Community Experience Distilled

Backbone.js Essentials

Build amazing high-performance web applications using Backbone.js

Jeremy Walker

[PACKT] PUBLISHING open source* community experience distilled

Backbone.js Essentials

Build amazing high-performance web applications using Backbone.js

Jeremy Walker

[PACKT] open source
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

Backbone.js Essentials

First published: May 2015

Production reference: 1260515

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

ISBN 978-1-78439-479-0

www.packtpub.com

Credits

Author
Jeremy Walker

Reviewers
Philippe Charrière
Peter deHaan
Jeff Smith
Hadrien Tulipe

Commissioning Editor
Amarabha Banerjee

Acquisition Editor
Kevin Colaco

Content Development Editor
Rahul Nair

Technical Editor
Utkarsha S. Kadam

Copy Editors
Dipti Kapadia
Tani Kothari

Project Coordinator
Sanjeet Rao

Proofreaders
Stephen Copestake
Safis Editing

Indexer
Tejal Daruwale Soni

Graphics
Abhinash Sahu

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Jeremy Walker is a writer and programmer who was born, raised, and presently resides in Silicon Valley. Jeremy currently works as a Staff Software Engineer at Syapse, where he develops the company's precision medicine platform and helps doctors treat serious medical conditions using genetic sequencing. Prior to Syapse, Jeremy worked in the multifamily tenant industry, where he drafted multiple XML standards while serving as Technical Vice Chair of the industry's standards committee.

In his free time, Jeremy enjoys reading, playing old computer games, and answering questions on Stack Overflow. If the weather is nice, he can often be found hiking or just lying on the beach in neighboring Santa Cruz. Jeremy is also the author of two programming libraries: Underscore Grease and BackSupport, and he is an active proponent of all things Backbone-related.

Acknowledgments

First, I would like to thank my employer Syapse. For the past three years, Syapse has given me the opportunity to learn and improve my proficiency with Backbone while doing meaningful work to improve the treatment of serious illnesses. I could not have asked for a better place to learn and refine the ideas that made up Backbone.js Essentials.

Second, I would like to thank the wonderful editorial staff at Packt Publishing. Without their support, this book could never have been made, and I am exceedingly grateful to Packt for taking a chance on an unproven author like myself.

I would also like to thank everyone who helped provide feedback on this book, but especially one reviewer in particular. Sep Dadsetan not only reviewed every chapter of this book, but he did so within hours of it being written, and his feedback was always invaluable. I could not ask for a better or more dedicated editorial assistant and friend.

I would also like to thank my family, particularly my deceased Uncle Larry. Many of my first experiences ever using a computer came while sitting in Uncle Larry's living room, and as I grew up his talks and gifts of literature helped shape me both as a writer and as a human being. He will always be missed.

Finally, I would like to thank the most important person in my life, my wife Stacy, for doing everything possible to help me write this book. I am incredibly lucky and grateful to have such a beautiful, supportive, and loving partner, who never once complained about the many hours I spent ignoring her while writing this book (even when it was during our trip to Hawaii).

About the Reviewers

Philippe Charrière is CTO at SQLI in France, and at night, he is an open source developer advocate at Golo project (`http://golo-lang.org/`) and a Backbone enthusiast: He has written a small open source book about Backbone.js in French (`https://github.com/k33g/backbone.en.douceur/`). He's also an occasional speaker on Backbone.js and mobile technologies. He focuses primarily on open web technologies (front and server-side).

He has also worked on the following books:

- Backbone Blueprints (`http://k33g.github.io/2014/07/29/BBBLUEPRINT.html`)
- Node.js Blueprints (`http://k33g.github.io/2014/07/22/NODEJSBLUEPRINT.html`)
- Backbone Patterns & Best Practices (`http://k33g.github.io/2014/02/28/BBPRACTICES.html`)
- Mastering Grunt

Peter deHaan likes Grunt a lot and thinks it's the best thing to happen to Node.js since npm. You can follow his Grunt npm-twitter-bot feed at `@gruntweekly`.

Jeff Smith is a web developer with about three and a half years of focusing on AngularJS, Backbone, Ember, and Node.js.

He has served as Technical Reviewer on the following Packt books:

- *Cake PHP 1.3 Application Cookbook*
- *Drupal 7 Business Solutions*
- *Drupal 7 Webforms Cookbook*

Manning books:

- *Fast Asp.net*
- *The Responsive Web* (Published 10/30/2014)
- *Clojure in Action, 2nd edition, 2nd Review now ()*
- *F# Deep Dives* (December 2014)

I would like to thank my girlfriend Dawn for putting up with the long evenings

---

Hadrien Tulipe is a French software engineer. He discovered programming in high school while creating his first HTML website. After graduating, he studied computer science for 5 years in France and Canada.

With his diploma, he got a job at Worldline, an Atos company, where he specialized in Geographical Information System engineering. As a GIS engineer, he had to conceive complex software architecture and develop robust software. He has also worked for the French National Geographic Institute (IGN) and French National Hydrographic Institute (SHOM).

He has developed front-end web applications using Javascript libraries and frameworks such as OpenLayers and Backbone. He has also developed a lot of back-end web services with Spring and Hibernate.

At work, he advocates software quality and clean code practices. After 4 years as a GIS engineer at Worldline, he moved to the position of Quality and Tests Support Engineer in the same company. He provides support to his colleagues with respect to software testing and good programming practices.

He is also a great partisan of the open source philosophy and likes to contribute to such projects.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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

`https://www2.packtpub.com/books/subscription/packtlib`

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

Why subscribe?

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

Free access for Packt account holders

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

Table of Contents

Preface

We JavaScript developers live in exciting times. Every month we're presented with an incredible new browser feature, open source library, or software methodology that promises to vastly improve on whatever came before it. And then, just as soon as one new technology arrives, another almost inevitably follows it, promising still further innovation.

Amid this ever-changing landscape of new technologies only a very few libraries manage to stay not just relevant but essential over a long period of time, despite having newer challengers. For many web programmers jQuery was the first such library, but in recent years another library has proven itself to be similarly indispensable. That library is Backbone.

Backbone offers developers a wide range of foundational pieces from which they can build any manner of web application. From its simple yet flexible class system, to its event-based and AJAX-simplifying data containers, to its DOM manipulation and single-page user-facing components, Backbone provides everything needed to form the underlying framework of a site.

However, at first Backbone's incredible power and flexibility can be intimidating. When faced with over a hundred different methods spread across four base classes it can be challenging for a new Backbone developer to determine which ones to use and when. In addition, while Backbone is very "opinionated" about its core functions, it takes a deliberately agnostic approach to just about everything else. This allows developers to choose the perfect approach for their application, but at the same time all those choices can be daunting to someone unfamiliar with Backbone.

In this book we offer two things, both drawn from years of experience using Backbone to create and maintain a real-world web-based application. First, we will provide you with an understanding of all of Backbone's essentials. By the time you complete this book, you will have learned everything you need to create powerful and maintainable web applications using Backbone.

But at the same time, in addition to just explaining Backbone itself, we'll also explain the wider, "meta" level of Backbone programming. From advice on how to implement important details of your classes, to examinations of how Backbone can be made even more powerful when connected to the larger JavaScript ecosystem, we have endeavored to not only show how to use Backbone, but how to use it well.

Welcome to Backbone Essentials.

What this book covers

Chapter 1, *Building a Single-Page Site Using Backbone*, introduces Backbone and explains why it is such a popular choice of framework

Chapter 2, *Object-Oriented JavaScript with Backbone Classes*, details Backbone's class system, a significant improvement over JavaScript's native system

Chapter 3, *Accessing Server Data with Models*, begins our exploration of Backbone's data management, event listening, and AJAX capabilities with Backbone's Model class

Chapter 4, *Organizing Models with Collections*, continues our exploration of Backbone's data management capabilities, only this time using multiple data sets with the Collection class

Chapter 5, *Adding and Modifying Elements with Views*, examines Backbone's DOM rendering and event-handling View class

Chapter 6, *Creating Client-side Pages with Routers*, introduces the heart of Backbone's single-page architecture, the Router class

Chapter 7, *Fitting Square Pegs in Round Holes – Advanced Backbone Techniques*, explores advanced Backbone patterns for solving tricky problems

Chapter 8, *Scaling Up – Ensuring Performance in Complex Applications*, looks at the causes of performance issues in Backbone applications, and how to prevent them

Chapter 9, *What Was I Thinking? Documenting Backbone Code*, considers different strategies for documenting your application, including JSDoc and Docco

Chapter 10, *Keeping the Bugs Out – How to Test a Backbone Application*, recommends best practices for testing a Backbone application, with examples from the Mocha and Sinon libraries

Chapter 11, *(Not) Re-Inventing the Wheel – Utilizing Third-Party Libraries*, previews a variety of 3rd party libraries, some Backbone-specific and some not, which can benefit a Backbone application

Chapter 12, *Summary and Further Reading*, looks back on the topics previously discussed and considers their application in the real world

What you need for this book

As a web-based technology book, Backbone Essentials requires very little, only some form of text editor and web browser. Backbone itself is compatible with almost all browsers, including Internet Explorer 7 and up, though a modern web browser is recommended. Similarly, while you *can* write Backbone code using a plain text editor, we recommend instead using a modern IDE such as Sublime, Eclipse, or WebStorm, as it can provide many useful features to aid development.

Of course to use Backbone you'll need to download (and include in your HTML) the library itself, which can be found at `http://www.backbonejs.com`. In addition, you'll also need to download Backbone's two dependencies: jQuery (`http://www.jquery.com`) and Underscore (`http://www.underscorejs.org/`). Technically only Underscore is required, but you'll want jQuery also to be able to take advantage of Backbone's View class.

Once you have an editor, browser, and all three libraries downloaded and included in your HTML, you're ready to get started using Backbone.

Who this book is for

This book is geared towards readers with a basic understanding of JavaScript and HTML and at least some familiarity with the jQuery library. While this book does not require any formal computer science knowledge, it does employ industry terms such as "reference" or "inheritance system" which may be unfamiliar to readers without at least some programming background.

If you are unfamiliar with jQuery, we recommend that you first read Packt's excellent *Learning jQuery*, by Jonathan Chaffer, before starting this book. If you are completely new to web development, you would be better served by first familiarizing yourself with JavaScript through one of the many excellent free tutorials available on the web.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
<tr>
    <td>Fake Book</td>
    <td>This is a description of a fake book</td>
    <td><a href=""/buy/book1"">Buy Fake Book</a></td>
</tr>
```

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: "For instance, you might have a **Submit** button on a page and after it triggers a save of the page's Model you want it to redirect the user to a different route."

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 about it. 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
Building a Single-Page Site Using Backbone

In this chapter, you'll learn both what Backbone is and why you will want to use it to create web applications. In particular, we'll examine the following topics:

- Backbone's history and how it fits into the larger history of web development
- The advantages of Backbone's **single-page** architecture
- How real-world companies are using Backbone to power their sites

What is Backbone?

Created in 2010 by Jeremy Ashkenas, Backbone is a part of an entirely new breed of JavaScript libraries. Depending on who you ask, this type of library can be referred to as a rich application framework, a single page library, a thick client library, or just a JavaScript framework. Whatever you choose to call them, Backbone and its related libraries, such as Angular, Ember, and CanJS, provide tools that can be used to build websites that are so powerful that they go beyond being mere sites and become full-fledged web applications.

Backbone is made up of the following five major tools:

- A class system, which makes it easy to practice object-oriented programming
- A `Model` class, which allows you to store and manipulate any kind of data as well as exchange this data with and from your remote server using AJAX
- A `Collection` class, which allows you to perform the same data manipulation and transmission but on groups of Models instead

- A `View` class, which can be used both to render the DOM elements that make up the page and to manage any user interactions that occur on them
- A `Router` class, which enables you to create an entire site, with any number of virtual pages, using only a single HTML file

While conceptually very simple, together these components allow you to create websites with a level of sophistication and robustness previously unseen on the **World Wide Web** (**WWW**).

Why choose Backbone?

The question of why you will want to choose Backbone for your project really has two parts. First, there's the question of "Why use a rich application framework at all?," and second, there's the question of "Why choose Backbone over its alternatives?." Let's start with the first question.

To truly appreciate the value of a **single-page application** (**SPA**), it's essential to understand what came before. All previous websites can be grouped into three major categories, which I'll call static, server-based, and JavaScript-assisted. Each of these types correlates with a different era in web development history.

A Brief History of Web Development

In many ways, the history of web development can be seen as a progression from server-based logic to client-based logic. The story begins in 1993, with the introduction of the world's first real web browser: Mosaic. At that time, the Web didn't even have JavaScript (or CSS, for that matter), just HTML. In those early days, most sites were simple static sites, and any site with dynamic elements had to be entirely server based. The very first version of JavaScript would only be introduced two years later at the end of 1995, and it would take several more years before the language was useful for anything other than simple form validation.

Yahoo!'s JavaScript-less website in 1997

Luckily, the Web did evolve, and soon JavaScript developers witnessed the birth of a whole new wave of JavaScript libraries, such as Dojo, MochiKit, YUI, and, of course, jQuery. These libraries allowed developers to easily manipulate the DOM, avoid the rampant cross-browser issues of the time, and take advantage of a newly introduced technology known as **AJAX**. In other words, they enabled developers to create a new type of site, the JavaScript-assisted but still largely server-based web application.

Even with these advances, the server still maintained control over two critical pieces of a site's infrastructure: navigation and page rendering. This problem wouldn't be solved until years later, with the introduction of the modern generation of JavaScript frameworks, the first and popular one being Backbone. Using Backbone, web developers were finally able to control an entire site using only the client-side technologies of JavaScript, HTML, and CSS, which meant that they could create an entirely new type of web application, the thick client or single page site.

Today, even with the advent of Backbone and related libraries, many developers still continue to create the three previous types of site, which is perfectly reasonable as long as their goals are modest. In other words, if you simply want to show off your wedding photos to friends, then you probably don't need the full power of Backbone. However, if your goal is to build a powerful and robust web application, then the advantages of a Backbone-powered site are clear.

Benefits of Backbone and single-page applications

While there are numerous benefits of adopting thick client architecture for a site, they can be grouped into three main categories: asset control, easier data management, and improved performance.

Full User Interface Asset Control

One of the challenges of developing a traditional multipage website is the sharing of HTML assets. On such a site, the HTML is generated using server-side tools, such as Django templates, ERBs, or **JavaServer Pages** (**JSPs**) but, of course, the client-side logic also depends heavily on that same HTML. In smaller organizations, this means that programmers frequently have to divide their focus between JavaScript and a server-side language, which can be frustrating due to the frequent context switching.

In large organizations where teams are separate, the HTML assets are usually managed by the server team. This sometimes makes it difficult for the client team to even make the most basic changes to the site's HTML, as they have to work across the aisle. When they fail to do so, often the result is such that they create parallel versions of the server team's work instead, with such duplication inevitably resulting in bugs.

Backbone-powered thick client applications solve these problems by leaving the site's HTML firmly under the control of the client team, either in the form of a template system, raw HTML files, or in DOM-manipulation JavaScript logic. Any interactions between the two teams happen through a carefully negotiated set of APIs, allowing both groups to focus on their core specialties without stepping on each others toes.

Simpler Data Management and Event Triggers

As an application scales, it may become difficult to manage the interactions between its various components. One powerful approach to solve this problem is to use event-based control systems, but before Backbone, such systems were rarely found in JavaScript. True, DOM events have long been a part of web development, but without a framework such as Backbone, developers have been limited to just the user-generated events. To truly realize the power of an event-based system, you also need data-driven events, which are an integral part of Backbone.

Another common scaling challenge comes from JavaScript's lack of support for **object-oriented programming** (**OOP**). OOP allows programmers to organize large, complicated logic into smaller, more manageable classes and is very useful when growing an application. While JavaScript has a built-in class system, it is fairly unconventional and often discourages developers from employing OOP techniques. Backbone solves this problem by providing a more friendly system that, while still built within the limits of the JavaScript language, looks much closer to what you'd find in a solid OOP language, such as Java.

Enhanced performance

On the Web, speed is paramount, and one significant factor in a site's speed is the weight of its HTML files. In a multipage application, every time the the user visits a new page, their browser has to send a request and wait for a response from the server. When the response comes back, it doesn't just contain a unique HTML for that page. Instead, the response contains the HTML for everything, including any common site components such as menus or footers. When the user visits the next page, they once again have to download that same common component HTML, even if it hasn't changed.

Moreover, that's not the only redundant HTML downloaded: multiple rows in a table, multiple search results in a list, or any other repeated content also has to have its HTML downloaded multiple times. For instance, consider the following HTML:

```
<tr>
    <td>Fake Book</td>
    <td>This is a description of a fake book</td>
    <td><a href=""/buy/book1"">Buy Fake Book</a></td>
</tr>
<tr>
    <td>Another Fake Book</td>
    <td>I hope you like fake book titles because plenty more are
coming in future chapters...</td>
    <td><a href=""/buy/book2"">Buy Another Fake Book</a></td>
</tr>
```

Only the names, descriptions, and URLs of the two books are unique in the preceding code, but even so all of the nonunique parts of the code have to be downloaded with it. If the site shows 50 books, the user downloads 50 copies of the book row HTML. Even when a site has no common components or repeated elements, there's still a performance cost when the user visits a new page because the browser has to go through an entire request-response cycle and then reload and redraw the page, all of which takes time.

In a single-page application, none of this is an issue. The site's foundation HTML is downloaded only once, and after that, all page transitions happen entirely through JavaScript. Since the client knows how to render both common and repeated components, there's no need to download any HTML for them at all. On a Backbone site, the server sends only the unique data via AJAX, and if there is no unique data to download, the user can progress without making a single new request to the server.

Backbone and Its Competitors

Many of the advantages we've just discussed apply to any single-page application, not just a Backbone one. This means that you can achieve many of those benefits even if you use one of Backbone's competing libraries, such as Ember or Angular. Whether you've considered using these frameworks or not, you're probably at least wondering, "Will Backbone provide me with everything I need to build my site, both now and in the future?"

The first thing to consider when answering this question is whether or not Backbone has an active community and will continue to be actively developed. Backbone users can feel safe in this regard: at the time of writing this book, Backbone's GitHub page had more than 1,500 watchers and more than 21,000 stars, beating its next closest competitor (Ember) by more than 400 watchers and 7,000 stars. Other frameworks such as CanJS and Google's Angular have even less interest on GitHub. While this certainly doesn't make Backbone better than those libraries, it shows the strength of its community and should provide you with the assurance that Backbone will be around for many years to come.

Another reason to feel confident when selecting Backbone is that it only tries to do a specific set of tasks, leaving everything else to external libraries. This means that if you find a better template system, dependency management tool, or any other library in the future, you can easily switch to using it. Other frameworks tightly couple things such as their template systems to their framework, leaving you with less options in the future.

However, perhaps the biggest indicator of Backbone's vitality is the companies that are already using it to accomplish amazing things. Companies as varied as USA Today, Pandora, Hulu, Gawker Media, AirBnB, Khan Academy, Groupon, and even Walmart use Backbone to create powerful web applications. If Backbone is powerful enough to support these major companies, it's almost certain to be powerful enough for your project.

There's one other company that uses Backbone, which is the company that I work for—Syapse. At Syapse, we've built a precision medicine data platform that helps hospitals receive genetic data in a structured format, pull patients' clinical data from a variety of internal health IT systems, and present this data together in an interactive web application. Through this interface, physicians see their patients' genetic and clinical data in context, enabling them to choose the most effective drugs possible tailored to a patient's own genetic profile.

Creating an application like Syapse did isn't easy, and with serious diseases such as cancer on the line, there's little room for error. However, using Backbone, Syapse has managed to grow from just one developer to a six-person client-side team with over 21,000 lines of code (not counting libraries) in just 3 years. Were it not for Backbone's ability to scale, there's simply no way we could have grown that quickly, at least without making major changes to our architecture along the way.

In short, while Backbone itself may be just under half a decade old, the real-world usage of the library has proven both its value and scalability. If your goal is to create a powerful and robust web application that a single developer can easily get off the ground but which can also grow and be supported by a full-sized team, you cannot go wrong with Backbone.

Summary

In this chapter, we explored how Backbone represents a new chapter in web development and why it's the best framework for your project if your goal is to make powerful and scalable web applications.

In the next chapter, we'll begin to take a look at the components that make up Backbone, in particular its easy-to-use class system. We'll also look at Backbone's sister library, Underscore, which was also created by Jeremy Ashkenas and is a requirement for Backbone itself.

2
Object-Oriented JavaScript with Backbone Classes

In this chapter, we will explore the following topics:

- The differences between JavaScript's class system and the class systems of traditional object-oriented languages
- How new, this, and prototype enable JavaScript's class system
- Extend, Backbone's much easier mechanism for creating subclasses
- Ways to take advantage of Underscore, which (like jQuery) is one of Backbone's dependencies

JavaScript's class system

Programmers who use JavaScript can use classes to encapsulate units of logic in the same way as programmers of other languages. However, unlike those languages, JavaScript relies on a less popular form of inheritance known as prototype-based inheritance. Since Backbone classes are, at their core, just JavaScript classes, they too rely on the prototype system and can be subclassed in the same way as any other JavaScript class.

For instance, let's say you wanted to create your own `Book` subclass of the Backbone `Model` class with additional logic that Model doesn't have, such as book-related properties and methods. Here's how you can create such a class using only JavaScript's native object-oriented capabilities:

```
// Define Book's Initializer
var Book = function() {
    // define Book's default properties
```

```
    this.currentPage = 1;
    this.totalPages = 1;
}

// Define book's parent class
Book.prototype = new Backbone.Model();

// Define a method of Book
Book.prototype.turnPage = function() {
    this.currentPage += 1;
    return this.currentPage;
}
```

If you've never worked with prototypes in JavaScript, the preceding code may look a little intimidating. Fortunately, Backbone provides a much easier and easier to read mechanism for creating subclasses. However, since that system is built on top of JavaScript's native system, it's important to first understand how the native system works. This understanding will be helpful later when you want to do more complex class-related tasks, such as calling a method defined on a parent class.

The new keyword

The `new` keyword is a relatively simple but extremely useful part of JavaScript's class system. The first thing that you need to understand about new is that it doesn't create objects in the same way as other languages. In JavaScript, every variable is either a function, object, or primitive, which means that when we refer to a *class*, what we're really referring to is a specially designed initialization function. Creating this class-like function is as simple as defining a function that modifies this and then using the new keyword to call that function.

Normally, when you call a function, its this is obvious. For instance, when you call the `turnPage` method of a book object, the `this` method inside `turnPage` will be set to this book object, as shown here:

```
var simpleBook = {currentPage: 3, pages: 60};
simpleBook.turnPage = function() {
    this.currentPage += 1;
    return this.currentPage;
}
simpleBook.turnPage(); // == 4
```

Calling a function that isn't attached to an object (in other words, a function that is not a method) results in this being set to the global scope. In a web browser, this means the window object:

```
var testGlobalThis = function() {
    alert(this);
}
testGlobalThis(); // alerts window
```

When we use the `new` keyword before calling an initialization function, three things happen (well, actually four, but we'll wait to explain the fourth one until we explain prototypes):

- JavaScript creates a brand new object ({})for us
- JavaScript sets the `this` method inside the initialization function to the newly created object
- After the function finishes, JavaScript ignores the normal return value and instead returns the object that was created

As you can see, although the `new` keyword is simple, it's nevertheless important because it allows you to treat initialization functions as if they really are actual classes. At the same time, it does so without violating the JavaScript principle that all variables must either be a function, object, or primitive.

Prototypal inheritance

That's all well and good, but if JavaScript has no true concept of classes, how can we create subclasses? As it turns out, every object in JavaScript has two special properties to solve this problem: `prototype and __proto__ (hidden)`. These two properties are, perhaps, the most commonly misunderstood aspects of JavaScript, but once you learn how they work, they are actually quite simple to use.

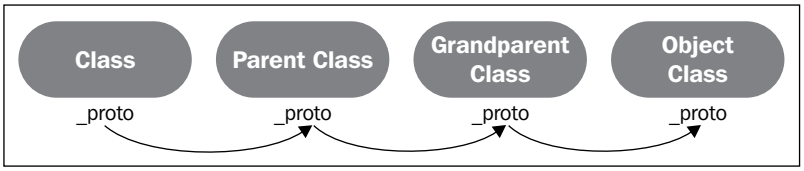
When you call a method on an object or try to retrieve a property JavaScript first checks whether the object has the method or property defined in the object itself. In other words if you define a method such as this one:

```
book.turnPage = function()
    this.currentPage += 1;
};
```

JavaScript will use that definition first when you call `turnPage`.

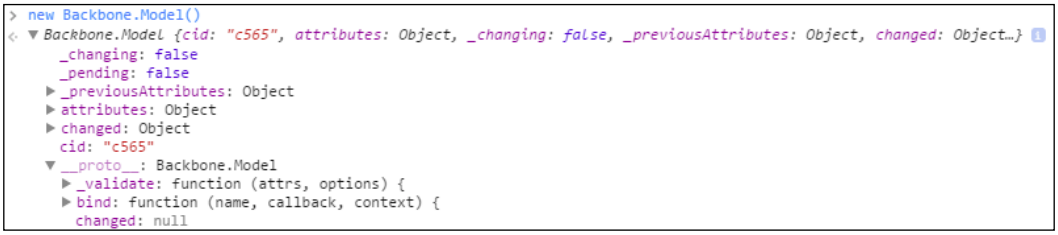
In real-world code, however, you will almost never want to put methods directly in your objects for two reasons. First, doing that will result in duplicate copies of those methods, as each instance of your class will have its own separate copy. Second, adding methods in this way requires an extra step, and that step can be easily forgotten when you create new instances.

If the object doesn't have a `turnPage` method defined in it, JavaScript will next check the object's `hidden __proto__` property. If this `__proto__` object doesn't have a `turnPage` method, then JavaScript will look at the `__proto__` property on the object's `__proto__`. If that doesn't have the method, JavaScript continues to check the `__proto__` of the `__proto__` of the `__proto__` and keeps checking each successive `__proto__` until it has exhausted the chain.



This is similar to single-class inheritance in more traditional object-oriented languages, except that instead of going through a class chain, JavaScript instead uses a prototype chain. Just as in an object-oriented language we wind up with only a single copy of each method, but instead of the method being defined on the class itself, it's defined on the class's prototype.

In a future version of JavaScript (ES6), it will be possible to work with the `__proto__` object directly, but for now, the only way to actually see the `__proto__` property is to use your browser's debugging tool (for instance, the Chrome Developer Tools debugger):



This means that you can't use this line of code:

```
book.__proto__.turnPage();
```

Also, you can't use the following code:

```
book.__proto__ = {
    turnPage: function() {
        this.currentPage += 1;
    }
};
```

But, if you can't manipulate `__proto__` directly, how can you take advantage of it? Fortunately, it is possible to manipulate `__proto__`, but you can only do this indirectly by manipulating `prototype`. Do you remember I mentioned that the `new` keyword actually does four things? The fourth thing is that it sets the `__proto__` property of the new object it creates to the `prototype` property of the initialization function. In other words, if you want to add a `turnPage` method to every new instance of `Book` that you create, you can assign this `turnPage` method to the `prototype` property of the `Book` initialization function, For example:

```
var Book = function() {};
Book.prototype.turnPage = function() {
    this.currentPage += 1;
};
var book = new Book();
book.turnPage();// this works because book.__proto__  == Book.
prototype
```

Since these concepts often cause confusion, let's briefly recap:

- Every object has a `prototype` property and a hidden `__proto__` property
- An object's `__proto__` property is set to the `prototype` property of its constructor when it is first created and cannot be changed
- Whenever JavaScript can't find a property or method on an object, it checks each step of the `__proto__` chain until it finds one or until it runs out of chain

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.

Extending Backbone classes

With that explanation out of the way, we can finally get down to the workings of Backbone's subclassing system, which revolves around Backbone's extend method. To use extend, you simply call it from the class that your new subclass will be based on, and extend will return the new subclass. This new subclass will have its `__proto__` property set to the `prototype` property of its parent class, allowing objects created with the new subclass to access all the properties and methods of the parent class. Take an example of the following code snippet:

```
var Book = Backbone.Model.extend();
// Book.prototype.__proto__ == Backbone.Model.prototype;
var book = new Book();
book.destroy();
```

In the preceding example, the last line works because JavaScript will look up the `__proto__` chain, find the `Model` method `destroy`, and use it. In other words, all the functionality of our original class has been inherited by our new class.

But of course, `extend` wouldn't be exciting if all it can do is make exact clones of the parent classes, which is why `extend` takes a `properties` object as its first argument. Any properties or methods on this object will be added to the new class's `prototype`. For instance, let's try making our `Book` class a little more interesting by adding a property and a method:

```
var Book = Backbone.Model.extend({
    currentPage: 1,
    turnPage: function() {
        this.currentPage += 1;
    }
});
var book = new Book();
book.currentPage; // == 1
book.turnPage(); // increments book.currentPage by one
```

The `extend` method also allows you to create static properties or methods, or in other words, properties or methods that live on the class rather than on objects created from that class. These static properties and methods are passed in as the second `classProperties` argument to `extend`. Here's a quick example of how to add a static method to our `Book` class:

```
var Book = Backbone.Model.extend({}, {
    areBooksGreat: function() {
        alert("yes they are!");
    }
});
```

```
Book.areBooksGreat(); // alerts "yes they are!"
var book = new Book();
book.areBooksGreat(); // fails because static methods must be
called on a class
```

As you can see, there are several advantages to Backbone's approach to inheritance over the native JavaScript approach. First, the word `prototype` did not appear even once in any of the previously mentioned code; while you still need to understand how `prototype` works, you don't have to think about it just to create a class. Another benefit is that the entire class definition is contained within a single `extend` call, keeping all of the class's parts together visually. Also, when we use `extend`, the various pieces of logic that make up the class are ordered the same way as in most other programming languages, defining the super class first and then the initializer and properties, instead of the other way around.

Applying parent methods

In order to realize the full power of a class system, however, it's not enough to just define new methods on subclasses; sometimes, we need to combine a method of a parent class with additional logic on a subclass. In traditional object-oriented languages, this is often done by referencing a special super object; but in JavaScript, no such object exists, which means that we have to utilize JavaScript's apply or call methods instead.

For instance, `Backbone.Model` has a `destroy` method, but what if we want our `Book` class to also have its own `destroy` method? This method might take a number of pages and destroy them (reduce the total number of pages by that amount), but at the same time, we might want to keep the Backbone version around for its original purpose (which is to destroy the server-side version of the `Book`).

Luckily, because Backbone properly configured our `Book` class's prototype for us, calling the parent method from the subclass method is fairly straightforward, as shown here:

```
var Book = Backbone.Model.extend({
    destroy: function(optionsOrPagesToDestroy) {
        if (typeof optionsOrPagesToDestroy === 'number') {
            // optionsOrPagesToDestroy is pagesToDestroy: call our
                version
            this.totalPages -=  optionsOrPagesToDestroy;
        } else {
            // optionsOrPagesToDestroy is an options object: call
                the Backbone version
```

```
            Backbone.Model.prototype.destroy.apply(this,
                arguments);
        }
    }
});
```

The key to making the preceding code work is the `apply` method, which is a method of every function in JavaScript (since functions are also objects in JavaScript, it is possible for them to have methods just like any other object). The `apply` method allows you to call its function as if it were called from the first argument given to apply. In other words, apply allows you to change a function's `this` method when it is invoked.

As mentioned before, under normal conditions, `this` will be set to the object from which the function was called. However, when you use apply, you can instead change this to whatever variable you want, as follows:

```
var Book = Backbone.Model.extend({
    currentPage: 1,
    turnPage: function() {
        this.currentPage += 1;
    }
});
var simpleBook = {currentPage: 20};
Book.prototype.turnPage.apply(simpleBook); //
simpleBook.currentPage == 21
```

You can also pass regular arguments to a method using `apply`, by providing them as a second (array) argument. In fact, `apply` can be used even if this isn't relevant to the function you're trying to call, and in that case, you can simply pass `null` as the first argument:

```
var Book = Backbone.Model.extend();
var book = new Book();
book.alertMessage = function(message, secondMessage) {
    alert(message + ' ' + secondMessage);
}
book.alertMessage.apply(null, ['hello', 'world']);// alerts "hello
world"
```

JavaScript functions also have a similar method: `call`. The difference between `call` and `apply` is how they provide arguments to the function they invoke. Unlike `apply`, `call` expects its arguments to be passed in separately, rather than as a single array argument. Take an example of the following code snippet:

```
var book = new Book();
book.alertMessage = function(message, secondMessage) {
    alert(message + ' ' + secondMessage);
}
book.alertMessage.call(null, 'hello', 'world'); // alerts "hello world"
```

By using one of these two methods and remembering that every method of a class is available to you on its `prototype` property, you can call not just the methods of a parent class but also the methods of a grandparent, great grandparent, or even a completely unrelated class. For readability reasons, you shouldn't utilize this technique too often on unrelated classes, but occasionally it can be quite helpful.

It's also worth noting that this technique is only possible in a prototype-based inheritance language such as JavaScript; it will be impossible to implement `apply` or `call` in a true object-oriented language such as Java.

Introducing Underscore

In addition to jQuery, Backbone also requires one library called **Underscore**. Underscore was written by Jeremy Ashkenas (the creator of Backbone), and many of its functions are relevant to the topics we've discussed so far. Because Underscore is required by Backbone, you'll already have it available to you if you use Backbone. All of these functions can be accessed via the _ character (similar to how jQuery functions are accessed via the `$` character).

As we discussed the `call` and `apply` methods, you may have realized that this is more fluid in JavaScript than in other languages. While a function that is called normally will preserve the `this` method automatically, when a function is called in an unusual way—such as through `window.setTimeout` or as a callback to a jQuery event handler or AJAX call—that won't be the case. The `window.setTimeout` will change this to the global window object, while jQuery event callbacks will change `this` to the element that triggered the event and jQuery AJAX callbacks will set it to the HTTP request created by the AJAX call. Here's a quick example:

```
var exampleObject = {};
exampleObject.alertThis = function() {
     alert(this);
};
window.setTimeout(exampleObject.alertThis); // alerts window
```

The solution to this problem is to use extra functions and use `apply` to wrap the original function so that we can force JavaScript to preserve our `this` method:

```
      var exampleObject = {};
exampleObject.alertThisBuilder = function() {
    var alertThis = function() {
        alert(this);
    }
    var correctThis = this;
    return function() {
        alertThis.apply(correctThis);
    }
};
var alertThis = exampleObject.alertThisBuilder();
window.setTimeout(alertThis); // alerts exampleObject
```

This works but let's face it: it's ugly. Luckily, Underscore has a two solution for this: `bind` and `bindAll`.

Let's start with `bind`. The `bind` functions allows you to force a function, which you provide as the first argument, to preserve a specific `this` value, which you provide as the second argument:

```
var simpleBook = {};
simpleBook.alertThis = function() {
     alert(this);
};
simpleBook.alertThis = _.bind(simpleBook.alertThis, simpleBook);
window.setTimeout(simpleBook.alertThis); // now alerts simpleBook,
not window
```

Underscore also has a related `bindAll` function, which can be used to permanently `bind` a method:

```
var Book = Backbone.Model.extend({
     initialize: function() {
          _.bindAll(this, 'alertThis');
     },
     alertThis: function() {
          alert(this);
     }
});
var  book = new Book();
window.setTimeout(book.alertThis); // alerts book, not window
```

As you can see, `bindAll` allows you to use your class's methods with `setTimeout` or as callbacks to jQuery event handlers or AJAX operations, without losing `this`.

While `bindAll` is very powerful, it is important not to overuse it, because it creates a new copy of every method it binds. If used inside a class, this will result in every instance of that class to have its own separate copy of that method. While this is perfectly fine when you only have a few bound methods and/or only a few instances, you will not want to use it on a large number of methods with a class that will be instantiated many times.

More Underscore

The `bind` and `bindAll` functions are just the tip of the iceberg of what Underscore has to offer web developers. While a full explanation of everything that Underscore has to offer is outside the scope of this book, it is worth taking a few moments to examine a few of Underscore's most useful functions. If you want to learn more about Underscore beyond what we have covered in this chapter, I strongly recommend that you read its web page (`http://underscorejs.org/`), which has well-written documentation for every method in the library.

Each, Map, and Reduce

Every JavaScript developer knows how to iterate using the `for` loops, but Underscore provides three powerful alternatives to those native loops: `each`, `map`, and `reduce`. While all three of these methods (along with many of the other Underscore methods) are included natively in ES5-supporting browsers, older browsers, unfortunately, do not have support for them. These alternative loops are so convenient that you may find yourself never using the native `for` loop again. Thankfully, Underscore provides its version of them, allowing you to bridge the gap until all major browsers support ES5.

Let's start with an example of the usage of each:

```
var mythicalAnimals = ['Unicorn', 'Dragon', 'Honest Politician'];
_.each(mythicalAnimals, function(animalName, index) {
    alert('Animal #' + index + ' is ' + animalName);
});
```

This will be the equivalent of the preceding code using JavaScript's native `for` loop:

```
      var mythicalAnimals = ['Unicorn', 'Dragon', 'Honest
Politician'];
for (var index = 0; index < mythicalAnimals.length; index++) {
    var animalName = mythicAnimals[index];
    alert('Animal #' + index + ' is ' + animalName);
}
```

Alternatively, if we instead use the `for/in` syntax:

```
var mythicalAnimals = ['Unicorn', 'Dragon', 'Honest Politician'];
for (var index in mythicalAnimals) {
    var animalName = mythicalAnimals[index];
    alert('Animal #' + index + ' is ' + animalName);
}
```

As you can see, both native implementations require you to extract the value (in this case, `animalName`) inside the loop, whereas Underscore's version provides it automatically.

Underscore's `map` and `reduce` methods are even more powerful. The `map` method allows you to convert an array of variables into another (different) array of variables, while reduce converts an array of variables into a single variable. For instance, let's say you've used jQuery's `text` method to extract a bunch of numbers, but because they came from the DOM, those numbers are actually strings (in other words, `5` instead of `5`). By using map, which has an almost identical syntax as each, you can easily convert all those strings into actual numbers, as follows:

```
var stringNumbers = ["5", "10", "15"];
var BASE = 10; // when we parse strings in to numbers in
               // JavaScript we have to specify which base to use
var actualNumbers = _.map(stringNumbers, function(numberString,
index) {
    return parseInt(numberString, BASE);
}); // actualNumbers == [5, 10, 15]
```

As you can see from the preceding example, the values returned inside the `map` function are added to the array returned by the overall `map` operation. This lets you convert any kind of array into another kind of array, as long as you can define a function that sits in the middle and performs the desired conversion.

However, what if you wanted to combine or sum up all those numbers? In that case, you'd instead want to use `reduce`, as shown here:

```
var total = _.reduce(actualNumbers, function(total, actualNumber) {
    return total +  actualNumber;
}, 0); // total == 30
```

The `reduce` function is slightly more complex than the previous functions because of its last argument, which is known as the *memo* argument. This argument serves as the starting value for the object that reduce will eventually return, and then as each value is iterated through that, memo is replaced by whatever value the `reduce` function (the function that is passed in to reduce) returns. The `reduce` function is also passed the previous memo as its first argument, so each iteration can choose to modify the previous value the way it wants to. This allows reduce to be used for far more complex operations than to simply add two numbers together.

Extend and defaults

Another common operation that is made easier with Underscore is copying the properties of one object onto another, which is solved with the Underscore methods `extend` (not to be confused with Backbone's `extend` class) and `defaults`. For instance, let's say you are using a third-party widget, such as a jQuery UI component, that takes a configuration argument when it is created. You might want to keep some configuration options the same for every widget in your site, but at the same time, you might want certain widgets to also have their own unique options.

Let's imagine that you've defined these two sets of configurations with two objects, one for the common configuration and one for a specific widget's configuration:

```
var commonConfiguration = {foo: true, bar: true};
var specificConfiguration = {foo: false, baz: 7};
```

The `extend` method takes the first argument given to it and copies the properties from each successive argument onto it. In this case, it allows you to take a new object, apply the common options to it first, and then apply the specific options, as follows:

```
var combined = _.extend({}, commonConfiguration,
specificConfiguration);
// combined == {foo: false, bar: true, baz: 7}
```

The `defaults` method works in a similar manner, except that it only copies over values that the object doesn't already have. We can instead rewrite the preceding example with `defaults`, simply by changing the argument order:

```
var combined = _.defaults({}, specificConfiguration ,
commonConfiguration);
// combined == {foo: false, bar: true, baz: 7}
```

As its name implies, the `defaults` method is very handy when you want to specify default values for an object but don't want to replace any values you've already specified.

Pluck and invoke

Two of the most common uses of `map` involve getting certain properties from an array of objects or calling a certain method on every object in an array. Underscore provides additional convenience methods to do exactly this: `pluck` and `invoke`.

The `pluck` method allows you to extract a single property value from each member of an array. For example, let's say we have an array of objects representing fake books, as follows:

```
var fakeBooks = [
    {title: 'Become English Better At', pages: 50, author: 'Bob'},
    {title: 'You Is Become Even Better at English', pages: 100,
      author: 'Bob'},
    {title: 'Bob is a Terrible Author', pages: 200, author: 'Fred
      the Critic'}
];
```

If you want to create a list of just the titles of these books, you can use the `pluck` method as shown here:

```
var fakeTitles = _.pluck(fakeBooks, 'title');// fakeTitles == ['Become
English Better At', ...]
```

The `invoke` method works in a similar way as `pluck` method, except that it assumes that the the provided property is a method and runs it and then adds the result to the returned array. This can be best demonstrated with the following example:

```
var Book = Backbone.Model.extend({
    getAuthor: function() {
        // the "get" method returns an attribute of a Model;
        // we'll learn more about it in the following chapter
        return this.get('author');
    }
});
var books = [new Book(fakeBooks[0]),
                    new Book(fakeBooks[1]),
                    new Book(fakeBooks[2])];
var authors = _.invoke(books, 'getAuthor'); // == ['Bob', 'Bob',
'Fred the Critic']
```

Further reading

Underscore also has many other useful functions, and as we have mentioned before, all of them are well documented at `http://underscorejs.org/`. Since every Backbone programmer is guaranteed to have Underscore available, there's no reason why you shouldn't take advantage of this great library; even spending just a few minutes on the Underscore site will help make you aware of everything that it has to offer.

Summary

In this chapter, we explored how JavaScript's native class system works and how the `new`, `this`, and `prototype` keywords/properties form the basis of it. We also learned how Backbone's `extend` method makes creating new subclasses much more convenient as well as how to use `apply` and `call` to invoke parent methods (or when providing callback functions) to preserve the desired `this` method. Finally, we looked at a number of ways in which Underscore, one of Backbone's dependencies, can solve common problems. In the next chapter, we'll dive into the first of the four Backbone classes, `Model`. We'll learn how to use Model to organize our data on the client side and to exchange this data to and from our remote server.

3
Accessing Server Data with Models

In this chapter, you'll learn how to:

- Use `Model`, Backbone's main class to work with data
- Create new `Model` subclasses
- Get and set `Model` attributes and trigger other code when these attributes change
- Store these attributes to a remote server and retrieve them from the server
- Use the several convenience methods that `Model` has borrowed from Underscore

The purpose of Models

Models in Backbone are the core of all data interactions, both within the client code itself and when communicating with a remote server. While one can simply use a plain JavaScript object instead of a `Model`, Models offer three major advantages:

- Models use Backbone's class system, making it easy to define methods, create subclasses, and so on
- Models allow other code to listen for and respond to changes in the `Model` attributes
- Models simplify and encapsulate the logic used to communicate with the server

As discussed in the previous chapter, we can create our own subclass of Model using `extend`:

```
var Cat = Backbone.Model.extend({
     // The properties and methods of our "Cat" class would go here
});
```

Once we've created that class, we can instantiate new `Model` instances using JavaScript's `new` keyword, and (optionally) we can pass in an initial set of attributes and options, as follows:

```
var garfield = new Cat({
    name: 'Garfield',
    owner: 'John'
}, {
    estimateWeight: true
});
```

Attributes, options, and properties

When we talk about attributes in Backbone, they often sound similar to regular JavaScript properties. After all, both attributes and properties are key-value pairs stored on a `Model` object. The difference between the two is that attributes aren't (in a technical sense) actually properties of a Model at all; instead, they are the properties of a property of a `Model`. Each `Model` class has a property called `attributes`, and the attributes themselves are stored as properties of that `attributes` property. Take an example of the following code snippet:

```
var book = new Backbone.Model({pages: 200});
book.renderBlackAndWhite = true;
book.renderBlackAndWhite // is a property of book
book.attributes.pages; // is an attribute of book
book.attributes.currentPage = 1; // is also an attribute of book}
```

Attributes versus Properties

As shown in the preceding code snippet, Backbone's `Models` class can also have regular, nonattribute properties. If you need to store a piece of data in a Model, you can choose between using a property or an attribute. In general, you should use an attribute only when the data is going to be synced to the server or when you want other parts of your code to be able to listen for data changes. If your data doesn't meet these requirements, it's best to store it as a regular JavaScript property instead of as an attribute, because storing such data as an attribute will create more work for you when you save the `Model` class.

On a purely conceptual level, any of the core persistent information that a `Model` class is designed to hold belongs in its attributes, while any information that is secondary or derived, or which is only designed to last until the user refreshes the page, should be stored as properties.

If you want to pass in attributes when you create a new `Model` class, you can simply provide them as the first argument and Backbone will automatically add them. You can also define default attributes, which all new `Models` of that class will have, using the defaults property when you extend your `Model` class, as shown here:

```
var Book = Backbone.Model.extend({

    defaults: {publisher: 'Packt Publishing'}
});
var book = new Book();
book.attributes.publisher; // 'Packt Publishing'
```

It is important to remember, however, that in JavaScript, objects are passed by reference, which means that any object you provide in `defaults` will be shared with, not copied between, instances of your `Model` class. In other words, check out the following code snippet:

```
var Book = Backbone.Model.extend({
    defaults: {publisher: {name: 'Packt Publishing'}}
});
var book1 = new Book();
book1.attributes.publisher.name // == 'Packt Publishing'
var book2 = new Book();
book2.attributes.publisher.name = 'Wack Publishing';
book1.attributes.publisher.name // == 'Wack Publishing'!
```

Since you probably don't want changes to one Model affecting your other Models, you should avoid setting objects in defaults. If you really need to set an object as a default, you can do so in the Model's `initialize` method or by using a more advanced function-based form of defaults, which we'll cover later on in *Chapter 7*, *Fitting Square Pegs in Round Holes – Advanced Backbone Techniques*.

While setting default attributes for Models is easy, the same cannot be said about adding default direct properties to a Model when it is first created. The best way to set these properties is to use the Model's `initialize` method. For instance, if you want to set a `renderBlackAndWhite` property when you create a `Book` Model, you can do what is described in the following code snippet:

```
var Book = Backbone.Model.extend({
    initialize: function(attributes, options) {
        this.renderBlackAndWhite =  options.renderBlackAndWhite;
    }
});
```

The preceding code sets the `renderBlackAndWhite` property of each newly created book to the `renderBlackAndWhite` option passed in when the book is created.

Getters and setters

While the `attributes` property is (behind the scenes) just a JavaScript object, that doesn't mean that we should treat it as such. For instance, you should never set an attribute directly by changing the *attributes* property of a `Model` class:

```
var book = new Book({pages: 200});
book.attributes.pages = 100; // don't do this!
```

Instead, the attributes of Backbone's `Models` should be set using the Model's `set` method:

```
book.set('pages', 100); // do this!
```

The `set` method has two forms or *signatures*. The first (shown previously) takes two arguments: one for the data's key and one for its value. This form is great if you only want to set one value at a time, but if you need to set multiple values, you can use the second form instead, which takes only a single argument containing all the values of `set`:

```
book.set({pages: 50, currentPage: 49});
```

In addition, Models also have an `unset` method, which takes only a single *key* argument and works in a similar way as JavaScript's `delete` statement, except that it also lets any code listening to the Model known about the change:

```
book.unset('currentPage');// book.attributes.currentPage == undefined
delete book.attributes.currentPage; don't do this!
```

As you might have guessed, retrieving attributes through Backbone is done using the `get` method, which takes only a single *key* argument and returns the value for that key:

```
book.set({pages: 50});
var bookPages = book.get('pages'); // == 50
bookPages = book.attributes.pages;// same effect, but again don't do this!
```

The `get` method is very simple; I'd actually like to show its entire source code here:

```
get: function(attr) {
    return this.attributes[attr];
}
```

Now, you may be thinking, if that's all there is to get, why not just use the `attributes` object directly? After all, there's no difference, right?

```
book.get('pages'); // good
book.attributes.pages; // bad?
```

In fact, there are two benefits of using the `get` method, as follows:

- The first benefit is that using `get` makes your code slightly short and more readable as well as more consistent since your `get` calls will match your `set` calls.
- The second benefit, and only a programmatic advantage, is that using `get` offers the option of future extensibility. Like many Backbone methods, `get` is not just designed for out of the box use but also for extension by you, the Backbone developer.

Let's imagine for a moment that you are building a Backbone application, and one day, you realize that you need to know when some piece of code accesses a certain attribute. Maybe, you find the need to do so as you're adding auditing or logging capabilities, or maybe you're just trying to debug a tricky problem.

If you've written your application to access attributes directly, you'll need to find every place in your code that gets an attribute and then update that code. However, if you have already been using the `get` method, you can simply override the `get` method on the relevant `Model` (or `Models`) to tap in to all of your existing `get` logic from a single place in the code.

Listening for events

One of the primary advantages of using the `set` and `unset` methods, as we just described, is that they allow other parts of your code to listen for changes to a Model's attributes. This works similar to listening for a DOM event in jQuery and is done using the `on` and `off` methods of the `Model`:

```
var book = new Backbone.Model({pages: 50});
book.on('change:pages', function() { // triggers when the "pages"
of book  change
    alert('the number of pages changed!');
});
book.set('pages', 51); // will alert "the number of pages has
changed"
book.off('changes:pages');
book.set('pages', 52); // will no longer alert
```

As you can see in the preceding code, the `on` method, just like jQuery's, takes two arguments: an `event` function and a `callback` function. When the event is triggered, Backbone invokes the `callback` function. The event listener can be removed using the `off` method. If you want your code to listen for multiple events, you can combine them with spaces, as shown here:

```
book.on('change destroy', function() {    // this callback will
trigger after a change or a destroy
});
```

Backbone offers one significant improvement over jQuery however, in the form of an optional third `context` argument. If this argument is provided, the callback will be bound (as if you had used the `_.bind` method mentioned in the previous chapter) as it's registered:

```
var Author = Backbone.Model.extend({
    listenForDeathOfRival: function(rival) {
        rival.on('destroy', function celebrateRivalDeath() {
            alert('Hurray!  I, ' + this.get('name') + ', hated ' +
                  rival.get('name') + '!');
        }, this); // "this" is our "context" argument
    }
```

```
});
var keats = new Author({name: 'John Keats');
var byron = new Author({name; 'Lord Byron'});
byron.listenForDeathOfRival(keats);
keats.destroy(); // will alert "Hurray!  I, Lord Byron, hated
John Keats!"
```

In the previous code, we define an `Author` class with a `listenForDeathOfRival` method, which takes a rival (another `Author` class) as an argument and sets up a listener for when the rival is destroyed. We pass the original author as the `context` argument, so when the callback resolves, its `this` method will be set to that author. We then call `listenForDeathOfRival` on `byron` and pass in `keats` so that `byron` listens for a destroy event on `keats`.

When we then trigger Keats' destruction in the final line, we trigger the event listener setup by `listenForDeathOfRival`, which gives the alert message. The message is able to include both the authors' names because when the callback resolves, Keats' name is available from the `rival` variable, while Byron's name is available as an attribute of the `Model` (because we passed his Model as the `context` argument when we set up the event listener).

Available events

`Models` have several different events available for you to listen to, as shown in the following table:

Event name	Trigger
`change`	When any attribute of a Model changes
`change:attribute`	When the specified attribute changes
`destroy`	When the Model is destroyed
`request`	When an AJAX method of the Model starts
`sync`	When an AJAX method of the Model has completed
`error`	When an AJAX method of the Model returns an error
`invalid`	When validation triggered by a Model's `save`/`isValid` call fails

There are also several other events that are related to **`Collection`**, which will be explained further in the next chapter:

Event name	Trigger
`add`	When the Model is added to a Collection
`remove`	When the Model is removed from a Collection

Event name	Trigger
`reset`	When a Collection that the Model belongs to is reset

Models also have one other, special, event called `all`. This event is triggered in response when any of the other `Model` events are triggered.

Custom events

Although you are unlikely to use them very often, you may find it useful to create your own custom events on certain occasions. This can be done by using the Model's `trigger` method, which lets you simulate an event coming from the `Model`, as follows:

```
someModel.trigger('fakeEvent', 5);
```

You can listen for these events in the way same as any other non-custom event, by using the `on` method. Any additional arguments passed to `trigger` (such as the `5` in the preceding example) will be passed as arguments to event handlers that listen for that event.

Server-side actions

Once we've filled a `Model` class with data, we might not want to lose that data, and that's where the AJAX features of Model come into play. Every Model has three methods to interact with the server, which can be used to generate four types of HTTP requests, as shown in the following table:

Method	RESTful URL	HTTP method	Server action
`fetch`	`/books/id`	`GET`	`retrieves data`
`save` (for a new `Model`)	`/books`	`PUT`	`sends data`
`save` (for an existing `Model`)	`/books/id`	`POST`	`sends data`
`destroy`	`/books/id`	`DELETE`	`deletes data`

The sample URLs in the preceding table are what Backbone will generate by default when it tries to perform any of the three AJAX methods. Backbone works best with a set of server-side APIs that are organized using this RESTful architecture. The basic idea of a RESTful server-side API is that it is made up of URL endpoints that expose various resources for the client to consume.

The different HTTP methods are used in such an architecture to control which action the server should take with that resource. Thus, to delete a book with an ID of 7, a RESTful API will expect a `DELETE` request to `/books/7`.

Of course, Backbone won't know that the server-side endpoint for a Book `Model` will be `/books` unless you tell it, which you can do by specifying a `urlRoot` property:

```
var Book = new Backbone.Model.extend({
    urlRoot: '/books'
});
new Book().save(); // will initiate a PUT request to "/books"
```

If, for some reason, your server-side API is on a different domain, you can also use `urlRoot` method to specify an entirely different domain. For instance, to indicate that our `Book` class should use the papers endpoint of `http://www.example.com/`, we can set `urlRoot` of `http://www.example.com/papers`. This approach will not work on most sites, however, because of cross-site scripting limitations imposed by browsers; so, unless you employ special techniques (for instance, a server-side proxy), you will most likely want your server-side API to be on the same domain that serves your JavaScript files.

If your server-side API isn't RESTful, you can still use Backbone's AJAX capabilities by overwriting its built-in `url` method. While the default version of this method simply combines the Model's `urlRoot` method with its ID (if any), you can redefine it to specify any URL that you'd prefer Backbone to use.

For instance, let's say our book Models have a `fiction` attribute and we want to use two different endpoints to save our books: `/fiction` for fiction books and `/nonfiction` for nonfiction books. We can override our `url` method to tell Backbone as much, as shown here:

```
var Book = Backbone.extend({
    url: function() {
        if (this.get('fiction')) {
            return '/fiction;
        } else {
            return  '/nonfiction';
        }
    }
});
var theHobbit = new Book({fiction: true});
alert(theHobbit.url()); // alerts "/fiction"
```

Storing URLs on the client

As we just covered, Backbone allows you to specify any URLs you want for your Models. If your site is fairly complex, however, it's a good idea to store the actual URL strings, or even URL-generating functions, in a separate `urls` object. Take an example of the following code snippet:

```
var urls = {
    books: function() {
        return this.get('fiction') ? '/fiction' : '/nonfiction';
    },
    magazines: '/magazines'
};
var Book = Backbone.Model.extend({url:  urls.books});
var Magazine = Backbone.Model.extend({urlRoot:  urls.magazines});
```

While this is not strictly necessary (and is may be overkill on smaller sites), this approach has several benefits on a larger site:

- You can easily share URLs between any of your Models
- You easily share URLs between your Models and non-Backbone AJAX calls
- Finding existing URLs is quick and easy
- If any URL changes, you only have to edit one file

Identification

Up until now, we've avoided the question of how exactly Backbone determines what a Model's ID is. As it turns out, Backbone has a very simple mechanism: it uses whatever attribute you specify as the `idAttribute` property of the `Model` class. The default `idAttribute` property is simply `id`; so, if the JSON returned by your server includes an `id` attribute, you don't even need to specify an `idAttribute` property. However, since many APIs don't use `id` (for example, some use `_id` instead), Backbone watches for changes to its attributes, and when it sees an attribute that matches the `idAttribute` property, it will set the Model's special `id` property to that attribute's value, as shown here:

```
var Book = Backbone.Model.extend({idAttribute:
'deweyDecimalNumber'})
var warAndPeace = new Book({{deweyDecimalNumber: '082 s
891.73/3'});
warAndPeace.get('deweyDecimalNumber'); // '082 s 891.73/3'
warAndPeace.id; // also '082 s 891.73/3'
```

In addition to telling Backbone what URL to use when saving the Model, the `ID` attribute also has another function: its absence tells Backbone that the Model is new, which you can see if you use the `isNew` method:

```
var warAndPeace = new Book({deweyDecimalNumber: '082 s
891.73/3'});
var fiftyShades = new Book();
warAndPeace.isNew(); // false
fiftyShades.isNew(); // true
```

This method can also be overridden if you need to use some other mechanism to determine whether a `Model` class is new or not. There's one other issue with `isNew` though: if new Models don't have IDs, how will you identify them? For instance, if you were storing a set of `Models` by `ID`, what will you do with the new ones?

```
var warAndPeace = new Backbone.Model({{id: 55});
var shades = new Backbone.Model();
var bookGroup = {};
bookGroup[warAndPeace.id] =  warAndPeace; // bookGroup = {55:
warAndPeace}
bookGroup[shades.id] = shades; // doesn't work because shades.id
is undefined
```

To get around this problem, Backbone provides a special client-side only `ID`, or `cid` property, on every Model. This `cid` property is guaranteed to be unique, but has no connection whatsoever with the Model's actual `ID` (if it even has one). It is also not guaranteed to be consistent; if you refresh the page, your Models' might have entirely different `cid` properties generated for them.

The presence of a `cid` property allows you to use a Model's ID for all server-related tasks and its `cid` for all client-side tasks, without the need to get a new ID from the server every time you create a new Model. By using the `cid` property, we can solve our previous problem and successfully index new books:

```
var bookGroup = {};
bookGroup[warAndPeace.cid] = warAndPeace; // bookGroup = {c1:
warAndPeace}
bookGroup[fiftyShades .cid] = fiftyShades;
// bookGroup = {c1: warAndPeace, c2: fiftyShades};
```

Fetching data from the server

The first of Backbone's three AJAX methods, `fetch`, is used to retrieve the data of a Model from the server. The `fetch` method doesn't require any arguments, but it takes a single optional `options` argument. This argument can take Backbone-specific options, such as silent (which will prevent the Model from triggering an event as a result of fetch), or any options that you will normally pass to `$.ajax`, such as *async*.

You can specify what to do when the `fetch` request is finished in one of the two ways. First, you can specify a `success` or `error` callback in the options passed to `fetch`:

```
var book = new Book({id: 55});
book.fetch({
    success: function() {
        alert('the fetch completed successfully');
    },
    error: function() {
        alert('an error occurred during the fetch');
    }
});
```

The `fetch` method also returns a jQuery promise, which is an object that lets you say *when the fetch is done, do ___* or *if the fetch fails, do ___*. We can use promises instead of a success callback to trigger logic after the AJAX operation finishes, and this approach even lets us chain multiple callbacks together:

```
var promise = book.fetch().done(function() {
        alert('the fetch completed successfully');
}).fail(function() {
        alert('an error occurred during the fetch');
});
```

While both approaches work, the promise style is slightly more powerful because it allows you to easily trigger multiple callbacks from a single `fetch` or trigger a callback only after multiple `fetch` calls complete. For instance, let's say we wanted to fetch two different Models and display an alert only after they've both been returned from the server. Using the success/error approach, this can be awkward, but with the promise style (combined with jQuery's `when` function), the problem is simple to solve:

```
var warAndPeace = new Backbone.Model({{id: 55});
var fiftyShades = new Backbone.Model({id: 56});
var warAndPeacePromise = warAndPeace.fetch();
var fiftyShadesPromise = fiftyShades.fetch();
$.when( warAndPeacePromise, fiftyShadesPromise).then(function() {
    alert('Both books have now been successfully fetched!');
});
```

Once a `fetch` method is complete, Backbone will take the server's response, assume that it's a JSON object representing the Model's attributes, and call `set` on that object. In other words, `fetch` is really just a `GET` request followed by a set of whatever comes back in the response. The `fetch` method is designed to work with a RESTful-style API that returns just the Model's JSON, so if your server returns something else, such as that same JSON wrapped inside an `envelope` object, you'll need to override a special `Model` method called `parse`.

When a `fetch` method finishes, Backbone passes the server's response through `parse` before it calls `set`, and the default implementation of the `parse` method simply returns the object given to it without modification. By overriding the `parse` method with your own logic, however, you can tell Backbone how to convert this response from whatever format the server sent it in into a Backbone attributes object.

For instance, let's say that instead of returning the data for a book directly, your server returned an object with the book's attributes contained inside a book key, as shown here:

```
{
    book: {
        pages: 300,
        name: 'The Hobbit'
    },
    otherInfo: 'stuff we don't care about'
}
```

By overriding the `parse` method of our `Book` class, we can tell Backbone to throw out the `otherInfo` and just use the `Book` property as our book's attribute:

```
var Book = Backbone.Model.extend({
    parse: function(response) {
        return response.pages; // Backbone will call this.
set(response.pages);
    }
});
```

Saving data to the server

Once you've started creating Models, you'll no doubt find yourself wanting to save their data to your remote server, and that's where Backbone's `save` method comes in. As its name suggests, `save` allows you to send your Model's attributes to your server. It does so in the JSON format, at the URL specified by the `url` method of your Model, via an AJAX `POST` or `PUT` request. Like `fetch`, `save` allows you to provide `success` and `error` callback options and (also like `fetch`) it returns a promise, which can be used to trigger code once the `save` method is completed.

Here's an example of `save` used with a promise-based callback:

```
var book = new Book({
    pages: 20,
    title: 'Ideas for Great Book Titles'
});
book.save().done(function(response) {
    alert(response); // alerts the the response's JSON
});
```

The preceding code reveals another problem: it will only work if our server is set up to receive all of the Model's attributes in the JSON format. If we want to save only some of those attributes or if we want to send other JSON, such as a wrapping `envelope` object, Backbone's `save` method won't work out of the box. Fortunately, Backbone provides a solution in the form of its `toJSON` method. When you save, Backbone passes your `Models` attributes through `toJSON`, which (like `parse`) normally does nothing, because by default `toJSON` simply returns whatever is passed in to it.

However, by overriding `toJSON`, you can gain complete control over what Backbone sends to your server. For instance, if we wanted to wrap our book JSON inside another object as a `book` property and then add some further information, we can override `toJSON` as follows:

```
var Book = Backbone.Model.extend({
    toJSON: function(originalJson) {
        return {
            data:  originalJson,
            otherInfo: 'stuff'
        };
    }
});
var book = new Book({pages: 100);
book.save(); // will send: {book: {pages: 100}, otherInfo:
'stuff'}
```

Validation

Before you send anything to the server, it's a good idea to make sure that the data you're sending is actually valid. To solve this problem, Backbone provides another method for you to optionally overwrite: `validate`. The `validate` method is called every time you save, but its default implementation simply returns `true`. If you overwrite it, however, you can add whatever validation logic you want, and if that logic returns `false`, then the whole `save` operation will fail and also return `false`. For instance, let's say we wanted to ensure that every new book has a minimum of 10 pages:

```
var Book = Backbone.Model.extend({
    validate: function(attributes) {
        var isValid = this.get('pages') >= 10;
        return isValid;
    }
});
var tooShort = new Book({pages: 5});
var wasAbleToSave = tooShort.save(); // == false
```

Note that if the validation fails, Backbone will not even return a promise; it will just return `false`.

As a result, if you add validation logic to your `Model` class, you will need to test for a failed validation case separately every time you `save`; you can't simply rely on the `fail` method of the returned promise since no such promise will be returned. In other words, the following won't work (and will cause an error since `false` has no `fail` method):

```
tooShort.save().fail(function() {
    // this code will never be reached
});
```

Instead, you should use the following code snippet:

```
var savePromise = tooShort.save();
if (savePromise) {
    savePromise.done(function() {
        // this code will be reached if both the validation and
             AJAX call succeed
    }).fail(function() {
        // this code will be reached if the validation passes but
            the AJAX fails
    });
```

```
} else {
    // this code will be reached if the validation fails
}
```

Note that you can also check the validity of your Models at any time by using the **`isValid`** method, which will only return the validation result (and not `save`).

Return of Underscore

That covers all the core functionality of `Model`, but before we move on to explore `Collections`, some of the convenience methods of `Model` are worth mentioning. In addition to requiring Underscore, Backbone alo incorporates many `Underscore` methods into its classes as shortcut methods, and `Model` is a perfect example. The main advantage of using these built-in shortcut methods, besides being a bit more readable, is that they operate on the Model's attributes rather than the `Model` itself.

For instance, Underscore has a method called `keys`, which you can use to get all the keys on an object. You can use this method directly to get all the keys of a Model's attributes, as follows:

```
var book = new Backbone.Model({pages: 20, title: 'Short Title'};
var attributeKeys = _.keys(book.attributes);
alert(attributeKeys); // alerts ['pages', 'title']
```

However, if you use Model's version of that same method instead, you can simplify your code slightly and make it slightly more readable:

```
var attributeKeys =  book.keys();
alert(attributeKeys); // alerts ['pages', 'title'];
```

There are a total of six of these methods on `Model`, and while we don't have time to explain all of them in this book, here's a brief summary of what each one does:

Name	What it does
`keys`	This returns every attribute key
`values`	This returns every attribute value
`pairs`	This returns an array of attribute key/value pairs
`invert`	This returns the attributes with keys and values switched; for instance, an attribute of `{'pages': 20}` will become `{'20': 'pages'}`

Name	What it does
`pick`	This returns both the keys and values of only the specified attributes; for instance, `book.pick('pages')` will return `{pages: 20}` without any title or other attributes
`omit`	This returns both the keys and values for every attribute except those specified; for instance, `book.omit('pages')` will return `{title: 'Short Title'}`

Summary

In this chapter, we explored Backbone's `Model` class. You learned how to use `get` and `set` to change attributes, how to use `on` and `off` to listen for events, and how to use `fetch`, `save`, and `destroy` to exchange data with a remote server. You also learned how you can customize Backbone to handle your server's API by modifying the `url`, `urlRoot`, and `idAttribute` properties and how to handle differently structured data with `parse` and `toJSON`.

In the next chapter, we'll take a look at Backbone's other data class, `Collection`. Collections allow you to store groups of Models together, and (just like Models) they allow you to listen for changes and send or retrieve data to/from the server.

4
Organizing Models with Collections

In this chapter, we'll learn how to do the following:

- Create new `Collection` subclasses
- Add and remove Models from a Collection
- Trigger other code in response to changes in the Collection
- Store and retrieve `Collections` of `Models` to and from a remote server
- Sand index the Models of a Collection
- Take advantage of the the convenience methods borrowed from Underscore

Working with Collections

In Backbone, Models may form the core of all data interaction, but you'll soon find that you need to work with multiple Models to do any real work, which is where the `Collections` class comes in. Just as a Model wraps an object and provides additional functionality, `Collections` wraps an array and offers several advantages over using the array directly:

- `Collections` uses Backbone's class system, making it easy to define methods, create subclasses, and so on
- `Collections` allows other code to listen for and respond when `Models` are added or removed from that `Collection` or when Models in a `Collection` are modified
- `Collections` simplifies and encapsulates the logic for communicating with the server

We can create a new `Collection` subclass, as follows:

```
var Cats = Backbone.Collection.extend({
     // The properties and methods of Cats would go here
});
```

Once we've created a subclass, we can instantiate new instances of it using JavaScript's `new` keyword, just as we did to create new `Model` instances. Like Models, Collections have two optional arguments. The first argument is an initial array of `Models` or `Model` attributes, and the second argument is the Collection's options. If you pass in an array of `Model` attributes, Backbone will convert them into Models for you, as follows:

```
var cartoonCats = new Cats([{id: 'cat1', name: 'Garfield'}]);
var garfield = cartoonCats.models[0]; // garfield is a  Model
```

Once a `Collection` subclass has been created, it stores its Model in a hidden property called `models`. Similar to attributes, models should not be used directly, and you should instead rely on the methods of `Collection` to work with its Models. Backbone also provides a `length` property on each `Collection`, which is always set to the number of Models in the `Collection` subclass:

```
var cartoonCats = new Cats([{name: 'Garfield'}, {name: 'Heathcliff'}]);
cartoonCats.models.length; // 2, but instead you can do ...
cartoonCats.length; // also 2
```

Collections and Models

Every `Collection` class has a `model` property, which defaults to `Backbone.Model`. You can set this property by providing it as an option when you use `extend` to create your `Collection` subclass:

```
var Cats = Backbone.Collection.extend({model: Cat});
var cats = new Cats(); // cats.model == Cat
```

However, this `model` property doesn't actually limit the type of Model a `Collection` can hold, and in fact, any `Collection` can hold any type of `Model`:

```
var Cat = Backbone.Model.extend();
var Cats = Backbone.Collection.extend({model: Cat});
var Dog = Backbone.Model.extend();
var snoopy = new Dog({name: 'Snoopy'});
var cartoonCats = new Cats([snoopy]);
cartoonCats.models[0] instanceof Dog; // true
```

This is because the `model` property of a `Collection` is only used when new Models are created through the `Collection`. One way in which this can happen is when the `Collection` is initialized with an array of attributes, as shown in the following example:

```
var snoopyAttributes = {id: 'dog1', name: 'Snoopy'};
var cartoonCats = new Cats([snoopyAttributes]);
cartoonCats.models[0] instanceof Cat; // true
```

Another way in which a Collection's `model` property can be used is via the Collection's `create` method. Calling `create` on a `Collection` and passing it an `attributes` object is essentially the same as calling new `Collection.model(attributes)`, except that after the provided attributes are converted into a `Model`, Backbone will add this `Model` to the `Collection` and then save it:

```
var cartoonCats = new Cats();
var garfield = cats.create({name: 'Garfield'}); // equivalent to:
// var garfield = new Cat({name: 'Garfield'});//
cats.models.push(garfield);
// garfield.save();
```

Adding to and resetting Collections

In addition to passing in `Models` or `attributes` when we first create a `Collection`, we can also add individual `Models` or `attributes`, or arrays of `Models` or `attributes`, to a `Collection` via the `add` method:

```
var cats = new Backbone.Collection();
cats.add({name: 'Garfield'});
cats.models[0] instanceof Cat; // true
```

Similar to `create`, the `add` method will use the Collection's `model` property to create the resulting `Models`. If instead you want to replace all the existing `Models` in a `Collection` rather than add more, you can use the `reset` method, as shown here:

```
var cats = new Backbone.Collection([{name: 'Garfield'}]);
cats.reset([{name: 'Heathcliff'}]);
cats.models[0].get('name'); // "Heathcliff"
cats.length; // 1, not 2, because we replaced Garfield with
Heathcliff
```

Indexing

In order to get or remove specific `Models` from a `Collection`, Backbone needs to know how to index these `Models`. Backbone does this in two ways:

- Using the Model's `id` property, if any (which, as we discussed in the previous chapter, can either be set directly or indirectly by setting the Model's `idAttribute` property)
- Using the Model's `cid` property (which all Models have)

When you add a `Model` or `attributes` to a `Collection`, Backbone uses both of the preceding forms of identification to register the `Model` in the Collection's `_byId` property. `_byId` is yet another one of Backbone's hidden properties, but it's a `private` property as well (because its name is prefixed by _). This means that, even more so than with other hidden properties, you should avoid using `_byId` directly and instead use it indirectly through methods such as `get`. The `get` method returns the `Model` with the provided ID (if any) by using `_byId`:

```
var garfield = new Cat({id: 'cat1', name: 'Garfield'});
var cats = new Backbone.Collection([garfield]);
cats.get('cat1'); // garfield
cats.get(garfield.cid); // also garfield
cats.get('cat2'); // returns undefined
```

The `_byId` property can also be used indirectly in another `Collection` method, which is `remove`. As its name implies, `remove` takes out the `Model` with the provided ID from the `Collection`:

```
var cats = new Backbone.Collection([{
    id: 'cat1',    name: 'Garfield'
}]);
cats.remove('cat1');
cats.length; // 0
```

You can also `get` and `remove` Models from a `Collection` using a set of methods you're already familiar with, because they also exist (and work the same way) on arrays in JavaScript. These are the methods that you can use:

- `push`
- `pop`
- `unshift`
- `shift`
- `slice`

Finally, if you want to retrieve a `Model` from a `Collection` using its index, instead of its `id`, you can use `at`, a Collection's final Model accessing method. The `at` method takes a zero-based index and returns the `Model` at that index, if any:

```
var cats = new Backbone.Collection([
    {name: 'Garfield'},    {name: 'Heathcliff'}
]);
cats.at(1); // returns heathcliff
```

Sorting

Backbone can automatically sort all `Models` added to a `Collection` if you tell it to by specifying a comparator. As with a model, a Collection's comparator can be provided as an option when the `Collection` class is created:

```
var Cats = Backbone.Collection.extend({comparator: 'name'});
var cartoonCats = new Cats();
cartoonCats.comparator; // 'name'
```

While the provided comparator controls how the Collection sorts its models internally, you can also sort a Collection using alternate comparators by using one of Underscore's sorting methods, which we'll detail at the end of the chapter.

The comparator itself can come in three forms. The first and simplest form is the name of an attribute of the Models in the Collection. If this form is used, Backbone will sort the collection based on the values of the specified attribute:

```
var cartoonCats = new Backbone.Collection([
    {name: 'Heathcliff'},
    {name: 'Garfield'}
], {
    comparator: 'name'
});
cartoonCats.at(0);// garfield, because his name sorts first alphabetically
```

The second form of comparator is a function that takes a single argument. If this form is used, Backbone will pass any Models it is trying to sort to this function one at a time and then use whatever value is returned to sort the Model. For instance, if you want to sort your Models alphabetically, except if that `Model` is `Heathcliff`, you can use a comparator of this form:

```
var Cats = Backbone.Collection.extend({
    comparator: function(cat) {
        if (cat.get('name') == 'Heathcliff') return '0';
        return cat.get('name');
    }
```

```
});
var cartoonCats = new Cats([
    {name: 'Heathcliff'},    {name: 'Garfield'}
]);
cartoonCats.at(0); // heathcliff, because "0" comes before
"garfield" alphabetically
```

The final form of `comparator` is a function that takes two `Model` arguments and returns a number, indicating whether the first argument should come before (`-1`) or after (`1`) the second. If there is no preference as to which Model should come first, the function will return `0`. We can implement our same first `Heathcliff` comparator using this style:

```
var Cats = Backbone.Collection.extend({
    comparator: function(leftCat, rightCat) {
        // Special case sorting for Heathcliff
        if (leftCat.get('name') == 'Heathcliff') return -1;
        // Sorting rules for all other cats
        if (leftCat.get('name') > rightCat.get('name')) return 1;
        if (leftCat.get('name') < rightCat.get('name')) return -1;
        if (leftCat.get('name') == rightCat.get('name')) return 0;
    }
});
var cartoonCats = new Cats([
    {name: 'Heathcliff'},    {name: 'Garfield'}
]);
cartoonCats.at(0);// heathcliff, because any comparison of his name
will return -1
```

Events

Just as with Models, Collections have `on` and `off` methods, which can be used to trigger logic when certain events occur in their Collection. The following table shows the events that can occur:

Event name	Trigger
`add`	When a Model or Models is/are added to the Collection
`remove`	When a Model or Models is/are removed from the Collection
`reset`	When the Collection is reset
`sort`	Whenever the Collection is sorted (typically after an add/remove)
`sync`	When an AJAX method of the Collection has completed
`error`	When an AJAX method of the Collection returns an error
`invalid`	When validation triggered by a Model's `save`/`isValid` call fails

Collections, such as Models, also have a special `all` event, which can be used to listen for any event occurring on the Collection. In addition, the `on` method of a Collection can also be used to listen for any Model events triggered by the Models in the Collection, as shown here:

```
var cartoonCats = new Cats([{name: 'Garfield'}]);
cartoonCats.on('change', function(changedModel) {
    alert( changedModel.get('name') + ' just changed!');
});
cartoonCats.at(0).set('weight', 'a whole lot'); // alerts
"Garfield just changed!"
```

Server-side actions

Just as with Models, Collections have a `fetch` method that retrieves data from the server and a `save` method to send data to the server. One minor difference, however, is that by default, a Collection's `fetch` will merge any new data from the server with any data it already has. If you prefer to replace your local data with server data entirely, you can pass a `{reset: true}` option when you fetch. Collections also have the `url`, `parse`, and `toJSON` methods that control how `fetch`/`save` work.

All these methods work in the same way as they do on Models. However, Collections do not have `urlRoot`; while the Models inside a Collection may have IDs, the Collections themselves don't, so Collections don't need to generate their URLs using a `.urlRoot`:

```
var Cats = Backbone.Collection.extend({
     url: '/cats'
});
var cats = new Cats({name: 'Garfield'});
cats.save(); // saves Garfield to the server
cats.fetch(); // retrieves cats from the server and adds them
```

Underscore methods

`Collections`, such as `Models`, have a number of methods inherited from Underscore, but Collections actually have a lot more methods than Models...28 methods to be exact. We can't possibly cover all these methods in detail here, so I'll just provide a quick summary of what each method does and then explore a few of the more important ones in depth.

Note that just as with the Underscore methods on Model, some of these methods operate on the attributes of the `Models` inside the `Collection` rather than on the `Collection` or `Models` themselves. At the same time, however, all the methods that return multiple results return plain old JavaScript arrays, not new Collections. This is done for performance reasons and shouldn't be an issue because if you need those results as a Collection, you can simply create a new one and pass in the results array.

Name	What it does
`each`	This iterates over every Model in the Collection
`map`	This returns an array of values by transforming every Model in the Collection
`reduce`	This returns a single value generated from all the Models in a Collection
`reduceRight`	The same as `reduce`, except that it iterates backwards
`find`	This returns the first Model that matches a testing function
`filter`	This returns all the Models that match a testing function
`reject`	This returns all the Models that don't match a testing function
`every`	This returns `true` if every Model in the Collection matches a test function
`some`	This returns `true` if some (any) Model in the Collection matches a test function
`contains`	This returns `true` if the Collection includes the provided Model
`invoke`	This calls a specified function on every Model in the Collection and returns the result
`max`	This applies a conversion function to every Model in the Collection and returns the maximum value returned
`min`	This applies a conversion function to every Model in the Collection and returns the minimum value returned
`sortBy`	This returns the Collection's Models sorted based on an indicated attribute
`groupBy`	This returns the Collection's Models grouped by an indicated attribute
`shuffle`	This returns one or more randomly chosen Models from the Collection
`toArray`	This returns an array of the Collection's Models
`size`	This returns a count of Models in the Collection, such as `Collection.length`
`first`	This returns the first (or first N) Model(s) in the Collection
`initial`	This returns all but the last Model in the Collection
`rest`	This returns all the Models in the Collection after the first N
`last`	This returns the last (or last N) Model(s) in the Collection

Name	What it does
`without`	This returns all the Models in the Collection except the provided ones
`indexOf`	This returns the index of the first provided Model in the Collection
`lastIndexOf`	This returns the index of the last provided Model in the Collection
`isEmpty`	This returns `true` if the Collection contains no Models
`chain`	This returns a version of Collection that can have multiple `Underscore` methods called on it successively (chained); call the `value` method at the end of the chain to get the result of the calls
`pluck`	This returns the provided attribute from each Model in the Collection
`where`	This returns all the Models in the Collection that match the provided attribute template
`findWhere`	This returns the first Model found in the Collection that matches the provided attribute template

Previously mentioned methods

Several methods in the preceding list should already be familiar, as we have already explained the `each`, `map`, `invoke`, `pluck`, and `reduce` methods in *Chapter 2*, *Object-Oriented JavaScript with Backbone Classes*. All these methods work the same if you call their Underscore version and then pass `Collection.models` to them, as shown here:

```
var cats = new Backbone.Collection([
    {name: 'Garfield'},    {name: 'Heathcliff'}
]);
cats.each(function(cat) {
    alert(cat.get('name'));
}); // will alert "Garfield", then "Heathcliff"
```

Testing methods

Several of the remaining methods focus on testing the Collection to see whether it passes a certain type of test. The `contains` and `isEmpty` methods allow you to check whether the `Collection` contains a specified `Model` or `Models` or whether it contains any models at all, respectively:

```
var warAndPeace = new Backbone.Model();
var books = new Backbone.Collection([warAndPeace]);
books.contains(warAndPeace); // true
books.isEmpty(); // false
```

For more advanced testing, you can use the `every` and `some` methods, which allow you to specify your own test logic. For instance, if you want to know whether any of the books in a Collection have more than a hundred pages, you can use the `some` method, as follows:

```
var books = new Backbone.Collection([
    {pages: 120, title: "Backbone Essentials 4: The Reckoning"},
    {pages: 20, title: "Even More Ideas For Fake Book Titles"}
]);
books.some(function(book)  {
    return book.get('pages') > 100;
}); // true
books.every(function(book)  {
    return book.get('pages') > 100;
}); // false
```

Extraction methods

Another way in which several of the `Underscore` methods can be used is by extracting a specific `Model` or `Models` from a `Collection`. The simplest way to do this is with the `where` and `findWhere` methods, which return all the (or in the case of `findWhere`, the first) Models that match a provided attributes object. For example, if you want to extract all the books in a Collection, which have exactly one hundred pages, you can use the `where` method, as shown here:

```
var books = new Backbone.Collection([
    {
        pages: 100,        title: "Backbone Essentials 5: The
Essentials Return"
    }, {
        pages: 100,        title: "Backbone Essentials 6: We're Not
Done Yet?"
    },{
        pages: 25,        title: "Completely Fake Title"
    }
]);
var hundredPageBooks = books.where({pages: 100});
//  hundredPageBooks array of all of the books except Completely Fake
Title
var firstHundredPageBook = books.findWhere({pages: 100});
firstHundredPageBook; // Backbone Essentials 5: The Essentials Return
```

What if we need a more complicated selection? For instance, what if instead of extracting all the books with exactly a hundred pages, we wanted to extract any book with a hundred or more pages? For this sort of extraction, we can use the more powerful `filter` method, or its inverse, the `reject` method, instead:

```
var books = new Backbone.Collection([
    {
        pages: 100,        title: "Backbone Essentials 5: The
Essentials Return"
    }, {
        pages: 200,        title: "Backbone Essentials 7: How Are We
Not Done Yet?"
    }, {
        pages: 25,        title: "Completely Fake Title"
    }
]);
var hundredPageOrMoreBooks = books.filter(function(book)  {
    return book.get('pages') >= 100;
});
hundredPageOrMoreBooks; // again, this will be an array of all
books but the last
var hundredPageOrMoreBooks = books.reject(function(book)  {
    return book.get('pages') < 100;
});
hundredPageOrMoreBooks; // this will still be an array of all
books but the last
```

Ordering methods

Finally, we have `toArray`, `sortBy`, and `groupBy`, all of which allow you to get an array of all the `Models` stored in a `Collection`. However, while `toArray` simply returns all the Models in the Collection, `sortBy` returns Models sorted by a provided criteria, and `groupBy` returns Models grouped into a further level of arrays. For example, if you want to get all the books in a Collection sorted alphabetically, you can use `sortBy`:

```
var books = new Backbone.Collection([
    {title: "Zebras Are Cool"},
    {title: "Alligators Are Also Cool"},
    {title: "Aardvarks!!"}
]);
var notAlphabeticalBooks = books.toArray();
notAlphabeticalBooks;// will contain Zebras, then Alligators, then
Aardvarks
var alphabeticalBooks = books.sortBy('title');
alphabeticalBooks;// will contain Alligators, then Aardvarks, then
Zebras
```

If, instead, you want to organize them into groups based on the first letter of their title, you can use `groupBy`, as follows:

```
var firstLetterGroupedBooks = books.groupBy(function(book) {
    return book.get('title')[0];
});
// will be an array of [Alligators,  Aardvarks], [Zebras]
```

Summary

In this chapter, we explored Backbone's `Collection` class. You learned how to add and remove Models and `Model` attributes, how to use the `on` and `off` methods to listen for events, how to control the sorting and indexing of Collections, and how to use `fetch` and `save` to exchange data with a remote server. We also examined the many `Underscore` methods of a Collection and how they can be used to realize the full power of a Collection.

In the next chapter, we'll take a look at Backbone's `View` class. Views allow you to render an HTML page, or a subset of one, using the data from the Models and Collections that we've already covered.

5
Adding and Modifying Elements with Views

In this chapter, we'll take a look at how to do the following:

- Create new `View` classes and instances
- Use Views to render DOM elements
- Connect Views to Models and Collections
- Respond to DOM events using Views
- Decide the rendering style(s) that best fits your project

Views are the core of Backbone-powered sites

While data is certainly important in any application, you can't have an application without a user interface at all. On the web, this means that you have a combination of DOM elements (to display information to the user) and event handlers (to receive input from the user). In Backbone, both of these things are managed by Views; in fact, it's only fair to say that Views pretty much control all the input and output on a Backbone-powered site.

Just as Models wrap an `attributes` object and Collections wrap a `models` array, `Views` wrap a DOM element inside a property called `el`. Unlike attributes and models, however, `el` is not hidden, and Backbone doesn't watch it for changes, so there's nothing wrong with referencing a View's `el` directly:

```
someModel.attributes.foo = 'bar'; // don't do this
$('#foo').append(someView.el); // feel free to do this
```

To create a new `View` subclass, simply extend `Backbone.View` in the same way as you created new `Model` and `Collection` subclasses, as shown here:

```
var MyView = Backbone.View.extend({
    // instance properties and methods of MyView go here
}, {
    // static properties and methods of MyView go here
});
```

Instantiating Views

As we mentioned in *Chapter 2, Object-oriented JavaScript with Backbone Classes*, Views take only a single `options` argument when instantiated. The most important part of these options is the `el` property, which defines the DOM element that the View will wrap as its `el`. A View's `el` option can be defined in one of the following three ways:

- HTML (`new Backbone.View ({el: <div id='foo'></div>})`)
- jQuery Selector (`new Backbone.View ({el: '#foo'})`)
- DOM element (`new Backbone.View ({el: document.getElementById('foo')})`)

You can also choose not to provide an `el` option, in which case Backbone will create the View's `el` option for you. By default, Backbone will simply create an empty `DIV` element (`<div></div>`), although you can change this by providing other options when you create your view. You can provide the following options:

- `tagName`: This changes the generated elements' tag from `div` to the specified value
- `className`: This specifies the HTML `class` attribute that the element should have
- `id`: This specifies the HTML `id` attribute that the element should have
- `attributes`: This specifies the HTML attributes that the element should have

Technically, you can specify both the class and ID of a View's element using the `attributes` option, but because they are important to the definition of a View, Backbone provides separate `id` and `className` options.

Instead of defining the preceding options when you instantiate your View, you can also choose to define them in a `View` class. For instance, if you want to create a `View` class that generates a `<form>` element with a class of `nifty`, you should do the following:

```
var NiftyFormView = Backbone.View.extend({
    className: 'nifty',
    tagName: 'form'
});
var niftyFormView = new NiftyFormView();
var niftyFormView2 = new Backbone.View({
    className: 'nifty',
    tagName: 'form'});
// niftyFormView and niftyFormView2 have identical "el" properties
```

Rendering view content

While Views can take an `el` option to define their initial element, it is rare for them to leave this `el` option unchanged. For instance, a `list` View might take an `<ul>` element as its `el` option but then fill this list with the `<li>` elements (possibly using data from a Collection). In Backbone, this generation of inner HTML is done inside the View's `render` method.

However, when you try to use the `render` method of an unmodified View, you quickly notice a problem with Backbone's default implementation, as follows:

```
render: function() {
    return this;
}
```

As you can see, the default `render` method doesn't actually render anything. This is because different Views can have entirely different content, so it doesn't make sense for Backbone to provide only one way of generating that content. Instead, Backbone leaves the implementation of your View's `render` method entirely up to you.

Later on, in this chapter, we'll consider the various strategies for how you might want to implement `render` methods on your site, but before we get to that, let's first examine how to connect Models and Collections to Views as well as how Views handle event bindings.

Connecting Views to Models and Collections

When a View is created, it can take two important options: `Model` and `Collection`. Both of these options are simple property options, which is to say that Backbone doesn't actually do anything with them other than add them as properties to the View. Even so, these properties can be very useful when you want to display or edit data that you've previously generated. For instance, if you want to associate a View with a `book` Model that you have created, you can do the following:

```
var book = new Backbone.Model({
    title: 'Another Fake Title? Why?'
});
var bookView = new Backbone.View({model: book});
// book == bookView.model;
```

When you write the `render` method for your `book` View, you can use that Model in order to get the data to generate the appropriate HTML. For instance, here's a simple implementation of `render`, loosely borrowed from the Backbone documentation:

```
render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
}
```

As you can see, the imaginary `render` method passes the output of the Model's `toJSON` to the View's templating system, presumably so that the templating system can use the Model's attributes to render the View.

Accessing a View's el element

Once you've created a View, you can access the element that it wraps at any time by referring to its `el` property. You can also access a jQuery-wrapped version of the same element by referring to the View's `$el` property. Take an example of the following code:

```
var formView = new Backbone.View({tagName: 'form'});
formView.$el.is('form'); // returns true
```

Backbone also provides another convenient shortcut when you want to access elements inside a View's element: the `$` method. When you use this method, it's effectively the same as calling jQuery's `find` method from the View's element. Because the search for the element is localized to only look through the View's `el` element and not through the entire page's DOM, it will perform much better than a global jQuery selection.

For example, if you create a View of a `<form>` element with an `<input>` element inside it, you can use the View's `$` method to access the `<input>` element , as shown here:

```
var formView = new Backbone.View({
    el: '<form><input value="foo" /></form>'
});
var $input = formView.$('input');
$input.val(); // == "foo"
```

A brief aside on $Variable names

When working with jQuery objects in Backbone (or even just in JavaScript, in general), it may often be difficult to tell whether a given variable refers to a `View` element or to its `el` element. In order to avoid confusion, many programmers (including the authors of both Backbone and jQuery) preface any variable that points to a jQuery object with the `$` symbol, as follows:

```
var fooForm = new Backbone.View({id: 'foo', tagName: 'form'});
var $fooForm = fooForm.$el;
```

While this practice is certainly not necessary to use Backbone, adopting it will likely save you from confusion in the future.

Handling events

While the Views we've described so far are great for creating and/or wrapping existing HTML elements, they're not very good at responding when a user interacts with them. One approach to solve this problem will be to hook up event handlers inside a View's `initialize` method, as follows:

```
var FormView = Backbone.View.extend({
    id: 'clickableForm',
    initialize: function() {
        this.$el.on('click', _.bind(this.handleClick, this));
    },
    handleClick: function() {
        alert('You clicked on ' + this.id + '!');
    }
});
var $form = new FormView().$el;
$form.click(); // alerts "You clicked on clickableForm!"
```

However, there are two problems with this approach, as follows:

- It's not terribly readable
- We have to bind the event handler when we create it so that we can still reference `this` from inside it

Luckily, Backbone offers a better way, in the form of an optional property called `events`. We can use this `events` property to simplify our previous example:

```
var FormView = Backbone.View.extend({
    events: {'click': 'handleClick'},
    id: 'clickableForm',

    handleClick: function() {
        alert('You clicked on ' + this.id + '!');
    }
});
var $form = new FormView().$el;
$form.click(); // alerts "You clicked on clickableForm!"
```

As you can see, using the `events` property saves us from even having to write an `initialize` method at all, while at the same time, it also takes care of binding the `handleClick` event handler to the View itself. In addition, by defining the event handlers in this way, we let Backbone know about them so that it can manage to remove or rebind them as needed.

When you instantiate a View, Backbone calls a `delegateEvents` method, which checks the `events` property, binds all the handlers found in it, and then creates listeners for the appropriate events using jQuery. There is also a corresponding `undelegateEvents` method, which can be used to remove all the event handlers. Normally, you won't need to call either of these methods yourself because Backbone will call them for you.

However, if you change a View's `el` element without telling Backbone (for example, `yourView.el = $('#foo')[0]`), Backbone won't know that it needs to hook up the events to the new element, and you will have to call `delegateEvents` yourself.

Alternatively, instead of changing a View's `el` element manually and then calling `delegateEvents` afterwards, you can use a View's `setElement` method to do both at the same time, as follows:

```
var formView = new Backbone.View({el: '#someForm'});
formView.setElement($('#someOtherForm'));
// formView.el == someOtherForm
```

Rendering strategies

Now that we've covered all of View's capabilities, it's time to return to the question of how to render a View. Specifically, let's look at the main options available to you, which are explained in the sections that follow, when you overwrite the `render` method.

Simple templating

The first, and perhaps the most obvious, approach for rendering is to use a simple, logic-less templating system. The `render` method provided in the Backbone documentation is a perfect example of this, as it relies on the Underscore library's `template` method:

```
render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
}
```

The `template` method takes a string, which contains one or more specially designated sections, and then combines this string with an object, filling in the designated sections with that object's values. This is best explained with the following example:

```
var author ={
    firstName: 'Isaac',
    lastName: 'Asimov',
    genre: 'science-fiction'
};
var templateString = '<%= firstName %> <%= lastName %> was a '+
                     'great <%= genre %> author.';
var template = _.template(templateString);
alert(template(author));
// will alert "Isaac Asimov was a great science-fiction author."
```

While the `template` function works with any string and data source, when used as part of a Backbone View, it is typically used with an HTML string and a `Backbone.Model` data source. Underscore's `template` function lets us combine the two to easily create a `render` method.

For instance, if you want to create a `<div>` tag with an `<h1>` tag inside containing an author's name and genre and then add an emphasis (in other words, an `<em>` tag) around the genre, you can create a `template` string with the desired HTML and placeholders for the first name, last name, and genre. We can then use the `_.template` to create a `template` function and then use this `template` function in a `render` method with an author Model's attributes.

Of course, as we mentioned in *Chapter 3, Accessing Server Data with Models*, it's safer if we don't access a Model's attributes directly; so, we'll want to use the Model's `toJSON` method instead. Putting all of this together, we get the following block of code:

```
var authorTemplate =  _.template(
    '<h1>' +
        '<%= firstName %> <%= lastName %> was a ' +
        '<em><%= genre %></em> author.' +
    '</h1>'
);
var AuthorView = Backbone.View.extend({
  template: authorTemplate,
  render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
  }
});
var tolkien = new Backbone.Model({
    firstName: 'J.R.R.', // Tolkien's actual first name was John
    lastName: 'Tolkien',
    genre: 'fantasy'
});
var tolkienView = new AuthorView({model: tolkien});
var tolkientHtml = tolkienView.render().$el.html();
alert(tolkienHtml);
// alerts "<h1>J.R.R. Tolkien was a <em>fantasy</em> author.</h1>"
```

One major advantage of this approach is that because our HTML is completely separated into a string, we can optionally choose to move it out into a separate file and then bring it in using jQuery or a dependency library, such as `Require.js`. HTML that is stored separately like this can be easier for a designer to work with, if you have such a person in your team.

Advanced templating

Instead of using Underscore's `template` method, you can also employ one of the many quality third-party templating libraries available, such as Handlebars, Mustache, Hamljs, or Eco. All these libraries offer the same basic ability to combine a data object with a template string, but they also offer the possibility to include logic inside the template. For instance, here's an example of a Handlebars template string that uses an `if` statement, which is based on a provided `isMale` data property, to select the correct gender pronoun inside a template:

```
var heOrShe = '{{#if isMale }}he{{else}}she{{/if}}';
```

If we use Handlebars' `compile` method to turn that into a template, we can then use it just as we will use an Underscore template:

```
var heOrSheTemplate = Handlebars.compile(heOrShe);
alert(heOrSheTemplate({isMale: false})); // alerts "she"
```

We'll discuss more about Handlebars in *Chapter 11, (Not) Re-inventing the Wheel: Utilizing Third-Party Libraries* but the important thing to understand for now is that no matter which templating library you choose, you can easily incorporate it as part of your View's `render` method. The difficult part is deciding whether or not you want to allow logic inside your templates and if so, how much.

Logic-based

Instead of relying on templating libraries, another option is to use Backbone's View logic, jQuery methods, and/or string concatenation to power your `render` methods. For instance, you can reimplement the preceding `AuthorView` without using templates at all:

```
var AuthorView = Backbone.View.extend({
  render: function() {
    var h1View = new Backbone.View({tagName: 'h1'});
    var emView = new Backbone.View({tagName: 'em'});
    emView.text(this.model.get('genre'));
    h1View.$el.html(this.model.get('firstName') + ' ' +
                    this.model.get('lastName') + ' was a ');
    h1View.$el.append(emView.el, ' author.');
    this.$el.html(h1View.el);
    return this;
  }
});
```

As you can see, there are both advantages and disadvantages to a purely logic-based approach. The main advantage, of course, is that we don't have to deal with a template at all. This means that we can see exactly what logic is being used, because nothing is hidden inside the template library's code. Also, there is no limit on this logic; you can do anything that you will normally do in JavaScript code.

However, we also lost a good deal of readability by not using a template, and if we had a designer on our team who wanted to edit the HTML, they would find `that` code very difficult to work with. In the first version, they will see a familiar HTML structure, but in the second version, they will have to work with the JavaScript code even though they (probably) aren't familiar with programming. Yet another downside is that because we've mixed the logic with the HTML code, there's no way to store the HTML in a separate file, the way we can if it were a template.

The combined approach

All three preceding approaches have advantages and disadvantages, and rather than settling for just one, you can choose to combine some of them instead. There's no reason why, for instance, you can't use a templating system (either Underscore's for simplicity or an external one for power) and then switch to using logic when you want to do something that doesn't fit neatly into a template.

For example, let's say you want to render a `<ul>` with a `class` HTML attribute derived from the attributes of a Model—this sounds like something that will be easier with JavaScript logic. However, let's say you also want this `<ul>` to contain `<li>` elements with text based on a template and filled in with the attributes of Models in a Collection; that sounds like something we can best handle with a template. Luckily, there is nothing stopping you from combining the two approaches, as shown here:

```
var ListItemView = Backbone.View({
    tagName: 'li',
    template: _.template('<%= value1 %> <%= value2 %> are both ' +
                         'values, just like <%= value3 %>'),
    render: function() {
        this.$el.html(this.template(this.model.toJSON()));
        return this;
    }
});
var ListView = Backbone.View.extend({
  tagName: 'ul',
  render: function() {
    this.$el.toggleClass('boldList', this.model.get('isBold'));
    this.$el.toggleClass('largeFontList',
                         this.model.get('useLargeFont'));
    this.$el.empty();
    this.collection.each(function(model) {
        var itemView = new ListItemView({model: model});
        this.$el.append(itemView.render().el);
    }, this);
    this.$el.html(this.template(this.model.toJSON()));
    return this;
  }
});
```

Of course, mixing approaches means that you won't always get the full benefits of both. The most notable issue will be that any HTML that you render via Backbone's Views or other JavaScript logic won't be accessible to a non-programming designer. If you don't have such a role in your team, then this limitation won't bother you, but if you do, you should probably try to keep as much HTML inside templates as possible.

Other render considerations

In addition to deciding whether you can rely on templates and how much you can rely on them in your render methods, there are a few other things that you need to consider when writing them. None of these choices are completely binary, but you should strive for as consistent an approach as possible. This means that the real question isn't about which one you should choose, as it is about when you will choose one over the other. Ultimately, a consistent strategy will make it such that both you and your co-workers don't have to think in order to decide which approach to use when writing new `render` methods or editing existing ones.

Child views

Very often, in Backbone, you will want to create one View for a container element, such as a `<ul>` element, and another View for its child elements, such as its `<li>` element. When creating these child Views, you can either choose to have a child View create the child elements, or you can have the parent View create them and pass a jQuery selector as the child View's `el` element, as shown here:

```
var ListView1 = Backbone.View.extend({
    render: function() {
        this.$el.append(new ListItemView().render().el);
    }
});
var ListView2 = Backbone.View.extend({
    render: function() {
        this.$el.append('<li>');
        new ListItemView({el: this.$('li')}).render();
    }
});
```

The primary advantage of generating the child element in the View is encapsulation. By taking this approach, you can keep all the logic related to the child element within the child View. This will make the code easier to work with, as you won't have to think about both the child and parent Views at the same time.

On the other hand, when you are using templates, it may be inconvenient to split the template into parts in order to render the overall container element. Also, rendering the child elements in the parent view provides a good separation between DOM-generating Views and purely event handling Views, which can help you to sort your logic better.

Neither approach has to be taken to the extreme. While you can have a single page View with a single template for all your content or you can have every single child View generate its DOM elements, you can also choose to have a mix of approaches in your Views. In fact, unless you want to have a single page template for a top-level render, which generates the entire DOM, you will probably want to take a combined approach, so the real question is how much do you want to rely on each technique? The more you rely on the parent View, the easier it is to see the big picture; but the more you rely on DOM generation in child Views, the easier it will be to encapsulate your logic.

Repeatable versus one-time renders

Another thing that you'll want to consider is whether to design your `render` methods to be rendered once (usually when the View is first created), or whether you want to be able to call your `render` method multiple times in response to changes. The render methods that are designed to render only once are the simplest to write and the simplest to work with; when something needs to change with your View's DOM element, you simply call the View's `render` method again and the DOM gets updated.

However, this approach can also lead to performance issues, particularly if the View involved contains a great number of child Views. To avoid this, you might instead want to write `render` methods that don't blindly replace their content but instead update them in response to changes. For instance, if you have a `<ul>` elements where you want some of the child `<li>` elements to appear or disappear in response to user actions, you might want to have your `render` method check whether the `<li>` elements already exist first and then simply apply changes to their display style rather than rebuilding the entire `<ul>` from scratch every time.

Again, you'll probably want to mix the two approaches based on the type of View you are writing. If performance is a high priority for you, then you'll likely want to think carefully about how you can reuse previously-generated DOM elements in your `render` methods. On the other hand, if performance is less important, designing your `render` method for one-time DOM generation will keep its logic a lot simpler. Also, if performance problems crop up, you can always change the `render` method in question to reuse existing elements.

Return value – this or this.$el

The sample `render` method from the Backbone documentation returns `this` when it completes. This approach allows you to easily access the `el` property of the render by simply calling it on the `render` method's return value, as follows:

```
var ViewThatReturnsThis = Backbone.View.extend({
    render: function() {
        // do the rendering
        return this;
    }
});
var childView = new ViewThatReturnsThis();
parentView.$el.append(childView.render().$el);
```

However, you can make it even easier to access the View's `el` by returning that `el` instead of this:

```
var ViewThatReturns$el = Backbone.View.extend({
    render: function() {
        // do the rendering
        return this.$el;
    }
});
var childView = new ViewThatReturns$el();
parentView.$el.append(childView.render());
```

The downside of returning the `$el` from a `render` method is that you can no longer chain other View methods from the return value of the render method:

```
parentView.$el.append(childView.render().nextMethod());// won't work
```

So, what this choice really comes down to is how much you plan to post-render your Views and how much you plan to call those post-rendering methods from outside the View itself (if you call them from within the render method, their return values are irrelevant). If you're not sure, the safest bet is to follow the `render` method in Backbone's documentation and return `this`, as it offers maximum flexibility (at the cost of a slightly less-concise code).

Whatever you choose, you'll likely want to keep it consistent across all your Views. If you don't, you will find yourself constantly having to look up a View's `render` method every time you want to use it.

Summary

In this chapter, we explored Backbone's `View` class. You learned how to create DOM elements by instantiating a View as well as how to pass in existing elements to new Views. You also learned how to use Views to hook up event handlers on the View's `el` and how to remove/reattach these handlers to the `undelegate` and `delegate` methods. Finally, we considered how Backbone's (deliberately) empty implementation of the `render` method leaves us many different options for how to implement our own `render` method and what factors influence those choices.

In the next chapter, we'll take a look at Backbone's final class, the `Router` class. This class lets us simulate traditional web pages, only using Views instead of separate HTML files to generate those pages.

6
Creating Client-side Pages with Routers

In this chapter, we'll examine Backbone's `Router` class to learn about the following:

- How routers simulate browser history to create virtual pages
- How to create new `Router` subclasses and instances
- The various mechanisms for defining routing patterns
- How to handle special cases, such as nonexistent routes
- How to use a page view to render routes consistently

Backbone routers enable single-page applications

Until now, all the Backbone features we've looked at have been enhancements to previously existing functionality; Backbone's classes improve on native JavaScript classes—`Models` and `Collections`—as well as enhance the functionality of objects and arrays, and `Views` improve on the DOM manipulation and event binding that jQuery already facilitates.

`Routers`, however, break this trend. Instead of enhancing something that already exists, `Routers` allow you to do something entirely new: create an entire site using only a single HTML page. As discussed in *Chapter 1*, *Building a Single-page Site Using Backbone*, single-page applications offer several advantages over traditional multipage websites: most notably, enhanced performance and complete client-side control over the application. Backbone's `Router` class is what makes all this possible.

How Routers work

On a traditional multipage site, the browser provides both routing and page history without the need for any extra effort. When a user visits a URL on the site, the browser asks the server for that URL's content, and then when the user moves to another URL, the browser tracks the change in the user's browsing history. In a Backbone-powered site, however, both of these must be handled by a `Router` instead. When the user visits a new page, the `Router` determines which `Views` to render, and when they leave that page, the `Router` informs the browser of the new history entry. This lets a Backbone-powered site act just like a traditional one, including allowing the user to use their browser's `Back` button.

There are two different approaches by which a `Router` can accomplish this, and Backbone lets you decide which approach you want to use. The first, and default, approach is hash-based routing, which takes advantage of URL fragments (also known as named anchors). You've probably seen URLs with these fragments before, for instance `www.example.com/foo#bar`. By using these fragments to define the pages of your site, Backbone can both know what page to load and can tell the browser how to track the user's movement across pages.

If you don't want your users to see hashes in your URLs, Backbone offers a second option that relies on the recently added HTML5 pushState technology. While this definitely helps URLs look cleaner, the pushState approach comes with a significant drawback. Although pushState is available in most web browsers, older browsers, such as Internet Explorer 9 and below, do not support it. If you try to use pushState-based routing in a browser without support for it, Backbone will instead fall back on hash-based routing. Even if only 1 percent of your users have an older browser, it still means that these 1 percent users will see a different URL from the rest of your users, which may cause confusion if (for instance) users with different browsers share links.

Routing based on pushState also has a minor drawback: if a user refreshes the page on a pushState-powered site, the browser will request that URL from the server. This can be solved by simply making your server return your application's single HTML page whenever the browser requests for such a URL. However, given both of these drawbacks, using pushState is only recommended if you care strongly about the appearance of your URLs and believe that all your users will have modern browsers.

Backbone.history

Because it's possible to use multiple `Routers` at once (although it's not generally recommended, as we'll discuss later in the chapter), Backbone has a separate global object known as `Backbone.history` that handles history management. It is important to understand, however, that this object doesn't actually replace your browser's history; instead, it simply helps to manage what gets added to the history and when.

When you load a page that uses a Backbone `Router`, you need to invoke the `start` method of this history object in order to tell Backbone to begin routing. This `start` method also lets you tell Backbone which routing technology you need to employ. If you want Backbone to rely on hash-based routing, you can simply call the `this` method without any arguments:

```
Backbone.history.start();
```

If instead you wish to use pushState-based routing, you need to provide one extra argument to indicate it:

```
Backbone.history.start({pushState: true});
```

Differences between routes and pages

Routes and pages are very similar concepts and are often used interchangeably by Backbone programmers. This is natural since routes are essentially just the Backbone implementation of pages. However, there are important differences between the two, at least when you compare the pages of a multi-page website to the routes of a Backbone single-page application.

In traditional, multi-page websites, each new page that the user visits requires you to make a new request to the site's server. In a Backbone-powered site, however, the user can navigate through as many pages (routes) as they want, making new requests only when it is necessary to fetch new data. This feature alone accounts for a significant increase in speed on a Backbone-powered site.

Another important difference is that standard web pages trigger an entire page load, whereas routes trigger only a specific JavaScript function. This means that unlike a traditional application, which has to limit the number of pages (and thus the number of HTTP requests) it makes, routes can be used for just about any possible change in state. Routes can be used in a similar way to traditional web pages, or they can be used for much smaller changes to a page. They can even (rarely) be used when there is no change to the DOM at all.

Of course, while Backbone routes have many advantages, they also have some downsides. The most important downside is that because they do not refresh the page, existing DOM elements, style changes, and event bindings are not cleared automatically. This means that you will need to handle these clean-up tasks yourself. Luckily, this isn't very difficult to do, especially if you rely on a page view, which we will introduce shortly.

Creating a new Router

As with all Backbone classes, you can create a new `Router` subclass by using `extend`, where the first argument provides the class's instance properties and methods and the second argument provides the static ones:

```
var MyRouter = Backbone.Router.extend({
    // instance methods/properties go here
}, {
    // static methods/properties go here
);
```

Similar to `Views`, `Routers` take only a single `options` argument when they are instantiated. This argument is completely optional, and the only real option it takes is the `routes` option. As mentioned, once the `Router` has been created, you will need to run `Backbone.history.start()` before it can handle routes:

```
myRouter = new Backbone.Router({
    routes: {
        'foo': function() {
            // logic for the "/foo" or "#foo" route would go here
       }
    }
});
Backbone.History.start(); // siteRouter won't work without this
```

Creating routes

The problem with passing in your routes when you create a `Router` is that you have to supply all the routing functions together with your routes, which (especially if you have a lot of routes on your site) can easily become messy. Instead, many Backbone programmers define their routes on the `Router` class itself, as shown here:

```
var SiteRouter = Backbone.Router.extend({
    routes: {
        'foo': 'fooRoute'
    },
```

```
    fooRoute: function() {
        // logic for the "/foo" or "#foo" route would go here
    }
});
var siteRouter = new SiteRouter();
Backbone.History.start(); // siteRouter won't work without this
```

As you can see, this approach simplifies the routes' definition and makes them more similar to the `events` properties of `Models` and `Collections`. Instead of defining routing methods with the routes themselves, you can simply give the route the name of a routing method and Backbone will look for that method inside the `Router` class.

There is also a third way to add routes and that is by using Backbone's `route` method. This method takes the `route` as the first argument, the route's name as the second argument, and the route's handler as the third argument. If the third argument is omitted, Backbone will look for a method on the `Router` with the same name as the `route` itself:

```
For example:SiteRouter = new Backbone.Router({
    initialize: function() {
        this.route('foo', 'fooRoute');
    },
    fooRoute: function() {
        // logic for the "/foo" or "#foo" route would go here
    }
});
```

Just as the `on` method of `Models` and `Collections` lets you create event bindings after an instance of the class is created, the `route` method of `Routers` allows you to define a new route at any time. This means that you can dynamically create all your routes on the fly after the `Router` class is created. However, just as you would normally want all the pages of your site to be available to your user when they first visit it, you will generally want all your routes to be available too, which means that you will usually create them in your `Routers` `initialize` method.

The main advantage of using the `route` method is that you can apply logic to your routes. For instance, let's say you have several routes on your site that are only for administrators. If your `t` can check an `isAdmin` attribute of a user `Model`, then it can use this attribute to dynamically add or remove these administrator-only routes when the `Router` class is initialized, as shown here:

```
// NOTE: In a real case user data would come from the server
var user = new Backbone.Model({isAdmin: true});SiteRouter = new
Backbone.Router({
     initialize: function(options) {
        if(user.get('isAdmin')) {
```

```
            this.addAdminRoutes();
        }
    },
    addAdminRoutes: function() {
        this.route('adminPage1', 'adminPage1Route');
        this.route('adminPage2', 'adminPage2Route');
        // etc.
    }
});
```

As you can see in the preceding example, another advantage of using the `route` method is that you can organize groups of routes into separate functions. This can be useful if you want to turn certain groups of routes on or off together, but it can also be useful simply to organize constant routes. In fact, you can even define some of your routes in an entirely different file as long as your `Router` has access to them, as shown here:

```
// File #1
window.addFooRoutes = function(router) {
    router.route('foo', function() {
        //...
    })
}

// File #2SiteRouter = new Backbone.Router({
    initialize: function(options) {
        if(options.includeFooRoutes) {
            addFooRoutes(this);
        }
    }
});
```

Route styles

In addition to having the choice to decide how you hook up your routes, you also have the choice to decide how to define the routes. So far, all the routes you've seen have been simple routes; they define a string of characters, and when Backbone sees that exact string of characters in the URL, it triggers the appropriate routing method. Sometimes, however, you'll want to have routes that are more flexible.

For example, imagine that we have a site which sells books and we want to have a "learn about a particular book" `route`. If we have a good number of books on our site, creating routes for each of them individually might get painful:

```
var SiteRouter = new Backbone.Router({
    initialize: function(options) {
        this.route('book/1', 'book1Route'); // for the book with ID 1
        this.route('book/2', 'book2Route'); // for the book with ID 2
        this.route('book/3', 'book3Route'); // for the book with ID 3
        // this will get old fast
    }
});
```

Luckily, there are two alternatives that solve this problem: routing strings and regular expression routes. First, let's look at how regular expression routes can solve this problem:

```
SiteRouter = new Backbone.Router({
    initialize: function(options) {
        // This regex will match "book/" followed by a number
        this.route(/^book\/(\d+)$/, 'bookRoute');
    },
    bookRoute: function(bookId) {
        // book route logic would go here
    }
});
```

As you can see in the preceding example, we were able to create a single route with a regular expression that matches all our possible book routes. Because this expression contains a group (the part of the regular expression that is surrounded by parentheses), Backbone will pass the matching part of the route to the routing function as an argument, allowing us to know the book's ID from within our routing function. If we were to include multiple groups in our regular expression, Backbone would have provided each matching part of the route as a separate argument to our routing function.

Regular expressions are powerful, which makes them a good choice for defining routes. However, they have one major downside: they are hard for humans to read, especially when you revisit them months after writing them. As the old programming adage goes: you had a problem and tried to solve it using regular expressions. Now you have two problems. For this reason, Backbone provides a third option for defining routes: routing strings.

Routing strings are similar to regular expressions; in that, they let you define dynamic routes but are much more limited. You can only define groups (also known as **parameters**), wildcard groups (also known as **splats**), and optional parts. By trading away some of the power of regular expressions, routing strings gain a great deal of readability. Instead of using `(\d+)` to match a part of the route for a book's ID, a routing string will use the more readable `bookId`.

Just as with a regular expression route, Backbone will provide the route's parameters as arguments to the routing function, as shown here:

```
SiteRouter = new Backbone.Router({
    initialize: function(options) {
        this.route('book/:bookId', 'bookRoute');
    },
    bookRoute: function(bookId) {
        // book route logic would go here
    }
});
```

If we wanted to make that part of the route optional, we can wrap it with parentheses (for example, `book/(:bookId)`). However, there's a problem; routing strings stop matching parameters (optional or not) when they see a `/` character, which means that they won't work if your book ID has a `/` in it. For instance, if we tried to navigate to `book/example/with/slash/5`, our route wouldn't get triggered. To get around this issue, Backbone provides a `wildcard` or `splat` option, which will match the remaining portion of the `route` class. These splats are like parameters, but they use a `*` character instead of a `:` character:

```
SiteRouter = new Backbone.Router({
    initialize: function(options) {
        this.route('book/*bookId', 'bookRoute');
    },
    bookRoute: function(bookId) {
        // book route logic would go here
    }
});
// if we now navigate to "book/example/with/slash/5 " our
bookRoute
// method will receive an argument of "example/with/slash/5 "
```

Because Backbone is not particular about whether you use simple routes, regular expressions, or routing strings, you can freely combine all the three types in any given `Router`. You'll probably find it best, however, to use the most readable form possible each time. In other words, you should start by trying to use a simple route, then switch to a routing string if you need to add a few variables, and only use a regular expression if you can't define the route with a routing string.

A note about routing conflicts

Whichever method you use to create your routes, you should be careful to avoid defining the same route twice or defining two routes that overlap each other, such as `foo` and `splatThatCouldBeFoo`. While Backbone will allow you to define such routes and will continue working fine in spite of them, it will silently ignore any routes after the first that match. Take an example of the following code snippet:

```
new Backbone.Router({
    routes: {
        'foo': function() {alert('bar')},
        ':splatThatCouldBeFoo': function() {alert('baz')},
    }
})
Backbone.history.start();
// navigating to #foo alerts('bar')
```

One could take advantage of this behavior by deliberately defining specific routes, and then less specific overlapping routes, but this is not generally recommended. While a few simple overlapping routes are unlikely to cause problems, if you have too many they can make your `Router` difficult to work with. When you have such overlapping routes you have to first stop and understand every route involved before you can make changes to any of them, and then if you make a mistake it's all to easy to create a difficult to find bug.

Trailing slashes

Before we move on from routes there is one last detail worth mentioning: trailing slashes. Normally web developers don't need to think about trailing slashes, because most web servers treat `foo` and `foo/` the same. Technically however they are different, and Backbone treats them as such, which means that a route of `foo` will not be triggered when the user navigates to `foo/`.

Luckily this can easily be solved if you are using routing strings by adding (`/`) to the end of each string. If you are using regular expressions you can achieve a similar effect by adding `\/?` to the end of each of your regular expressions. However, if you are using simple strings you will either need to define two separate routes (`foo` and `foo/`), or you will just need to be careful never to create any links to `foo/`.

Redirects

Normally users traverse a site by clicking on anchor tags, better known as hyperlinks. The same is true for Backbone-powered sites; even if you use hash-based routes you can still create normal hyperlinks for them:

```
<a href="#foo">Click here to go to the "foo" route</a>
```

Sometimes however you will want to move the user to a different route using JavaScript instead. For instance, you might have a **Submit** button on a page and after it triggers a save of the page's `Model` you want it to redirect the user to a different route. This can be done by using the `Routers` `navigate` method, like so:

```
var router = new Backbone.Router({
    routes: {
        foo: function() {
            alert('You have navigated to the "foo" route!');
        }
    }
});
router.navigate('foo', {trigger: true});
```

It's important to note the addition of the `trigger` option as the second argument to `navigate`. Without this extra argument, Backbone will take the user to the route's URL but won't actually trigger the route's logic.

The `navigate` method can also take one other option: `replace`. If this option is provided, Backbone will navigate to the specified route but will not add an entry to the browser's history. Take an example of the following line of code:

```
router.navigate('bar', {replace: true, trigger: true});
```

Not creating a history entry means that, for instance, if the user visits another page and then clicks on their browser's **Back** button instead of going back to the replaced page, they will be taken to the page before it in the browser's history. Since this sort of behavior will likely be confusing for your user, it is recommended that you limit the use of `replace: true` to routes that are temporary, such as the route for loading a page.

404s and other errors

Until now, we've focused on what happens when a route matches, but what happens when the user visits a URL that doesn't have a matching route? This can happen, for instance, because of a stale link or because the user mistyped a URL. On a traditional website, the server will handle this by throwing a `404` error, but on a Backbone-powered site, the `Router` class will just do nothing, by default. This means that if you want to have a `404` page on your site, you'll have to create it yourself.

One way to do this is to rely on the `start` method. This method returns `true` or `false` depending on whether any matching routes are found for the current URL:

```
if (!Backbone.history.start()) {
    // add logic to handle the "404 Page Not Found" case here
}
```

However, since that method will only be called once, when the page is first loaded, it won't allow you to catch nonmatching routes that occur after the page is loaded. To catch these cases, you'll need to define a special `404` route, which you can do using the routing string's `splat` syntax:

```
var SiteRouter = Backbone.Router.extend({
    initialize: function(options) {
        this.route('normalRoute/:id', 'normalRoute');
        this.route('*nothingMatched', 'pageNotFoundRoute');
    },
    pageNotFoundRoute: function(failedRoute) {
        alert( failedRoute + ' did not match any routes');
    }
});
```

Routing events

Normally, route-based logic is handled by the `Router` class itself. However, sometimes you may wish to *trigger* additional logic. For instance, you may want to add a certain element to the page whenever the user visits one of several routes. In such a case, you can take advantage of the fact that Backbone triggers events each time routing occurs, allowing you to listen and respond to route changes from outside the `Router` class itself.

You can listen for routing events the same way as events in `Models` and `Collections`, by using the `on` method, as follows:

```
var router = new Backbone.Router();
router.on('route:foo', function() {
    // do something whenever the route "foo" is navigated to
});
```

The following table shows the three different routing events that you can listen for on a `Router` class:

Event name	Trigger
`route`	This is triggered when any route is matched
`route:name`	This is triggered when the route with the specified name is matched
`all`	This is triggered when any event is triggered on the `Router` class

In addition, Backbone also provides a `route` event that you can listen to on the `Backbone.history` object, as follows:

```
Backbone.history.on('route', function() { ... });
```

The advantage of listening to `history` instead of a particular `Router` is that it will catch events from all the `Routers` on your site, instead of just a particular one.

Multiple routers

In general, you only need a single `Router` to power your entire site. However, if you have reason to do so, you can easily include multiple `Routers`, and Backbone will happily allow this. If two or more `Routers` on the same page match a particular route, Backbone will trigger the route from the first `Router` with a matching route defined.

There are two main reasons why you should use multiple `Routers`. The first reason is to separate your routing into logical groups. For instance, in an earlier example, we used a conditional to add certain admin-only routes to the `Router` class when the current user was an administrator. If our site has enough of these routes, it might make sense to create a separate `Router` for the admin-only routes, as follows:

```
var NormalRouter = Backbone.Router({
     routes: {
        // routes for all users would go here
    }
};
var AdminRouter = Backbone.Router({
```

```
        routes: {
            // routes for admin users only would go here
        }
    };
    new NormalRouter();
    if (user.get('isAdmin') {
        new AdminRouter();
    }
```

The other reason why you need to use multiple `Routers` is when you have two different sites that need to share some common code. For instance, a book-selling site might want a main site for their shoppers to buy books and an entirely different site for publishers to submit new books. However, even though the sites are different, they might both want to share some common code, such as a `Book Model` or a book-rendering `View`. By using multiple `Routers`, our bookseller can share any code they want between the two sites, while keeping the pages/routes of each entirely separate from one another.

In practice though, using multiple `Routers` can be confusing, especially if they have overlapping routes. Since you can easily add routes dynamically (as we did with our earlier administrator example), there typically won't be any need for you to rely on multiple `Routers` and you will likely avoid confusion by sticking to only one.

Page views

Before we finish this chapter, it's worth discussing how `Views` interact with `Routers`. Throughout this chapter, we've deliberately been vague about what you should actually do inside your routing functions, and part of the beauty of Backbone is that it leaves this decision entirely up to you.

For many Backbone users, however, a very common pattern is to create a special `Page View` and then instantiate a different subclass of that `View` in every route-handling method. Take an example of the following code snippet:

```
var Book = Backbone.Model.extend({urlRoot: '/book/'});
var Page = Backbone.View.extend({
    render: function() {
        var data = this.model ? this.model.toJSON() : {};
        this.$el.html(this.template(data));
        return this;
    }
});
var BookPage = Page.extend({
   template: 'Title: <%= title %>'
```

```
});
var SiteRouter = new Backbone.Router({
    route: {
        'book/:bookId(/)': 'bookRoute',
    },
    bookRoute: function(bookId) {
        var book = new Book({id:  bookId});
        book.fetch().done(function() {
            var page = new BookPage({model: book});
            page.render();
        });
    }
});
```

If your site has multiple sections, for instance a side navigation bar and a main content area, you can split up these parts into their own `View` classes and then make those parts the defaults used by your `Page View` class. You can then override these defaults as needed in your `Page View` subclasses to allow these `Views` to change the different parts of your site.

For instance, let's say most of the time, your site has a single side navigation bar but on certain pages, you wish to add some extra links to it. By using the `Page View` architecture, such a scenario is easy to implement:

```
var StandardSidebar = Backbone.View.extend({
    // Logic for rendering the standard sidebar
});Page = Backbone.View.extend({
    sideBarClass:  StandardSidebar,

    render: function() {
        // render the base page HTML
        this.$el.html('<div id="sidebar"></div>' +
                      '<div id="content"></div>');

        // render the sidebar
        var sidebar = new this.sideBarClass({
            el: this.$('#sidebar')
        });
        sidebar.render();

        // logic for rendering the content area would go here
        return this;
    }
```

```
});
var AlternateSidebar  = Backbone.View.extend({
    // Logic for rendering the alternate sidebar
});
var AlternateSidebarPage = Page.extend({
   sidebarClass: AlternateSidebar
});
```

As you can see in the preceding example, our `AlternateSidebar` and `AlternateSidebarPage` classes are very short. Thanks to the power of inheritance, they don't need to redefine any of your existing logic and can instead focus entirely on what makes them different: the logic for rendering the alternate sidebar. While your site may not even have a sidebar, it is no doubt made up of composable parts, and by breaking these parts into separate `View` classes, your routes can simply use the appropriate class for whichever `View` they wish to render.

Summary

In this chapter, we explored Backbone's `Router` class. You learned how Backbone simulates pages by creating `routes` on a `Router` class and how `Routers` can operate using either hash-based or pushState-based routing. You also learned about the three different ways to add routes (via a `routes` option, a `routes` property, or the `route` method) and the three types of routes (simple, route strings, and regular expressions). Finally, you saw how to handle missing routes, respond to routing events, use multiple `Routers`, and most importantly how to combine a page view with composable sub-`Views` to power your routing methods.

In the next chapter, we'll take a look at some more advanced uses of Backbone, such as using methods in place of Backbone properties, such as `model`, or mixing sets of methods into multiple classes.

7
Fitting Square Pegs in Round Holes – Advanced Backbone Techniques

In this chapter, we will look at various advanced uses of Backbone, including the following:

- Using methods in place of properties
- Overriding a class's constructor
- Using mixins to share logic between classes
- Implementing the publish/subscribe pattern
- Wrapping widgets from other libraries in Backbone `Views`

Taking it up a notch

In the previous chapters, we've discussed the basics of using Backbone's four classes, and while these basics are more than enough to make you productive with Backbone, we've deliberately left out a few of the more complicated details. In this chapter, we'll explore those left-out details and how they relate to certain advanced Backbone techniques.

Before we proceed, however, it is important to note that all of the techniques discussed in this chapter are intended to be used to solve specific problems, and not as general-purpose patterns. While there is certainly nothing "wrong" with using any of these techniques, they will make your code more complex and thus, less intuitive and harder to maintain. Thus, it is recommended that you avoid using any of the techniques discussed in this chapter unless you specifically need to do so to solve a particular problem.

Methods in place of properties

Many of the properties of Backbone classes have the option of being defined as functions instead of as primitives. In fact, we've already seen one instance of this behavior in the `url` property of `Models` and `Collections`. As we learned in *Chapter 3, Accessing Server Data with Models*, this property can be a simple string, but if you have a more complex URL that requires logic to generate, you can instead use a function that returns a string.

This same approach can be used with a number of Backbone properties, including the following:

- For Models:
  - `defaults`
  - `url`
  - `urlRoot`
- For Views:
  - `attributes`
  - `className`
  - `events`
  - `id`
  - `tagName`
- For Routers:
  - `routes`

For instance, let's say you wanted to have `View` that can generate either `<input>` or `<select>` based on whether or not it has `Collection`. By using a function in place of a string for our `Views tagName` property of the `Views`, we can do exactly that:

```
var VariableTagView = Backbone.View.extend({
    tagName: function() {
        if (this.collection) {
            return 'select';
        } else {
            return 'input';
        }
    }
});
```

Collection.model as a factory method

The `Collection` class also has a property that can be replaced with a function, but unlike the other properties, this one isn't normally a primitive… it's a Backbone `Model` subclass. As discussed in *Chapter 4*, *Organizing Models with Collections*, the *model* property of a `Collection` class determines what type of `Model` will be created by that `Collection`. Normally, each `Collection` class can have only a single *model* property and thus, can only create a single type of `Model`.

However, there is a way around this limitation: If we replace the *model* property of our `Collection` with a function that returns new `Models`, Backbone will cheerfully overlook the fact that the `model` property isn't actually a `Model` subclass at all. For example, let's say our server has a "/book" API endpoint that returns JSON for two types of books: fiction and nonfiction. Let's also say that we have two different `Model` classes, one for each type of book. If we wanted to be able to have a `Collection` class that fetches the JSON for both types of books from a single end point but uses the appropriate `model` function to instantiate each type of book, we could use a factory "model" function, as follows:

```
var FictionBook = Backbone.Model.extend({
    // insert logic for fiction books here});
var NonFictionBook = Backbone.Model.extend({
    // insert logic for non-fiction books here});
var FictionAndNonFictionBooks = Backbone.Collection.extend({
    model: function(attributes, options) {
        if (attributes.isFiction) {
            return new FictionBook(attributes, options);
        } else {
            return new NonFictionBook(attributes, options);
        }
    },
    url: '/book'});
```

Overriding a class constructor

Whenever a Backbone class is instantiated, it runs the code defined in its `initialize` method. However, this code isn't run until after the new object has been instantiated. This means that even if you define an `initialize` method for a `Model` class, the attributes of that `Model` class will already have been set before your `initialize` code is even called.

Normally, this is a good thing, as it is convenient to have things like the attributes of the `Model` class already set, but sometimes, we want to take control before this happens. For example, let's reimagine our previous scenario of fiction and nonfiction books, only this time instead of having a single `Collection` class that can create both types of books, we want a `Collection` that only creates one type and we want this type decided by the first book that we give the `Collection` class when its instantiated.

In other words, if the first book given to our `Collection` class has the `isFiction` property, we want our `Collection` class to have the `Models` property of `FictionBook`; otherwise, we want it to have the `Models` property of `NonFictionBook`. If we do this inside an `initialize` method, the "model" option will already have been set on the `Collection` class before the `initialize` method is run, and any `Model` attributes that we pass in it will already have been converted into `Models`. Therefore, we need to add logic that runs before `initialize`. We can do this by overriding the constructor of the `Collection Constructor` method, as follows:

```
var FictionOrNonFictionBooks = Backbone.Collection.extend({
    constructor: function(models, options) {
        if (models[0].isFiction) {
            options.model = FictionBook;
        } else {
            options.model = NonFictionBook;
        }
        return Backbone.Collection.apply(models, options);
    }
});
```

Notice how we still called the original `Collection Constructor` method from within our overridden version; if we hadn't done so, none of Backbone's normal logic would run and we wouldn't even have a proper `Collection` class when we're done.

As with the other techniques discussed in this chapter, it is not recommended that you override constructors very often. If at all possible you should override the `initialize` method instead. Constructor overriding should mainly be left to cases where you want to preprocess the arguments that go into a Backbone object before that object actually gets created.

Class mixins

Backbone provides a very powerful class system, but sometimes, even this class system isn't enough. For instance, let's say you have several classes that all contain several identical methods. The popular **Don't Repeat Yourself** (**DRY)** programming principle suggests that you should eliminate the duplicate versions of the methods and instead, define them all together in just one place in your code.

Normally, you could do so by refactoring these methods into a common parent class of these classes. But what if these classes can't share a common parent class? For instance, what if one of these classes is a `Model` class and the other a `Collection` class?

In this case, you need to rely on something called a mixin. A mixin is just an object that holds one or more methods (or even primitive properties) that you want to share between several classes. For example, if we wanted to share a couple of logging-related methods between several classes, we could create a mixin of these methods, as follows:

```
var loggingMixin = {
    startLogging: function() {
        Logger.startLogging(this);
    },
    stopLogging: function() {
        Logger.stopLogging(this);
    }
}
```

Once you've defined your mixin, the next step is to "mix it in" to the definition of your class. This means modifying the `prototype` of the class to include the methods from our mixin. For instance, let's say we have a book `Model`:

```
var Book = Backbone.Model.extend({
    defaults: {currentPage: 1},

    read: function() {
        // TODO: add logic to read a book here
    }
});
```

Because Backbone's syntax makes creating our new class so simple, it's easy to miss the fact that we're actually defining a prototype object here, but if we change things just a little, it suddenly becomes more obvious:

```
var bookPrototype = {
    defaults: {currentPage: 1},

    read: function() {
        // TODO: add logic to read a book here
    }
};Book = Backbone.Model.extend(bookPrototype);
```

Once we've separated the Book prototype, we can augment it with our mixin. We could do this manually:

```
bookPrototype.startLogging = loggingMixin.startLogging;
bookPrototype.stopLogging = loggingMixin.stopLogging;
```

However, this won't work well if we have a lot of methods to mixin, or if our mixin later gets new methods added. Instead, we can use Underscore's `extend` function to extend all of the methods in our mixin on to the `Book` prototype in a single command: `_.extend(bookPrototype, loggingMixin);`

With just this one line, we've added a whole set of logging-related methods to our `Book` class. Plus, we can now share these methods with any other class, again by using only a single line, all without having to change the parent class of any of the classes involved.

Of course, it's important to note that because we are applying our mixin on top of the existing prototype, it will overwrite any properties that it has in common with the prototype. In other words, assume that our `Book` class already has its own `startLogging` method that prevents logging:

```
bookPrototype = {
    defaults: {currentPage: 1},

    read: function() {
        // TODO: add logic to read a book here
    },
    startLogging: $.noop // don't log this class
};
```

We will erase this method by applying our mixin, giving us logging on our class even though we don't want it. Further, even if `Book` doesn't have such a method, a future developer might add one and then, be confused when it doesn't work. If the future developer doesn't see (or doesn't understand) the mixin line, he might spend hours looking through the parent classes of `Book`, trying to figure out what's happening.

Because of this issue, it is usually better to rely on standard object-oriented programming as much as possible when designing your classes, and only use mixins when you can't share methods through a normal class hierarchy. In these cases, however, mixins can be a powerful tool and serve as yet another example of what is possible in JavaScript but not in most other languages (Good luck mixing methods into a Java class!).

Publish/subscribe

Backbone classes often wind up being tightly coupled together. For instance, a `View` class might listen to a `Model` class for changes in its data, and then, when this data changes, it might look at the `attributes` of the `Model` to determine what to render. This practice couples the `View` class to the `Model` class, which normally is a good thing as it lets you define the exact relationship you need between the two classes, while still keeping your code fairly simple and maintainable.

When you only have a few `Models` and `Views`, it's easy enough to manage their relationships in this way. However, if you are building a particularly complex user interface, then this same coupling can instead become a hindrance. Imagine having a single page with a large number of `Collections`, `Models`, and `Views`, all listening and responding to changes in one another. Whenever a single change occurs, it can cause a ripple effect, resulting in further changes, which can then result in still further changes, and these changes can … well, you get the idea! This can make it nearly impossible to understand what changes are occurring and what effect they will have; in the worst case scenario, they can create infinite loops or make it difficult to fix bugs.

The solution for a complex system like this is to decouple the various components (`Collections`, `Models`, and `Views`) involved with each other by using the Publish/Subscribe pattern (or pub/sub for short). In this pattern, all of the components still communicate via events, but instead of each `Collection` and `Model` class having its own event bus, all of the `Collection` and `Model` classes involved share a single, global event bus, eliminating direct connections between them. Whenever one component wants to communicate with another, it publishes (triggers) an event that the other component has subscribed (listened) to.

To implement this pattern in Backbone, we need a global event bus, and it turns out that Backbone already includes one for us: the Backbone object itself. Like `Models` and `Collections`, the Backbone object has both an "on" (subscription) method and a "trigger" (publishing) method. Both these methods work the same way as they do on other Backbone objects.

One type of application that often benefits from the pub/sub pattern is games, because they often have many different UI pieces being updated by a variety of different data sources. Let's imagine that we're building a three-player game where each player is represented by a `Model` class. Since we want to keep the player `Models` updated with changes from the server, we fetch those `Model` at periodic intervals:

```
var Player = Backbone.Model.extend();
var bob = new Player({name: 'Robert', score: 2});
var jose = new Player({name: 'Jose', score: 7});
```

```
var sep = new Player({name: 'Sepehr', score: 4});
window.setInterval(1000, function() {
    bob.fetch();
    jose.fetch();
    sep.fetch();
});
```

Let's further imagine that we have a scoreboard `View`, which needs to update itself in response to changes in the players' scores. Now, of course, in the real world, if we just had a single `View`, we most likely wouldn't even need the pub/sub pattern, but let's keep this example as simple as possible. To avoid coupling our `View` to our player `Models` directly we'll instead have it listen for a `scoreChange` event on the Backbone object:

```
var Scoreboard = Backbone.View.extend({
    renderScore: function(player, score) {
        this.$('input[name="' + player + '"]').html(score);
    });
});
var scoreboard = new Scoreboard();
Backbone.on('scoreChange', scoreboard.renderScore, scoreboard);
```

Now, in order to have these `scoreChange` events occur, we'll need to make our `Player` class trigger them. We'll also need a way to tell if a score has changed, but luckily for us, each of the `Model` classes has a `hasChanged` method, which does exactly that. We can use this method, inside an overwritten `fetch` method, to trigger our `scoreChange` event as follows:

```
Player = Backbone.Model.extend({
    fetch: function() {
        var promise = Backbone.Model.prototype.fetch.apply(
                        this, arguments
                      );
        promise.done(function() {
            if (this.hasChanged('score')) {
                Backbone.trigger('scoreChange', this.get('name'),
                                 this.get('score'));
            }
        });
        return promise;
    }
});
```

As you can see, the `Model` class and the `View` are now completely separate. The `Model` class passes any information that the `View` requires through the event itself, when it makes the `Backbone.trigger` call; otherwise, all classes involved remain oblivious to one another.

Of course, as with any of these advanced techniques, there are also disadvantages to this approach, primarily the lack of encapsulation. When you have a `View` fetch a `Model` class directly, you know exactly what code connects the two, and if you want to refactor or otherwise change that code, it's easy to determine what will be affected. However, with the pub/sub pattern, it can be much more difficult to determine what code will be impacted by a potential change.

In many ways, this is very similar to the trade-off between using global variables and using local variables: While the former are much more powerful and easy to work with, initially, their very lack of limits can make them much harder to work with in the long run. Therefore, as much as possible, you should avoid relying on the pub/sub pattern and instead, rely on event listeners bound to particular Backbone objects. However, when you do have a problem that requires communication between many disparate parts of your code, relying on this pattern and listening for/triggering events on the Backbone object itself can allow for a much more elegant solution.

Wrapping widgets of other libraries

Backbone is an incredibly powerful framework, but it can't do everything. If you want to add a calendar widget, a rich text editor, or a node tree to your site, you'll probably want to use another library, which can provide such a component for you (for example, jQuery UI, TinyMCE, or jsTree). However, just because you want to use a tool other than Backbone, it doesn't mean you have to give up all of the convenience and power of Backbone classes. In fact, there are a number of benefits of creating a Backbone `View` class that wraps your third-party component.

First, wrapping the component in a `View` allows you to specify a common way of using this component. For instance, let's say you wanted to use the JQuery UI calendar widget (or `datepicker`) in your site. In jQuery UI, if you want your calendar to include a month-picking control, you have to supply a `changeMonth: true` option in every place in your code that creates a calendar:

```
// File #1
$('#datepicker1').datepicker({changeMonth: true});
// File #2
$('#datepicker2').datepicker({changeMonth: true});
```

However, assume that you create a `View` class that wraps the `datepicker` widget as follows:

```
var CalendarView = Backbone.View.extend({
    initialize: function() {
        this.$el.datepicker({changeMonth: true});
    }
});
```

You can use this class anywhere in your code that needs a calendar, and you won't have to worry about forgetting to provide the `changeMonth` option:

```
// File #1
new CalendarView({el: '#datepicker1'});
// File #2
new CalendarView({el: '#datepicker2'});
```

If a completely different developer on your team wants to add a calendar, even if he knows nothing about jQuery UI or the `changeMonth` option, he'll still be able to create a calendar with the appropriate options for your site just by using the `View` you've created. This ability to encapsulate all logic related to a particular component and define your own interface for using it is another major benefit of this wrapping approach.

Yet another benefit is the ability to easily make changes to a component. For instance, let's say one day, we decide that we want all the calendars on our site to also have a year-picking drop-down (in other words, we want them all to have a `changeYear: true` option). Without a wrapping Backbone `View`, we'd have to find every place on our site that uses the `datepicker` widget and change its options, but with our `View`, we only have a single place in the code to update.

Of course, there are disadvantages to this technique as well. One obvious one is that if we add any components to our site that don't use the same options as the rest, such as a `datepicker` widget without a month selector, we'll need to refactor our wrapping `View` to allow for such a possibility.

Another problem is that wrapped components can remove themselves from the DOM without calling remove on their wrapping `View`, preventing this `View` from being garbage collected. Normally, this won't be a problem, as a few extra "zombie" `Views` widgets in the memory won't meaningfully impact performance. However, if you have a significant number of these wrapped widgets, then you may want to use the widget's event system to ensure that the wrapping `View` always has `remove` called on it when its wrapped widget leaves the page. For instance:

```
CalendarView = Backbone.View.extend({
    initialize: function() {
        this.$el.datepicker({
            changeMonth: true,
                         onClose: _.bind(function() {
                             this.remove();
                         }, this)
        });
    }
});
```

Finally, while hiding the details of a widget's usage inside a `View` may be convenient, it can also inadvertently hide relevant details about this widget from your fellow developers. For instance, a widget might have a certain side effect, but because other developers just use the widget's wrapping `View` and never read the original widget's documentation, they won't be aware of this side effect.

For this reason, it is important to ensure that such `Views` are properly documented. If the wrapped component has any sort of side effect, performance or otherwise, this side effect should also be carefully documented on the `View` itself or you should ensure that all developers on the team understand the workings of the underlying component.

Summary

In this chapter, we explored several advanced techniques for using Backbone. First, we learned how many of Backbone's properties can be replaced with methods, and in particular, how the `model` property of a `Collection` class can be used in this way to generate different types of `Models`. We then learned how to override a class `constructor` to gain access to and/or manipulate its arguments before `initialize`, and how to use mixins to share methods between otherwise unrelated classes. Finally, we examined how to use the Backbone object itself to implement the pub/sub pattern, and how Backbone `Views` can be used to "wrap" components from other JavaScript libraries.

In the next chapter, we'll look at some of the performance implications of Backbone and how you can avoid the most common performance pitfalls.

8
Scaling Up – Ensuring Performance in Complex Applications

In this chapter, we will look at the most common performance issues in Backbone, as well as how to avoid them. In particular, we'll cover the following:

- CPU-based performance issues
- Bandwidth-based performance issues related to content size
- Bandwidth-based performance issues related to the number of requests
- Memory-based performance issues

Backbone and performance

At its core, Backbone is just a JavaScript library, and as such, it doesn't add any new performance challenges that weren't already in JavaScript to begin with. In fact, the creators of Backbone have taken great care to make the library perform well, and in performance comparisons with rival libraries, Backbone typically comes out ahead, if not on top.

However, while Backbone itself doesn't create performance issues, it does enable entirely new ways of creating web applications, and because such applications can be far more complex than traditional websites, a whole new realm of potential performance issues is exposed. In this chapter, we will explore these issues, as well as their underlying technical details, and address ways to avoid or mitigate them.

Causes of performance issues

Whenever a user experiences performance problems, it is because he has exceeded the capacity of one of his system resources, either bandwidth memory or processing power. Before debugging any performance issue, it's essential to understand which of these factors is responsible, and this can be determined in one of two ways. First, a profiling tool (such as the tools included with all major browsers) can be used to measure how much of each resource is being used, which should quickly make obvious which resource is being used excessively. An explanation of these tools falls outside the scope of this book, but I strongly encourage you to familiarize yourself with the tools available in your favorite browser.

For most problems, however, a profiling tool won't even be necessary, because their source can be determined by how they manifest. Bandwidth problems only occur when retrieving or sending data from your server (and in most applications, only retrieval operations involve enough bandwidth to be problematic). If they occur at load time, then they could be caused by a significant number of large static resources, such as images, but if they occur afterwards, they are far more likely to be the result of AJAX calls. In Backbone applications, this means *fetch* operations (or, rarely, *save* or *destroy* operations) from `Models` or `Collections`.

CPU performance issues only occur when the user's computer is forced to think hard about something you are making it do. For example, a series of nested `for` loops, or the rendering of a complex visualization such as a chart can cause such performance issues. This type of performance issue is usually easy to identify because it only occurs when the user triggers such computationally intensive code.

The final, and by far the most difficult, source of performance issues is the memory. Unlike the other two issues, which usually have obvious triggers such as the start of an AJAX operation or the rendering of a chart, memory issues can occur without any clear or obvious source. In fact, memory issues can begin seconds or even minutes before the user actually starts noticing problems, forcing you to trace back through all the code they hit to try and find a cause. Because memory issues are the most common type, and because they are the hardest to understand and resolve, we will be focusing most heavily on them in this chapter. However, before we do so, let's examine the other two sources, and some common sense approaches to avoiding them.

CPU-related performance issues

As mentioned earlier, Backbone itself will not usually be the source of CPU-related performance issues, because these issues tend to be caused by specific components rather than the overall site architecture. However, there is one way that Backbone can contribute to such problems, and that is by making it easy to repeat the same work in multiple places. For instance, let's imagine that you are creating a dashboard page that will use one main `View` class to show your user various pieces of data, and a number of child `Views` to render each of these pieces of data. In addition, let's imagine that each of these pieces of data will update periodically. Normally, you would tie the updating of that data to an AJAX response or a user event, but under certain circumstances, you might instead want to use a `setInterval` statement. For instance, `onScroll` events are known to be problematic, so many developers avoid them and instead rely on `setInterval` to check for scrolling periodically.

This approach will work fine as long as there is only a single `setInterval` event running, but what if you instead decide to create a separate `setInterval` event for each child `View`? With only a few child `Views`, this still might work, but eventually, too many such intervals will become a drain on the user's CPU, causing performance problems. In the worst case scenario, while your development machine will be able to handle the page, your user's (less powerful) machine might not, causing the user to report bugs that you can't reproduce.

The solution in such cases is straightforward: Don't repeat processing-intensive tasks unnecessarily. In the preceding example, instead of having each child `View` update in response to its own `setInterval` event, you could start only a single `setIonterval` process in the main `View` and then, have it trigger updates in your child `Views` (possibly by using the pub/sub pattern described in the previous chapter).

Event delegation

One other way developers can easily create unnecessary strain on the user's browser is by creating too many event handlers. For instance, let's say you want to create a large table (perhaps 20 rows × 20 columns), so you create a parent `View` for the table and a large number of child `Views` for each cell. So far so good! Now let's say, you add a `click` event handler to each of these child `Views`. Without realizing it, you just created 400 event handlers. If you add another event handler, such as a `change` handler for `<input>` elements inside the cell, you add another 400, and so on.

Given enough `Views` and enough event handlers, this can eventually create a performance issue, but luckily, JavaScript comes with a built-in mechanism that we can use to solve this problem: event bubbling. Whenever an event occurs in the DOM, it first triggers event handlers on the relevant DOM element and then bubbles up to each successive parent of that element. In other words, if a `click` event occurs on a `<td>` element, the browser will resolve any event handlers bound to that `<td>` element first, then (unless one of the event handlers returned false) it will call the handlers on the `<td>` element's parent `<tr>` element and then that `<tr>` parent's `<table>`:

```
$(document.body).append('<table><tr><td></td></tr></table>');
$('td').on('click', function(e) {
    e.target; // will be the td element
    e.currentTarget; // will also be the td element
});
$('tr').on('click', function(e) {
    e.target; // will still be the td element
    e.currentTarget; // will be the tr element
});
$('table').on('click', function(e) {
    e.target; // will still be the td element
    e.currentTarget; // will be the table element
});
$('td').click(); // first the td handler will trigger, then the tr
                 // handler, and finally the table handler
```

We can take advantage of this fact and change our event binding strategy to improve performance by binding our events to the parent `table` element's `View` rather than to each child `View`. This parent `View` event handler can then trigger the appropriate logic on the relevant child `View` by using the event's `target` property to determine which child `View` caused the event. While this approach requires slightly more work, it allows us to reduce our 400 click event handlers down to a single event handler, and on particularly complex pages (such as our hypothetical table page), the use of such event delegation can significantly reduce the strain on the browser.

Bandwidth-related performance issues

While it's true that the typical user's bandwidth has grown significantly in the recent years, the bandwidth nevertheless remains a constant issue for web developers. However, many developers don't realize that there are actually two main sources of bandwidth problems. The first of these is fairly obvious: forcing your users to download files that are too large. But, there is also a second, less obvious source: forcing your users to download too many files at once.

Downloading excessively large files

Let's start with the obvious source first. If the files that your users need to download are too large, it doesn't matter whether they are images, videos, or JavaScript code files: Your site is going to load slowly. However, you can make a big difference in the size of any file you use by enabling compression at the web server level. On an Apache web server, this can be done by using `mod_deflate`, and most other web servers have similar options. Doing so will make your server compress the files that it sends to your users in such a way that your users' browser can easily decompress them … all without the user even knowing that any decompression is going on.

However, if turning on compression doesn't help enough, then your next steps depend on the file type. If your issues come from images or videos, then you simply have to find a way to use smaller files, for instance, by lowering their resolution. However, if JavaScript files are your main concern, there is another option: using a minification program.

Minification programs parse your code to create a new optimized version of it that eliminates comments, removes extra whitespace, and renames variables with shorter names. The only downside to using such a program is that it will make it harder for you to debug problems on your production servers, which shouldn't be an issue as long as you have a matching development environment where you don't minify your files. Further, if the minification truly becomes a problem, you can always temporarily switch your server back to the unminified files to do your debugging.

Together the two techniques of web server compression and minification can result in a major difference in file size. For example, the uncompressed jQuery library (version 1.11.0) is 276 KB, while the zipped version is only 82 KB and the zipped version of the minified jQuery code is only 33 KB. In other words, just by using these two techniques, it's possible to reduce jQuery's footprint by almost a factor of ten!

Downloading excessive number of files

Unfortunately though, even if you reduce the bandwidth of your code files and assets, there is also another, more subtle bandwidth issue to worry about that has nothing to do with the number of bytes your user downloads. To understand this issue, you have to understand how browsers handle requests for data, both those that come from the DOM (such as the `<link>` and `<script>` tags) and AJAX requests.

When a browser opens up a connection to a particular remote computer, it keeps track of how many other connections have already been opened to that computer's domain, and if too many are already open, it pauses until one of the previous requests completes. The exact number of connections that can occur before this happens varies by browser: In Internet Explorer 7, it's only two, but in most modern browsers, it's six or eight. Because of this limit, and because each request, no matter how small, has a certain minimum amount of time that it will take (also called latency), the actual amount of bandwidth used can be irrelevant. There are two main approaches for solving a bandwidth issue.

The first is, obviously, to make fewer requests. If your problem is too many `Models` being fetched at once, `Collections` can be very helpful in solving it; instead of fetching each `Model` class individually, simply create an endpoint on your server that can return all of the `Models` at once and then, use a `Collection` class to fetch them. Even though you will be downloading the same amount of data, this change in API will result in significantly less requests. Similarly, if your problem is too many images, you can combine all of the images into a single `sprites` file and then, use CSS to only display one image at a time.

The other option, if your application truly does require a large number of requests, is to use subdomains. When a browser counts how many connections it has outstanding, it doesn't just look at the source's domain but also at its subdomain. This means that you can fetch the maximum number of requests from `http://example.com/` and then, fetch that same number of requests from `foo.example.com`, `bar.example.com`, and so on. This trick is often used to serve CSS and images more quickly, but it can just as easily be used to make a large number of simultaneous fetches (as long as you update your `url` methods appropriately to fetch from the correct subdomain).

Finally, there is one last solution to bandwidth issues, which doesn't really solve these issues so much as make them more palatable to the user. If you know that you're going to be making a request that will take long enough for the user to notice, you can give the user a visual wait indicator, such as adding an animated spinner image or changing the cursor's CSS property to wait. If you make these changes just before you start a `fetch` operation, you can then use that operation's success and failure callbacks (or, if you use the deferred style, a single `complete` callback) to undo the changes. While this won't make your data download any faster, it will make a difference in your user's experience.

Memory-related performance issues

Memory-related issues are the hardest to debug and solve, and unfortunately, they are also the most likely to be encountered when you first start using Backbone. Again, this is not because Backbone itself has memory issues, but because the possibilities that Backbone enables can allow developers to shoot themselves in the foot if they're not careful.

However, before we proceed, it's important to first explain just how browsers manage memory. As you probably already know, the memory in JavaScript is managed by the browser, not the developer, using something called **garbage collector**. What this means is that you don't have to tell the browser I'm done using this variable. Instead, you can simply stop using that variable and the browser will figure out that it has become `garbage`, which will usually make it clean that variable up automatically.

The problem is that the garbage collector operates on a very simple interpretation of what is or is not garbage. In essence, any variable that is not referenced by another variable is considered to be garbage. For instance:

```
var bookReference = {
    fakeBook: new Book({title: 'Hamlet, Part 2: The Reckoning'})
};
delete bookReference.fakeBook;
// fakeBook has no references, and will be collected as "garbage"
```

The problem is that developers often don't realize when they leave behind references to a variable, and as such, they force the browser to keep using its memory even though the programmer considers it garbage. Let's look at an example:

```
var exampleModel = new Backbone.Model();
var exampleView = new Backbone.View();
exampleModel.on('change', exampleView.render);
$(document.body).append(exampleView.el);
exampleView.remove();
// exampleView will NOT be garbage collected
```

In this example, it seems like we eliminated all references to `exampleView` when we called `exampleView.remove()` and removed it from the DOM, but in fact, there was still one reference left behind, hidden inside `exampleModel`. This reference was created when we called the `on` method of `exampleModel` and passed it `exampleView.render`. By doing so, we told the `Model` to wait until a change happens and then call `exampleView.render`, which required it to store a reference to `exampleView.render`. Since we didn't delete `exampleModel`, this reference remains and won't be garbage-collected, leaving a so-called zombie `View` in the browser's memory.

One way to solve this problem would be to remove this reference manually by using the `off` method:

```
example.model.off('change');
```

However, having to manage such references can quickly become tedious. Luckily, the creators of Backbone added a method to `View` (as well as the other three Backbone classes) that helps solve this problem, called `listenTo`. This method works very similarly to `on` with two important differences. First, it is called on the listening object (in this case, the `View`), rather than on the object being listened to (in this case, the `Model`), and second, it does not take a context argument. Instead, the context of the callback will always be set to the object that `listenTo` was called on.

Just as there is an `off` method for `on`, there is a `stopListening` method that removes listeners created by `listenTo`. However, you won't need to call `stopListening` yourself very often, because it's called automatically as part of the `remove` method of a `View`, which is what makes it so convenient.

Let's retry our last example using `listenTo`:

```
exampleModel = new Backbone.Model();exampleView = new
Backbone.View();
exampleModel.listenTo('change', exampleView.render);
$(document.body).append(exampleView.el);
exampleView.remove();
// exampleView WILL be garbage collected
```

This time, just as before, we have a `View` listening for changes in a `Model`. However, because we used `listenTo` instead of `on`, the reference created as a side effect will get removed whenever `stopListening` is called. Since we called `remove` on `exampleView` and since this method automatically calls `stopListening` for us, our `View` gets garbage-collected correctly without us having to do any extra work.

Unfortunately, however, `listenTo` can't solve all potential leaky references. For one thing, you may still want to use the `on` method from time to time. The primary reason for doing so is to listen for events from non-Backbone code, such as a jQuery UI widget. You might also be tempted to use `on` because (unlike `listenTo`) it takes a context argument, but thanks to Underscore's `bind` method, you don't need to do so; you can simply bind your desired context in your callback function before passing it to `listenTo`. However, even if you do avoid using `on` entirely, you still have to remember to call `remove` on your `View`. If you don't, you still need to use `off` or `stopListening` to clear the event binding references. Finally, event handlers aren't the only source of references.

For instance, parent `Views` and child `Views` often reference each other, and unless you delete the referencing `Views` entirely, the `Views` that it references won't actually be garbage-collected. The good news is that there's no need to worry too much about such references on a small scale, and in fact, trying to optimize performance too heavily on every last bit of code in your application can wind up being counterproductive. Any given `Model` or `View` will normally take up only a small amount of memory on its own, so even if you do create a leaky reference that prevents it from being garbage-collected, the actual effect on your application's performance will be minimal. If the user never even notices the leak and then reclaims the memory when he closes his browser or hits refresh, then clearly there was no need for you to have spent time worrying about it.

Instead, you mainly want to focus on managing your references when dealing with large numbers of objects. If you are designing a page `View` that will be used throughout your application, or creating a `View` for a large table with many separate child `Views`, then you will likely want to be extra careful with each reference you create and ensure that all these references get cleaned up when you are done with them.

Summary

In this chapter, we learned how JavaScript code in general, and Backbone code in particular, can cause performance problems. We learned the three main causes of such problems (namely bandwidth, CPU, and memory), as well as techniques and methods that can be used to solve them. In particular, we learned how leaky event bindings or other references can prevent garbage collection, and how using `listenTo` or manually cleaning up references can enable garbage collection to work as expected.

In the next chapter, we'll examine the benefits of proper code documentation, learn how to solve some of the Backbone-specific documentation challenges, and consider which of the many quality documentation tools are the best for documenting your projects.

9
What Was I Thinking? Documenting Backbone Code

In this chapter, we will consider various options for documenting your Backbone code, including the following:

- "Non-documentation" approach to documentation
- How to explicitly document your code without a formal structure
- How to explicitly document your code using a formal structure such as JSDoc
- How to use Docco, an alternate documentation tool from the creator of Backbone'

Backbone and documentation

Documentation is important in any software project, but Backbone projects have some unique considerations when it comes to documentation. However, before we delve into these considerations, it's important to identify why you are documenting your code in the first place, and what strategy you plan to use to document it.

Many developers have the false impression that documentation is for other developers, when in fact, nothing could be further from the truth. Most software isn't written just once but is instead iterated on over time. Each new revision can occur days, months, or even years after the previous one, and when you go back to modify code you wrote months/years ago, the code can almost look as if it was written by another person.

Because of this, even if you are the only developer on your team, and even if you know exactly how every last line in your code base works at the moment, it is important to document your code, if only for the benefit of your future self. Further, if you aren't the only developer on your team, documentation becomes all the more important, as it can also serve as a bridge between you and your coworkers.

Documentation approaches

There are three main approaches to documenting JavaScript code, none of which are a one-size-fits-all solution. You will need to determine which approach makes the most sense for you and your team on the basis of your team size and the ambitions of your project.

The three main approaches can be referred to as the non-documentation, simple documentation, and robust documentation approaches.

In general, the larger your team is, and the larger the organization to which this team belongs is, the more likely are you to want robust documentation, although size isn't the only factor to consider here. Other factors include your team members' desire for external documentation and whether or not any of your code is customer-facing (for instance, if you offer a public API for your customers to extend your application).

The non-documentation approach

First, let me clarify that the non-documentation approach doesn't mean avoiding documentation entirely. Rather, it relies on using forms of documentation other than explicit code comments.

For instance, consider the following line of code:

```
var Book = Backbone.Model.extend();
```

Now, if we had wanted to, we could have written a comment describing this line of code, as follows:

```
/**
 * This defines a book model.
 */Book = Backbone.Model.extend();
```

However, adding such a comment doesn't really tell us anything we don't already know, because our choice of variable name (`Book`) already tells us what the class is. Simply by choosing a descriptive variable name for our class, we have documented what it does, without the need of supplemental documentation.

However, the names of class variables aren't the only important names. Function names, and particularly, method names, can also be very helpful in explaining what the function/method does. Consider the following:

```
        bookNav: function() {
            router.navigate('bookPage', {silent: true, trigger:
              true});
        }
    vs.:
        navigateSilentlyToBookPage: function() {
            router.navigate('bookPage', {silent: true, trigger:
              true});
        }
```

Just by typing a few extra characters we've eliminated the need to read the method's definition in order to understand it. Further, while reading, the definition might not seem so hard in this particular example; however, if the function were longer or more complex, it might not be so easy.

One other way that functions can be used to convey documentation is by breaking them up into separate functions. For instance, let's say we had a `View` method that incremented a counter, saved a `Model`, updated and then rendered another `View` method, and finally changed the URL, as follows:

```
example: function(router) {
    this.model.set('counter', this.model.get('counter') + 1);
    this.model.save();

    this.siblingView.model = this.model;
    this.siblingView.render();

    var url = this.url();
    router.navigate(url, {silent: true, trigger: false});
}
```

Naming such a method would be difficult:

`"incrementCounterAndSaveModelAndUpdateAndRenderSiblingViewAndRefreshURL"` doesn't roll off the tongue very well. You could simplify it to `updateCounterAndRefresh`, but then you'd lose some of the non-documentation that the more verbose name would provide.

A better approach is to use the simpler name but separate the parts of the method into their own methods, as follows:

```
incrementCounterAndSave: function() {
    this.model.set('counter', this.model.get('counter') + 1);
    this.model.save();
},
updateAndRenderSiblingView: function() {
    this.siblingView.model = this.model;
    this.siblingView.render();
},
refreshURL: function(router) {
    var url = this.url();
    router.navigate(url, {silent: true, trigger: false});
},
updateCounterAndRefresh: function(router) {
    this.incrementAndSaveCounter();
    this.updateAndRenderSiblingView();
    this.refreshURL(router);
}
```

As you can see the contents of our `updateCounterAndRefresh` method now read almost like an English documentation string, effectively documenting what's going on without any actual documentation. Furthermore, if we wanted to unit test this code (which we'll discuss in the following chapter), it will be far easier to do so with these separate methods than it would have been originally.

Moreover, the preceding code is a relatively simple example, while your actual methods may in fact be much longer than six lines. In real-world projects, it is even more important to utilize this technique of using many (well-named) methods rather than long monolithic methods, as it will greatly improve both the code's readability and ease of testing.

Class, function, and other variable names are not the only forms of non-documentation. File names and folder structure can also provide a great deal of information if used correctly. Consider a file named `Book.js`. On its own, we have no way of knowing whether this file contains a book `Model`, a book `View`, both, or something else entirely. However, if this file were renamed "`BookView.js`", it would be obvious. Similarly, if the file retained the name "`Book.js`" but was stored inside a "`views`" folder, the contents would again be apparent, without the requirement of any additional documentation.

Benefits of non-documentation for other approaches

Before we move on to the other two documentation approaches, it is worth noting that the strategies we just described aren't only useful if you choose the non-documentation approach. In fact, they are great advice for all programmers. No matter what documentation you generate to supplement your code, at the end of the day, you and your fellow developers are going to have to work with the code itself, not the documentation. By working to keep that code as readable and instructive as possible, you provide benefits that while not the same as those offered by explicit documentation, are none the less very valuable, particularly in the long run.

The simple documentation approach

Non-documentation is incredibly important to writing maintainable code, but it does have its limits. For instance, while you can use expressive variable names to describe many variables, you can't use them to describe parts of Backbone itself. For example, when a `View` takes a `model` option, the only way to rename it more expressively would be to create an entirely new property:

```
var BookView = Backbone.View.extend({
    initialize: function() {
        this.bookModel = this.model;
    }
});
```

Now, there is nothing wrong with the above code, but in some sense, it crosses the line between *code as documentation* and *replacing documentation with code*. In cases like these, a more natural solution may instead be to simply use a comment, as follows:

```
// This View takes a ""Book"" ModelBookView =
Backbone.View.extend();
```

However, many programmers find that it's easy to miss such single-line comments, and therefore, they save them only for documenting specific lines of code. For class or method documentation, these developers rely on a specialized form of the multiline comment, which is marked with an extra leading asterisk and (optional) leading asterisks on each subsequent line:

```
/**
 * This View takes a ""Book"" Model
 */BookView = Backbone.View.extend();
```

```
When used this way throughout your code, these documentation
sections form easy-to-read alternating blocks, making it trivial
to skim through to what you're looking for without having to
actually read the code in between:BookView =
Backbone.View.extend({
    /**
     * This ""foo"" method does foo stuff
     */
    foo: function() {
        doSomeFooStuff();
    },
    /*
     * This ""bar"" method takes a ""Baz"" argument and does bar stuff
     */
    bar: function(baz) {
        doBarStuffWith(baz);
    }
});
```

Most modern code editors will color such documentation differently from the rest of the code as well, making it even easier to skim through.

By using a combination of these multiline comments for your classes and methods, and using single-line comments to explain the tricky code inside of functions, you can very effectively document what class of `Model` a `View` has or which routes use a particular `View`. Further, while writing such comments does take more time than relying on non-documentation, the few extra seconds that they take you to write could save you hours or even days of work later on.

The robust documentation approach

Either of the two preceding approaches are enough for most programmers to reap the benefits of documentation without having to spend too much time writing it. However, if you are looking for a more universal structure to your documentation, or if you and your team are not the documentation's only audience, then it might be beneficial to use an external tool such as JSDoc or Docco.

In larger organizations, it's not uncommon to have different teams working on different parts of the code. When one team needs to use a component or library that another team manages, the first team may not want to read the second team's code, or for that matter, they might not even have access to it themselves. Robust documentation can be useful in these cases by providing a way for teams to understand each other's code without having to read it directly.

Another important scenario is user customization. With the robust and powerful applications that Backbone enables, it often is only a matter of time before a customer requests a way to customize something or, if they can't write JavaScript themselves, have a consultant customize it for them. In these cases, the best solution is often to expose a subset of the code for the customer to work with, and this public customization API will require documentation so that the customer (or their contractor) can learn how to use it.

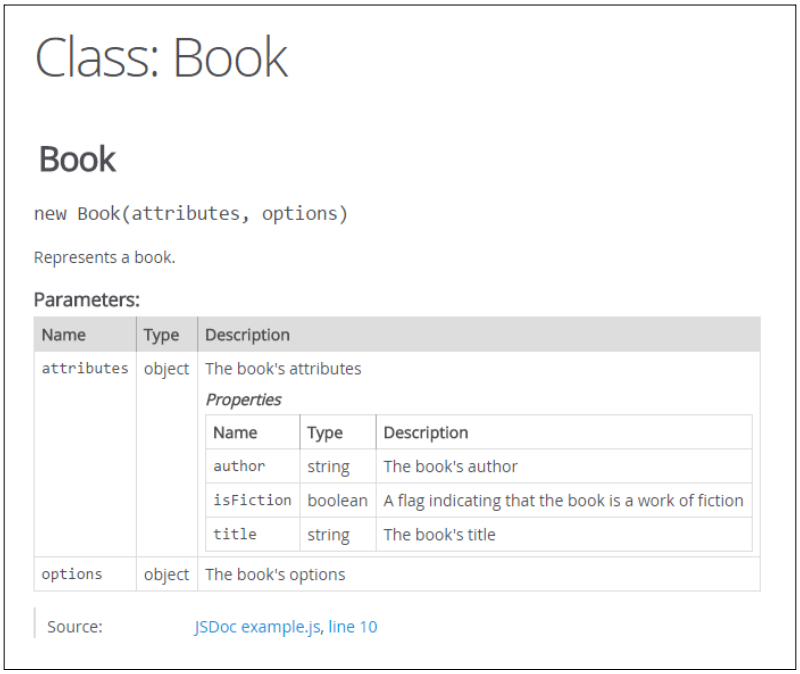
In either of these cases, a tool such as JSDoc or Docco can be used to generate separate documentation files that can be shared without sharing the code itself. However, at the same time, developers can still write their documentation along with their code; there's no need for them to maintain (for instance) a separate MS Word document.

Many teams also use the structure of JSDoc within their documentation, without ever actually using this tool to generate externally facing documentation. For these teams, the main benefit of JSDoc lies in its annotations, which let the teams write documentation in a similar manner and in a way that future team members are more likely to understand.

JSDoc

JSDoc (`http://www.usejsdoc.org`) is the oldest and most popular JavaScript documentation tool available. As far back as 1999, before the JSDoc tool even existed, developers used its syntax (borrowed from JavaDoc) to document the very early JavaScript code. The JSDoc tool itself, now in its third edition, is incorporated into several different other tools (including Google's Closure compiler), and support for its syntax can be found in almost every major code editor from Sublime to Eclipse to Visual Studio.

Using JSDoc, a developer can easily convert his inline documentation into external HTML files, as shown in the following screenshot:



Class: Book

Book

`new Book(attributes, options)`

Represents a book.

Parameters:

Name	Type	Description
`attributes`	object	The book's attributes *Properties* Name / Type / Description `author` / string / The book's author `isFiction` / boolean / A flag indicating that the book is a work of fiction `title` / string / The book's title
`options`	object	The book's options

Source: JSDoc example.js, line 10

JSDoc works by relying on annotations included in documentation strings. For instance, consider the following code:

```
/**
 * This ""bar"" method takes a ""Baz"" argument and does bar stuff
 */
bar: function(baz) {
    doBarStuffWith(baz);
}
```

Instead of using the above code, a developer using JSDoc would use an `@param` annotation, as follows:

```
/**
 * Does bar stuff
 * @param {Baz} baz this argument is used to do bar stuff
 */
bar: function(baz) {
    doBarStuffWith(baz);
}
```

The "`@param` annotation tells JSDoc that the method takes a *baz* argument and that it should be an instance of the "`Baz`" class. The full list of annotations can be found on the JSDoc website, and they are fairly straightforward, so we won't discuss all of them here. However, two of them (`@property` and `@param`) can be slightly problematic when used with the Backbone code, so let's examine how to use them properly.

Let's imagine that we want to document a Backbone `Model`. For doing so, we might write the following code:

```
/**
  * This model represents a book in our application.
  * @class
  */
var Book = Backbone.Model({ ...
```

Now, let's assume that our `Book Model` can have three different attributes: title, description, and page length. Now, the problem is how do we document them? There is no `@attribute` annotation in JSDoc since *attributes* are Backbone-specific and the `@param` annotation can seemingly only specify the argument of a single attribute; it can't (for example) tell us that two of the attributes should be strings and the others should be integers.

Luckily, one can use multiple `@param` or `@property` annotations to solve this problem (which one you use is up to you; because Backbone creates properties for both attributes and options, both are in fact parameters and properties). Consider the following:

```
/**
  * This model represents a book in our application.
  * @param {object} attributes
  * @param {string} attributes.title book's title
  * @param {string} attributes.description description of the book
  * @param {integer} attributes.pageLength number of pages
  */Book = Backbone.Model({ ...
```

If you are using JSDoc informally (that is, you don't plan to generate external documentation), you can even simplify the above slightly by leaving out the initial `@param {object}` attribute annotation.

Docco

Docco (`http://jashkenas.github.io/docco/`) is notable both because it takes a very different approach from JSDoc and because it was written by the creator of Backbone itself (Jeremy Ashkenas). Unlike JSDoc, which focuses on creating API documentation, Docco focuses on generating tutorials and/or explanations of how a given block of code works. Docco also differs from JSDoc in that it uses single-line comments, rather than multiline ones, to generate its documentation.

Here's an example of documentation generated using Docco; in fact, it was generated using Backbone's own source code (you can find the original version on the Backbone website):

As you can see from the preceding example, Docco generates documentation with two columns. On the right is the original source code being documented, minus any comments, and on the left are the comments that correspond to this source code. For example, here are the original lines from Backbone that were used to generate the preceding example:

```
//     Backbone.js 1.1.2

//     (c) 2010-2014 Jeremy Ashkenas, DocumentCloud and
Investigative Reporters & Editors
```

```
//     Backbone may be freely distributed under the MIT license.
//     For all details and documentation:
//     http://backbonejs.org

(function(root, factory) {

  // Set up Backbone appropriately for the environment. Start with
AMD.
  if (typeof define === 'function' && define.amd) {
    define(['underscore', 'jquery', 'exports'], function(_, $,
exports) {
      // Export global even in AMD case in case this script is
loaded with
      // others that may still expect a global Backbone.
      root.Backbone = factory(root, exports, _, $);
    });

  // Next for Node.js or CommonJS. jQuery may not be needed as a
module.
  } else if (typeof exports !== 'undefined') {
    var _ = require('underscore');
    factory(root, exports, _);

  // Finally, as a browser global.
  } else {
    root.Backbone = factory(root, {}, root._, (root.jQuery ||
root.Zepto || root.ender || root.$));
  }

}(this, function(root, Backbone, _, $) {

  // Initial Setup
  // -------------

  // Save the previous value of the `Backbone` variable, so that
it can be
  // restored later on, if `noConflict` is used.
  var previousBackbone = root.Backbone;
```

A key part of Docco's appeal is just how simple it is: There are no annotations or even multiline comments, just plain old `//` single-line comments. Docco is also valuable because of the type of documentation it generates: If you want to create a tutorial or walk through of your code Docco is a much better choice than JSDoc.

Ultimately which documentation system or systems you use will depend on your needs and future expectations. Even if you opt for simple documentation, it is important that you not underestimate the need to properly document your code. If you do, your future self (and possibly your coworkers) will regret it.

Of course, as with most things in programming, it is also possible to have too much documentation. When adding documentation, you should always remember that as you refactor and update your code, any documentation written for that code must similarly be updated. This certainly shouldn't stop you from adding documentation, given its many benefits, but before we move on to testing (which shares the same downside), we'd be remiss not to mention this ongoing maintenance cost.

Summary

In this chapter, we learned how to document Backbone JavaScript code and what specific areas to focus on when doing so. We explored three different documentation options—non-documentation, simple documentation, and robust documentation—and considered two popular tools for generating robust documentation: JSDoc, and Docco.

In the next chapter, we'll look at how to test your Backbone code. In particular, we'll explore the popular test running frameworks QUnit and Mocha, as well as the Sinon library for creating spies, stubs, and mocks.

10

Keeping the Bugs Out – How to Test a Backbone Application

In this chapter, we'll look at the challenges of testing a Backbone application, including the following:

- Choosing a unit testing framework
- Deciding between the BDD and the TDD testing styles
- Mocking out related components in tests
- Using Selenium to create acceptance-level tests

Testing in JavaScript?

For many years, the idea of testing JavaScript code was laughable. After all, who would write a testing suite for a bunch of form validation scripts? Moreover, even if you were doing something more interesting that could actually benefit from testing, there were no JavaScript libraries that facilitated testing, so it had to be done from scratch.

In 2008, all this changed as a wave of new automated JavaScript testing tools emerged. All of these libraries focused on unit testing or testing of a specific part of the code (usually, a single method or small class). Two of the first unit testing libraries, QUnit and Jasmine, offered dueling approaches (TDD versus BDD) for testing in the browser, while a third library, JS Test Driver, offered testing at the command line. Soon after them came a number of other unit testing libraries, including Buster.js and Mocha.js, as well as test support or `mock` libraries such as JsMockito, JSMock, and Sinon.JS.

Today, as JavaScript developers create ever more advanced web applications (particularly, the kind made possible by Backbone), testing has become absolutely essential. Testing can prevent bugs that, because of the code's complexity, would otherwise have been difficult or even impossible to find. Further, perhaps, just as importantly, a proper suite of tests allows developers to safely refactor their code, which is a frequent and necessary task for keeping a code base maintainable.

At this point, it's no exaggeration to say that if you are a building a serious web application using Backbone, a testing suite isn't just a luxury, it's a necessity.

Which library to use

Building proper testing capabilities for a modern web application requires the following three main pieces:

- A unit-testing framework: a tool to test specific units (usually functions or small classes) of code
- A mocking library: a tool that facilitates testing by creating fake versions of objects
- An acceptance testing framework: a tool for testing complete user experiences, such as logging in to your site or ordering a product

Depending on which unit-testing library you select, you may also want to download additional tools. For instance, many libraries offer alternate styles of test reports, which must be downloaded and included separately. Moreover, if you use a library that lacks the ability to run tests at the command line, you may wish to add this functionality with a headless web browser such as the very popular PhantomJS, Doing so will make it easier to automate your tests so that they can, for instance, run periodically or in response to code checks-ins. While there are several different mocking libraries to choose from (we'll explain why you want one in a moment), we recommend using the most popular, Sinon.JS. Sinon is both powerful enough and easy enough to use, and thus, it's a great fit for just about any project. Similarly, there is really only one main option for acceptance testing and that is Selenium. We'll discuss Selenium at the end of the chapter, so for now, we will focus on the choice of a unit testing library.

There are far too many high-quality testing libraries, with far too many differences in features, to possibly cover them all properly in this book. However, we do need to choose a library for the examples in this chapter, and so, we've opted to use Mocha.js. Mocha is one of the newer libraries available as well as one of the most robust and powerful. Mocha can be used either with the traditional xUnit/TDD style used by libraries such as QUnit, or with the BDD style popularized (in JavaScript at least) by Jasmine. Further, while it can be run in the browser, it can also be run (by using PhantomJS) at the command line as well.

On top of all this, Mocha offers its users the choice of four different assertion libraries. While other major testing libraries include assertions as part of the library itself (which admittedly is more convenient: there is one less file to download), Mocha prefers to give developers a choice. Because there are subtle differences possible in the way assertions are implemented, Mocha lets you select the style so that you can have the style that most appeals to you.

However, while we will be using Mocha throughout this chapter, much of what we'll be demonstrating will in fact be similar (if not identical) to how many of the other popular testing libraries work.

Getting started with Mocha

To use Mocha, you will first need to download the `mocha.css` and `mocha.js` files from the project's GitHub page (which can be found at `https://github.com/mochajs/mocha`). You will also need to download an assertion library; for this example, we'll be using `Expect.js` (which you can download from `https://github.com/Automattic/expect.js`).

Next, you will need to create a new HTML page and add script tags for all of your application's JavaScript files, as well as all library files (for instance, Backbone and jQuery). You'll also need to add a few bits of Mocha boilerplate code. All together you should wind up with an HTML file that looks something like the following:

```
<html>
<head>
<!-- External Library Files -->
<script src="/underscore.js"></script>
<script src="/jQuery.js"></script>
<script src="/Backbone.js"></script>

<!-- Application-Specific Files -->
<script src="/SomeModel.js"></script>
<script src="/SomeView.js"></script>
<script src="/SomeOtherNonBackboneCode.js"></script>

<!-- Test Library Files -->
<script src="/mocha.js"></script>
<link rel="stylesheet" type="text/css" href="/mocha.css"></link>
<script src="/expect.js"></script>

</head>
<body>
```

```
<div id="mocha"></div>
<!-- Test Code -->
<script>
mocha.setup('bdd'); // start Mocha in BDD mode
// *INSERT TESTS HERE*
mocha.run();
</script>
</body>
</html>
```

TDD versus BDD: What's the difference?

In the preceding code, we call `mocha.setup` (bdd), which starts Mocha in the BDD mode (as opposed to the TDD mode). But what does this even mean?

Test Driven Development (**TDD**) is a style of development where the developer begins all his work by writing a (failing) test case. The developer then adds just enough code to his application to make that test pass, but no more than that. As soon as the test passes, the developer writes a new (failing) test and repeats the cycle.

The advantages of TDD are two-fold. First, by always writing a test first, the developer guarantees that his code will always be completely covered by tests. Second, the TDD style naturally forces developers to write their code in a testable fashion, which means adopting practices like writing several short functions instead of one long one. As we mentioned in the previous chapter, these practices have benefits that go beyond the test environment.

However, TDD also has its downsides. Perhaps, the most significant one is that it can result in excessive, useless tests, which require significant time to maintain. Also, by its very nature, TDD focuses heavily on *how* the code does instead of what it does, but many programmers would argue that *what* the code does is far more important to test.

This distinction is particularly important when refactoring. By definition, a refactoring operation is required when you change how the code does what it does without changing what it does. Because TDD-based tests focus so heavily on the how, whenever a refactoring occurs, the tests need to be updated to match the new code.

Behavior Driven Development (**BDD**) is very similar to TDD but attempts to solve the problems related to TDD by focusing on what the code does. This is achieved largely by using a slightly more verbose syntax for organizing tests and test suites. Of course, one doesn't necessarily need a different syntax to write tests that focus on what the code does, but the benefit of the BDD syntax is that it naturally encourages such behavior.

Consider the following imaginary test of the Backbone set method of `Model`, written using Mocha's TDD syntax and the better-assert assertion library. It works by creating `testModel`, setting an attribute (`a`) with a value (`1`), and then confirming that the attribute/value pair has been added to the test attributes of `Model`:

```
suite('Model', function() {
    var testModel;
    setUp(function() {
        testModel = new Backbone.Model();
    });
    suite('set', function() {
        test('set adds a value to the model\'s attributes',
            function() {
            testModel.set('a', 1);
            assert(testModel.attributes.a === 1);
        });
    });
});
```

As you can see, in the TDD code, a test is registered using a `test` function and groups of tests are organized into suites. Also, the test verifies the validity of the code by using an `assert` statement. Now, let's look at a similar imaginary test created using the BDD syntax:

```
describe('Model', function() {
    var testModel;
    beforeEach(function() {
        testModel = new Backbone.Model();
    });
    describe('#set', function() {
        it('sets a value that "get" can retrieve', function() {
            testModel.set('a', 1);
            expect(testModel.get('a')).to.be(1);
        });
    });
});
```

Notice how similar the two tests are: While the first example uses *suite* and the second uses *describe*, both functions have the same effect of grouping tests. Similarly, in the BDD example, tests are still defined and still contain validity checks, but the tests are defined with an `it` function, and instead of assertions, we have expectations (expect calls).

However, there are two important differences that go beyond just the names of the functions. First, the latter example reads much more like an English sentence, making it easier to read both in the code, and later on, when viewing the test results. Second, and more importantly, the first test emphasizes how `set` works, while the second one emphasizes what it should do. While there is nothing inherent in either the TDD or the BDD style that forces either test to be written in a certain way, this example highlights how the syntax can nevertheless influence a test's design. This can become important later on if the Backbone developers ever change Backbone to use a different name instead of *attributes*, as they will then have to rewrite the first test while the second one would continue working without adjustment.

Ultimately it is possible to test what your code does with either style of test, and so, it's really just a question of which style you prefer. However, if you have no opinion one way or the other, the BDD syntax is probably a better starting point, both for its improved readability and for its natural emphasis on describing what the code should do. For these reasons, we'll continue to use the BDD syntax in the rest of this chapter.

Describe, beforeEach, and it

If you've used testing libraries in other languages before you might already understand the preceding code, but if not allow me to explain. The test starts with a call to Mocha's `describe` function. This function creates a *suite* (or grouping) of tests, and you optionally nest more `describe` functions to further organize your tests. One benefit of using `describe` statements is that when you go to run your tests, you can choose to only run a specific suite defined by `describe`, rather than running all of them at once. The `describe` function takes two arguments: a string indicating what is being described, and a function that is used to wrap all of the tests within the `describe` function.

Next, we have a `beforeEach` call that, as you might imagine, defines code that will run before each test. This can be a useful place to separate code that is common to all of the tests in the `describe` statement (such as the creation of a new `Backbone.Model` class in the preceding example). Mocha also has an equivalent `afterEach` function that runs similarly after each test completes and can be used to clean up the side effects of the tests. Mocha also has `before` and `after` functions, which are similar except that they only run once per `describe` statement (that is per suite), rather than once per test.

After another `describe` (this time to group tests related to `set`), we get to the `it` function, which actually defines a specific test. Like `describe`, `it` takes two arguments: a string that describes what it (the code being tested) should do, and a function that tests whether the indicated behavior is actually occurring. If the function passed to the `it` function completes without throwing an error, Mocha will consider this test to have passed, whereas if an error is thrown, it will instead mark that test as failed (and provide the stack trace of the thrown error in the test output so that you can easily debug the relevant code).

Finally, we have the `expect` function (which could just as easily be called `assert` or something else had we chosen a different assertion library). The `expect` function takes a single argument and then uses jQuery-like chaining to assert something about the first argument. Here are a few examples:

```
expect(1).to.be(1); // asserts that 1 === 1
expect(1).not.to.be(2); // asserts that 1 !== 2
expect(1).to.eql("1") // asserts that 1 == "1"
expect(1).to.be.ok(); // asserts that 1 is "truthy" (evaluates
true when used as a boolean)
expect(1).to.be.a('number'); // asserts typeof 1 === 'number'
```

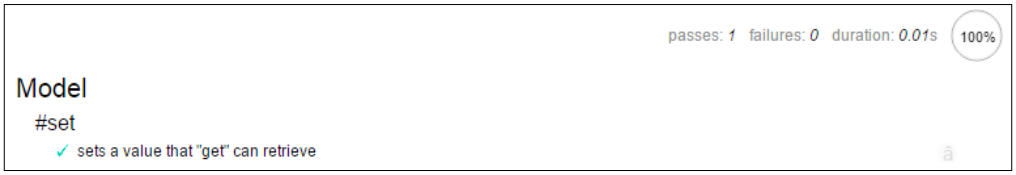
The `Expect.js` library has several other forms of assertions as well, all of which you can find detailed on their GitHub page.

Running our test

Now that we understand how everything works, let's try to actually run our test. To do this, simply replace the following line in our HTML file with the `describe` code provided earlier:

```
// *INSERT TESTS HERE*
```

Alternatively, you can also choose to put the `describe` code into a separate file and replace the insert line with a script tag that references this file. Either way, save the file and open it up in your favorite web browser. You should see something like this:



Our test passes!

Introducing mocks

As long as the code we're testing remains relatively simple, Mocha and Expect.js are all that we need to test it. However, the code rarely remains simple, and there are two complications in particular that can make us need another tool, known as a `mocking` library, to let us create fake versions or mocks of our objects.

First, let's imagine we want to test the following method of an imaginary `ExampleModel` class:

```
foo: function() {
    this.bar += 1;
    this.baz();
}
```

Now, we'll want to test whether our `foo` method calls the `baz` method. However, at the same time, we will (presumably) already have a separate test of the `baz` method itself. This leaves us with a dilemma: how can we test that `foo` calls `baz` without repeating the test code for `baz` inside the test code for `foo`?

Alternatively, let's consider another imaginary method that we might want to test:

```
fetchThenDoFoo: function() {
    this.fetch().done(this.foo);
}
```

In this case, we want to test whether `foo` is called after the `fetch` operation completes, but to truly test this, we'd need to actually fetch a `Model` class from the server by using AJAX. This, in turn, would make our tests require an active server, making them significantly slower and opening up the possibility of the tests causing side effects on our server.

The solution to both of these problems is to use a mocking library such as `Sinon.js`. To do this, simply download sinon.js from `http://sinonjs.org/` and then include this file (via a `script` tag) on your test running the HTML page.

Once Sinon is available, we can use it to solve our testing problems. First, let's use it to create a special kind of mock called `stub` for a test of our `foo` method as follows:

```
describe('foo', function() {
    var bazStub,
        example;
    beforeEach(function() {
        example = new ExampleModel();
        // Replace the real "baz" with a fake one that does
            nothing
```

```
        bazStub = sinon.stub(example, 'baz');
    });
    it('calls baz', function() {
        example.foo();
        expect(bazStub.calledOnce).to.be(true); // did foo call
          baz?
    });
    afterEach(function() {
        // Restore the original baz (in case another test uses it)
        baz.restore();
    });
});
```

As you can see in the preceding code, we were able to use Sinon to create `stub` that replaced the normal `baz` method. While this `stub` didn't actually do anything, it did keep track of how many times it was called (as well as which arguments were used, although we didn't test this), which let us write a test that ensured that `foo` would call `baz` without having to repeat any of the test code for `baz`.

For our second problem, that of testing an AJAX method, we could use an AJAX-specific mock tool such as MockJax. However, Sinon is so powerful that we really don't need to use anything else; consider the following test:

```
describe('fetchThenDoFoo', function() {

    var fetchStub,
        fooStub,
        example;
    beforeEach(function() {
        example = new ExampleModel();
        // Replace the real "fetch" with a fake one that returns
an
        // already-resolved $.Deferred
        var deferred = new $.Deferred().resolve();
        fetchStub = sinon.stub(example, 'fetch').returns(deferred);
        // Since we only want to test whether or not foo was
called,
        // we can also use stub for it
        fooStub = sinon.stub(example, 'foo');
    });
    it('calls foo after fetch completes', function() {
        example.fetchThenDoFoo();
        expect(fooStub.calledOnce).to.be(true);
    });
```

```
        afterEach(function() {
            // Restore the original versions of our stub functions
            fetchStub.restore();
            fooStub.restore();
        });
    });
```

In this example, we used two `stub` functions. We used the `fooStub` function in a manner similar to how we used the `bazstub` function in the previous example, to check whether or not `foo` got called, but our `fetchStub` served a different purpose. By chaining a `returns` method call off of our stub's creation, we created a `stub` function that (unlike the previous `stub` functions) actually did something: it returned our resolved deferred function. Since jQuery treats a resolved deferred function the same way as it treats a completed AJAX call (by invoking any done code for the call), we simulated the return of an AJAX call without involving an actual server.

Sinon has many other useful methods related to `stub`, as well as other types of mocking functions such as `spy` and `mock`, all of which are very well documented on their website. Sinon also has a feature known as `sandbox`, which can allow you to eliminate all of the `baz.restore()` code by automatically activating it after every test run (when `sandbox` is cleaned up). Again, you can find all the details of this feature on Sinon's website.

Selenium

Up to this point, we've been focusing entirely on unit testing, but it is worth mentioning that there are many other forms of tests that can benefit a Backbone application, such as load tests, smoke tests, and usability tests. However, a detailed discussion of all these tests is outside the scope of this book, and in most workplace environments, these tests won't be the purview of the development team anyway. Instead, they will be managed by the quality assurance (QA) department.

However, there is one type of testing, acceptance testing, which is worth mentioning. In smaller organizations that lack a QA department, acceptance tests will often be created and maintained by developers, and even in larger organizations, it's not unheard of for developers to assist with the creation and maintenance of these tests.

Acceptance tests test the functionality of your site, by checking whether or not a user can perform a specific action. A given acceptance test might check whether the user can log in to a site, change his password, or place an order. Unlike unit tests, which test a small piece of the larger functionality (such as a single `placeOrder` function), acceptance tests validate all of the disparate functions that are used to complete a particular user interaction.

On the web, there is a single tool that almost completely dominates acceptance testing: the Selenium web driver (`http://www.seleniumhq.org/`). The Selenium web driver allows you to create automated tests that perfectly simulate the actions of a real user on your site. A Selenium test can click a button, fill in a text field, scroll, and do just about anything else that an actual user can do.

Selenium is available for many different languages, including JavaScript. It can either be used directly via the Selenium web driver (`https://code.google.com/p/selenium/wiki/WebDriverJs`) or be wrapped with a library that offers an alternate syntax such as Nightwatch.js (`http://nightwatchjs.org/`). Here's an example from the Nightwatch.js home page that demonstrates a simple acceptance test written with Nightwatch.js:

```
module.exports = {
    'Demo test Google' : function (client) {
        client
          .url('http://www.google.com')
          .waitForElementVisible('body', 1000)
          .assert.title('Google')
          .assert.visible('input[type=text]')
          .setValue('input[type=text]', 'rembrandt van rijn')
          .waitForElementVisible('button[name=btnG]', 1000)
          .click('button[name=btnG]')
          .pause(1000)
          .assert.containsText('ol#rso li:first-child',
            'Rembrandt - Wikipedia')
          .end();
  }
};
```

As you can see from the preceding example, Nightwatch.js (and Selenium itself) uses the same selectors that you already use for CSS and jQuery, along with special methods such as `waitForElementVisibility` to control timing and prevent the automation script from moving too quickly through the test. This allows you to simulate any user story you want, no matter how simple or how complex it is, and then, repeat this story over and over to test your site.

However, Selenium-based tests do have their limits with the biggest being that because Selenium operates on the user level, it has no awareness of what's going on at the code level. If a Selenium test fails, it won't provide a stack trace or point you to a line of failing code; instead, it will simply alert you that a particular operation failed, and it will be up to you to debug it manually, just as you would with a user-submitted bug. Ideally, you should try and catch as many failures as you can in your unit test suite and only rely on Selenium-based acceptance tests to find bugs that "slip through the cracks" in your unit testing.

Summary

In this chapter, we learned how to combine a variety of tools to test your Backbone applications. We learned the difference between TDD and BDD, as well as the differences between major testing libraries such as QUnit and Jasmine. We explored in depth how to use the Mocha framework to create a testing suite and how to use the Sinon library to mock out parts of your code within that suite.

In the next chapter, we'll look into a variety of other third-party tools that can help you create Backbone applications. We'll look at general-purpose tools, such as the Require.js dependency management system, and Backbone-specific tools, such as BackGrid and BackSupport.

11
(Not) Re-Inventing the Wheel – Utilizing Third-Party Libraries

In this chapter, we'll take an introductory look at a mix of general-purpose and Backbone-specific third-party libraries, all of which can benefit you as a Backbone developer. In particular, we'll look at the following:

- Dependency management tools Require.js and Bower
- Table-generation tools Backbone Paginator and BackGrid
- HTML templating tool Handlebars
- Task automation tool Grunt
- Alternate language CoffeeScript
- General-purpose Backbone utility library BackSupport

The Backbone ecosystem

In the five years that Backbone has existed, its popularity has resulted in the development of hundreds of related third-party libraries. In addition, numerous other general-purpose libraries have been released, which can also be of great value to a Backbone developer. While we don't have room to explore these libraries in depth, this chapter will provide you with a preview of each library so that you can identify the ones that would be most beneficial to you.

As we cover the libraries in this chapter, keep in mind that for each library that we preview, there exist several (and sometimes several dozen) competing libraries that we didn't feature. While we could try to list every available library in each category, such a list would quickly become out of date, so instead we've chosen to focus only on the most popular offerings. If any of the libraries in this chapter strike you as useful, we strongly recommend searching the Internet to see what other similar libraries are available, as you may well find one that better suits you.

Dependency management with RequireJS

With modern IDEs, it is possible for a developer to open a file by typing just a few characters of that file's name. This feature, along with a general desire to keep code organized, has motivated developers to separate their code into multiple files. In Backbone, this typically means creating one file for every `Collection`, `Model`, `View`, and `Router`, and even in a small project, this can add up to a lot of files.

All these files create two problems. First, each file needs to be downloaded separately, and as we learned in *Chapter 8*, *Scaling Up: Ensuring Performance in Complex Applications*, browsers can only download between 2 and 8 files at once. Second, the order in which files are loaded can become increasingly hard to manage, because of the dependencies among the different files (`View A` needs `Collection B`, which needs `Model C`, which needs …). **RequireJS** (`http://requirejs.org/`) solves both of these problems.

RequireJS does this by organizing your code into `modules`. Each module can optionally depend on one or more other modules, and RequireJS will take care of stitching the modules together in a way that preserves all of their dependencies. RequireJS also provides a related tool, called RequireJS Optimizer, which allows you to combine multiple modules into a single file. The optimizer can also "minify" your code, as well as "uglify" it, making it harder for others to understand (to prevent competitors from reading your source files).

Here's an example RequireJS module for a BookList `View` class:

```
// All RequireJS modules start by calling a special "define" function
define([
    // The module's dependencies are the first argument
    'collections/Books', // dependency on collections/Books.js
    'models/Book'       // dependency on models/Book.js

// The function that defines the module is the second argument
], function(
    // The variable names for each dependency make up the
       arguments to that function    Books, // alias the
       "collections/Books" module as "Books"
    Book   // alias the "models/Book" module as "Book"
) {
    // The actual module itself goes here
    var BookList = Backbone.View.extend({
        // Logic for our BookList View would go here; presumably
           it
```

```
        // would use both Book and Books
    });
    // To tell RequireJS what variable this module should "define"
       simply return
    // that variable at the end
    return BookList;
    // in other words, whatever is returned will be what is passed
       in
    // to other modules that depend on this one
});
```

RequireJS's style of managing dependencies is known as the **AMD** style. There is also a competing style, known as **Common JS**, which is used by other dependency management libraries such as Browserify or Hem. Common JS modules look significantly different; here's our previous example rewritten using Browserify's Common JS syntax:

```
// Dependencies are brought in by using the "require" function
// The module aliases are defined on the same line using the
// standard JavaScript syntax for declaring a variable
var books = require('collections/Books');
var book = require('models/Book');
// Just as before, the contents of the module are defined using
// standard JavaScriptBookList = Backbone.View.extend({
    // Logic for our BookList View would go here; presumably it
       would
    // use both Book and Books
});
// Instead of returning what the module defines, in CommonJS
modules
// are "exported" by assigning them to a
"module.exports"module.exports.BookList = BookList;
```

The downside to this approach is that the actual ordering of the modules isn't handled for you, as it is in RequireJS. Instead, the order of the modules must be spelt out separately using `require` statements:

```
<script src="fileWithModuleDefinitions.js"/>
<script>
require('collections/Books');
require('models/Book');
require('views/BookList');
</script>
```

External dependency management with Bower

In addition to using RequireJS or a similar library for managing your code's dependencies, many programmers also use a second tool called **Bower** (`http://bower.io/`) to manage their external library dependencies. Much like Python's `pip` or Node.js's NPM, Bower offers a simple command line interface for easily installing external libraries. It's worth noting that NPM itself can also be used for managing libraries on the client-side, but this tool is designed primarily for server-side developers, whereas Bower is designed primarily for use on the client-side. To install a library, such as jQuery, one would simply run the following command at the command line:

```
bower install jquery
```

Multiple Bower dependencies can be stored in a `bower.json` file, allowing you to install all of your application's dependencies with a single command. Here's an example of what such a file might look like:

```
{
    "name": "your-project",
    "version": "0.0.1",
    "ignore": [
        "**/*.txt"
    ],
    "dependencies": {
        "backbone": "1.0.0",
        "jquery": "~2.0.0"
    },
     "devDependencies": {
        "mocha": "^1.17.1"
    }
}
```

As you can see, the preceding file defines the dependencies for your project. It includes Backbone and jQuery but not Underscore; as a dependency of Backbone, Underscore will be downloaded automatically (jQuery isn't technically a dependency of Backbone, as the library will work without it, so we have to require it separately). It also lets us include Mocha as `devDependency`, which means that it will be downloaded in a development environment but not in a production one. Using a requirements file like this allows you to keep your external libraries out of your source control system and to easily update them when new versions come out. It also allows you to easily manage different builds of these libraries (for instance, the debugging version versus the minified version).

Paginating with Backbone Paginator

One very common task many Backbone developers are faced with is rendering paginated data. For example, you might have a search page that can return hundreds of results, but you only want to show the first twenty results. To do this, you essentially need two `Collection` classes: one for all of the results and the other for the results being displayed. However, switching between the two can be tricky, and you may not actually want to fetch hundreds of results at once. Instead, you might just want to fetch the first twenty, but still be able to know how many total results are available so that you can display that information to your users.

Backbone Paginator (`https://github.com/backbone-paginator/backbone.paginator`) is a specialized `Collection` class created expressly for this purpose. Backbone Paginator was originally two separate libraries, but these libraries have since merged making Backbone Paginator the primary tool for handling paginated data in Backbone.

Backbone Paginator can be used in one of the following three modes:

- `client`: For when you want to fetch with your entire `Collection` at once
- `server`: For when you want to leave most of the `Collection` on the server and only fetch the relevant parts
- `infinite`: For creating a `Collection` class to power a Facebook-like infinite scroll view.

To use Backbone Paginator, you simply extend its `PageableCollection` class to create your own pageable `Collection` class, as follows:

```
var BookResults = Backbone.PageableCollection.extend({
    model: BookResult,
    queryParams: {
        currentPage: 'selected_page',
        pageSize: 'num_records'
    },
    state: {
        firstPage: 0,
        currentPage: 5,
        totalRecords: 500
    },
    url: 'www.example.com/api/book_search_results'
});
```

As you can see a `PageableCollection` class is very similar to a normal `Collection` class: It has the `model` and `url` properties, can be extended, and so on. However, it also has two special properties.

The first of these properties is `queryParams`, which tells `PageableCollection` how to interpret the pagination information in your server's response, in much the same way that a `parse` method normally tells Backbone how to interpret your server's response. The second property is `state`, which `PageableCollection` uses to keep track of which page of results the user is currently on, how many results there are in a page, and so on.

The full Backbone Paginator library has several other options for both `queryParams` and `state`, as well as a set of pagination methods such as `getPage` (for moving to a specific page of results) or `setSorting` (for changing how the results are sorted). If you want to implement a paginated view yourself, you can find full documentation of Backbone Paginator on its GitHub page. However, as it turns out, you may not want to create your own `View` of paginated data at all, because there's already a very powerful existing one that you can leverage instead: BackGrid.

Rendering tables with Backgrid.js

There are several different table `View` libraries available specifically for rendering tables in Backbone, and there are also many popular non-Backbone specific libraries (such as `jqGrid` or `DataTables`) that you can easily adapt for use in Backbone. However, BackGrid (`http://backgridjs.com/`) stands out from the rest because it combines a powerful feature set, simple design, and support for Backbone Paginator *out of the box*.

Here's an example of a table generated using BackGrid:

ID	Name	Population	% of World Population	Date	URL
1	Afghanistan	25,500,100	0.36	2013-01-01	http://en.wikipedia.org/wiki/Afghanistan
2	Albania	2,831,741	0.04	2011-10-01	http://en.wikipedia.org/wiki/Albania
3	Algeria	37,100,000	0.53	2012-01-01	http://en.wikipedia.org/wiki/Algeria
4	American Samoa (USA)	55,519	0.00	2010-04-01	http://en.wikipedia.org/wiki/American_Samoa
5	Andorra	78,115	0.00	2011-07-01	http://en.wikipedia.org/wiki/Andorra
6	Angola	20,609,294	0.29	2012-07-01	http://en.wikipedia.org/wiki/Angola
7	Anguilla (UK)	13,452	0.00	2011-05-11	http://en.wikipedia.org/wiki/Anguilla
8	Antigua and Barbuda	86,295	0.00	2011-05-27	http://en.wikipedia.org/wiki/Antigua_and_Barbuda
9	Argentina	40,117,096	0.57	2010-10-27	http://en.wikipedia.org/wiki/Argentina
10	Armenia	3,275,700	0.05	2012-06-01	http://en.wikipedia.org/wiki/Armenia
11	Aruba (Netherlands)	101,484	0.00	2010-09-29	http://en.wikipedia.org/wiki/Aruba
12	Australia	22,808,690	0.32	2012-11-11	http://en.wikipedia.org/wiki/Australia
13	Austria	8,452,835	0.12	2012-07-01	http://en.wikipedia.org/wiki/Austria
14	Azerbaijan	9,235,100	0.13	2012-01-01	http://en.wikipedia.org/wiki/Azerbaijan

1 2 3 4 5 6 7 8 9 10

To use BackGrid, you simply extend it, just as you would with any other `View` and then, instantiate it with one extra `columns` option:

```
var BookResultsGrid =  Backgrid.Grid.extend();
var grid = new BookResultsGrid({
    columns: [
        {name: 'bookTitle', label: 'title', cell: 'string'},
        {name: 'numPages', label: '# of Pages', cell: 'integer'},
        {name: 'authorName', label: 'Name of the Author',
         cell: 'string'}
    ],
    collection: bookResults
});
grid.render();
```

As you can see the extra `columns` option that you provide tells BackGrid which attribute of `Model` to use for the column's data (`name`), what this column's header text should be (`label`), and how BackGrid should format cells in this column (`cell`). Once you provide BackGrid with these columns and a `Collection` class, BackGrid will generate a `<table>` element as its `el` by using each `Model` in the provided `Collection` to generate a `<tr>` element.

If you only want a table that displays information that's all you need to do, but BackGrid can also be optionally used to edit information. To use this feature, you just need to pass one other `editable: true` option in each column that you want to make editable. When BackGrid renders its table, users will be able to click on any cells in the editable columns to switch the selected cell into the *edit* mode (for instance, replacing plain text with an HTML `<input>` tag), and any changes that the user makes will be automatically updated in the appropriate `Model`.

BackGrid has many other features as well, such as the ability to define your own custom cell types by extending Backgrid.Cell. You can find the full list of these features on Backgrid's website, which has excellent tutorial-style documentation and API reference documentation.

Templating with Handlebars

As we discussed in *Chapter 5*, *Adding and Modifying Elements with Views*, using templates to render your View's HTML offers many benefits. While you can simply use Underscore's `template` function, if you want a more powerful templating language there are many different libraries to choose from. For this chapter, we are going to use Handlebars (`http://handlebarsjs.com/`) as our templating engine. Other libraries you may want to consider include Mustache (`https://github.com/janl/mustache.js`), Embedded JS (`http://embeddedjs.com/`), or Hogan.js (`http://twitter.github.io/hogan.js/`).

Handlebars, which was created from another templating library (Mustache), offers a significant amount of in-template logic in the form of "helper". For instance, here's a Handlebars template that uses an "each" helper and an "if" helper to render a list of last names prefaced with either "Mr." or "Ms.," depending on the person's gender:

```
<ul>
    {{#each people}}
    <li>
        {{#if this.isMale}}Mr.{{else}}Ms.{{/if}} {{this.lastName}}
    </li>
    {{/each}}
</ul>
```

Handlebars also allows you to define your own custom helpers, which makes it very extensible. These helpers are so flexible that you could create an entire sub-language within your templates if you so desire.

Once you have a template written, you can either include it directly in your JavaScript code (wrapped in quotation marks to make it a valid string), or you can store it in a separate file and use RequireJS or a similar tool to bring it in. Thereafter, it's a simple matter of compiling the template and writing a `View` `render` method that uses this template, as follows:

```
var template = '<ul>' +
    '{{#each people}}' +
    '<li>' +
        '{{#if this.isMale}}Mr.{{else}}Ms.{{/if}} {{this.lastName}}' +
    '</li>' +
    '{{/each}}' +
'</ul>';
var compiledTemplate = Handlebars.compile(template);
var TemplatedView = Backbone.View.extend({
    render: function() {
        var templatedHtml = compiledTemplate(this.model.toJSON());
        this.$el.html(templatedHtml);
        return this;
    }
});
new TemplatedView({
    model: new Backbone.Model({isMale: 'true', lastName: 'Smith'})
}).render().$el.html(); // will be "Mr. Smith"
```

Automating tasks with Grunt

In many software projects, there are a number of tasks that are typically automated. For instance, just as a traditional C++ or Java project would need to compile the source code into byte code, a JavaScript project might need to use RequireJS or CoffeeScript to compile its source. The project might also need to concatenate files, run linting programs to validate the source, or assemble other web components such as sprite image files or SCSS/Less-generated CSS files.

Most of these tasks aren't language-specific: you don't need to actually use JavaScript code in order to start RequireJS Optimizer; you just need a command line. Because of this, it's possible to use a tool designed for another language (such as Ant or Maven for Java, or Fabric for Python) to automate these tasks, and if your server-side team is using such a language, it might be helpful to have everyone using the same tool.

However, if you don't have a server-side team (or if that team uses `Node.js`), you may wish to have a JavaScript-specific build tool, and this is where Grunt comes in. Here's an example Grunt configuration file that can be used to run RequireJS Optimizer:

```
module.exports = function(grunt) {
    grunt.initConfig({
        requirejs: {
            app: {
                options: {
                    findNestedDependencies: true,
                    mainConfigFile: 'public/js/config.js',
                    baseUrl : 'public/js',
                    name : 'app',
                    out : 'build.js',
                    optimize : 'none'
                }
            }
        }
    });
    grunt.loadNpmTasks('grunt-contrib-requirejs');
    grunt.registerTask('default', ["requirejs"]);
};
```

While a full explanation of the preceding code is outside the scope of this book, it is sufficient to say that by using this configuration file, you can run a single command at the command line to invoke RequireJS Optimizer. More importantly, you can make RequireJS Optimizer just one step of your entire deployment process and then, invoke that entire process by using a single command. You can also use Grunt to set up different processes for different environments, such as one process for setting up a development environment and a different one for setting up a production server.

New syntax with CoffeeScript

You would think that as the creator of Backbone (and **Underscore** and **Docco**), Jeremy Ashkenas would have enough on his hands … but you'd be wrong. Somewhere in between creating these three libraries, Jeremy also found the time to create an entirely new programming language called **CoffeeScript** (`http://coffeescript.org/`).

CoffeeScript is interesting to many web developers because of the following two key features:

1. CoffeeScript compiles to JavaScript, which means that you can use it for development but then compile it into a language that your user's browsers can actually understand.
2. CoffeeScript offers a syntax and feature set that have more in common with languages such as Python or Ruby than with "vanilla" JavaScript.

This is best explained by example. Here's how you would create a `View` class for an `h1` element in CoffeeScript:

```
class HeaderView extends Backbone.View
    tagName: 'h1'
    initialize: ->
        @render
    render: ->
        $(@el).text 'Hello World!'
header = new  HeaderView
```

Here are some of the key differences that we just demonstrated:

- CoffeScript has its own syntax for declaring classes
- CoffeeScript uses indentation rather than curly braces ("`{}`") to mark where a function starts/stops
- CoffeeScript declares functions using `->` instead of `function() {}` `this` can be referenced as `@` the parentheses in function calls are optional

Together these (and many other) features make CoffeeScript a significant improvement over JavaScript. However, there is a catch to using it. Because it compiles to JavaScript, debugging can be tricky, as the browser will use the JavaScript line numbers instead of the CoffeeScript line numbers when reporting errors. Also, when using the browser's debugger, you will walk through the JavaScript code, not the CoffeeScript code. While a new technology called "source maps" can help reduce these issues, it doesn't eliminate them entirely.

If you are willing to put up with the abovementioned inconveniences, CoffeeScript can provide you with a syntax and a feature set that JavaScript itself won't have for many years, if ever. Further, because it was created by Jeremy Ashkenas, you can be rest assured that Backbone will always be compatible with CoffeeScript. In fact, Jeremy added at least one feature to Backbone, the hidden `__super__` property, specifically to support CoffeeScript.

Making life easier with BackSupport

When I first started working with Backbone a few years ago, I was inspired to write BackSupport (`https://github.com/machineghost/BackSupport`) to help eliminate a lot of boilerplate code that I found myself writing over and over again. For instance, consider this basic `View` class:

```
var BookView = ParentBookView.extend({
    className: ParentBookView.prototype.className + ' book-view',
    initialize: function(options) {
        this.template = options.template ;
        if (!this.template ) {
            throw new Error('The template option is required!');
        }
        _.bindAll(this, ''render');
    },
    render: function() {
        this.$el.html(this.template(this.model.toJSON()));
   }
});
```

Now, what if there were a way to reduce all that common code to just the parts that are specific to our class? This is where BackSupport comes in. Let's look at that same `View` class recreated using BackSupport:

```
BookView = ParentBookView.extend2({
    boundMethods: ['render'],
    className: "book-view",
    propertyOptions: ['template'],
    requiredOptions: ['template'],
});
```

As you can see, BackSupport simplified so much of our logic that we didn't even need an `initialize` method in our second version!

Here are all of the BackSupport features demonstrated:

- `extend2`: This alternate form of Backbone's `extend` is smart enough to combine, rather than replace, properties such as `className`, `events`, or `defaults`. This allows you to more easily create subclasses that use these properties, without losing their values from the parent class.
- `boundMethods`: BackSupport will automatically call `_.bindAll` on every method included in this property, so we don't have to do it manually in an `initialize` method.
- `propertyOptions`: BackSupport will automatically transform any option included in this property into a new property of instances of this class, saving us from the tedium of doing `this.foo = options.foo` in our `initialize`.
- `requiredOptions`: BackSupport will throw an error if a class is instantiated without providing these options, giving us a simple way to ensure that they are provided without adding extra `initialize` logic.
- `render`: BackSupport provides a set of methods to make using templates easier. These methods are completely agnostic as to which templating system you use, and to choose a particular templating system, you need to only override the relevant BackSupport methods.

While this simple example provides a taste of what BackSupport offers, it also has many other great convenience features, which you can learn about on its GitHub page.

Summary

In this chapter, we learned about the great variety of tools available to the Backbone community. In particular, we looked at RequireJS and Bower for dependency management, Backbone Paginator and BackGrid for rendering paginated tables, and Handlebars for templating. We also looked at using Grunt for build management, CoffeeScript for an alternative syntax, and this author's own tool, BackSupport, for general-purpose solutions to many of the minor inconveniences of Backbone.

In the next chapter, we'll wrap things up by reviewing a summary of everything we've covered in this book. We'll also take a brief look at how the lessons learned can be applied to a real-world use case. Finally, we'll finish up with a look into the sources for learning even more about Backbone.

12
Summary and Further Reading

In this chapter, we begin by looking back over everything we've covered in the previous chapters. We then look at how Backbone is being used to power a real-world medical application, and finally, we look forward to how you can continue your Backbone education. In particular we will cover the following:

- Summarize the roles of various Backbone components
- Learn how Backbone is being used today to power Syapse
- Consider how everything we've learned is applied to the Syapse use case
- Look at further opportunities to learn about Backbone

Putting it all together

In the first half of this book, we examined the four Backbone classes (`Collection`, `Model`, `Router`, and `View`), and how each of them fit together to build a web application. To refresh, a Backbone-powered site starts with a `Router` class, which is used to map URLs to the virtual pages of the application. `Router` makes up half of the MVC `Controller` layer, while the `View` class makes up the other half. The `View` class is also responsible for the MVC `View` layer of a web application, as `View` classes both render the pages that make up the site and listen for and respond to user-generated events.

Of course, a data or MVC `Model` layer is also essential for almost any application, and this layer is handled by the `Model` and `Collection` classes. `Model` classes, which represent individual data objects, are used both to manage data on the client-side and to send and receive data to and from the server. `Collection` classes hold sets of `Model` classes but are otherwise used similarly to manage data and transmit it to/from the server. `Model` and `Collection` classes are used primarily by `View` classes, which render their data into DOM elements.

All of these classes are designed not just to be used directly but also to be extended into new subclasses with logic specific to your application. This same extension mechanism is used by third-party Backbone libraries, such as BackGrid and BackSupport, to provide components that further extend the capabilities of your application. However, Backbone-specific libraries aren't the only third-party libraries that you can use in your Backbone application. By wrapping non-Backbone components with your own custom Backbone classes, you can cleanly incorporate tools such as a template system or a jQuery UI widget in to your application. Even libraries that can't be wrapped in a Backbone class, such as Underscore, RequireJS, or Mocha, can be used independently to add functionality without losing any of the benefits of Backbone itself.

This, in short, summarizes everything we've learned so far, but as we covered that information originally, we kept our focus on component at a time. As we finish this book, it would be beneficial to look at how all these pieces can be used together to implement a real-world use case. That use case is Syapse.

What's Syapse?

As mentioned in *Chapter 1*, *Building a Single-Page Site Using Backbone*, Syapse (`http://www.syapse.com`) is a Backbone-powered web application for requesting and providing precision medicine results. Syapse's customers are laboratories and hospitals that use genetic sequencing to profile patients with serious diseases like cancer. Once sequenced, these genetic profiles can be combined with large bodies of research to determine the best treatments and dosages for a given patient based on the patient's own DNA.

Let's take a look at how Syapse was put together.

The 10,000-foot view

Syapse's client-side code is organized using Require.js (see *Dependency management with RequireJS* in *Chapter 11, (Not) Re-Inventing the Wheel: Utilizing Third-Party Libraries*). Every module is either a class, a singleton instance (for utility libraries), or a function (for routes). Syapse has two different sites, one for laboratories and the other for clinics, so we use RequireJS to compile separate JavaScript files for each site.

Each of these files has its own Backbone `Router`, allowing each site to have a completely distinct set of URLs and pages (see *Multiple routers* in *Chapter 6*,

Creating Client-Side Pages with Routers). These `Router` classes form the top of the RequireJS dependency tree, bringing in (or depending on) all of the site's routes. These route modules then bring in the site's `Views` classes, which in turn bring in the site's `Model` and `Collection` classes.

There are four "base" classes that every other class in the application inherits from. These base classes are based on the BackSupport classes (see *Making life easier with BackSupport* in *Chapter 11, (Not) Re-Inventing the Wheel: Utilizing Third-Party Libraries*), and are used to define new features that are Syapse-specific. For instance, since Syapse uses the Handlebars templating library, the base `View` class includes the logic for rendering Handlebars templates (see *Templating with Handlebars* in *Chapter 11, (Not) Re-Inventing the Wheel: Utilizing Third-Party Libraries*).

Syapse uses the `Page View` pattern (see *Page views* in *Chapter 5, Adding and Modifying Elements With Views*) with one base `Page View` class for the laboratory interface and one for the clinic interface. These render all of the parts of the site that are shared between pages, such as the navigation menu, and both of these `Page View` patterns share a common base `View` class, allowing them to reuse generic page-rendering logic that is common to both sites.

The View layer

Pages in the Syapse laboratory site are divided into three sections, each with their own `View` class: a left navigation section, a header area, and a main content area. Each of these can optionally be overwritten when the page is instantiated, letting each route modify only its unique section(s). For instance, most routes don't change the left navigation bar, so when these routes instantiate the page `View` class, they simply rely on the default navigation bar of this `View` class.

Each of the subclasses of `View` handles the rendering of its part of the page. Because Syapse relies on *The combined approach* (see *Chapter 5*, *Adding and Modifying Elements with Views*) to do its rendering, these `View` classes use a combination of Handlebars templates and other child `View` classes to generate their content. For particularly complex pages, this can result in not just child `View` classes but also grandchild, great grandchild, and sometimes, even great-great grandchild `View` classes. Whenever possible, Syapse uses multiple smaller `View` classes rather than one large one, to keep the logic for each `View` class as simple as possible.

For consistency, all of the render methods in these `View` classes return this. Further, for consistency, each `View` class is designed to be re-renderable, so that it can easily listen and respond to change events in its `Model` classes (see *Other render considerations* in *Chapter 5*, *Adding and Modifying Elements with Views*).

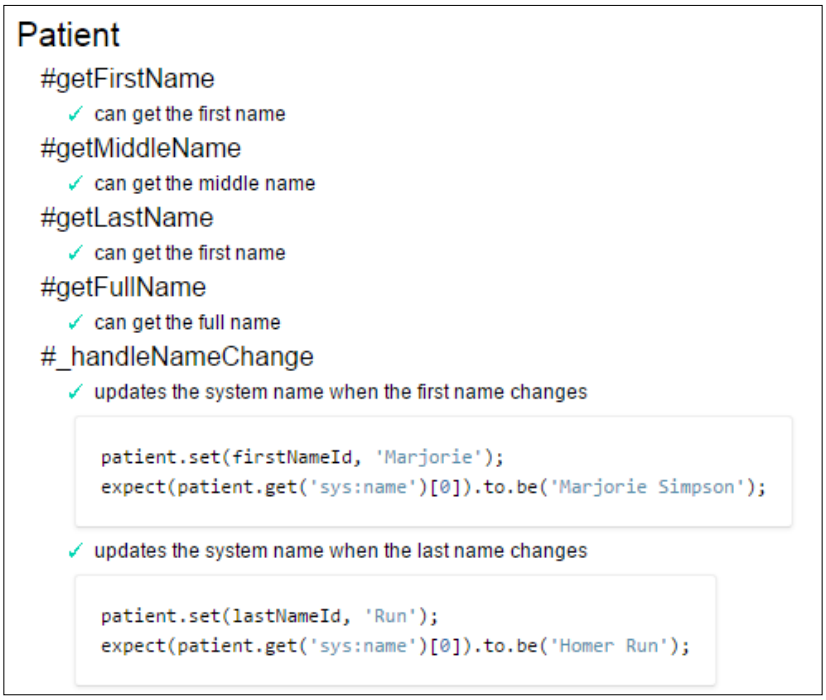
The Data layer

On the server-side, Syapse uses Python and the Django REST Framework library to power all of its APIs, and while this gives the server team a great deal of control over the API, they are still somewhat limited by the library and can't always make APIs that return ideal Backbone JSON. Because of this many of Syapse's `Model` and `Collection` classes make use of `parse` and `toJSON` method overrides to extract or send back the correct JSON to/from the API (see *Fetching data from the server* and *Saving data to the server* in *Chapter 3*, *Accessing Server Data with Models*).

Sometimes, Syapse's APIs return not just the data for a given `Model` or `Collection` class but also supplemental information. For instance, an API that returns a patient `Model` class might include the ID of that patient's doctor, but since the end user wants to see a name instead of an ID, the API response also includes a separate map of IDs to names. To keep track of this supplemental information, Syapse relies on a global pub/sub system (see *Publish/Subscribe* in *Chapter 7*, *Fitting Square Pegs in Round Holes: Advanced Backbone Techniques*). Whenever a `Model` class parses such supplemental information during a fetch, it triggers a special supplemental information" event with this information as an extra argument. This event is then listened for by one or more site-wide `Collection` caches, which aggregate the supplemental information and make it available for the `View` classes.

The Support layer

All of Syapse's code is tested using Selenium for acceptance testing and Mocha/Sinon for unit testing (see *Testing? In JavaScript?* and *Selenium* in *Chapter 10*, *Keeping the Bugs Out: How to Test a Backbone Application*). By using expressive test names and the BDD test output style (see *TDD versus BDD: What's the difference?* in *Chapter 10*, *Keeping the Bugs Out: How to Test a Backbone Application*), we ensure that Syapse's test output is very specific and looks like the following:



For documentation, Syapse relies primarily on inline documentation, using JSDoc annotations without actually rendering documentation pages (see *The robust documentation approach* in *Chapter 9*, *What Was I Thinking? Documenting Backbone Code*). However, Syapse also has a customer-facing JavaScript API, which is considerably more heavily documented than both generated JSDoc API pages and a Docco-based tutorial.

Building your own Syapse

Your application might be, like Syapse, a serious tool designed to help with a critical problem such as fighting cancer. Alternatively, your application might be something more fun, like a game or personal project. In either case, Backbone offers everything you need to not just build your site but also continue adding to and maintaining the site over its entire lifetime.

However, no book can possibly explain every possible nuance of a library that is as flexible and as powerful as Backbone. At its core, Backbone strives to do only a few important things well and leaves everything else up to you, the programmer. This not only means a great deal of power and flexibility, but it also means that you have to make a large number of decisions for yourself as to how you want to use Backbone. To make the correct decisions, and truly take advantage of everything Backbone has to offer, you will no doubt want to continue learning as much as you can about both Backbone in particular and web development in general.

Further reading

There are many places to learn about Backbone, but perhaps, the best place is Backbone's own source code. As the author of both Backbone and Docco, it's no surprise that Jeremy Ashkenas uses the Docco documentation tool to provide an annotated version of Backbone's source code, which you can find at `http://backbonejs.org/docs/backbone.html`.

However, you don't need the annotated version to read Backbone's code. In fact, whenever some part of Backbone seems confusing or difficult to understand, one of the best ways to learn more is to throw debugger statements into the Backbone source code itself and then, run your application using the browser's debugging tool. By walking through the code in the debugger, you can see the logic progress through Backbone's classes and methods, and because the source code is so well-written and readable, this task will be much less daunting than it would be with almost any other major JavaScript library.

Backbone also offers a wiki at `https://github.com/jashkenas/backbone/wiki`. In addition to basic Backbone information, this wiki includes collections of plugins and development tools, lists of companies that use Backbone, and a great variety of tutorials and informative blog posts. This latter list (which can be found at `https://github.com/jashkenas/backbone/wiki/Tutorials,-blog-posts-and-example-sites`) is particularly valuable, with more than fifty different high-quality sites where you can learn more about Backbone.

Another great source of information on Backbone is Packt Publishing, which offers many great Backbone-focused books. While they do cover some of the same basics as this book, you might find the sample applications built in *BackboneJS Blueprints* by Andrew Bugess or the recipes of *Backbone.js Cookbook* by Vadim Mirgood to be valuable. If you'd prefer a more advanced look at useful Backbone patterns you can employ, then you might instead find *Backbone.js Patterns and Best Practices* by Swarnendu De useful. Further, if you find your interest piqued by *Chapter 10*, *Keeping the Bugs Out: How to Test a Backbone Application*, then *Backbone.js Testing* by Ryan Roemer is a perfect text for you to learn more about Backbone testing.

However, as great as books are, they can never keep completely up-to-date with the very latest in emerging Backbone technologies, and that's where certain websites can be invaluable. One incredible resource is Stack Overflow (`htttp://www.stackoverflow.com`), where programmers of any persuasion can find answers to their technical questions, including ones about Backbone. However, Stack Overflow can also be valuable when you don't have a specific question. Because the site has both *tags* and *votes* on every question, you can search for questions with the `backbone.js` tag, and then sort them by votes; any questions at the top will most likely be educational. At the time of writing, Stack Overflow featured more than 12,000 different Backbone questions with answers (and more than 17,000 questions in total).

Another similar question and answer site, which isn't programming-focused, is Quora (`https://www.quora.com/`). While Stack Overflow limits itself exclusively to objective questions, Quora has no such limitation and so is perfect for more subjective questions, such as "What are the advantages of Backbone.js" (`http://www.quora.com/What-are-the-advantages-of-Backbone-js`).

Another excellent place to learn about Backbone is the Backbone Google Group (`https://groups.google.com/forum/#!forum/backbonejs`), which has an active community. If, on the other hand, you prefer to follow a stream of news articles, the Backbone sub-feed on Reddit (`http://www.reddit.com/r/backbonejs/`) is a great resource for keeping up to date. Finally, on a more general level, Hacker News (`https://news.ycombinator.com/`), Lobsters (`https://lobste.rs`), and Dzone (`http://www.dzone.com/links/index.html`) all feature continuous feeds of various programming-related news and articles, including many on JavaScript in general and Backbone in particular.

Summary

In this chapter, we reviewed the previous eleven chapters and summarized patterns of how to use Backbone to build a robust web application. We also looked specifically at how Backbone was used to build the cancer-fighting application Syapse. Finally, we examined other places where you can learn even more about Backbone, including other great books from Packt Publishing. We hope you enjoyed this book, and we wish you the best of luck in creating powerful web applications using Backbone.

Index

D

E

F

G

H

I

J

L

M

N

O

U

V

W

Thank you for buying

Backbone.js Essentials

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.

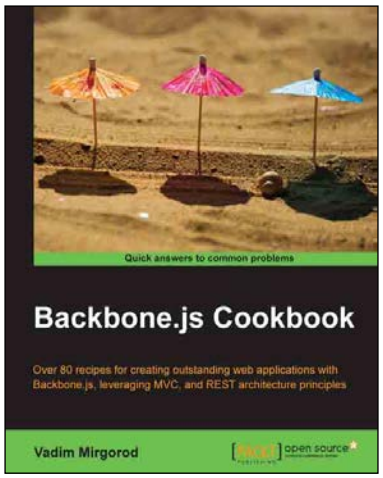
[PACKT] open source
community experience distilled
PUBLISHING

Backbone.js Blueprints

ISBN: 978-1-78328-699-7 Paperback: 256 pages

Understand Backbone.js pragmatically by building seven different applications from scratch

1. Gain insights into the inner working of Backbone to leverage it better.
2. Exploit Backbone combined with the features of a Node powered server.
3. Learn how to build seven step-by-step frontend applications.



Backbone.js Cookbook

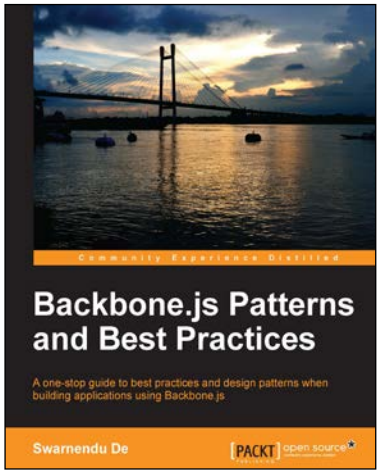
ISBN: 978-1-78216-272-8 Paperback: 282 pages

Over 80 recipes for creating outstanding web applications with Backbone.js, leveraging MVC, and REST architecture principles

1. Easy-to-follow recipes to build dynamic web applications.
2. Learn how to integrate with various frontend and mobile frameworks.
3. Synchronize data with a RESTful backend and HTML5 local storage.
4. Learn how to optimize and test Backbone applications.

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING

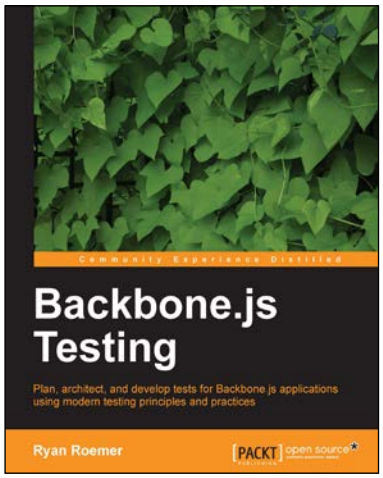


Backbone.js Patterns and Best Practices

ISBN: 978-1-78328-357-6 Paperback: 174 pages

A one-stop guide to best practices and design patterns when building applications using Backbone.js

1. Offers solutions to common Backbone.js related problems that most developers face.
2. Shows you how to use custom widgets, plugins, and mixins to make your code reusable.
3. Describes patterns and best practices for large scale JavaScript application architecture and unit testing applications with QUnit and SinonJS frameworks.



Backbone.js Testing

ISBN: 978-1-78216-524-8 Paperback: 168 pages

Plan, architect, and develop tests for Backbone.js applications using modern testing principles and practices

1. Create comprehensive test infrastructures.
2. Understand and utilize modern frontend testing techniques and libraries.
3. Use mocks, spies, and fakes to effortlessly test and observe complex Backbone.js application behavior.
4. Automate tests to run from the command line, shell, or practically anywhere.