of the state. Suppose you make an entry in the calendar, a to-do item, or make some notes in the scratchpad. Then, if you navigate away and come back or reload the page, all your changes will be lost. It would be nice to have the changes remembered, and this is exactly what we will do next. In the third, and last, major section in this chapter, we will introduce a cheap, homegrown HTML5 localStorage-based persistence solution that works surprisingly well. It does not let you access your data from multiple computers, but let's leave that alone for now and just work on persistence on the same computer.

The overall application is intended to handle personal information management/ logistics: a scratchpad for any information, a to-do list, a calendar, and a stub of whining artificial intelligence that is meant to be replaced by something interesting of your own devising.

Adding a WYSIWYG scratchpad, courtesy CKeditor

There are multiple **What You See Is What You Get** (**WYSIWYG**) editors out here, and the choice of CKeditor is not much a judgment that CKeditor is the uncontested king of free and paid editors as a common choice. We will see how to ask ReactJS to leave part of the DOM alone (and in this case, not clobber our CKeditor instance). We will cover these topics:

- Why use something such as CKeditor when it doesn't work similar to ReactJS?
- Installing a "small is beautiful" take on CKeditor, with a look at which edition is the best
- Including CKeditor in our page, with an emphasis on JSX

Bringing all things together into one web page

We've done almost everything together. We will cover the following topics:

- Adjusting the JSX so that all our features are now uncommented. This is a very simple step.
- CSS styling that lets everything fit. We arrange the components in a 2 x 2 grid, but this could be replaced by pretty much any styling approach that would fit the components on a page.
- Bringing in elementary data persistence which displays the components together. This will include some basic, non-exhaustive CSS work.
- In the interests of offering a complete sample application, the user-interface-centric application we have been developing will include basic persistence on your computer, in this case humbly implemented with HTML5 localStorage. This means that one computer, without sign-in or other annoyances, will be able to persistently use data.

- A few simple `JSON.stringify()` calls work and lay a foundation for a more common style of remote, server-based persistence. The data is stringified through `JSON.stringify()`, which is not specifically needed with localStorage, but makes the code slightly more ready to swap out the localStorage references alone and replace them with a potentially remote server of choice.
- Causing the CKeditor state to persist. Some experienced programmers, on being asked to create a localStorage persistence solution for the component state, might reasonably guess our solution for everything but the scratchpad. The scratchpad has some gotchas for Web 2.0 work, because CKeditor has some gotchas for Web 2.0 work.

The entire system working together may be seen at `http://demo.pragmatometer.com/`.

This book is about ReactJS, so why use CKeditor?

It might, in general, be suggested that it is better to have something that works with the declarative spirit of ReactJS and one-way data bindings. If you have a choice between a good implementation of something such as CKeditor that doesn't particularly work in a similar way to ReactJS, and some other component made that fits in nicely with ReactJS and handles WYSIWYG well, you should go with the component that fits ReactJS well.

This book is intended to help you with either side of a fork on the road. It includes development with and without JSX, both legacy and green field development, one-way and two-way data binding, and (here) purely ReactJS components versus integrating with non-ReactJS JavaScript tools. The good news is that ReactJS is good at playing nicely with other children. Tools from all across the world of JavaScript are at least potentially available for you, not just the sliver explicitly developed for work with ReactJS. Perhaps you may have the luxury of working with pure ReactJS components. Perhaps you want, need, or have to use some JavaScript tool that was not built with any ReactJS integration in mind. Here's the good news: in either case, ReactJS probably has covered. In this chapter, we will use a standard non-ReactJS tool—the famous and well-established CKeditor, which ReactJS lets us integrate into our web page quite well.

CKeditor – small free offerings, and small is beautiful

There are several free and commercial editors available; one such editor is CKeditor, (its home page is at `http://ckeditor.com/`). CKeditor comes with four basic options: *Basic*, *Standard*, *Full*, as well as a *Custom* option that allows complete freedom in selecting and deselecting optional functionality. For this project, we will use the *Basic* option. This is not a service for a user to present an overwhelming array of rows of buttons, and the correct question regarding which features to include is, "What is the minimum that will work well for us?"

Including CKeditor in our page

The **Basic** option (and also the Standard, Full, and Custom arrays of options) is available for download or from CDN. At the time of writing this book, the Basic option can be obtained from CDN using the following:

```
<script src="//cdn.ckeditor.com/4.4.7/basic/ckeditor.js"></script>
```

This should be our HTML. We also need to work on JSX. The code used to put up the scratchpad is the simplest and shortest of our four subcomponents:

```
var Scratchpad = React.createClass({
  render: function() {
    return (
      <div id="Scratchpad">
        <h1>Scratchpad</h1>
        <textarea name="scratchpad"
          id="scratchpad"></textarea>
      </div>
    );
  },
  shouldComponentUpdate: function() {
    return false;
  }
});
```

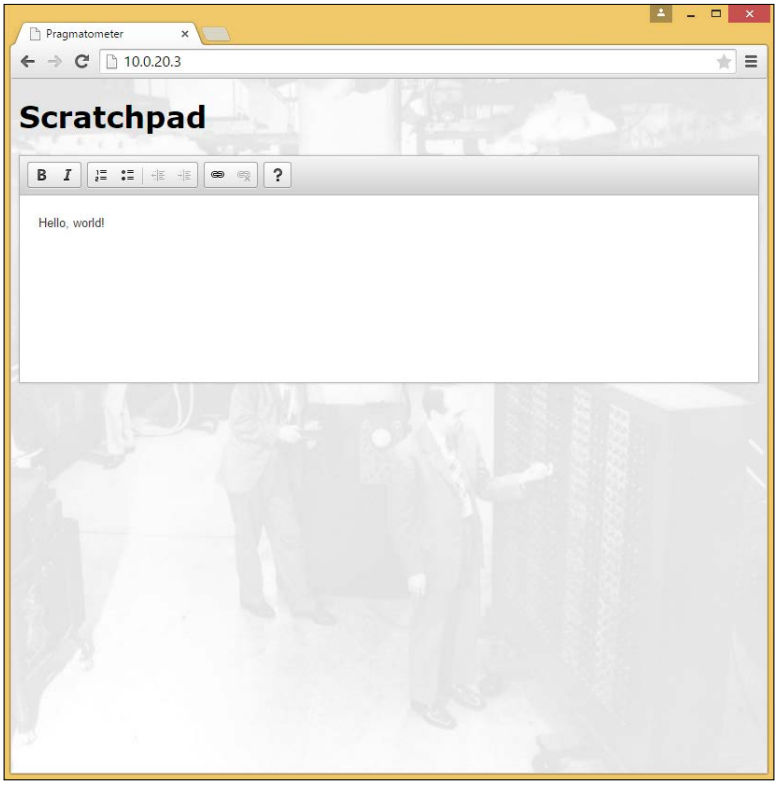
The `render()` method is as simple as it looks. Note that it defines a `TEXTAREA` rather than a CKeditor widget. The different versions of CKeditor work by Hijaxing specific `TEXTAREAs` rather than writing their widget in the code. The `shouldComponentUpdate()` method is also as simple as it looks, but it is worthy of a comment. This method is intended to facilitate optimization for the rare case where ReactJS's virtual DOM diffing isn't as fast as what you can do. For instance, under ClojureScript, Om has immutable data structures and can therefore test equality via reference comparison alone, without the need for deep equality checking, which is why Om plus ClojureScript is about twice as fast as ReactJS plus JavaScript. And, as stated in earlier chapters, 99 percent of the time, micromanaging ReactJS's virtual DOM is simply not needed, even if you want to be very performant.

However, here we have a separate use case for the mechanism of `shouldComponentUpdate()`. Its use here is unrelated to optimization and obtaining the same result with less comparison. Instead, it is used to disclaim the jurisdiction of part of the DOM. For some other tools that you might like to include, such as CKeditor, it is desirable to ask ReactJS to create part of the DOM and then leave it alone, without later clobbering another tool's changes; this is exactly what we have done here. Hence, `shouldComponentUpdate()`—besides constituting a mechanism to prune unnecessary comparisons in lightning-fast virtual DOM diffing—can also be used to affix a label that says, "Something other than ReactJS is responsible for maintaining this part of the DOM. Please do not clobber it."

After the first rendering of the web application, we ask CKeditor to replace the `TEXTAREA` that has an ID of scratchpad, which should give us a live widget:

```
    React.render(<Pragmatometer />,
      document.getElementById('main'));
    CKEDITOR.replace('scratchpad');
  We temporarily comment out the other subcomponents:
    var Pragmatometer = React.createClass({
      render: function() {
        return (
          <div className="Pragmatometer">
            {/* <Calendar /> */}
            {/* <Todo /> */}
            <Scratchpad />
            {/* <YouPick /> */}
          </div>
        );
      }
    });
```

And now we have an interactive scratchpad. Here is a screenshot of our web application, showing only the scratchpad:



Integrating all four subcomponents into one page

Having created four subcomponents—a calendar, a scratchpad, a to-do list, and a **You Pick** slot with a placeholder—we will now integrate them.

We start by uncommenting all the commented subcomponents in the Pragmatometer's `render()` method:

```
<div className="Pragmatometer">
  <Calendar />
  <Todo />
  <Scratchpad />
  <YouPick />
</div>
```

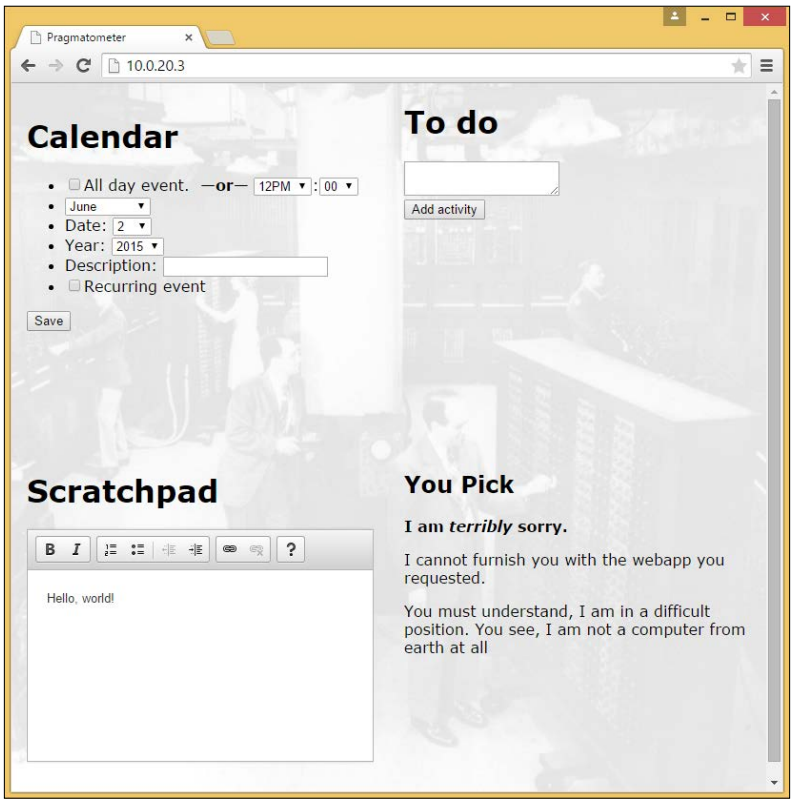
Our next step is to add styling, with just a bit of responsive design. At least one of the main contenders in responsive design is to simply not try to know and address each screen resolution, but to have several steps of responsiveness depending on screen width. You can see this if, for example, you have a wide desktop monitor, load `http://therussianshop.com/`, and progressively narrow your browser window. Different adaptations kick in, and the page as a whole makes a whole when seen at desktop width, tablet on either orientation, or smartphone. We will not be attempting a serious solution here, but there is some responsiveness in the fact that we make our styling conditional to a minimum width of 513 px. Without any styling, the four elements will displayed one on top of the other; with styling, they will be corralled into a 2 x 2 grid.

The CSS that styles the subcomponents essentially divides a sufficiently large window into quarters, adds some padding, and ensures that any overflow on each application will scroll:

```
@media only screen and (min-width: 513px) {
  #Calendar {
    height: 46%;
    left: 2%;
    overflow-y: auto;
    position: absolute;
    top: 2%;
    width: 46%;
  }
  #Scratchpad {
    height: 46%;
    left: 2%;
    overflow-y: auto;
    position: absolute;
    top: 52%;
    width: 46%;
  }
  #Todo {
    height: 46%;
    left: 52%;
    overflow-y: auto;
    position: absolute;
    top: 0;
    width: 46%;
  }
  #YouPick {
    height: 50%;
    left: 52%;
    overflow-y: auto;
```

```
        position: absolute;
        top: 52%;
        width: 46%;
      }
    }
```

This allows us to display and the following is a screenshot of all the parts of our web application put together:



Persistence

Some frameworks are all-purpose frameworks that are intended to do everything; ReactJS is not. It does not even offer any method by which to make an AJAX call, even though (practically) any important project that uses ReactJS will have AJAX requirements. This is entirely by design. The reason is that ReactJS is specialized as a framework for working on user interfaces or making Views, and it is intended to be combined with other technologies to make whatever complete package is appropriate for your site.

One feature that is desirable in this Pragmatometer application is that it remembers the data you enter. If you enter an appointment for tomorrow at 2:00 p.m., then leave the page and come back, it would be nice if the page remembered the appointment instead of presenting a completely blank slate every single time you load it. Persistence is one part of a complete web application, but it's not the responsibility of the view or user interface, and ReactJS obviously does not offer a solution for persistence. Nor should it—perhaps. Some recent chapters covered how to use ReactJS to do "X"; this chapter is about how to do something else that complements ReactJS.

For mainstream use, persistence is usually handled by communication with a backend; there are several good technologies available. But perhaps it will not be terribly useful to try to cram the treatment of properly implementing a backend into one section of one chapter of a book on frontend development with ReactJS.

As an exercise that remains squarely in frontend territory, we will handle persistence by a well-known frontend route—HTML5's localStorage (the persistence code does nothing if Modernizr fails to detect localStorage). The functions used, `save()` and `restore()`, save in localStorage if such is found. They call `JSON.stringify()` and `JSON.parse()` directly, even though such a step is not strictly needed to cause JSON-serializable objects to persist in localStorage. This is intended to provide a direct hook to change the code to talk to a remote backend. The simplest adaptation would, like the implementation here, monolithically save and restore the entire state for an application, but remember that premature optimization remains the root of all evil. Extremely heavy use of an application in this way might lead to an amount of state comparable to a single large PNG file. The code is, of course, further adaptable to more surgical approaches to saving or restoring a diff that is lighter, but the point here is to lay a solid groundwork, not push optimization as far as it can go.

We will use Crockford's JSON from `https://github.com/douglascrockford/JSON-js/blob/master/json2.js` and Modernizr from `http://modernizr.com/`. We will only use Modernizr in this application to test the localStorage availability, so if you're looking for a "minimum weight that's sufficient for this project" Modernizr build, opt to test for localStorage and opt out of everything else. Let's include these files in `index.html`:

```
<script src="js/json2.js"></script>
<script src="js/modernizr.js"></script>
```

In our `site.jsx` file, we define the `save()` and `restore()` functions. These will be used to cause the entire state to persist for different applications. A different approach might be to make more and smaller saves instead of a few monolithic saves, but a few monolithic saves are easy to keep track of mentally. Therefore, they are easier to maintain and debug than a mesh of different saves for minor aspects of data (if we need to optimize later, we can, but premature optimization is still the root of all evil). The save() function reads as follows:

```
var save = function(key, data) {
  if (Modernizr.localstorage) {
    localStorage[key] = JSON.stringify(data);
  }
}
```

The most obvious way to connect this to a remote backend, besides taking care of details such as account management (which are not addressed in this example), would be to replace the `localStorage[key]` assignment with a call to notify the server of the new stringified data associated with that key. That would make the Modernizr check unnecessary. However, be warned: even IE8 supports localStorage, and clients that don't support it will be a little archaic and probably not supported by ReactJS, which does not advertise support for IE versions earlier than 8 (also, IE8 support is now based on a shim instead of being native; see `http://tinyurl.com/reactjs-ie8-shim`).

The `restore()` function takes an optional parameter besides key—`default_value`. This is used to support an initialization that pulls the saved state if it exists and falls back to the normal value that would otherwise be used in initialization. Initialization code can be reused to work with this `restore()` function, which pulls non-null and defined saved data for that key if it exists, or the default if nothing interesting is found. The line with `JSON.parse()` and the `if` statement that probes localStorage are the lines you would most directly replace with calls to talk to a remote backend. Alternatively, and to take a step further, the `restore()` function would probably be gutted and replaced with a function with the same signature and semantics, but would talk to a remote server that owned more of the work of checking whether any pre-existing data has been saved. This perhaps leaves the client to return the default value if the server has nothing to return:

```
var restore = function(key, default_value) {
  if (Modernizr.localstorage) {
    if (localStorage[key] === null || localStorage[key]
      === undefined) {
      return default_value;
    } else {
```

```
      return JSON.parse(localStorage[key]);
    }
  } else {
    return default_value;
  }
}
```

Now, all the `getInitialState()` functions are modified to go through the restore() function. See what happens hence. Consider the `Todo` initializer of this code:

```
getInitialState: function() {
  return {
    'items': [],
    'text': ''
  };
},
```

It is simply wrapped in a call to `restore()`:

```
getInitialState: function() {
  return restore('Todo', {
    'items': [],
    'text': ''
  });
},
```

There are a few functions that alter the state on one component or another, and we make any function that changes part one of our component's states save the whole state. Hence, in the appropriately named `Calendar#handle_submit`, numerous details of `this.state.entry_being_added` are filled in to match what is on the (Hijaxed) form. Then the filled-in entry is added to the list of live filled-in entries, and a new one is put in its place:

```
this.state.entries.push(this.state.entry_being_added);
this.state.entry_being_added = this.new_entry();
```

These two lines alter `this.state`, and so we save the state after them:

```
this.state.entries.push(this.state.entry_being_added);
this.state.entry_being_added = this.new_entry();
save('Calendar', this.state);
```

One detail – persisting the CKeditor state

Most of this section is relatively predictable. Some programmers, who were told that we were adding persistence through HTML5 localStorage, might have guessed something close to what was written earlier, and most likely, they would be pretty close to the mark. There is one detail about CKeditor, however, that is less than obvious and less than ideal.

CKeditor does what you might naïvely expect under "un-fancy" Web 1.0 form usage. If you have a form, include a `TEXTAREA` with a name and ID of `foo`, call CKeditor to convert it, and submit the form. The form will be submitted as if the HTML, which was then on the CKeditor instance, had been the content of the `TEXTAREA`. All of this is as it should be.

However, if you use CKeditor in almost any "AJAXian" way, querying the value of the text area without having a full-page form submission, you will run into a problem. The reported value of the CKeditor instance is neither more nor less than the text that it was initialized to. The reason is that the value of `TEXTAREA` is synced for you on a whole-page form submission, but this is not automatically done at the intermediate steps. This means that unless you take an extra step, you cannot usefully query the CKeditor instance.

Fortunately, the extra step is not particularly difficult or slippery; CKeditor provides an API to sync the `TEXTAREA`, so you can query `TEXTAREA` to get the CKeditor instance's value. Before connecting the CKeditor scratchpad, we initialized the entire display and set an interval so that the display is updated every 100 milliseconds (there is nothing necessary or magical about this length for an interval; it could be updated more or less often, with longer intervals being choppier but basically the same):

```
var update = function() {
  React.render(<Pragmatometer />,
    document.getElementById('main'));
};
update();
var update_interval = setInterval(update,
  100);
```

To accommodate CKeditor, we shuffle and unbundle things slightly. Our code will be a little messier in order to accommodate calling things in a particular order. For our `TEXTAREA` to exist in the first place, we need to render the Pragmatometer master component once (or more than once, if we want). Then, after that call, we ask CKeditor to convert the `TEXTAREA`.

Next, we start an update function. This both updates the display and synchronizes CKeditor's `TEXTAREAs` to where they can be queried. The loop that synchronizes the `TEXTAREA` is not strictly necessary. If we have only one editor instance, we need only one line of code, but the code we have is generic for any number of CKeditor instances with any ID. Finally, within the loop, we call `save()` on the editor contents. One optimization, if `save()` and `restore()` are gutted and replaced to talk to a backend server, would be to save the current editor state in a variable and only `save()` if the editor's contents differ from the previous saved value. This should diminish the frequent network chatter:

```
React.render(<Pragmatometer />,
  document.getElementById('main'));
CKEDITOR.replace('scratchpad');
var update = function() {
  React.render(<Pragmatometer />,
    document.getElementById('main'));
  for(var instance in CKEDITOR.instances) {
    CKEDITOR.instances[instance].updateElement();
  }
  save('Scratchpad',
    document.getElementById('scratchpad').value);
};
var update_interval = setInterval(update,
  100);
```

There are a few more changes so that all of the initialization wraps our earlier code in a call to `restore()`. Also, each time we change a component's state, we call `save()`. And we're done!

Summary

In this chapter, we added a fourth component. It differs from the others by not being built from the ground up in ReactJS, but integrating a third-party tool. This can work well enough; just be careful to have a `shouldComponentUpdate()` method that returns `false` as a way of saying, "Don't clobber this; let the other software do its work here."

Despite the fact that we covered three basic topics—a component, integration, and persistence—this chapter was easier than some others. We have a live, working system, and you can see it at `http://demo.pragmatometer.com/`.

Now let's take a step back to look at the conclusion, discussing what you learned over the course of this book.

12
How It All Fits Together

Google Maps was a big hit when it came out, and it remains quite important, but the new functionality it introduced was pretty much nothing. The contribution Google made with its maps site was taking things previously only available with a steep learning cliff and giving them its easy trademark simplicity. And that was quite a lot.

Similar things might be said about ReactJS. No one at Facebook invented functional reactive programming. No one at Facebook appears to have significantly expanded functional reactive programming. But ReactJS markedly lowered the bar to entry. Previously, with respect to functional reactive programming, there were repeated remarks among seasoned C++ programmers; they said, "I guess I'm just stupid, or at least, I don't have a PhD in computational mathematics." And it might be suggested that proficiency in C++ is no mean feat; getting something to work in Python is less of a feat than getting the same thing to work in C++, just as scaling the local park's winter sledding hill is less of an achievement than scaling Mount Everest. Also, ReactJS introduces enough of changes so that competent C++ programmers who do not have any kind of degree in math, computational or otherwise, stand a fair chance of using ReactJS and being productive in it. Perhaps they may be less effective than pure JavaScript programmers who are particularly interested in functional programming. But learning to effectively program C++ is a *real* achievement, and most good C++ programmers have a fair chance of usefully implementing functional reactive programming with ReactJS. However, the same cannot be said for following the computer math papers on Wikipedia and implementing something in the academic authors' generally preferred language of Haskell.

In this conclusion, we will explore the following topics in this chapter:

- A retrospective of the terrain covered
- Immunity to the problems that gave rise to *The Mythical Man-Month*
- ReactJS as just a view—but what a view!

- The joy of programing ReactJS
- Whole new vistas opening for ReactJS beyond the web. The ReactJS work introduced here is not the *end* of ReactJS's possibilities: it is only the *beginning*.

A review of the terrain covered

We've covered a lot in this book, both theoretically and practically. We covered the basics of functional programming, reactive programming, and functional reactive programming. We also covered a technology, Facebook's ReactJS. It makes some of the strengths of functional reactive programming available to frontend developers who are not necessarily steeped in computational mathematics (this is, unfortunately, something of a distinctive feature to this text). The text here is meant to follow the lead of ReactJS and is specifically intended to make sense to programmers without a special math background. Along the way, we met interesting technologies, such as Om, Brython, and Jest, and took a look at what future frontend web development might be like. We may be able to do web development in the language of our choice, rather than necessarily be limited to JavaScript.

We also built two systems, one smaller and one larger, and tried to demonstrate slight variations in how a problem is solved: with or without JSX, with controlled values for form elements, and by classic form Hijaxing. The point is not exactly that one is better than the other, as that needs will call for different solutions, and we want to increase the chances that at least one of the approaches covered here is helpful in a particular situation.

Along the way, there were reasons to say, as has been said about Python, "Programming is fun again!" Every system has its quirks, but somehow, there seem to be fewer speed bumps along the road when traveling with ReactJS. This title's brief treatment of CKeditor necessarily included a workaround for a speed bump that will frustrate a first-time CKeditor user. There have been precious few necessary warnings about workarounds for problems that persist in ReactJS code.

Could the Mythical Man-Month have been avoided?

Fred Brooks' 1975 book *The Mythical Man-Month* (40 plus years old by the time you read this book) is the most heavily cited work in all of software engineering literature. Tanenbaum's classic textbook *Operating Systems: Design and Implementation* makes a (brief) mention of Brooks' title:

> *"One of the designers of OS/360, Fred Brooks, wrote a witty and incisive book (Brooks, 1975) describing his experiences with OS/360. While it would be impossible to summarize the book here, suffice it to say that the cover shows a herd of prehistoric beasts stuck in a tar pit..."*

Here, there is a direct relevance. To explain why, let's make a variant of the Big-Coffee notation as introduced by Steve Luscher. While perhaps no one outside Luscher's head knows what inspired him to make his point as he did, Luscher is clearly familiar with the classical big-O notation for runtime complexity, and presumably with the fact that it is also used to assess the complexity in other kinds of resource usage, such as memory. But I might suggest that a possible additional inspiration might be helpful as explained by Big Organization complexity. If Big-Coffee complexity can be nightmarish and quadratic at $\text{☕}(n \wedge 2)$ — or as Luscher writes, $\text{☕}(n*(n-1))$ — something eerily familiar manifests in the communication complexity within a monolithic project.

If there is one programmer on a monolithic project, the complexity is zero, as there is no need to avoid stepping on other programmers' feet. If there are two programmers, the complexity in communication is exactly one connection. If there are three programmers, there are three connections; and if we scale up to 10 programmers, the paperwork scales up to 45 connections. IBM's approach to the OS/360 project was the so-called Big Blue Solution, and it said, "Since there's lots and lots of work we want done, let's hire lots and lots and lots of programmers!" IBM had well over 10 programmers, and therefore had vastly more than 45 connections.

One character that might be appropriate to indicate the complexity of organizational communication is the interconnected HTML dingbat, encodable as `☸` or `☸`:



If we are able to retrofit a "Big Organization" complexity of how much communication is needed to prevent programmers from breaking others' work, perhaps no dingbat or emojicon is perfect. But we can speak of monolithic software projects having a quadratic communication complexity—of $\Theta(n \wedge 2)$, or if you prefer, $\Theta(n*(n-1))$—for developers to keep up with other changes and partially avoid collisions with other developers' work. On the scale of the OS/360 project, this resulted in developers spending over half their day only keeping track of memos, to keep abreast of what other programmers have done.

There is reason to believe that if the OS/360 project had been done with an approach like what Facebook uses for ReactJS plus Flux, there might have been no need to write *The Mythical Man-Month*.

The combination of ReactJS and Flux is explicitly written so that you don't need to put your hands in every other component's pockets. In fact, it is written so that if everybody is observing the approach, you can't put your hands in other components' pockets unless you find a way to breach the security. The communication complexity is not quadratic (as it was in the OS/360 project) at $\Theta(n \wedge 2)$ or $\Theta(n*(n-1))$; the number of directions of communication if you have at most one developer per component is vastly smaller, and may possibly be barely above linear at $\Theta(n \log n)$. The difference is enormous in its implications.

While it is not clear that Facebook is being insistently purist enough to achieve the best theoretically possible results, it does seem evident that Facebook—which is one of the largest organizations on the Internet, and may have a frontend development staff comparable to (or greater than) the OS/360 project in size—has communications that scale much better than the monolithic OS/360 endeavor. Perhaps Facebook is one of the many organizations that are quicker to advertise their strengths than their weaknesses. But nothing that I've been able to find in any resource on the Web suggests that the volume of communication between Facebook developers is out of hand, as it was in the OS/360 project, or that the necessary volume of internal communications is onerous enough to be an issue that makes developers' lives genuinely more difficult.

ReactJS is just a view, but what a view!

Charles Cézanne famously said, "Monet is just an eye, but what an eye!" Monet didn't try to show off his knowledge of structure and anatomy, but just copy what his eye saw. The consensus judgment of his work holds on to both "just an eye," and "what an eye!" And indeed, the details may be indistinct in Monet, who rebelled against artistry that tried to impress with deep knowledge of anatomy and knowledge of structure that is far beyond what jumps out to the eye.

ReactJS is a framework rather than a library, which means that you are supposed to build a solution within the structure provided by ReactJS instead of plugging ReactJS into a solution that you structure yourself. The canonical example of a library is jQuery, where you build a solution your way, and call on jQuery as it fits into a structure that you design.

However, ReactJS is specialized as a view. It's not that this is necessarily good or bad, but ReactJS is not a complete web development framework, and does not have even the intension of being the only tool you will ever need. It focuses on being a view, and in Facebook's offering, this does not include any form of AJAX call. This is not a monumental oversight in developing ReactJS; the expectation is that you use ReactJS as a View to provide the user interface functionality, and other tools to meet other needs as appropriate. This text hasn't covered using ReactJS together with your favorite tools, but do combine your favorite tools with ReactJS if they are not going to step on each other's feet. ReactJS may or may not collide with other Views, but it is meant to work with non-View technologies.

Programming is fun again!

When the Web first appeared, I had my first "Programming is fun again!" experience. I had gotten my bearings in programming in Unix and C, and when I was informed that it was possible to include an image in a web page, I anticipated the amount of work that it would take in a C-like environment from scratch to tell how to display an image. I implicitly thought, "Too much work for me." But I was positively astonished to find that an image could be included in a web page with nothing more than `<IMG SRC=Portrait.GIF>`, and that an image itself didn't need to be embedded in a web page; it could equally gracefully be made available as `<A HREF=Portrait.GIF>Click here!</A>`. Thus began my first exposure to a language that was declarative rather than imperative. Perhaps it was not strictly programming; certainly, before JavaScript, it was not a Turing approximant. However, it let me easily do things with computers that I hadn't dreamed of.

Some years later, I had my second "Programming is fun again!" experience after a friend suggested that I try Python. By the time that occurred, I was a sort of a language collector; the only languages I wanted to know but didn't were Icon, C++, and some assembler. A common reality for language collectors is that their first project in a new language is slower, more difficult, and more frustrating than any further work. It gets better after that, but for the first project, "It always takes longer than you think, even when you take into account the fact that it always takes longer than you think." However, with Python, my surprise was, "What? *Is it working already*?" and this was just the tip of an iceberg.

My story of being a language collector, finding Python, and then stopping the effort to learn new languages is not an especially unusual story among Pythoneers. Eric Raymond certainly picked up some deeper notes in his article Why Python? at `http://www.linuxjournal.com/article/3882`. Python is an enchanted kingdom where the streets are paved with glue, and it's not just masters who can benefit.

What was not mentioned earlier is that if you leave your mouse hovering over the cartoon image at `http://xkcd.com/353/`, this message appears: **I wrote 20 short programs in Python yesterday. It was wonderful. Perl, I'm leaving you...** Now Perl is also a good language and, for a time, my favorite, but there's still something about Python.

Finally, my last and greatest "Programming is fun again!" moment came when I began appreciating ReactJS. ReactJS delivers something that XHTML and HTML5 don't, in terms of creating useful components that can be used like tags.

Whatever the "X" in XHTML stands for, it does not mean, "In mainstream use, people will build and deploy lots of interesting new tags." HTML5 offers a number of new components, such as `<input type="date" />`, but they are not universally supported, and this is not another case of IE having to be the life of a party. The mainstream and current non-Microsoft browsers have very inconsistent coverage of features that were announced loudly and clearly when HTML5 was in the limelight. There are polyfills available, and integrating one of the many JavaScript date pickers that existed before HTML5 may make just as much sense today as it did when they were new. But while a site like `http://html5please.com/` is praiseworthy and worth using, it is also a symptom of a major problem.

ReactJS and JSX succeed where these fail. This text has not covered how to make a `<DatePicker />` function, but once this has been made, you can include it in your JSX almost as easily as a native HTML tag. If someone nostalgic for fractals draws on an HTML5 canvas and makes scrollable and zoomable `<LogisticMap />`, `<VonKochSnowflake />`, `<MandelbrotSet />`, and `<SierpinskiGasket />`, these can be as easily included in JSX as the plain old simple `<img />` tags. Components defined in ReactJS differ in an important sense from manually configuring and wiring a JavaScript date picker to work with your form. They are like subroutines in classic structured programming, in the sense that they can be readily reused wherever reusing makes sense, and combined to make bigger building blocks.



It might be suggested that there is money to be made in producing libraries of useful components that can be used to almost extend the basic set of useful tags open to other web developers.

Moreover, if I may borrow from Robin Martin's "What Killed Smalltalk Could Kill Ruby" and use slightly more polite language, the key metric in a code review (among other things) is the number of times the reviewer has to ask, "What were they thinking?" For the "What were they thinking?" metric, an acceptable score for the code being reviewed is 0. Anything higher than this is unacceptable. Also, this metric is relevant well outside of code reviews.

In Python, such moments are rare: they do exist, as a search for "Python mutable default argument" will show, but they are significant because they are rare. This is different from "What were they thinking?" arguments in JavaScript, such as "You can use variables without declaring them (but if you do, they'll be global)" and "You can write pseudo-classical constructors (but if you forget to use the new keyword when calling them, they'll clobber things in the global namespace.)" The environment in JavaScript is such that a key language advocate, such as Douglas Crockford, sharply warns people to steer clear of large chunks of the basic language, and appears to be getting pickier over time.

Ultimately, it seems that ReactJS and Python have the same heart. Both are, at heart, small and simple. Perhaps both have flaws, but flaws where the "What were they thinking?" moment is the exception and not the norm. Both have, as a sarcastic tweet said when ReactJS was announced, "Facebook: Rethink established best practices." ESR mentioned quite a hang-up about Python's strange choice of significant white spaces.

> *"And like most hackers, upon realizing this fact, I recoiled in reflexive disgust.*
>
> *I was barely old enough to have programmed in batch Fortran for a few months back in the 1970s. These days, most hackers aren't, but somehow our culture seems to have retained a pretty accurate folk memory of how nasty those old-style fixed-field languages were. Indeed, the term "free format" used back then to describe the newer style of token-oriented syntax in Pascal and C has almost been forgotten. All languages have been designed that way for decades now, or almost all; anyway. On seeing this Python feature, it's hard to blame anyone for initially reacting as if they had unexpectedly stepped in a steaming pile of dinosaur dung."*

ReactJS also has the courage to say that those people who create CSS can create very simple JavaScript, instead of only working in deliberately underpowered templating languages. Now, JavaScript was chosen as a Domain-specific Language to deliberately leave as much power as is needed. However, it is not necessary for designers to summon the full power of JavaScript. They can create the 99 percent of the simple JavaScript that would have been done in an underpowered templating language, and JavaScript developers can create the remaining 1 percent of powerful JavaScript, because this would be problematic to address in an underpowered templating language.

Summary

In this chapter, we looked at a slightly higher level than the nitty-gritty. Other chapters had detailed a couple of projects, but here we looked at some of the major wins that ReactJS represents, and to some of the most famous problems in computing.

This book is intended to cover both the theory and practice of functional reactive programming with Facebook's ReactJS. It is hardly the first title covering functional programming, reactive programming, or functional reactive programming, but it may be an early title among treatments of functional reactive programming that do not assume PhD-level mathematical prowess. Part of this was accomplished by making the text somewhat philosophical. In some sense, this is the price paid to be merely challenging for some veteran programmers to understand, but not impossible for most veteran programmers to understand. The best functional reactive programming in JavaScript and ReactJS is based on functional programming proficiency, and the best functional reactive programming in Haskell is also based on functional programming proficiency; there is no real difference there. There is, however, a difference in the fact that a typical veteran C++ programmer has a fighting chance of achieving a useful proficiency with ReactJS. And Facebook has done quite an impressive job of putting things in easier reach.

JavaScript is a versatile language, and you can get a significant amount of productivity if you are approaching it while thinking in a Scheme way (of course!) or a Python way, or C#, Erlang, Perl, Ruby, Java, Haskell, PHP, Lisp, or Visual Basic. Perhaps no other programming language's way of thinking will reach the upper echelons of pure, functionally driven JavaScript thinking, but there's a lot that you can say in JavaScript, without needing to be a native speaker sporting a flawless JavaScript accent!

And nothing of this is lost in ReactJS. Perhaps the last ounce of power cannot be squeezed out of functional reactive programming unless you have a very high proficiency in some very specific areas of math, but ReactJS has markedly lowered the barrier to entry for the benefits of functional reactive programming. Functional reactive programming used to have an unwritten sign over its door saying, "Math Programming Jocks Only." Now it doesn't. It may be purely an advantage to grok functional reactive programming to work with ReactJS, but all the other specialties mentioned can get a lot out of ReactJS without needing to know much math beyond the significant proficiency that is normally embedded in computer science and information technology.

Some people have said that developers very rarely pay for *books*; they pay for *chapters*. This book has been intended to work as a whole, with different parts illustrating complementary approaches to others so that each part adds to the whole. But it is also seriously intended to provide chapters that work perfectly well as standalone assets for people who want to orient themselves to something.

The next steps from here

There are countless directions that you can explore. You can dig deeper and explore the core of ReactJS. You can also explore the integration of ReactJS into projects that use other technologies to address other concerns.

Then, you can program ReactJS from Lisp or Python. (It's not merely true that you can program with ReactJS in JavaScript only if you come from a "Lispy" or "Pythonic" background. You can create animated web pages using ReactJS without leaving Lisp or Python to author a single line of JavaScript code.)

You can create a much richer set of components than any version of HTML5 can offer, and use them as easily as components that HTML 1.0 offered.

Perhaps the most exciting possibility of all is that ReactJS is no longer only for HTML/Web. It now offers a killer application for "learn once, write anywhere." Now your stellar JavaScript skills and hard work with learning functional programming unlock a good deal more than the Web. It's about as easy to write for iOS, for instance. Now see the home page at `https://facebook.github.io/react-native/`. This book has placed you in a position to not only use ReactJS on the Web but also learn ReactJS Native quickly and well, and this is profoundly significant.

You've climbed to the top of the diving board. Now it's time to jump in and make the biggest splash that's available.

Perhaps the best thing to be said about the best way to work with functional reactive programming and ReactJS is not to *work* with ReactJS at all. It is to *play* with ReactJS, as you play with fresh snow. What has been said of the Glasgow Haskell Compiler applies in entirety to ReactJS: forget that it's something you use for work. Play with it as if you were given a huge Lego set as a young child. See what you can build, and what you can't.

Programming is new again; there's pixie dust. Just like programmers once gained the ability to use programmer-contributed subroutines on par with built-in functions, now frontend web developers have gained the ability to use developer-made components as easily and with as little fuss as including an `IMG` tag in their JSX. Gone are the days when we needed to go to `http://html5please.com/` and learn that `<input type="date">` has an amber alert for a "Caution [even] with polyfill" warning label. Gone are also the days when you needed to—even worse manually—wire up a JavaScript date picker with your form on each page, and you might use 58 lines of repetitive-stress-injury-inducing code to get a date picker on one single page. Once someone has made a proper ReactJS `<DatePicker />` function, the problem is solved to the point that it is less than one line of code to include, and you can include it zero, one, or many times on a page. "Even. When. Deploying. With. A Shim. to Internet Exploder 8" was deliberate use of punctuation, as also used by Terry Pratchett in the speech of trolls, and is emphatic. As regarding the term "Internet Exploder", that title has been around for a long time and developers, or me at any rate, had two problems in implementing a solution that would work with any normal browser, and then again get things working, this time for the browser that was the, um, life of the party.

Wikipedia aims for a neutral "POV" (point of view), and unflinchingly writes of an earlier version:

This version of Internet Explorer has been widely criticized for its security issues and lack of support for modern web standards, making frequent appearances in "worst tech products of all time" lists, with PC World labeling it "the least secure software on the planet."[2] And this isn't even beginning to mention all of the pixie dust that means that you don't have to manage 

transitions, because you conceptually blow everything up and rebuild things as they should be at a particular point in time.

There is a world of possibilities to explore, and there is perhaps one thing to be said:

When this book began, the author's home was Python. When this book ended, the author's home was ReactJS.

A Node.js Kick start

In web development, it is desirable to have a backend server of some sort. The list of servers and languages available is extensive. But one server that has generated an extraordinary level of excitement is Node.js, which lets you use the same JavaScript for frontend and backend development, and allows truly interesting possibilities.

The purpose of this book is to introduce Facebook's frontend user interface framework, called ReactJS. The purpose of this appendix is to provide just enough of a backend to run an authenticated server, and while there are many good choices, Node.js works without an appendix asking you to deal with a new language. The basic work done in this appendix covers territory comparable to what you might approach with another server and backend language.

In this appendix, we will be covering these topics:

- How Node.js takes a cue from INTERCAL
- How Node.js, like JavaScript, has minefields
- Porting the Pragmatometer

But let's first look at Node.js and INTERCAL.

Node.js and INTERCAL

INTERCAL, properly named **The Compiler Language With No Pronounceable Acronym**, was first announced by Princeton students Don Woods and Jim Lyon in 1972. The archetypal example of a language designed to satirize various trends and fashions in programming languages, it is perhaps better known as the archetypal example of a language intended, not to be easy to work with, but to be deliberately and unnecessarily hard to use. Its *Hello, world!* code contains full, repetitive-stress-injury-inducing 16 lines; its legendary ROT-13 encryptor/decryptor (a straightforward one-liner for Perl or Unix shell commands) has been described on `alt.folklore.computers` as "four pages of completely undecipherable code." INTERCAL was originally published on punch cards in EBCDIC, a character encoding that has been called an encryption standard.

One trend that was satirized was Edgser Dikjstra's "Go to statement considered harmful," a work that was beyond being merely seminal, arguably being the single most important article in the history of computer science. Wags have said, for instance, that a programmer is someone who is offended on being told, "Go to hell!" not by the "hell" but by `Go to`. One INTERCAL variant (C-INTERCAL) took the premise that `Go to` statements are indeed rightly considered harmful, and they wanted to go as far away from `Go to` statements as possible—much farther than wimpy IF-THEN-ELSE statements and `while` loops. They offered a much more radical departure from the `Go to` statement than IF-THEN-ELSE and `while` loops—the `come from` statement. While the `Go to` statement says, "If the execution reaches this point of the code, go to that area of the code," the `come from` antonym says, "If the execution reaches that other area of the code, switch over and pick up things at this point of the code."

The suggestion that might be made is as follows: the genius of Node.js, for which we are providing a kick start in this chapter, is that it bends over backward to have flow control based on `come from` statements, or something very similar to them. Now, Node.js is also a server programmable in JavaScript, which is nothing to sneeze at, but it has completely eclipsed all other servers programmable in JavaScript. Its genius stems, not from JavaScript alone, but from a development environment that works best when you can solve problems in terms of `come from` as a primary tool for flow control. The usual preferred term is asynchronous callback function, rather than `come from`, but you will work best with Node.js when you realize that Node.js is interesting as a live example of `come from` programming being performant by default, and outperforming its competitors by an order of magnitude.

Someone who has formative C experience and comes to Perl might well be told, "You're not really thinking of Perl unless you're thinking of hashes," or an old-line Java programmer might be told, "You're not really thinking of JavaScript unless you're thinking of closures." In like fashion, someone coming from any other mainstream web server at all might be told, "You're not really thinking of Node.js unless you're thinking of `come from`-style asynchronous callback functions."



Anonymous functions have been glorious since they appeared in Lisp, and they're a great feature in JavaScript. But when dealing with Node.js callbacks, consider using non-nested named functions as an alternative to deeply nesting layers of anonymous inner functions.

Technically speaking, using asynchronous callback functions in Node.js is strictly optional. However, it may be strongly suggested that unless you are using a learning tool for Node.js—such as the excellent "Learn You the Node.js For Much Win!" (a title that clearly alludes to *Learn You a Haskell For Great Good* as an excellent predecessor), the Node.js learning tool being the one promoted at `http://nodeschool.io`—you should remember Knuth's two rules:

- **Rule 1 (for all programmers)**: Don't optimize
- **Rule 2 (for advanced programmers only)**: Optimize later

In the context of Node.js, this becomes:

- **Rule 1 (for all Node.js hackers)**: Don't use synchronous methods where an asynchronous approach would work
- **Rule 2 (for advanced Node.js hackers only)**: Add any synchronous functionality later

For an example of code implemented synchronously, the way a non-Node.js person would likely see, we can read and print a file such as `/etc/passwd` (on Windows, a different full path would be appropriate; you can create and save one with Notepad or your favorite editor):

```
var fs = require('fs');
var contents = fs.readFileSync('/etc/passwd');
console.log(contents);
```

Implemented with the `come from` asynchronous callback functionality:

```
var fs = require('fs');
fs.readFile('/etc/passwd', 'utf8', function(err, data) {
  if (err) {
    console.log('There was an error:');
    console.log(err);
    return;
  }
  console.log(data);
});
```

The question of whether `console.log()` is blocking or not does not concern us here.

This is a slightly more sophisticated *Hello, world!* program for node, or perhaps a program just after *Hello, world!* which, in Node.js, is simply as follows:

```
console.log('Hello, world!');
```

Let's comment on the asynchronous example in detail.

The standard way of importing a package is to call `require()`, and save the result in the variable you want to use to access the package. The `fs` package is one of the few packages that come automatically with Node.js, but Node.js comes packed with a whole universe of packages available through **Node Package Manager** (**npm**), a package manager that may appear familiar to people who use Linux package managers. With npm, you can search for, for example, Express.js, which will be covered briefly in this chapter. Express.js is popular among the Node.js community, works well with Node.js, and is a bit like Rails for Ruby or Django for Python. A search for Express.js can be like this:

```
npm search express
```

Once you've identified which package name you want (or you think you want), you can install it:

```
npm install express
```

In the preceding code, the `fs.readFile()` function call is what sets up the `come from` behavior. Like other asynchronous calls, it has two required arguments: a basic argument (possibly an array) that is given to `fs.readFile()`, and a callback function. What happens when this is called—instead of blocking for an expensive amount of time while the file is being read—is that the program registers a request to read a file with the specified parameters, and then the single Node.js leaves it where it is, and attends to other requests. This is very important. Instead of blocking and doing nothing while it waits, the program services other needs, and after the file operation comes back with the file result (having been busy in the meantime tucking in other requests), and then `come from` wherever it is and services the callback function provided. It is very hard to accidentally get properly used Node.js to block, except with something that burns the CPU, and with the current CPU speeds, it is rare for an innocuous request to block the CPU so much to be a problem (interested parties can use Node.js to mine bitcoins via tools such as those provided at `http://bitcoinjs.org/`, but presumably few people worried about their Node.js server being as performant as possible will have it mine bitcoin on the side). There is a cluster module intended to take advantage of multiple cores where Node.js, by default, runs in one single-threaded process on one core. But if you have any doubt about whether your use case is extreme enough to need something like cluster to perform Node's performant-by-default structure, you probably don't need cluster yet.

The way Node works (without using cluster, at least) has the additional advantage of avoiding concurrency issues, because it is single-threaded and de facto not concurrent. This is a very good thing. Concurrency is a slippery, treacherous thing to deal with. There are highly proficient programmers who work well with concurrency, but overall, concurrency should be considered harmful—a sort of Pandora's Box that perennially confuses most normal programmers. There may be a reason to consider purely functional languages using immutable data to be a separate case as far as concurrency goes, but here we will stick to saying that it is good that Node.js can, by default, handle an enormous number of requests without requiring the developer to cope with slippery concurrency difficulties.

The second argument given to `fs.readFile()` is optional, and `fs.readFile()` allowing it is a little unusual. Normal asynchronous calls look like identifier(data, callback). In this case, the second, optional argument is worth a closer look.

The argument is what encoding to use to make a string from an array of bytes, and the argument given is the normal default encoding, `'utf-8'`, although there is a slight temptation here to fall back, in this case on `'ascii'`. This is particularly because Unix `/etc/passwd` files precede `UTF-[anything]` by decades. But we will be good netizens and use `'utf-8'`.

The behavior would probably be similar, in our case, between using UTF-8 and ASCII encoding. The output will, in fact, be identical if `/etc/passwd` is like many `/etc/passwd` files over the years in containing ASCII characters alone and possibly supporting ASCII characters alone. But either of them will be a different beast by not specifying some encoding. Without further changes, the callback will be given in bytes rather than any JavaScript string in the ordinary sense. And here, we have a hint of something big about Node.js.

There has been an arms race for the fastest JavaScript engine among browsers, and Node.js took a release of Google Chrome's V8 engine (forks of Node.js may use something newer), and extended it in certain ways, so that it would be able to function as a general-purpose runtime environment, including being well-adapted to serving as a web server. This included adding several extensions that don't exist in client-side web browser JavaScript. The ability to handle sockets as a server was one example, and this, along with I/O, was brilliantly developed along an asynchronous model, as discussed earlier.

Another gap has to do with binary data. The standard browser JavaScript, at the time of writing this book, doesn't really provide a direct way to deal with binary data. While there may be obvious work of handling binary data behind the scenes in the following code, there is no clear way to say "I want 128 (octet) bytes of alternating 1s and 0s, starting with a 1":

```
<img id="portrait" />
<script>
(document.getElementById('portrait').src =
'http://cjsh.name/images/portrait.png');
</script>
```

Node.js extends V8's capabilities to handle proper binary data, and tried and true JSON is complemented with the new and binary-friendly BSON. The handling of binary data is low-level, possibly too low-level for its C-like character. For example, a productivity boost and decrease in frustration comes when a C programmer switches completely from using `malloc()` ("memory allocate") to using `calloc()` ("cleared memory allocate"). The `malloc()` function allocates a raw block of memory with whatever detritus was left over from the memory's previous occupant, leading to strange and wickedly magical effects if you fail to properly initialize any portion of the memory.

The `calloc()` function allocates a raw block of memory and clobbers any previous content with zeroes. Remember Pete Hunt's words, "I'd rather be predictable than right?" Stopping ever using `malloc()` directly again in favor of `calloc()` is a major way in which C programmers can opt to be predictable rather than right. However, out of misguided optimization concerns (it is a fraction of a second faster not to wipe the allocated byte memory), Node.js offers the equivalent of a C `malloc()` only, without any furnished `calloc()` equivalent, as far as I can tell. Fortunately, Node.js JavaScript (or C, for that matter) is so powerful that it is a straightforward exercise to port the `calloc()` functionality. Just make a wrapper that handles everything that the Node.js byte allocation handles, follows up by making all bits zeroes, and use this wrapper exclusively when you allocate bytes.

Returning to the immediately highlighted code, in Node.js thinking at least, reading from a file or from the network does not return a string. It returns binary bytes. Now those bytes may be easily convertible via a given encoding, and if you furnish the encoding you want, `fs.readFile()` will give you a proper string, not just bytes. But let's look at some code similar to what we had earlier. Node.js, like many good environments, offers a **Read-Eval-Print-Loop** (**REPL**) to try things out (invoking the node executable without any following arguments will activate the REPL). From the REPL:

```
> fs.readFile('/etc/passwd', function(err, data){console.log(data)});
undefined
> <Buffer 23 23 0a 23 20 55 73 65 72 20 44 61 74 61 62 61 73 65 0a 23
20 0a 23 20 4e 6f 74 65 20 74 68 61 74 20 74 68 69 73 20 66 69 6c 65
20 69 73 20 63 6f 6e 73 ...>
```

The various bytes in the file are represented in hexadecimal code.

Returning to our last code sample, the `function(err, data)` callback signature is the normal callback signature for the programming contract. The callback should be eventually called, and perhaps called very quickly. This should be done with either a "truthy" `err`, in which case the callback should optionally take steps to respond to any information contained in the error feedback, and non-optionally return without going further, or a null err, in which case the function's precondition is met and the callback should take whatever action is appropriate to receive the data that was requested. The preceding code illustrates the pattern: check for null `err`, optionally act on it by logging diagnostics, and print the file contents if err is null.

Warning – Node.js and its ecosystem are hot, and hot enough to burn you badly!

When I was a teacher's assistant, one of the nonobvious suggestions I was told was not to tell a student that something was "easy." The reason was somewhat obvious in retrospect: if you tell people that something is easy, someone who doesn't see a solution may end up feeling (even more) stupid, because not only do they not get how to solve the problem, but also the problem that they are too stupid to understand is an easy one!

There are gotchas that don't just annoy people coming from Python/Django, which immediately reloads the source if you change anything. With Node.js, the default behavior is that if you make one change, the old version continues to be active until the end of time or until you manually stop and restart the server. This inappropriate behavior doesn't just annoy Pythonistas; it also irritates native Node.js users who provide various workarounds. The StackOverflow question "Auto-reload of files in Node.js" has, at the time of this writing, over 200 upvotes and 19 answers; an edit directs the user to a nanny script, node-supervisor, with homepage at `http://tinyurl.com/reactjs-node-supervisor`. This problem affords new users with great opportunity to feel stupid because they thought they had fixed the problem, but the old, buggy behavior is completely unchanged. And it is easy to forget to bounce the server; I have done so multiple times. And the message I would like to give is, "No, you're not stupid because this behavior of Node.js bit your back; it's just that the designers of Node.js saw no reason to provide appropriate behavior here. Do try to cope with it, perhaps taking a little help from node-supervisor or another solution, but please don't walk away feeling that you're stupid. You're not the one with the problem; the problem is in Node.js's default behavior."

This section, after some debate, was left in, precisely because I don't want to give an impression of "It's easy." I cut my hands repeatedly while getting things to work, and I don't want to smooth over difficulties and set you up to believe this: getting Node.js and its ecosystem to function well is a straightforward matter, and if it's not straightforward for you too, you don't know what you're doing. If you don't run into obnoxious difficulties while using Node.js, that's wonderful. If you do, I would hope that you don't walk away feeling, "I'm stupid. There must be something wrong with me." You're not stupid if you experience nasty surprises dealing with Node.js. It's not you! It's Node.js and its ecosystem!

Next, we will explore a sample project, a remote equivalent of the quick and dirty localStorage-based persistence, which was covered in *Chapter 11*, *Demonstrative Functional Reactive Programming in JavaScript with a Live Example Part IV – Adding a Scratchpad and Putting It All Together*. That was a success, but it was way too hard with way too many trip-ups along the way. I have, at times, compared Python and JavaScript; but it may be worth a moment to look at why JavaScript's Node.js is really nasty compared to Python's Django.

My first experience with Django, after years of experience, was a feeling that it was a brilliant power tool that, for some strange reason, had accidentally not been enshrined in Python's standard library. Now, in fact, there is excellent reason for this, and one that need criticize neither Python nor Django: as Python's Benevolent Dictator for Life observed, you put something into the standard library when it is "dead," not when it is still growing. Django is still growing and it's still getting better. Therefore, Django does not belong in the standard library for Python no matter how good it is.

There is, for many contexts, a principle of least astonishment, and once you start to know Python well, it does not give you unpleasant surprises too often. Django does come with some surprises, such as its templating system, which at the time of ASP and JSP was a striking proposition. (Now it has had its 15 minutes of fame, and even Python/Django developers start off by replacing the templating system with something more powerful. It was entirely the right choice for ReactJS to essentially do the opposite.) But paradoxically, both Django and ReactJS offered templating that reflected a genius from Mars technique, as defined in The New Hacker's Dictionary:

> *[TMRC] A visionary quality which enables one to ignore the standard approach and come up with a totally unexpected new algorithm. An attack on a problem from an offbeat angle that no one has ever thought of before, but in retrospect makes perfect sense. Compare grok, zen.*

Working with Node.js doesn't feel anything like working with Python, or even like working with ReactJS. It is more frustrating, is more difficult, and has more things that don't make sense.

Here is one example: at the time of the initial research, I intended to use passport.js to offload the dirty work of authentication. I originally meant to use Facebook authentication, but the instructions involved creating something on the Facebook developer site, and taking down information from the Facebook application. And even after exploring the Facebook developer site and asking, "passport.js says to get XYZ information from my application on the Facebook developer site," I completely failed to obtain any timely answer.

Scaling back my ambitions, I decided to use the most basic proper authentication from passport.js—username and password—until I learned that what was provided as username and password support was almost entirely useless.

The reason it is useless is that just as **Create, Read, Update, Destroy** (**CRUD**) offers an enumeration of basic responsibilities—bases that need to be covered in any serious and complete tool for handling data and records (whether it is an SQL database, any stripe of a NoSQL database, pickled data saved in a programming environment, an editor, or an e-mail client)—there is a basic set of bases that need to be covered in mainstream account management, whether username/password or any newer and gentler alternative to making users keep track of yet another login and password. While individual sites may opt out of certain functionality, repeated and basic functionality that provide the CRUD of account management includes the following:

- Allowing users to create new accounts
- Allowing users to log in with an existing account
- Replacing a lost password without an unencrypted password being e-mailed to the user
- Possibly an extended set of features for a site's administrative members, such as account moderation (if desired), locking and unlocking accounts, and account deletion

The only one of these bases covered by passport.js's functionality is logging in with an already existing account, created by some means that I have not been able to ascertain, and being successfully or unsuccessfully authenticated. Perhaps the only thing more pathologically incomplete in terms of a CRUD was offered in the April 1 issue of Byte magazine some decades ago, when someone advertised an exceptionally good deal on write-only memory.

Now we might note in passing that supporting 100 percent of CRUD is not, strictly speaking, the only possible approach. Years ago, **Write Once, Read Many** (**WORM**) disk drives spent some time in the limelight. While there is possibly no modern laptop that has shipped with a genuine WORM drive, ClojureScript includes an extraordinary amount of effort to offer WORM data. WORM, in this context, means that data is designed to exclude Updating (although you can make modified copies easily enough), and Deletion is reserved to garbage collection: out of full CRUD support, ClojureScript's WORM data only allows unimpaired Creation and Reading of data. ClojureScript reflects a carefully thought-out decision to offer WORM data in a context where full CRUD support would have been considerably easier. This decision now deserves obvious respect.

Now, the lack of an equivalent of full CRUD support is not the end of the world for authentication, as at least one other group has approached "authentication CRUD" more appropriately. Stormpath advertises offerings for Node, Python, Java, and REST. One of their developers rewrote my code for authentication for me to use their system. While this may just be being nice to an author who might cover their product, Stormpath inclusion isn't even a proper integration challenge; it's really simple. Now it should be stated for the record that Stormpath is not open source, but a SaaS with freemium pricing. The full-featured, free, and "no credit card required" developer tier has a quota of 100,000 API calls per month, and they estimate user logins as using about three API calls. They definitely have a profit motive, but if you have enough traffic for you to need a paid tier of service, you shouldn't care about the trifle you have to pay them. The system gives an overall impression of being a bit young, with people working out the remaining kinks, but it has happened that someone told them politely about something that was immature, and the problem was resolved quickly.

Another basic difficulty surrounds databases. There's a good case to be made that MongoDB is important, and together with the "access MongoDB from Node.js" package of mongoose, it is worth the learning curve. In preparation for this chapter, top-of-search tutorials proved to explain how to create a Schema and save something in it, but left guesswork as to how to usefully approach a database where all the necessary schemas already exist. Subsequent work turned up an existing Stack Overflow solution that appeared to cover database CRUD with a database with already existing schemas and databases. I may have come within an inch of pay dirt before giving up, but I intended to use mongoose/MongoDB for its database work almost from the beginning, and I have not yet attained proficiency.

Another database that seemed like a perfect fit, and may yet be redeemable, is a server-side implementation of the HTML5 key-value store. This approach has the cardinal advantage of an API that most good frontend developers understand well enough. For that matter, it's also an API that most not-so-good frontend developers understand well enough. But with the `node-localstorage` package, while `dictionary-syntax` access is not offered (you would want to use `localStorage.setItem(key, value)` or `localStorage.getItem(key)`, but not `localStorage[key])`, the full localStorage semantics is implemented, including a default 5 MB quota. Why? Do server-side JavaScript developers need to be protected from themselves?

For client-side database capabilities, a 5 MB quota per website is really a generous and useful amount of breathing space to let developers work with it. You could set a much lower quota and still offer developers an immeasurable improvement over limping along with cookie management. A 5 MB limit doesn't lend itself very quickly to big data client-side processing, but there is a really generous allowance that resourceful developers can use to do a lot. On the other hand, 5 MB is not a particularly large portion of most disks purchased any time recently. This means that if you and a website disagree about what the reasonable use of disk space is, or if a site is simply hoggish, it does not really cost you much and you are in no danger of a swamped hard drive, unless your hard drive was already too full. We may be better off if the balance were a little less, or a little more, but overall it's a decent solution for addressing the intrinsic tension with respect to a client-side context.

However, it might gently be pointed out that when you are the one writing code for your server, you don't need any additional protection from making your database more than a tolerable 5 MB in size. Most developers will neither need nor want tools acting like nannies and protecting them from storing more than 5 MB of server-side data. Also, this 5 MB quota, which is a golden balancing act on the client side, is rather a bit silly on a Node.js server.

Moreover, for a database for multiple users—such as is covered in this appendix—it might be pointed out, slightly painfully, that it's not 5 MB per user account, unless you create a separate database for list on each account. It is 5 MB shared between all user accounts together. This could get painful if you go viral!

The documentation states that the quota is customizable, but an e-mail a week ago to the developer asking how to change the quota is unanswered, as was a Stack Overflow question asking the same thing. The only answer I have been able to find is in the GitHub CoffeeScript source, where it is listed as an optional second integer argument to a constructor. This is easy enough, and you can specify a quota equal to the disk or partition size. But besides porting a feature that does not make sense, the tool's author has also completely failed to follow a very standard convention of interpreting 0 as "unlimited" for a variable or function wherein an integer is to specify the maximum limit for the concerned resource use. The best thing to do with this disfeature is probably to specify that the quota is infinity:

```
if (typeof localStorage === 'undefined' || localStorage === null)
  {
  var LocalStorage = require('node-localstorage').LocalStorage;
  localStorage = new LocalStorage(__dirname + '/localStorage',
    Infinity);
  }
```

A similar amateur roughness kept cropping up in elements that surfaced in my research. Express.js is of a higher level than Node.js, but in terms of ways of shooting yourself in the foot, Node.js is closer to offering C's way than any other technology I have used recently. C is for those who prefer to load their own rounds before shooting themselves in the foot.

A sample project – a server for our Pragmatometer

Let's work toward a simple project. We will create a generic server backend that can serve a modification of the Pragmatometer project covered in the last chapters of this book, which handled persistence by saving and restoring from a few JSON strings locally in HTML5 localStorage. We will work on a server that can provide static content, like what has already been developed, provide an API to save or restore a string and an identifying key, and handle basic authentication and account management. The client-side programming should be barely more interesting than it was before, essentially by swapping saving to localStorage to saving to our remote Node.js server.

We will work with several technologies. The most attention will be given to working within Express.js compared to, for instance, Stormpath. Stormpath appears not to have taken credit for inventing something fundamentally new, original, or stunning, or for a breakthrough in an authentication mechanism. They might perhaps take credit for solving a well-known problem in such a way that a large chunk of busy work will be taken off your plate. Adding Stormpath is small and unobtrusive. Most users will not use it as a platform to build some great work on top of. Consequently, we will give significant attention to Express.js (and getting our client to talk to Express.js), which is a platform to work with. On the framework's site, Express.js is promoted as a "fast, unopinionated web framework for Node.js." They pretty much deliver what they boast of.

We will need to build a server, but also alter the client side for the Pragmatometer project in chapter 8 to chapter 11 There are `save()` and `restore()` functions, and they will be altered and expanded.

Install Express.js via `npm install express`. Then create an express project using `express [the directory name for your project]`. You will have a framework fleshed out. You can add packages to the `package.json` file, and run `npm install` to populate your local copy. There will be a public or static directory that you can make available, and `routes/index.js` handles routing in a way that people who know other frameworks may feel like home.

Client-side preparations

Everything that was under the `js/` directory in chapter 8 to chapter 11 is moved to `public/javascripts`. The full details of the changes will be posted on the website. Here, we adapt the `save()` and `restore()` functions from being (client-side) localStorage-specific to retaining localStorage for a slight perceived speed boost, but restore from and save to a remote server. In this case, the server is a Node.js server built with Express.js, but it could essentially be any server serving the same, simple, and implicit API.

Ordinarily, with ReactJS, an object's state is set within a `setInitialState()` call. Theoretically, we can preserve the relevant semantics by loading with the synchronous equivalent of an Ajax call, but it's also possible to populate a stub and then make available a callback that will really set things in motion. The function used to populate an object's state upon successful return from an Ajax call is `populate_state()`:

```
var populate_state = function(state, new_data) {
  for (field in new_data) {
    if (new_data.hasOwnProperty(field)) {
      state[field] = new_data[field];
    }
  }
state.initialized = true;
}
```

The `restore()` function is slightly complex, but that's because it is written to build perceived layers of responsiveness. It makes an Ajax call, setting a state as initialized and marking `state.initialized` as `false`. It also restores from JSON (if anything has been saved). Its check of whether localStorage is available, and graceful degradation if it isn't, is probably historical, as ReactJS is only claimed to work with browsers (IE8 and higher) that offer localStorage. Nonetheless, it provides an example of how we might go about graceful degradation:

```
var restore = function(identifier, default_value, state, callback) {
  populate_state(state, default_value);
  state.initialized = false;
  var complete = function(jqxhr) {
    if (jqxhr.responseText === 'undefined' || jqxhr.responseText.
length &&
      jqxhr.responseText[0] === '<') {
      // We deliberately do nothing.
    } else {
      populate_state(state, JSON.parse(jqxhr.responseText));
    }
    callback();
```

```
        state.initialized = true;
      }
      jQuery.ajax('/restore', {
        'complete': complete,
        'data': {
          'identifier': identifier,
          'userid': userid
        },
        'method': 'POST'
      });
      if (Modernizr.localstorage) {
        if (localStorage[identifier] === null ||
  localStorage[identifier]
          === undefined) {
          return default_value;
        } else {
          return JSON.parse(localStorage[identifier]);
        }
      } else {
        return default_value;
      }
    }
```

As a limitation of scope, neither the implementation of the `restore()` function in the preceding code, nor the `save()` function in the following code address resilience in the face of a failed Ajax call (or calls). One way of addressing this concern is to check for failure and keep retrying, with exponentially increasing delays between retries to be a good netizen and not add persistent heavy traffic to the network. This pattern is followed (roughly) at a high level by Gmail and at a low level baked into TCP/IP. For our implementation, anything that a failed Ajax call might not have conveyed should be available afresh in the key-value store, unless there has been a subsequent update, in which case both changes will usually be saved.

The `save()` function is slightly simpler, but it represents the other side of a coin: make an Ajax call to save/restore, and save to and restore from localStorage as an approximation before it is available:

```
    var save = function(identifier, data) {
      if (Modernizr.localstorage) {
        localStorage[identifier] = JSON.stringify(data);
      }
      jQuery.ajax('/save', {
        'data': {
          'data': JSON.stringify(data),
          'identifier': identifier,
```

```
          'userid': userid
        },
        'method': 'POST'
      });
    }
```

While we are pulling stuff from localStorage, we try to prevent the user from being able to enter data. This is because under unpredictable race conditions, this data gets clobbered when data from Ajax calls comes back. In other words, the user is blocked from adding any input until data is restored from Ajax (even if a value has already been restored from localStorage). This means, in particular, that submit buttons are disabled, and for now, the only application for which the callback function is given to `restore()` is for enabling submit buttons that have been disabled. For the calendar, the `render()` method has a disabled **Submit** button (you can be more purist and disable all the input fields, but disabling the **Submit** button is enough to prevent user data from being clobbered by race conditions):

```
render: function() {
  var result = [this.render_basic_entry(
    this.state.entry_being_added)];
  if (this.state.entry_being_added &&
    this.state.entry_being_added.hasOwnProperty('repeats') &&
    this.state.entry_being_added.repeats) {
    result.push(this.render_entry_additionals(
      this.state.entry_being_added));
  }
  return (<div id="Calendar">
    <h1>Calendar</h1>
    {this.render_upcoming()}<form onSubmit={
    this.handle_submit}>{result}
    <input type="submit" value="Save" id="submit-calendar"
      disabled="disabled" /></form></div>);
},
```

The Calendar's `getInitialState` function only arranges for a bare stub of data to be synchronously put in place. The Ajax call, upon returning, gives it a more appropriate value and re-enables the disabled save button, as race conditions are no longer a concern here:

```
getInitialState: function() {
  default_value = {
    entries: [],
    entry_being_added: this.new_entry()
    };
```

```
        restore('Calendar', default_value,
          default_value, function()
          {
          jQuery('#submit-calendar').prop('disabled', false);
          });
```

There are a few more details on the client side, but they are not particularly difficult. For instance, we add a Logout link (positioned with CSS to be at the top right), and the JavaScript behavior (without calling the usual `preventDefault()` method, because we do not want to prevent the default behavior) of erasing account data from the key-value store:

```
jQuery('#logout').click(function() {
  if (Modernizr.localstorage) {
    localStorage.removeItem('Calendar');
    localStorage.removeItem('Todo');
    localStorage.removeItem('Scratch');
  }
});
```

The server side

When we need packages, we should add them to our `package.json` file. One way of doing this is backwards. Perform npm install XYZ and then add a line to the `package.json` file under "dependencies," specifying "XYZ": "~1.2.3" and recording the version number for the installation. The dependencies presently included are as follows:

```
{
  "name": "Pragmatometer",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "connect-flash": "~0.1.1",
    "debug": "~2.2.0",
    "ejs": "~2.3.2",
    "express": "~4.12.4",
    "express-stormpath": "~1.0.5",
    "jade": "~1.9.2",
    "morgan": "~1.5.3",
```

```
    "serve-favicon": "~2.2.1",
    "stormpath": "~0.10.0"
  }
}
```

We create an account at `https://stormpath.com/`, probably a free developer account (unless you know you need more), and specify the various details in `app.js`. This setup uses HTML-like EJS over Markdown-like Jade for views:

```
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'ejs');

app.use(logger('dev'));
// uncomment after placing your favicon in /public
app.use(favicon(__dirname + '/public/images/favicon.ico'));
app.use('/public', express.static(path.join(__dirname, 'public')));

// Authentication middleware.
app.use(stormpath.init(app, {
  apiKeyId: '[Deleted]',
  apiKeySecret: '[Deleted]',
  application:
    'https://api.stormpath.com/v1/applications/[Deleted]',
  secretKey: '[Deleted]',
  sessionDuration: 365 * 24 * 60 * 60 * 1000
  }));

app.use('/users', users);
```

All but one of the items marked `[Deleted]` are things that you get from your setup on Stormpath. Some people have advised trying to be clever in making your own secret key; Don't! Under Mac, Unix, Linux, or Cygwin (Cygwin is freely available from `http://cygwin.org` and runs under Windows), pull up Command Prompt and type the following line:

```
python -c "import binascii; print binascii.hexlify(open('/dev/random').
read(1024))"
```

This will get you a kilobyte of cryptographically strong and random data that is encoded to be copy and paste friendly.

Here's a note on hygiene: the recommended practice is to be very careful with your secret key and, in particular, not to include it in version control. Instead, put it into a dot-file directory under your home directory with permissions that don't let anyone else do anything with it.

Probably, the one file with the heaviest work in it is `routes/index.js`. We pull in several dependencies, including a body parser that will be able to get data out of Ajax saves that POST JSON in the body of the request:

```
var body_parser = require('body-parser');
var json_parser = body_parser.json();
var express = require('express');
var stormpath = require('express-stormpath');
```

We include localStorage , specifying Infinity as our quota, and then provide a sanitizer for characters in keys. This specific sanitizer leaves alphanumeric characters intact, which with the rest of the application is sufficient to ensure that it doesn't make key collisions. It also ensures that the characters are on a whitelist that excludes colons. This allows us to create keys with names similar to `username:component-name`, perform a string split on colons, and always get the username in the zeroth slot and a component name in the first slot:

```
var sanitize = function(raw) {
  var workbench = [];
  for(var index = 0; index < raw.length; ++index) {
    if
('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_.'
      .indexOf(raw[index]) !== -1) {
      workbench.push(raw[index])
    }
  }
  return workbench.join('');
}
```

The router works in a way that should seem familiar to users who've seen routers in almost any context before. Although route and non-route functions will be mixed, the router is created and connected to the first two routes:

```
var router = express.Router();

router.get('/', stormpath.loginRequired, function(request, response) {
  response.render('index.ejs');
});
```

```
router.post('/', stormpath.loginRequired, function(request, response)
{
  response.render('index.ejs');
});
```

The included `stormpath.loginRequired`, once Stormpath is included, is really all that you need to get a view to be login protected. We go ahead to define two non-view functions: the functions used to `save()` and `restore()` a key for a particular user:

```
var save = function(userid, identifier, value) {
  localStorage.setItem(sanitize(userid) + ':' + sanitize(identifier),
value);
  return true;
}

var restore = function(userid, identifier) {
  var value = localStorage.getItem(sanitize(userid) + ':' +
    sanitize(identifier));
  if (value) {
    return value;
  } else {
    return 'undefined';
  }
}
```

We add the routes that will service the POST Ajax requests. If we want to add support for GET or other verbs, we can call `router.get()` and the like:

```
router.post('/restore', json_parser, function(request, response, next)
{
  var result = restore(request.user.href, request.body.identifier);
  response.type('application/json');
  response.send(result);
});

router.post('/save', function(request, response, next) {
  var success_or_failure = save_identifier(request.user.href,
  request.query.identifier,
    request.query.data);
  response.type('application/json');
  response.send(String(success_or_failure));
  });
```

Then there is a boilerplate line that we keep intact:

```
module.exports = router;
```

We also use Express.js's hierarchy for static data; the revised `index.ejs` pulls from a place different from our earlier js/:

```
    <script
      src="//cdnjs.cloudflare.com/ajax/libs/react/0.13.1/react-with-
addons.js">
    </script>
    <script
      src="//cdnjs.cloudflare.com/ajax/libs/showdown/1.0.2/showdown.
js">
    </script>
    <script src="//cdn.ckeditor.com/4.4.7/basic/ckeditor.js"></script>
    <script src="//code.jquery.com/jquery-1.9.1.js"></script>
    <script src="/public/javascripts/json2.js"></script>
    <script src="/public/javascripts/modernizr.js"></script>
    <script src="/public/javascripts/site.js"></script>
```

And that's it! We provide full details in the electronic resource packet. Now we have provided a server-side key-value store with account management.

Summary

When this appendix was being contemplated, one of the questions I considered was "JavaScript plus Node.js or Python plus Django?" The focus of this book is on a frontend with ReactJS, and the focus on the backend was just to have enough to support the frontend. I naturally thought that Python is so easy, even to newcomers, and Django also is so easy (again, even to newcomers) that even with the introduction of a new language, a basic key-value store with authentication should be an easy appendix to read and to write. However, the author thought then that I would go the high road of JavaScript plus Node.js, the combination everybody wants in on, and I have been paying for his decision not to provide a Python-plus-Django appendix ever since.

The code provided in a bundle is, of course, freely offered for any mileage you can pull from it that won't violate Packt Publishing's licensing. But the basic task of implementing a key-value store with account management is perhaps of the level of an undergraduate's homework assignment. It does not, in any sense, demonstrate the peaks of wonder offered by any server.

Now, Node.js really does offer peaks of wonder. These, however, were not explored here because the goal was to provide just enough "Node.js plus Express.js" to create a server-based adaptation of the Pragmatometer project covered in chapters 8 to 11. Moreover, given the degree of enthusiasm and the sheer work hours on all projects, it may be warranted in a year, 2 years, or 3 years to sharply temper any remarks about an immature ecosystem made at the time of writing this book. After 5 years, it may really make sense to say, "The 2015 Node.js ecosystem had several minefields. The 2020 Node.js ecosystem has multiple paradises."

But to publish, as passport.js did, to have a simple animation between `passport.authenticate('twitter')`, `passport.authenticate('google')`, `passport.authenticate('facebook')`, and so on slide by with a quite seductive apparent ease, and then have the user searching and asking at length for a passport.js way to handle username-password authentication that allowed users to create a new account, doesn't happen. This is extremely inappropriate, and it is inappropriate in a way that happened more than once in the Node.js ecosystem. The transition between finding a Node.js tool with a slick website that seems to offer exactly what you need, and activating "Hello, world!" levels of functionality meets perhaps 50 percent success. It represents a gulf bigger than I anything have seen in the entire history of the Web.

I can see people thinking, not exactly that I am brilliant because of how I made the to-do list with multiple statuses available for each item, but that I was practical, and overall the book significantly reduced the amount of legwork involved in the readers' getting up to speed with ReactJS. However, I would be mystified if people told me that I was brilliant, because I thought of making an authenticated key-value store, as covered in this appendix. The achievement is not at all because I managed to get some technology to work as an authenticated key-value store—which is a task on par with undergraduate homework—but because it was accomplished in an environment that exists in continuity with INTERCAL.

People have needlessly shot themselves in the foot by constantly using JavaScript as a whole, and crucial to JavaScript being made a respectable language was Douglas Crockford saying, in essence:

> "*JavaScript as a language has some really good parts and some really bad parts. Here are the good parts. Just forget that anything else is there.*"

Perhaps the hot Node.js ecosystem will grow its own "Douglas Crockford," who will say:

The Node.js ecosystem is a coding Wild West, but there are some real gems to be found. Here's a road map. Here are the areas to avoid at almost any cost. And here are the areas with some of the richest pay dirt to be found in any language or environment.

Perhaps someone else can take these words as a challenge, follow Crockford's lead, and write *The Good Parts* and/or *The Better Parts* for Node.js and its ecosystem. I'd buy a copy!

Index

Symbols

10,000-foot
  overview 1
  user interface programming, handling 2
.length method 23

B

Bacon.js 110, 112
Big-Coffee Notation 40-42
Brython
  about 112-115
  URL 112

C

calendar system
  about 131
  boring code, writing 169
  calendar entries, adding 157-159
  calendar entries, displaying 179
  calendar entries, segregating 178
  challenges 154-156
  day activities, sorting 180, 181
  dropdown menus, adding for months 174, 175
  greatest one-time calendar entry, searching 178
  holidays, accommodating 183
  interface, displaying for repeating calendar entries 172, 173
  Markdown, using 181
  Plain Old JavaScript Objects, using 165, 166
  progressive disclosure, implementing 166
  render() method, using 168
  successor() function, using 175-177
  Todo, displaying 182
  UI, creating for non-recurring entries 170, 171
  user interface, creating 160-164
CKeditor
  about 190
  including, in page 190, 191
  state, persisting 198, 199
  URL 190
  using 189
ClojureScript 108, 109
complementary tools
  URL 108
componentDidMount() function 55
computer folklore
  history 90-92
content distribution network (CDN) 48
Create, Read, Update, Destroy (CRUD) 220
custom sort functions
  about 69-71
  array.filter() 72, 73
  Array.prototype, extending 76, 77
  filter 75
  filter toolbox 78
  filter windows 81
  global pollution, avoiding 77
  Illusionism 74, 75
  map 75-78
  reduce 75-78
  reduce function 78-80
Cygwin
  URL 228

D

data types, Immutable.js
collection 116
IndexedCollection 116
IndexedIterable 116
IndexedSeq 116
Iterable 116
KeyedCollection 116
KeyedIterable 116
KeyedSeq 116
List 116
Map 116
OrderedMap 116
OrderedSet 116
Record 116
Seq 116
Set 116
SetCollection 117
SetIterable 117
SetSeq 117
Stack 117
declarative programming 33
Django templating
backend, URL 39
frontend, URL 39
Domain-specific Language (DSL)
JavaScript as 39, 40

E

ECMAScript 6
about 19, 29
URL 29

F

Facebook
BDD unit testing 122-125
flow control 19
Flux Architecture
about 35
implementing, with Fluxxor 126
Fluxxor
URL 126
used, for implementing Flux Architecture 126
function
about 19, 22, 98
comments 24-28
functional reactive programming (FRP)
about 1, 96-102
actual display, triggering 142, 143
complete component 136-138
demonstrating, example 129-135
features , distinguishing 95
features, distinguishing 96
render method 139

G

game with multiple ports
history 47
generators
ECMA6-proposed syntax, URL 72
get_level() function 54
Google Chrome
installing 6-9
URL 6
Google Maps 201
green field project 130

H

Haskell
URL 92, 100
hasOwnProperty() method 29
Heisenbugs
about 33, 34
URL 33
Hello, World!
advanced prerequisites 92-95
HTML, for web page
about 47, 48
content distribution network (CDN), using 48
minimal page body 50, 51
simple styling 48-50

I

IE8 support
URL 196

Immediately Invoked Function Expression (IIFE)
about 51
function, for creating game levels 54, 55
function, for game restart 53
function, for game start 53
GAME OVER screen 62, 64
initialization 52, 53
ReactJS classes, using 55-59
tick() function 59-62
variable declaration 52, 53
Immutable.js
about 115, 116
data types 116
URL 115
Immutable.List methods
asImmutable 117
asMutable 117
butLast 117
concat 117
contains 117
count 117
countBy 117
delete 117
deleteIn 117
entries 117
entrySeq 117
equals 117
every 118
filter 118
filterNot 118
find 118
findIndex 118
findLast 118
findLastIndex 118
first 118
flatMap 118
flatten 118
forEach 118
fromEntrySeq 118
get 118
getIn 118
groupBy 118
has 118
hashCode 118
hasIn 118
indexOf 118
interleave 118
interpose 118
isEmpty 118
isList 118
isSubset 119
isSuperset 119
join 119
keys 119
keySeq 119
last 119
lastIndexOf 119
List 119
map 119
max 119
maxBy 119
merge 119
mergeDeep 119
mergeDeepIn 119
mergeDeepWith 119
mergeIn 119
mergeWith 119
min 119
minBy 119
of 119
pop 119
push 119
reduce 119
reduceRight 120
rest 120
reverse 120
set 120
setIn 120
setSize 120
shift 120
skip 120
skipLast 120
skipUntil 120
skipWhile 120
slice 120
sort 120
sortBy 120
splice 120
take 120
takeLast 120
takeUntil 120
takeWhile 120
toArray 120

toIndexedSeq 121
toJS 121
toKeyedSeq 121
toList 121
toMap 121
toObject 121
toOrderedMap 121
toSeq 121
toSet 121
toSetSeq 121
toStack 121
update 121
updateIn 121
values 121
valueSeq 121
withMutations 121
zip 121
zipWith 121
information hiding
JavaScript closures, using 85-87
JavaScript, using 82-84
INTERCAL 212-217

J

JavaScript
about 67-105
arrays, sorting 69-71
as DSL 39, 40
for page animation 51
information hiding 82-84
URL 105
Jest 122-125
JSON
URL 195
JSX scripts
URL 51

L

linear 41
loops 28, 29

M

memoization 74
Monads
URL 95

N

NaN 19-22
Node.js
about 212-217
installing 10-15
limitations 218-223
URL 6, 10
Node Package Manager (npm)
about 214
jsx, URL 51
novelty flowchart
URL 84

O

Observer pattern 5
Om 110

P

persistence 194-197
Plain Old Simple HTML (POSH) 57
Pragmatometer project
client-side preparations 224-227
server, creating 223
server side preparations 227-231
URL 130, 131
programming
evolution 205, 207
language 104

Q

quadratic 41

R

reactive programming 31, 32
ReactJS
about 189
advantages 201
as view 205
declarative programming 33
properties 56
Starter Kit, installing 16-18
state 56
URL 32

Read-Eval-Print-Loop (REPL) 217
reduce function 78-81
render() method
about 96, 191
implementing 139-141
restore() function 196
Reverse Polish Notation (RPN)
URL 47
Room to Grow 131

S

Scratchpad 131
second system effect
URL 35
sesquipedalian 98
shouldComponentUpdate() method 191
Showdown library
URL 133
side effect 99
sort() function 69
Starter Kit
for ReactJS, URL 6
installing, for ReactJS 16-18
URL 16
strict mode 19, 20
subcomponent
integrating, into page 192-194

T

Todo class 147
to-do list
about 131, 145
adding, to application 146
appropriate initial state, setting 147
colors, using 151
inner functions. using 148, 149
ReactJS add-ons, including 147
render() function, used for heavy lifting 148
result, rendering 151
result table, building 149, 150
text, making editable 147
tools
Google Chrome, installing 6
installing 6
Node.js, installing 10
Starter Kit, installing for ReactJS 16

U

UI teardown and rebuild 38, 39
unit testing 134, 135
updates
cross-synchronization
display, refreshing 142, 143
user interface programming, paradigms
about 3
aspect-oriented programming 3
declarative programming 3
defensive programming 3
functional reactive programming 4, 5
handling 2
imperative programming 4
information hiding 4
object-oriented programming 5
patterns 5
procedural programming 5
reactive programming 5

V

values 19, 22
variables and assignment
about 19-21
comments 21
flow control 22

W

web page
HTML 47
What You See Is What You Get (WYSIWYG) scratchpad
adding 188
merging, into web page 188, 189
Write Once, Read Many (WORM) 220



Thank you for buying
Reactive Programming with JavaScript

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

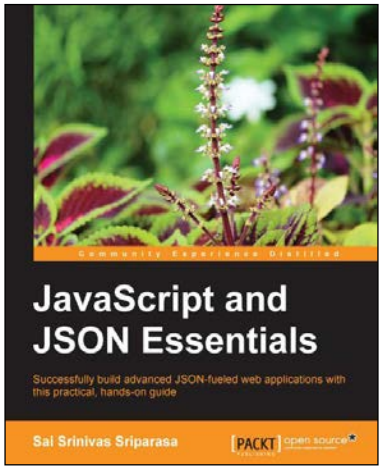
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced 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.

[PACKT] open source
community experience distilled
PUBLISHING

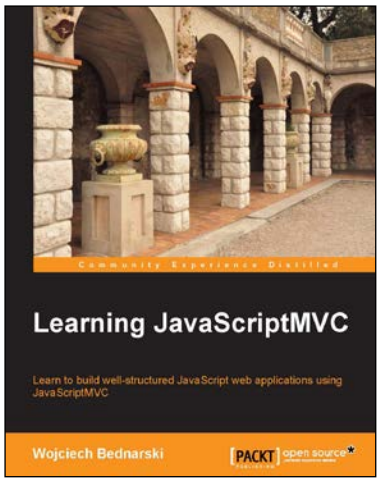


JavaScript and JSON Essentials

ISBN: 978-1-78328-603-4 Paperback: 120 pages

Successfully build advanced JSON-fueled web applications with this practical, hands-on guide

1. Deploy JSON across various domains.
2. Facilitate metadata storage with JSON.
3. Build a practical data-driven web application with JSON.



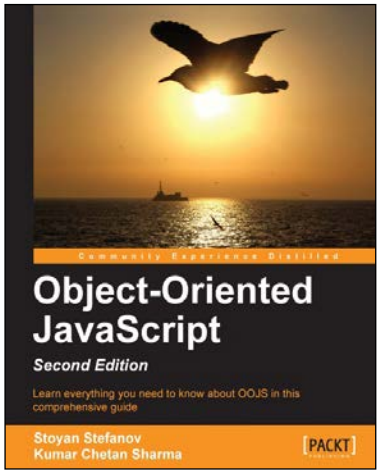
Learning JavaScriptMVC

ISBN: 978-1-78216-020-5 Paperback: 124 pages

Learn to build well-structured JavaScript web applications using JavaScriptMVC

1. Install JavaScriptMVC in three different ways, including installing using Vagrant and Chef.
2. Document your JavaScript codebase and generate searchable API documentation.
3. Test your codebase and application as well as learning how to integrate tests with the continuous integration tool, Jenkins.

[PACKT] open source
PUBLISHING community experience distilled



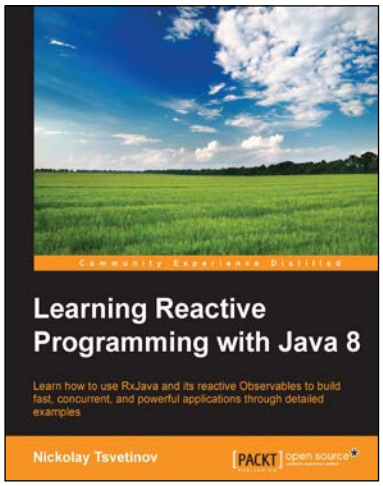
Object-Oriented JavaScript

Second Edition

ISBN: 978-1-84969-312-7 Paperback: 382 pages

Learn everything you need to know about OOJS in this comprehensive guide

1. Make object-oriented programming accessible and understandable to web developers.
2. Apply design patterns to solve JavaScript coding problems.
3. Learn coding patterns that unleash the unique power of the language.
4. Write better and more maintainable JavaScript code.



Learning Reactive Programming with Java 8

ISBN: 978-1-78528-872-2 Paperback: 182 pages

Learn how to use RxJava and its reactive Observables to build fast, concurrent, and powerful applications through detailed examples

1. Learn about Java 8's lambdas and what reactive programming is all about, and how these aspects are utilized by RxJava.
2. Build fast and concurrent applications with ease, without the complexity of Java's concurrent API and shared states.
3. Explore a wide variety of code examples to easily get used to all the features and tools provided by RxJava.

Please check **www.PacktPub.com** for information on our titles