

Contents

Prelude	7
Target Audience	9
Acknowledgements	9
Credits	9
Reading	9
Introduction	10
What Is MVC?	10
What is Backbone.js?	11
When Do I Need A JavaScript MVC Framework?	12
Why Consider Backbone.js?	13
Setting Expectations	13
Fundamentals	15
MVC	15
Smalltalk-80 MVC	15
MVC Applied To The Web	16
MVC In The Browser	18
Client-Side MVC - Backbone Style	19
Implementation Specifics	22
What does MVC give us?	25
Delving Deeper into MVC	26
Summary	26
Further reading	27
Fast facts	27
Backbone.js	27
Used by	27

Backbone Basics	30
Models	30
Views	40
Collections	47
RESTful Persistence	56
Events	59
Routers	65
Backbone's Sync API	70
Dependencies	73
Summary	73
 Exercise 1: Todos - Your First Backbone.js App	 73
Static HTML	74
Todo model	77
Todo collection	78
Application View	80
Individual Todo View	85
Startup	88
In action	88
Completing & deleting todos	89
Todo routing	93
Summary	95
 Exercise 2: Book Library - Your First RESTful Backbone.js App	 95
Setting up	95
Wiring in the interface	103
Adding models	103
Removing models	104
Creating the back-end	105
Install node.js, npm, and MongoDB	106
Install node modules	106
Create a simple web server	107

Connect to the database	109
Talking to the server	117
Summary	123
Backbone Extensions	123
MarionetteJS (Backbone.Marionette)	123
Boilerplate Rendering Code	125
Reducing Boilerplate With Marionette.ItemView	126
Memory Management	127
Region Management	130
Marionette Todo app	132
Is the Marionette implementation of the Todo app more main- tainable?	143
Marionette And Flexibility	144
And So Much More	145
Thorax	145
Hello World	145
Embedding child views	146
View helpers	147
collection helper	148
Custom HTML data attributes	149
Thorax Resources	151
Common Problems & Solutions	152
Modular Development	177
Introduction	177
Organizing modules with RequireJS and AMD	177
Maintainability problems with multiple script files	178
Need for better dependency management	178
Asynchronous Module Definition (AMD)	179
Writing AMD modules with RequireJS	179
Getting Started with RequireJS	181

Require.js and Backbone Examples	184
Keeping Your Templates External Using RequireJS And The Text Plugin	188
Optimizing Backbone apps for production with the RequireJS Optimizer	189
Exercise: Your First Modular Backbone + RequireJS App	192
Overview	192
Markup	193
Configuration options	194
Modularizing our models, views and collections	195
Route-based module loading	201
JSON-based module configuration	201
Module loader Router	202
Using NodeJS to handle pushState	204
An asset package alternative for dependency management	204
Paginating Backbone.js Requests & Collections	205
Paginator's pieces	206
Live Examples	206
Paginator.requestPager	206
Paginator.clientPager	210
Plugins	214
Backbone Boilerplate And Grunt-BBB	214
Getting Started	216
Backbone Boilerplate and Grunt-BBB	216
Creating a new project	216
index.html	218
config.js	219
main.js	221
app.js	222
Creating Backbone Boilerplate Modules	224

router.js	227
Related Tools & Projects	228
Conclusions	228
Mobile Applications	229
Backbone & jQuery Mobile	229
Resolving the routing conflicts	229
Exercise: A Backbone, Require.js/AMD app with jQuery Mobile	230
Getting started	230
jQuery Mobile: Going beyond mobile application development	232
Unit Testing	232
Jasmine	233
Behavior-Driven Development	233
Suites, Specs, & Spies	234
beforeEach() and afterEach()	239
Shared scope	241
Getting set up	242
TDD With Backbone	243
Models	243
Collections	245
Views	247
View testing	249
Conclusions	257
Exercise	257
Further reading	257
QUnit	257
Introduction	257
Getting Setup	258
Assertions	259
Adding structure to assertions	262

Assertion examples	264
equal - a comparison assertion. It passes if actual == expected .	264
notEqual - a comparison assertion. It passes if actual != expected	264
strictEqual - a comparison assertion. It passes if actual ===	
expected.	264
notStrictEqual - a comparison assertion. It passes if actual !==	
expected.	265
deepEqual - a recursive comparison assertion. Unlike strictEqual(),	
it works on objects, arrays and primitives.	265
notDeepEqual - a comparison assertion. This returns the opposite	
of deepEqual	265
raises - an assertion which tests if a callback throws any exceptions	265
Fixtures	266
Fixtures example:	266
Asynchronous code	268
SinonJS	270
What is SinonJS?	270
Stubs and mocks	273
Stubs	273
Mocks	275
Exercise	275
Models	276
Collections	277
Views	279
App	280
Further Reading & Resources	281
Resources	281
Books & Courses	281
Extensions/Libraries	282
Conclusions	282

Appendix	284
A Simple JavaScript MVC Implementation	284
Event System	284
Models	285
Views	286
Controllers	287
Practical Usage	287
MVP	290
Models, Views & Presenters	290
MVP or MVC?	291
MVC, MVP and Backbone.js	291
Namespacing	293
Backbone Dependency Details	298
DOM Manipulation	298
Utilities	299
RESTful persistence	299
Routing	300
Upgrading to Backbone 0.9.10	300
Model	301
Collection	302
View	303
Events	304
Routers	307
Sync	307
Other	309

Prelude

In the past, building data-rich web applications with JavaScript was a challenge. The transition from writing an app that requires complete page reloads into something more dynamic often requires rethinking our architecture and it's easy for things to get out of hand if no effort is put into keeping your code organized.



We can't just hack together some jQuery code and hope that this will scale as your application grows.

Thankfully, we now have a number of libraries that can help improve the structure and maintainability of our code, making it easier to build ambitious interfaces without a great deal of effort. [Backbone.js](#) is one such foundation for solving these problems and in this book we will be taking you through an in-depth walkthrough of it.

Begin with the fundamentals, work your way through the practicals and learn how to build an application that is both cleanly organized and maintainable. If you are a developer looking to write code that can be more easily read, structured and extended - this guide will hopefully be of help.

This (in-progress) book is released under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported [license](#) meaning you can both grab a copy of the book for free or help to further [improve](#) it. Corrections to existing material are always welcome and I hope that together we can provide the community with an up-to-date resource that is of help.

My extended thanks go out to [Jeremy Ashkenas](#) for creating Backbone.js and [these](#) members of the community for their assistance making this project far better than I could have imagined.

Target Audience

This book is targeted at novice to intermediate developers wishing to learn how to better structure their client-side code. An understanding of JavaScript fundamentals is required to get the most out of it, however we have tried to provide a basic description of these concepts where possible.

Acknowledgements

I am indebted to the fantastic work done by the technical reviewers who helped improve this book. Their knowledge, energy and passion have helped shape it into a better learning resource and they continue to serve as a source of inspiration. Thanks go out to:

- [Marc Friedman](#)
- [Derick Bailey](#)
- [Jeremy Ashkenas](#)
- [Samuel Clay](#)
- [Mat Scales](#)
- [Alex Graul](#)
- [Dusan Gledovic](#)
- [Sindre Sorhus](#)

Credits

None of this would have been possible without the time and effort invested by the other developers and authors in the community who contributed to it. I would like to extend my thanks to Derick Bailey, Ryan Eastridge, Jack Franklin, Mike Ball, Ugis Ozols, Björn Ekengren and our other excellent [contributors](#) that made this project possible.

Reading

I assume your level of knowledge about JavaScript goes beyond the basics and as such certain topics such as object literals are skipped. If you need to learn more about the language, I am happy to suggest:

- [JavaScript: The Definitive Guide](#) by David Flanagan (O'Reilly)
- [Effective JavaScript](#) by David Herman (Pearson)
- [JavaScript: The Good Parts](#) by Douglas Crockford (O'Reilly)
- [Object-Oriented JavaScript](#) by Stoyan Stefanov (Packt Publishing)

Introduction

Frank Lloyd Wright once said “You can’t make an architect. You can however open the doors and windows toward the light as you see it.” In this book, I hope to shed some light on how to improve the structure of your web applications, opening doors to what will hopefully be more maintainable, readable applications in your future.

The goal of all architecture is to build something well; in our case, to craft code that is enduring and delights both ourselves and the developers who will maintain our code long after we are gone. We all want our architecture to be simple, yet beautiful.

Modern JavaScript frameworks and libraries can bring structure and organization to your projects, establishing a maintainable foundation right from the start. They build on the trials and tribulations of developers who have had to work around callback chaos similar to that which you are facing now or may in the near future.

When developing applications using just jQuery, the piece missing is a way to structure and organize your code. It’s very easy to create a JavaScript app that ends up a tangled mess of jQuery selectors and callbacks, all desperately trying to keep data in sync between the HTML for your UI, the logic in your JavaScript, and calls to your API for data.

Without something to help tame the mess, you’re likely to string together a set of independent plugins and libraries to make up the functionality or build everything yourself from scratch and have to maintain it yourself. Backbone solves this problem for you, providing a way to cleanly organize code, separating responsibilities into recognizable pieces that are easy to maintain.

In “Developing Backbone.js Applications,” I and a number of other experienced authors will show you how to improve your web application structure using the popular JavaScript library, Backbone.js

What Is MVC?

A number of modern JavaScript frameworks provide developers an easy path to organizing their code using variations of a pattern known as MVC (Model-View-Controller). MVC separates the concerns in an application into three parts:

- Models represent the domain-specific knowledge and data in an application. Think of this as being a ‘type’ of data you can model — like a User, Photo, or Todo note. Models can notify observers when their state changes.
- Views typically constitute the user interface in an application (e.g., markup and templates), but don’t have to be. They observe Models, but don’t directly communicate with them.

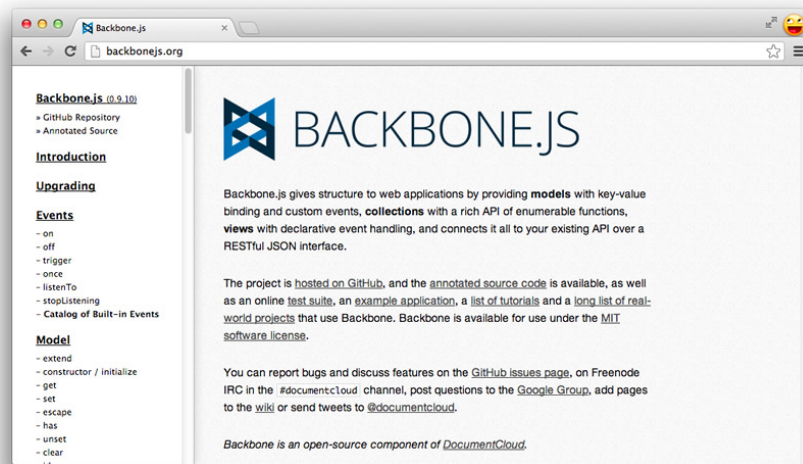
- Controllers handle input (e.g., clicks, user actions) and update Models.

Thus, in an MVC application, user input is acted upon by Controllers which update Models. Views observe Models and update the user interface when changes occur.

JavaScript MVC frameworks don't always strictly follow the above pattern. Some solutions (including Backbone.js) merge the responsibility of the Controller into the View, while other approaches add additional components into the mix.

For this reason we refer to such frameworks as following the MV* pattern; that is, you're likely to have a Model and a View, but a distinct Controller might not be present and other components may come into play.

What is Backbone.js?



Backbone.js is a lightweight JavaScript library that adds structure to your client-side code. It makes it easy to manage and decouple concerns in your application, leaving you with code that is more maintainable in the long term.

Developers commonly use libraries like Backbone.js to create single-page applications (SPAs). SPAs are web applications that load into the browser and then react to data changes on the client side without requiring complete page refreshes from the server.

Backbone is mature, popular, and has both a vibrant developer community as well as a wealth of plugins and extensions available that build upon it. It has been used to create non-trivial applications by companies such as Disqus, Walmart, SoundCloud and LinkedIn.

Backbone focuses on giving you helpful methods for querying and manipulating your data rather than re-inventing the JavaScript object model. It's a library, rather than a framework, that plays well with others and scales well, from embedded widgets to large-scale applications.

As it's small, there is also less your users have to download on mobile or slower connections. The entire Backbone source can be read and understood in just a few hours.

When Do I Need A JavaScript MVC Framework?

When building a single-page application using JavaScript, whether it involves a complex user interface or is simply trying to reduce the number of HTTP requests required for new Views, you will likely find yourself inventing many of the pieces that make up an MV* framework.

At the outset, it isn't terribly difficult to write your own application framework that offers some opinionated way to avoid spaghetti code; however, to say that it is equally as trivial to write something as robust as Backbone would be a grossly incorrect assumption.

There's a lot more that goes into structuring an application than tying together a DOM manipulation library, templating, and routing. Mature MV* frameworks typically include not only the pieces you would find yourself writing, but also include solutions to problems you'll find yourself running into later on down the road. This is a time-saver that you shouldn't underestimate the value of.

So, where will you likely need an MV* framework and where won't you?

If you're writing an application where much of the heavy lifting for view rendering and data manipulation will be occurring in the browser, you may find a JavaScript MV* framework useful. Examples of applications that fall into this category are Gmail, NewsBlur and the LinkedIn mobile app.

These types of applications typically download a single payload containing all the scripts, stylesheets, and markup users need for common tasks and then perform a lot of additional behavior in the background. For instance, it's trivial to switch between reading an email or document to writing one without sending a new page request to the server.

If, however, you're building an application that still relies on the server for most of the heavy-lifting of page/view rendering and you're just using a little JavaScript or jQuery to make things more interactive, an MV* framework may be overkill. There certainly are complex Web applications where the partial rendering of views can be coupled with a single-page application effectively, but for everything else, you may find yourself better sticking to a simpler setup.

Maturity in software (framework) development isn't simply about how long a framework has been around. It's about how solid the framework is and more

importantly how well it's evolved to fill its role. Has it become more effective at solving common problems? Does it continue to improve as developers build larger and more complex applications with it?

Why Consider Backbone.js?

Backbone provides a minimal set of data-structuring (Models, Collections) and user interface (Views, URLs) primitives that are helpful when building dynamic applications using JavaScript. It's not opinionated, meaning you have the freedom and flexibility to build the best experience for your web application how you see fit. You can either use the prescribed architecture it offers out of the box or extend it to meet your requirements.

The library doesn't focus on widgets or replacing the way you structure objects - it just supplies you with utilities for manipulating and querying data in your application. It also doesn't prescribe a specific template engine - while you are free to use the Micro-templating offered by Underscore.js (one of its dependencies), views can bind to HTML constructed using your templating solution of choice.

Looking at the [large](#) number of applications built with Backbone, it's clear that it scales well. Backbone also works quite well with other libraries, meaning you can embed Backbone widgets in an application written with AngularJS, use it with TypeScript, or just use an individual class (like Models) as a data backer for simpler apps.

There are no performance drawbacks to using Backbone to structure your application. It avoids run loops, two-way binding, and constant polling of your data structures for updates and tries to keep things simple where possible. That said, should you wish to go against the grain, you can of course implement such things on top of it. Backbone won't stop you.

With a vibrant community of plugin and extension authors, there's a likelihood that if you're looking to achieve some behavior Backbone is lacking, a complementary project exists that works well with it. This is made simpler by Backbone offering literate documentation of its source code, allowing anyone an opportunity to easily understand what is going on behind the scenes.

Having been refined over two and a half years of development, Backbone is a mature library that will continue to offer a minimalist solution for building better web applications. I regularly use it and hope that you find it as useful an addition to your toolbox as I have.

Setting Expectations

The goal of this book is to create an authoritative and centralized repository of information that can help those developing real-world apps with Backbone. If you come across a section or topic which you think could be improved or

expanded on, please feel free to submit an issue (or better yet, a pull-request) on the book's [GitHub site](#). It won't take long and you'll be helping other developers avoid the problems you ran into.

Topics will include MVC theory and how to build applications using Backbone's Models, Views, Collections, and Routers. I'll also be taking you through advanced topics like modular development with Backbone.js and AMD (via RequireJS), solutions to common problems like nested views, how to solve routing problems with Backbone and jQuery Mobile, and much more.

Here is a peek at what you will be learning in each chapter:

Chapter 2, Fundamentals traces the history of the MVC design pattern and introduces how it is implemented by Backbone.js and other JavaScript frameworks.

Chapter 3, Backbone Basics covers the major features of the core Backbone.js framework and technologies and techniques you will need to know in order to apply it.

Chapter 4, Exercise 1: Todos - Your First Backbone.js App takes you step-by-step through development of a simple client-side Todo List application.

Chapter 5, Exercise 2: Book Library - Your First RESTful Backbone.js App walks you through development of a Book Library application which persists its model to a server using a REST API.

Chapter 6, Backbone Extensions describes Backbone.Marionette and Thorax, two extension frameworks which add features to Backbone.js that are useful for developing large-scale applications.

Chapter 7, Common Problems and Solutions reviews common issues you may encounter when using Backbone.js and ways of addressing them.

Chapter 8, Modular Development looks at how AMD modules and RequireJS can be used to modularize your code.

Chapter 9, Backbone Boilerplate And Grunt BBB introduces powerful tools you can use to bootstrap a new Backbone.js application with boilerplate code.

Chapter 10, Mobile Applications addresses the issues that arise when using Backbone with jQuery Mobile.

Chapter 11, Jasmine covers how to unit test Backbone code using the Jasmine test framework.

Chapter 12, QUnit discusses how to use the QUnit for unit testing.

Chapter 13, SinonJS discusses how to use SinonJS for unit testing your Backbone apps.

Chapter 14, Resources provides references to additional Backbone-related resources.

Chapter 15, Conclusions wraps up the our tour through the world of Backbone.js development.

Chapter 16, Appendix returns to our design pattern discussion by contrasting MVC with the Model-View-Presenter (MVP) pattern and examines how Backbone.js relates to the two patterns. It also provides useful information for existing Backbone users who may be upgrading from Backbone 0.9.2 to version 0.9.10 and beyond.

Fundamentals

Design patterns are proven solutions to common development problems that can help us improve the organization and structure of our applications. By using patterns, we benefit from the collective experience of skilled developers who have repeatedly solved similar problems.

Historically, developers creating desktop and server-class applications have had a wealth of design patterns available for them to lean on, but it's only been in the past few years that such patterns have been applied to client-side development.

In this chapter, we're going to explore the evolution of the Model-View-Controller (MVC) design pattern and get our first look at how the Backbone.js framework allows us to apply this pattern to client-side development.

MVC

MVC is an architectural design pattern that encourages improved application organization through a separation of concerns. It enforces the isolation of business data (Models) from user interfaces (Views), with a third component (Controllers) traditionally managing logic, user-input, and coordination of Models and Views. The pattern was originally designed by [Trygve Reenskaug](#) while working on Smalltalk-80 (1979), where it was initially called Model-View-Controller-Editor. MVC was described in depth in "[Design Patterns: Elements of Reusable Object-Oriented Software](#)" (The "GoF" or "Gang of Four" book) in 1994, which played a role in popularizing its use.

Smalltalk-80 MVC

It's important to understand the issues that the original MVC pattern was aiming to solve as it has changed quite heavily since the days of its origin. Back in the 70's, graphical user-interfaces were few and far between. An approach known as [Separated Presentation](#) began to be used as a means to make a clear division between domain objects which modeled concepts in the real world (e.g.,

a photo, a person) and the presentation objects which were rendered to the user's screen.

The Smalltalk-80 implementation of MVC took this concept further and had an objective of separating out the application logic from the user interface. The idea was that decoupling these parts of the application would also allow the reuse of Models for other interfaces in the application. There are some interesting points worth noting about Smalltalk-80's MVC architecture:

- A Domain element was known as a Model and was ignorant of the user-interface (Views and Controllers)
- Presentation was taken care of by the View and the Controller, but there wasn't just a single View and Controller. A View-Controller pair was required for each element being displayed on the screen and so there was no true separation between them
- The Controller's role in this pair was handling user input (such as keypresses and click events) and doing something sensible with them
- The Observer pattern was used to update the View whenever the Model changed

Developers are sometimes surprised when they learn that the Observer pattern (nowadays commonly implemented as a Publish/Subscribe system) was included as a part of MVC's architecture decades ago. In Smalltalk-80's MVC, the View and Controller both observe the Model: anytime the Model changes, the Views react. A simple example of this is an application backed by stock market data - for the application to show real-time information, any change to the data in its Model should result in the View being refreshed instantly.

Martin Fowler has done an excellent job of writing about the [origins](#) of MVC over the years and if you are interested in further historical information about Smalltalk-80's MVC, I recommend reading his work.

MVC Applied To The Web

The web heavily relies on the HTTP protocol, which is stateless. This means that there is not a constantly open connection between the browser and server; each request instantiates a new communication channel between the two. Once the request initiator (e.g., a browser) gets a response the connection is closed. This fact creates a completely different context when compared to the one of the operating systems on which many of the original MVC ideas were developed. The MVC implementation has to conform to the web context.

A typical server-side MVC implementation implements the Front Controller design pattern. This pattern layers an MVC stack behind a single point of entry. This single point of entry means that all HTTP requests (e.g., `http://www.example.com`, `http://www.example.com/whichever-page/`, etc.)

are routed by the server's configuration to the same handler, independent of the URI.

When the Front Controller receives an HTTP request it analyzes it and decides which class (Controller) and method (Action) to invoke. The selected Controller Action takes over and interacts with the appropriate Model to fulfill the request. The Controller receives data back from the Model, loads an appropriate View, injects the Model data into it, and returns the response to the browser.

For example, let's say we have our blog on `www.example.com` and we want to edit an article (with `id=43`) and request `http://www.example.com/article/edit/43`:

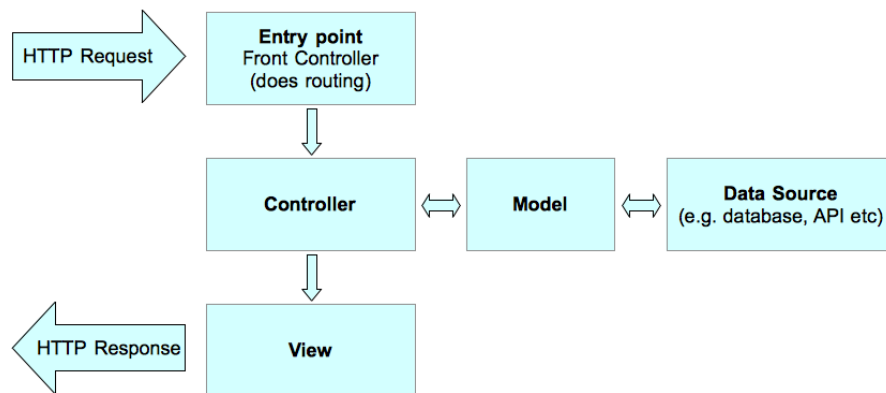
On the server side, the Front Controller would analyze the URL and invoke the Article Controller (corresponding to the `/article/` part of the URI) and its Edit Action (corresponding to the `/edit/` part of the URI). Within the Action there would be a call to, let's say, the Articles Model and its `Articles::getEntry(43)` method (43 corresponding to the `/43` at the end of the URI). This would return the blog article data from the database for edit. The Article Controller would then load the (`article/edit`) View which would include logic for injecting the article's data into a form suitable for editing its content, title, and other (meta) data. Finally, the resulting HTML response would be returned to the browser.

As you can imagine, a similar flow is necessary with POST requests after we press a save button in a form. The POST action URI would look like `/article/save/43`. The request would go through the same Controller, but this time the Save Action would be invoked (due to the `/save/` URI chunk), the Articles Model would save the edited article to the database with `Articles::saveEntry(43)`, and the browser would be redirected to the `/article/edit/43` URI for further editing.

Finally, if the user requested `http://www.example.com/` the Front Controller would invoke the default Controller and Action; e.g., the Index Controller and its Index action. Within Index Action there would be a call to the Articles model and its `Articles::getLastEntries(10)` method which would return the last 10 blog posts. The Controller would load the `blog/index` View which would have basic logic for listing the blog posts.

The picture below shows this typical HTTP request/response lifecycle for server-side MVC:

The Server receives an HTTP request and routes it through a single entry point. At that entry point, the Front Controller analyzes the request and based on it invokes an Action of the appropriate Controller. This process is called routing. The Action Model is asked to return and/or save submitted data. The Model communicates with the data source (e.g., database or API). Once the Model completes its work it returns data to the Controller which then loads the appropriate View. The View executes presentation logic (loops through articles and prints titles, content, etc.) using the supplied data. In the end, an HTTP response is returned to the browser.



The need for fast, complex, and responsive Ajax-powered web applications demands replication of a lot of this logic on the client side, dramatically increasing the size and complexity of the code residing there. Eventually this has brought us to the point where we need MVC (or a similar architecture) implemented on the client side to better structure the code and make it easier to maintain and further extend during the application life-cycle.

And, of course, JavaScript and browsers constitute another context to which the traditional MVC paradigm must be adapted.

MVC In The Browser

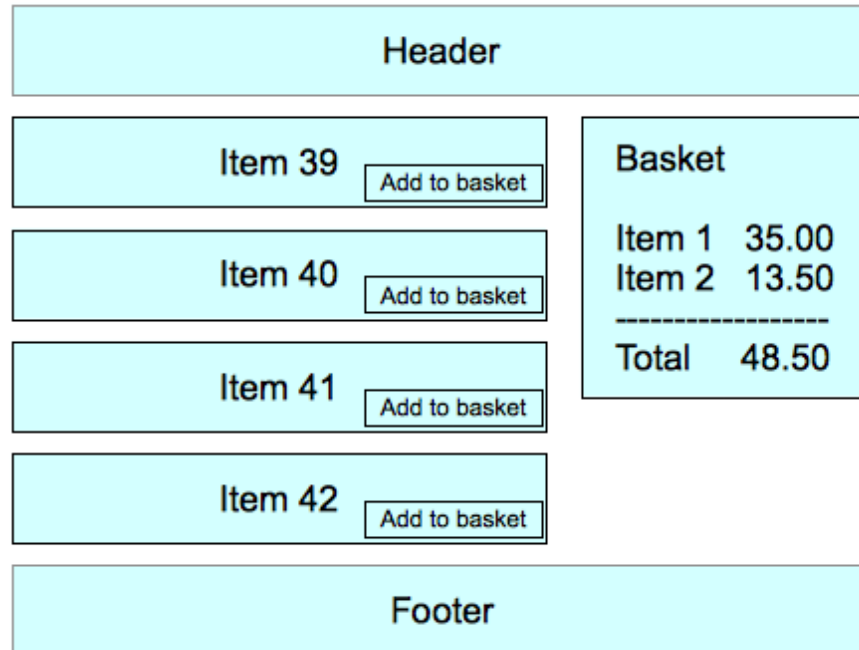
In complex JavaScript single-page web applications (SPA), all application responses (e.g., UI updates) to user inputs are done seamlessly on the client-side. Data fetching and persistence (e.g., saving to a database on a server) are done with Ajax in the background. For silky, slick, and smooth user experiences, the code powering these interactions needs to be well thought out.

The problem

A typical page in a SPA consists of small elements representing logical entities. These entities belong to specific data domains and need to be presented in particular ways on the page.

A good example is a basket in an e-commerce web application which can have items added to it. This basket might be presented to the user in a box in the top right corner of the page (see the picture below):

The basket and its data are presented in HTML. The data and its associated View in HTML changes over time. There was a time when we used jQuery (or a similar DOM manipulation library) and a bunch of Ajax calls and callbacks to keep the two in sync. That often produced code that was not well-structured or easy to maintain. Bugs were frequent and perhaps even unavoidable.



Through evolution, trial and error, and a lot of spaghetti (and not so spaghetti-like) code, JavaScript developers have, in the end, harnessed the ideas of the traditional MVC paradigm. This has led to the development of a number of JavaScript MVC frameworks, including Ember.js, JavaScriptMVC, Angular.js, and of course Backbone.js.

Client-Side MVC - Backbone Style

Let's take our first look at how Backbone.js brings the benefits of MVC to client-side development using a Todo application as our example. We will build on this example in the coming chapters when we explore Backbone's features but for now we will just focus on the core components' relationships to MVC.

Our example will need a div element to which we can attach a list of Todo's. It will also need an HTML template containing a placeholder for a Todo item title and a completion checkbox which can be instantiated for Todo item instances. These are provided by the following HTML:

```
<!doctype html>
<html lang="en">
<head>
```

```

    <meta charset="utf-8">
    <title></title>
    <meta name="description" content="">
  </head>
  <body>
    <div id="todo">
    </div>
    <script type="text/template" id="item-template">
      <div>
        <input id="todo_complete" type="checkbox" <%= completed ? 'checked="checked"' : '' %>
        <%- title %>
      </div>
    </script>
    <script src="jquery-min.js"></script>
    <script src="underscore-min.js"></script>
    <script src="backbone-min.js"></script>
    <script src="example.js"></script>
  </body>
</html>

```

In our Todo application, Backbone Model instances are used to hold the data for each Todo item:

```

// Define a Todo Model
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

// Instantiate the Todo Model with a title, allowing completed attribute
// to default to false
var todo1 = new Todo({
  title: 'Check attributes property of the logged models in the console.'
});

```

Our Todo Model extends Backbone.Model and simply defines default values for two data attributes. As you will discover in the upcoming chapters, Backbone Models provide many more features but this simple Model illustrates that first and foremost a Model is a data container.

Each Todo instance will be rendered on the page by a TodoView:

```

var TodoView = Backbone.View.extend({

  tagName: 'li',

  // Cache the template function for a single item.
  todoTpl: _.template( $('#item-template').html() ),

  events: {
    'dblclick label': 'edit',
    'keypress .edit': 'updateOnEnter',
    'blur .edit': 'close'
  },

  // Re-render the titles of the todo item.
  render: function() {
    this.$el.html( this.todoTpl( this.model.toJSON() ) );
    this.input = this.$('.edit');
    return this;
  },

  edit: function() {
    // executed when todo label is double clicked
  },

  close: function() {
    // executed when todo loses focus
  },

  updateOnEnter: function( e ) {
    // executed on each keypress when in todo edit mode,
    // but we'll wait for enter to get in action
  }
});

// create a view for a todo
var todoView = new TodoView({model: todo1});

```

TodoView is defined by extending Backbone.View and is instantiated with an associated Model. In our example, the `render()` method uses a template to construct the HTML for the Todo item which is placed inside a li element. Each call to `render()` will replace the content of the li element using the current Model data. Thus, a View instance renders the content of a DOM element using the attributes of an associated Model. Later we will see how a View can bind its `render()` method to Model change events, causing the View to re-render whenever the Model changes.

So far, we have seen that `Backbone.Model` implements the Model aspect of MVC and `Backbone.View` implements the View. However, as we noted earlier, Backbone departs from traditional MVC when it comes to Controllers - there is no `Backbone.Controller`!

Instead, the Controller responsibility is addressed within the View. Recall that Controllers respond to requests and perform appropriate actions which may result in changes to the Model and updates to the View. In a single-page application, rather than having requests in the traditional sense, we have events. Events can be traditional browser DOM events (e.g., clicks) or internal application events such as Model changes.

In our `TodoView`, the `events` attribute fulfills the role of the Controller configuration, defining how events occurring within the View's DOM element are to be routed to event-handling methods defined in the View.

While in this instance events help us relate Backbone to the MVC pattern, we will see them playing a much larger role in our SPA applications. `Backbone.Event` is a fundamental Backbone component which is mixed into both `Backbone.Model` and `Backbone.View`, providing them with rich event management capabilities.

This completes our first encounter with Backbone.js. The remainder of this book will explore the many features of the framework which build on these simple constructs. Before moving on, let's take a look at common features of JavaScript MV* frameworks.

Implementation Specifics

Models

- The built-in capabilities of Models vary across frameworks; however, it's common for them to support validation of attributes, where attributes represent the properties of the Model, such as a Model identifier.
- When using Models in real-world applications we generally also need a way of persisting Models. Persistence allows us to edit and update Models with the knowledge that their most recent states will be saved somewhere, for example in a web browser's `localStorage` data-store or synchronized with a database.
- A Model may also have single or multiple Views observing it. Depending on the requirements, a developer might create a single View displaying all Model attributes, or they might create separate Views displaying different attributes. The important point is that the Model doesn't care how these Views are organized, it simply announces updates to its data as necessary through the framework's event system.

- It is not uncommon for modern MVC/MV* frameworks to provide a means of grouping Models together. In Backbone, these groups are called “Collections.” Managing Models in groups allows us to write application logic based on notifications from the group when a Model within the group changes. This avoids the need to manually observe individual Model instances. We’ll see this in action later in the book.
- If you read older texts on MVC, you may come across a description of Models as also managing application “state.” In JavaScript applications state has a specific meaning, typically referring to the current state of a view or sub-view on a user’s screen at a fixed time. State is an important topic to consider when writing single-page applications. In JavaScript apps we care about state memory - that is, recalling a previous state or sharing it with someone else.

Views

- Users interact with Views, which usually means reading and editing Model data. For example, in our Todo application, Todo Model viewing happens in the user interface in the list of all Todo items. Within it, each Todo is rendered with its title and completed checkbox. Model editing is done through an “edit” View where a user who has selected a specific Todo edits its title in a form.
- We define a `render()` utility within our View which is responsible for rendering the contents of the Model using a JavaScript templating engine (provided by Underscore.js) and updating the contents of our View, referenced by `this.el`.
- We then add our `render()` callback as a Model subscriber, so the View can be triggered to update when the Model changes.
- You may wonder where user interaction comes into play here. When users click on a Todo element within the View, it’s not the View’s responsibility to know what to do next. A Controller makes this decision. In Backbone, this is achieved by adding an event listener to the Todo’s element which delegates handling of the click to an event handler.

Templating

In the context of JavaScript frameworks that support MVC/MV*, it is worth looking more closely at JavaScript templating and its relationship to Views.

It has long been considered bad practice (and computationally expensive) to manually create large blocks of HTML markup in-memory through string concatenation. Developers using this technique often find themselves iterating through their data, wrapping it in nested divs and using outdated techniques

such as `document.write` to inject the ‘template’ into the DOM. This approach often means keeping scripted markup inline with standard markup, which can quickly become difficult to read and maintain, especially when building large applications.

JavaScript templating libraries (such as Handlebars.js or Mustache) are often used to define templates for Views as HTML markup containing template variables. These template blocks can be either stored externally or within script tags with a custom type (e.g ‘text/template’). Variables are delimited using a variable syntax (e.g `<%= title %>`).

JavaScript template libraries typically accept data in a number of formats, including JSON; a serialisation format that is always a string. The grunt work of populating templates with data is generally taken care of by the framework itself. This has several benefits, particularly when opting to store templates externally which enables applications to load templates dynamically on an as-needed basis.

Let’s compare two examples of HTML templates. One is implemented using the popular Handlebars.js library, and the other uses Underscore’s ‘microtemplates’.

Handlebars.js:

```
<div class="view">
  <input class="toggle" type="checkbox" {{#if completed}} "checked" {{/if}}>
  <label>{{title}}</label>
  <button class="destroy"></button>
</div>
<input class="edit" value="{{title}}">
```

Underscore.js Microtemplates:

```
<div class="view">
  <input class="toggle" type="checkbox" <%= completed ? 'checked' : '' %>>
  <label><%- title %></label>
  <button class="destroy"></button>
</div>
<input class="edit" value="<%= title %>">
```

You may also use double curly brackets (i.e `{{}}`) (or any other tag you feel comfortable with) in Microtemplates. In the case of curly brackets, this can be done by setting the Underscore `templateSettings` attribute as follows:

```
_.templateSettings = { interpolate : /\{\{(.+)\}\}/g };
```

A note on Navigation and State

It is also worth noting that in classical web development, navigating between independent views required the use of a page refresh. In single-page JavaScript applications, however, once data is fetched from a server via Ajax, it can be dynamically rendered in a new view within the same page. Since this doesn't automatically update the URL, the role of navigation thus falls to a "router", which assists in managing application state (e.g., allowing users to bookmark a particular view they have navigated to). As routers are neither a part of MVC nor present in every MVC-like framework, I will not be going into them in greater detail in this section.

Controllers In our Todo application, a Controller would be responsible for handling changes the user made in the edit View for a particular Todo, updating a specific Todo Model when a user has finished editing.

It's with Controllers that most JavaScript MVC frameworks depart from the traditional interpretation of the MVC pattern. The reasons for this vary, but in my opinion, Javascript framework authors likely initially looked at server-side interpretations of MVC (such as Ruby on Rails), realized that the approach didn't translate 1:1 on the client-side, and so re-interpreted the C in MVC to solve their state management problem. This was a clever approach, but it can make it hard for developers coming to MVC for the first time to understand both the classical MVC pattern and the "proper" role of Controllers in other JavaScript frameworks.

So does Backbone.js have Controllers? Not really. Backbone's Views typically contain "Controller" logic, and Routers are used to help manage application state, but neither are true Controllers according to classical MVC.

In this respect, contrary to what might be mentioned in the official documentation or in blog posts, Backbone isn't truly an MVC framework. It's in fact better to see it a member of the MV* family which approaches architecture in its own way. There is of course nothing wrong with this, but it is important to distinguish between classical MVC and MV* should you be relying on discussions of MVC to help with your Backbone projects.

What does MVC give us?

To summarize, the separation of concerns in MVC facilitates modularization of an application's functionality and enables:

- Easier overall maintenance. When updates need to be made to the application it is clear whether the changes are data-centric, meaning changes to Models and possibly Controllers, or merely visual, meaning changes to Views.
- Decoupling Models and Views means that it's straight-forward to write unit tests for business logic

- Duplication of low-level Model and Controller code is eliminated across the application
- Depending on the size of the application and separation of roles, this modularity allows developers responsible for core logic and developers working on the user-interfaces to work simultaneously

Delving Deeper into MVC

Right now, you likely have a basic understanding of what the MVC pattern provides, but for the curious, we'll explore it a little further.

The GoF (Gang of Four) do not refer to MVC as a design pattern, but rather consider it a “set of classes to build a user interface.” In their view, it's actually a variation of three other classical design patterns: the Observer (Publish/Subscribe), Strategy, and Composite patterns. Depending on how MVC has been implemented in a framework, it may also use the Factory and Decorator patterns. I've covered some of these patterns in my other free book, “JavaScript Design Patterns For Beginners” if you would like to read about them further.

As we've discussed, Models represent application data, while Views handle what the user is presented on screen. As such, MVC relies on Publish/Subscribe for some of its core communication (something that surprisingly isn't covered in many articles about the MVC pattern). When a Model is changed it “publishes” to the rest of the application that it has been updated. The “subscriber,” generally a Controller, then updates the View accordingly. The observer-viewer nature of this relationship is what facilitates multiple Views being attached to the same Model.

For developers interested in knowing more about the decoupled nature of MVC (once again, depending on the implementation), one of the goals of the pattern is to help define one-to-many relationships between a topic and its observers. When a topic changes, its observers are updated. Views and Controllers have a slightly different relationship. Controllers facilitate Views' responses to different user input and are an example of the Strategy pattern.

Summary

Having reviewed the classical MVC pattern, you should now understand how it allows developers to cleanly separate concerns in an application. You should also now appreciate how JavaScript MVC frameworks may differ in their interpretation of MVC, and how they share some of the fundamental concepts of the original pattern.

When reviewing a new JavaScript MVC/MV* framework, remember - it can be useful to step back and consider how it's opted to approach Models, Views, Controllers or other alternatives, as this can better help you understand how the framework is intended to be used.

Further reading

If you are interested in learning more about the variation of MVC which Backbone.js uses, please see the MVP (Model-View-Presenter) section in the appendix.

Fast facts

Backbone.js

- Core components: Model, View, Collection, Router. Enforces its own flavor of MV*
- Event-driven communication between Views and Models. As we'll see, it's relatively straight-forward to add event listeners to any attribute in a Model, giving developers fine-grained control over what changes in the View
- Supports data bindings through manual events or a separate Key-value observing (KVO) library
- Support for RESTful interfaces out of the box, so Models can be easily tied to a backend
- Extensive eventing system. It's [trivial](#) to add support for pub/sub in Backbone
- Prototypes are instantiated with the `new` keyword, which some developers prefer
- Agnostic about templating frameworks, however Underscore's micro-templating is available by default. Backbone works well with libraries like Handlebars
- Doesn't support deeply nested Models, though there are Backbone plugins such as [Backbone-relational](#) which can help
- Clear and flexible conventions for structuring applications. Backbone doesn't force usage of all of its components and can work with only those needed.

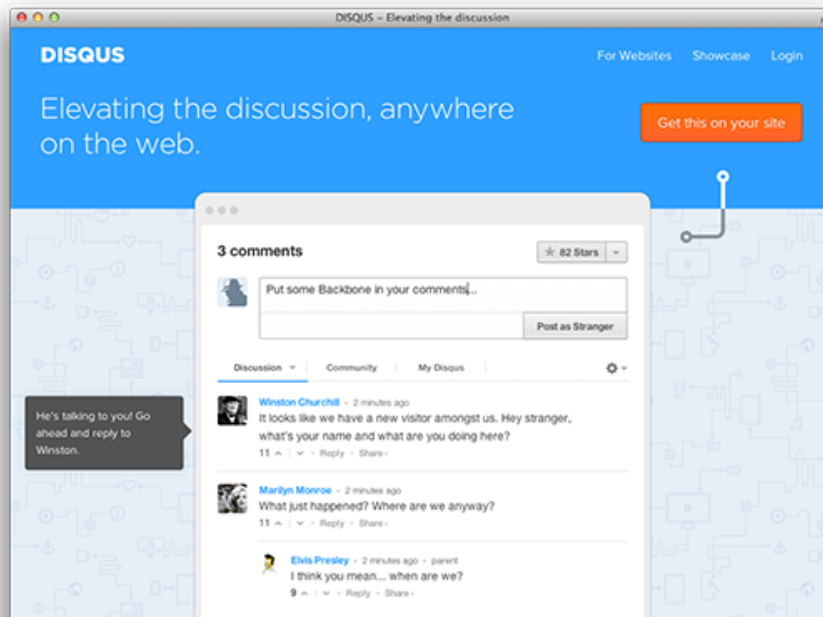
Used by

Disqus

Disqus chose Backbone.js to power the latest version of their commenting widget. They felt it was the right choice for their distributed web app, given Backbone's small footprint and ease of extensibility.

Khan Academy

Offering a web app that aims to provide free world-class education to anyone anywhere, Khan use Backbone to keep their frontend code both modular and organized.



MetaLab

MetaLab created Flow, a task management app for teams using Backbone. Their workspace uses Backbone to create task views, activities, accounts, tags and more.

Walmart Mobile

Walmart chose Backbone to power their mobile web applications, creating two new extension frameworks in the process - Thorax and Lumbar. We'll be discussing both of these later in the book.

AirBnb

Airbnb developed their mobile web app using Backbone and now use it across many of their products.

Code School

Code School's course challenge app is built from the ground up using Backbone, taking advantage of all the pieces it has to offer: routers, collections, models and complex event handling.

The screenshot displays the Khan Academy Exercise Dashboard in a web browser. The page features a green header with the Khan Academy logo, a search bar, and navigation links for WATCH, PRACTICE, COACH, VOLUNTEER, and ABOUT. A user profile for Ben Alpert is visible in the top right corner.

On the left side, there is a 'Knowledge Map' section with a 'Vital Statistics' link. It includes a 'Suggested' list of topics with progress bars:

- Equation of a line (0/11 skills)
- Factors and multiples (4/11 skills)
- 4-digit subtraction with borrowing
- Least common multiple and greatest common divisor word problems
- Congruent angles
- Line graph intuition
- Direct and inverse variation

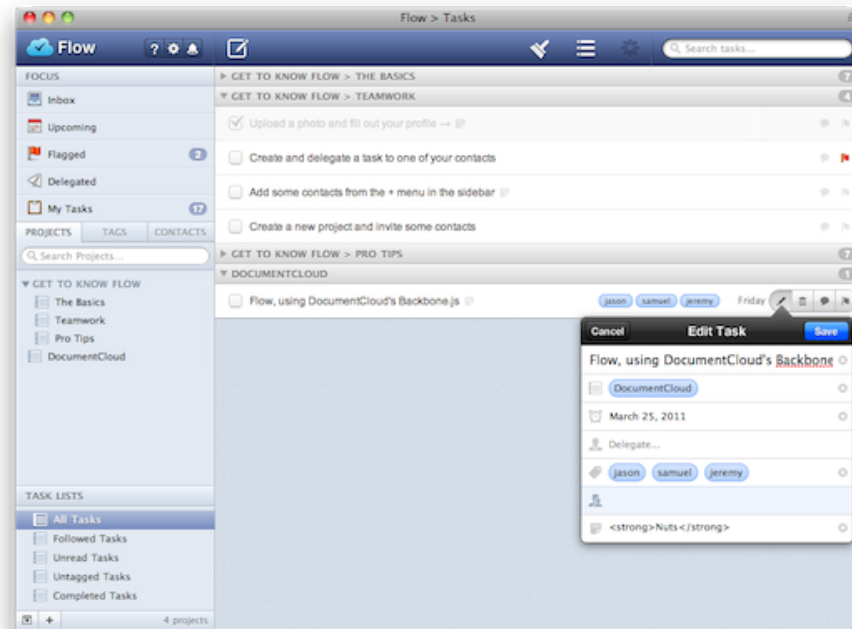
A 'Show All' link is at the bottom of this list.

The main area of the dashboard is a 'Knowledge Map' visualized as a star map. It shows a hierarchy of skills:

- Top level: 1-digit subtraction (Proficient)
- Second level: 2 and 3-digit subtraction (Suggested)
- Third level: Addition with carrying (Suggested), Addition and subtraction word problems (Suggested), Subtraction with borrowing (Suggested)
- Bottom level: 4-digit addition with carrying (Suggested), 4-digit subtraction with borrowing (Suggested)

Each skill is represented by a star icon with a corresponding label. The background of the map is a dark space with colorful stars.

The footer contains links for Contact, Report a Problem, Blog, FAQ, Careers, ToS, Privacy Policy, and Store, along with social media icons and a copyright notice for 2012 Khan Academy.



Backbone Basics

In this section, you'll learn the essentials of Backbone's models, views, collections, events, and routers. This isn't meant as a replacement for the official documentation, but it will help you understand many of the core concepts behind Backbone before you start building applications with it.

Models

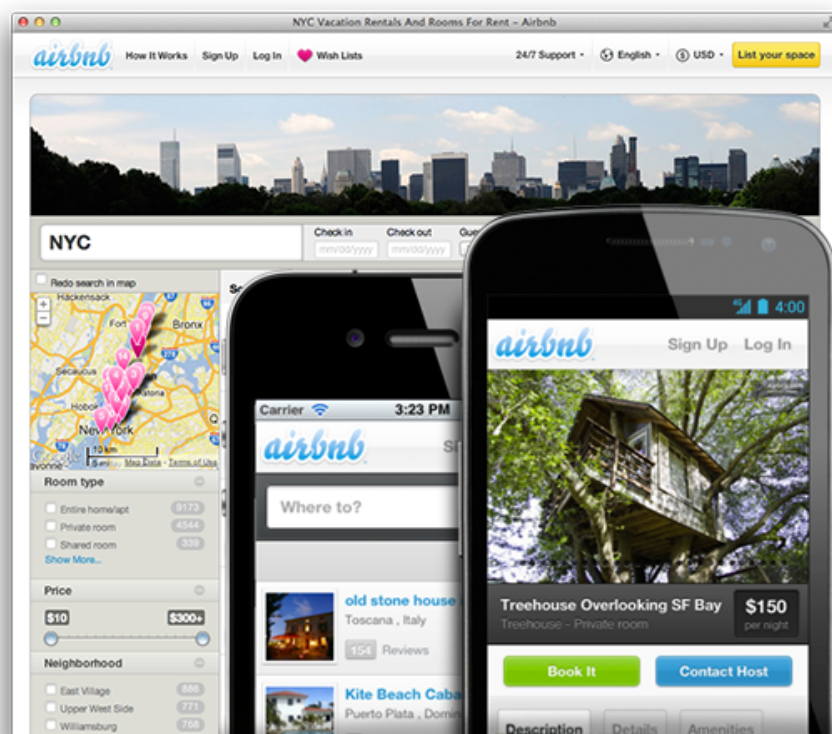
Backbone models contain data for an application as well as the logic around this data. For example, we can use a model to represent the concept of a todo item including its attributes like title (todo content) and completed (current state of the todo).

Models can be created by extending `Backbone.Model` as follows:

```
var Todo = Backbone.Model.extend({});

// We can then create our own concrete instance of a (Todo) model
// with no values at all:
var todo1 = new Todo();
```





code school
Home
Code TV
My Account
Support
Sign Up

Functions III

Given the code below, change the `greet` function so that it accepts two arguments instead of just one. It should alert both arguments, separated by a single white space.

CoffeeScript
live js
Hints

Next Hint (+75 Points)
1 Hint Remaining

- When declaring functions that take arguments, you must use parens

```
myFunction = (arg1, arg2) -> myOtherFunction(arg1, arg2)
```
- When calling functions that take arguments, you can omit the parens

```
myFunction = (arg1, arg2) -> myOtherFunction arg1, arg2
```

Submit Code

0
Total Points

Level 1
Change Level

- Variable Assignment
- Functions
- Functions II
- Functions III
- Functions IV
- Functions V
- Sum Function

Rematch Video
Shortcuts
Download Slides
Download Video

```

// Following logs: {}
console.log(JSON.stringify(todo1));

// or with some arbitrary data:
var todo2 = new Todo({
  title: 'Check the attributes of both model instances in the console.',
  completed: true
});

// Following logs: {"title":"Check the attributes of both model instances in the console.",
console.log(JSON.stringify(todo2));

```

Initialization The `initialize()` method is called when a new instance of a model is created. Its use is optional; however you'll see why it's good practice to use it below.

```

var Todo = Backbone.Model.extend({
  initialize: function(){
    console.log('This model has been initialized.');
```

Default values

There are times when you want your model to have a set of default values (e.g., in a scenario where a complete set of data isn't provided by the user). This can be set using a property called `defaults` in your model.

```

var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

// Now we can create our concrete instance of the model
// with default values as follows:
var todo1 = new Todo();

// Following logs: {"title":"","completed":false}
console.log(JSON.stringify(todo1));

```

```

// Or we could instantiate it with some of the attributes (e.g., with custom title):
var todo2 = new Todo({
  title: 'Check attributes of the logged models in the console.'
});

// Following logs: {"title":"Check attributes of the logged models in the console.,"completed":false}
console.log(JSON.stringify(todo2));

// Or override all of the default attributes:
var todo3 = new Todo({
  title: 'This todo is done, so take no action on this one.',
  completed: true
});

// Following logs: {"title":"This todo is done, so take no action on this one.,"completed":true}
console.log(JSON.stringify(todo3));

```

Getters & Setters Model.get()

Model.get() provides easy access to a model's attributes.

```

var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

var todo1 = new Todo();
console.log(todo1.get('title')); // empty string
console.log(todo1.get('completed')); // false

var todo2 = new Todo({
  title: "Retrieved with model's get() method.",
  completed: true
});
console.log(todo2.get('title')); // Retrieved with model's get() method.
console.log(todo2.get('completed')); // true

```

If you need to read or clone all of a model's data attributes, use its `toJSON()` method. This method returns a copy of the attributes as an object (not a JSON string despite its name). (When `JSON.stringify()` is passed an object with a `toJSON()` method, it stringifies the return value of `toJSON()` instead of the

original object. The examples in the previous section took advantage of this feature when they called `JSON.stringify()` to log model instances.)

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

var todo1 = new Todo();
var todo1Attributes = todo1.toJSON();
// Following logs: {"title":"","completed":false}
console.log(todo1Attributes);

var todo2 = new Todo({
  title: "Try these examples and check results in console.",
  completed: true
});

// logs: {"title":"Try these examples and check results in console.,"completed":true}
console.log(todo2.toJSON());
```

Model.set()

`Model.set()` allows us to pass attributes into an instance of our model. Backbone uses `Model.set()` to know when to broadcast to its listeners that the model's data has changed.

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

// Setting the value of attributes via instantiation
var myTodo = new Todo({
  title: "Set through instantiation."
});
console.log('Todo title: ' + myTodo.get('title')); // Todo title: Set through instantiation.
console.log('Completed: ' + myTodo.get('completed')); // Completed: false

// Set single attribute value at a time through Model.set():
```

```

myTodo.set("title", "Title attribute set through Model.set().");
console.log('Todo title: ' + myTodo.get('title')); // Todo title: Title attribute set through
console.log('Completed: ' + myTodo.get('completed')); // Completed: false

// Set map of attributes through Model.set():
myTodo.set({
  title: "Both attributes set through Model.set().",
  completed: true
});
console.log('Todo title: ' + myTodo.get('title')); // Todo title: Both attributes set through
console.log('Completed: ' + myTodo.get('completed')); // Completed: true

```

Direct access

Models expose an `.attributes` attribute which represents an internal hash containing the state of that model. This is generally in the form of a JSON object similar to the model data you might find on the server but can take other forms.

Setting values through the `.attributes` attribute on a model bypasses triggers bound to the model. Additionally you can pass `{silent: true}`, which silences individual “change:attr” events entirely, but can build up and then hit in unexpected ways later on.

Remember where possible it is best practice to use `Model.set()`, or direct instantiation as explained earlier.

Listening for changes to your model If you want to receive a notification when a Backbone model changes you can bind a listener to the model for its change event. A convenient place to add listeners is in the `initialize()` function as shown below:

```

var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  },
  initialize: function(){
    console.log('This model has been initialized.');
```

```

    this.on('change', function(){
      console.log('- Values for this model have changed.');
```

```

    });
  }
});

```

```

var myTodo = new Todo();

myTodo.set('title', 'The listener is triggered whenever an attribute value changes.');
```

console.log('Title has changed: ' + myTodo.get('title'));

```

myTodo.set('completed', true);
console.log('Completed has changed: ' + myTodo.get('completed'));

myTodo.set({
  title: 'Changing more than one attribute at the same time only triggers the listener once.',
  completed: true
});

// Above logs:
// This model has been initialized.
// - Values for this model have changed.
// Title has changed: The listener is triggered whenever an attribute value changes.
// - Values for this model have changed.
// Completed has changed: true
// - Values for this model have changed.
```

You can also listen for changes to individual attributes in a Backbone model. In the following example, we log a message whenever a specific attribute (the title of our Todo model) is altered.

```

var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  },

  initialize: function(){
    console.log('This model has been initialized.');
```

this.on('change:title', function(){

console.log('Title value for this model has changed.');

```

  });
},

  setTitle: function(newTitle){
    this.set({ title: newTitle });
  }
});
```

```

var myTodo = new Todo();

// Both of the following changes trigger the listener:
myTodo.set('title', 'Check what\'s logged.');
myTodo.setTitle('Go fishing on Sunday.');

// But, this change type is not observed, so no listener is triggered:
myTodo.set('completed', true);
console.log('Todo set as completed: ' + myTodo.get('completed'));

// Above logs:
// This model has been initialized.
// Title value for this model has changed.
// Title value for this model has changed.
// Todo set as completed: true

```

Validation Backbone supports model validation through `Model.validate()`, which allows checking the attribute values for a model prior to setting them.

Validation functions can be as simple or complex as necessary. If the attributes provided are valid, nothing should be returned from `.validate()`. If they are invalid, an error value should be returned instead. If an error is returned, an `invalid` event will occur and listeners will be passed the unmodified model and the error value.

Validation automatically occurs when the model is persisted using the `.save()` method or when `.set()` is called with the `validate` option set to `true`. A basic validation example can be seen below:

```

var Todo = Backbone.Model.extend({
  defaults: {
    completed: false
  },

  validate: function(attrs){
    if(attrs.title === undefined){
      return "Remember to set a title for your todo.";
    }
  },

  initialize: function(){
    console.log('This model has been initialized.');
    this.on("invalid", function(model, error){
      console.log(error);
    });
  }
}

```

```
});
```

```
var myTodo = new Todo();  
myTodo.set('completed', true, {validate: true}); // logs: Remember to set a title for your  
console.log('completed: ' + myTodo.get('completed')); // completed: false
```

Note: the `attributes` object passed to the `validate` function represents what the attributes would be after completing the current `set()` or `save()`. This object is distinct from the current attributes of the model and from the parameters passed to the operation. Since it is created by shallow copy, it is not possible to change any Number, String, or Boolean attribute of the input within the function, but it *is* possible to change attributes in nested objects.

An example of this (by @fivetanley) is available [here](#).

Views

Views in Backbone don't contain the HTML markup for your application; they contain the logic behind the presentation of the model's data to the user. This is usually achieved using JavaScript templating (e.g., Underscore Microtemplates, Mustache, jQuery-tmpl, etc.). A view's `render()` method can be bound to a model's `change()` event, enabling the view to instantly reflect model changes without requiring a full page refresh.

Creating new views Creating a new view is relatively straight-forward and similar to creating new models. To create a new View, simply extend `Backbone.View`. We introduced the sample `TodoView` below in the previous chapter; now let's take a closer look at how it works.

```
var TodoView = Backbone.View.extend({  
  
  tagName: 'li',  
  
  // Cache the template function for a single item.  
  todoTpl: _.template( "An example template" ),  
  
  events: {  
    'dblclick label': 'edit',  
    'keypress .edit': 'updateOnEnter',  
    'blur .edit': 'close'  
  },  
  
  // Re-render the titles of the todo item.  
  render: function() {
```



```

        this.$el.html( this.todoTpl( this.model.toJSON() ) );
        this.input = this.$('.edit');
        return this;
    },

    edit: function() {
        // executed when todo label is double clicked
    },

    close: function() {
        // executed when todo loses focus
    },

    updateOnEnter: function( e ) {
        // executed on each keypress when in todo edit mode,
        // but we'll wait for enter to get in action
    }
});

var todoView = new TodoView();

// log reference to a DOM element that corresponds to the view instance
console.log(todoView.el); // logs <li></li>

```

What is `el`? The central property of a view is `el` (the value logged in the last statement of the example). What is `el` and how is it defined?

`el` is basically a reference to a DOM element and all views must have one. Views can use `el` to compose their element's content and then insert it into the DOM all at once, which makes for faster rendering because the browser performs the minimum required number of reflows and repaints.

There are two ways to associate a DOM element with a view: a new element can be created for the view and subsequently added to the DOM or a reference can be made to an element which already exists in the page.

If you want to create a new element for your view, set any combination of the following properties on the view: `tagName`, `id`, and `className`. A new element will be created for you by the framework and a reference to it will be available at the `el` property. If nothing is specified `tagName` defaults to `div`.

In the example above, `tagName` is set to `'li'`, resulting in creation of an `li` element. The following example creates a `ul` element with `id` and `class` attributes:

```

var TodosView = Backbone.View.extend({
    tagName: 'ul', // required, but defaults to 'div' if not set
    className: 'container', // optional, you can assign multiple classes to this property like

```

```

    id: 'todos', // optional
  });

var todosView = new TodosView();
console.log(todosView.el); // logs <ul id="todos" class="container"></ul>

```

The above code creates the DOM element below but doesn't append it to the DOM.

```
<ul id="todos" class="container"></ul>
```

If the element already exists in the page, you can set `el` as a CSS selector that matches the element.

```
el: '#footer'
```

Alternatively, you can set `el` to an existing element when creating the view:

```
var todosView = new TodosView({el: $('#footer')});
```

el and \$()

View logic often needs to invoke jQuery or Zepto functions on the `el` element and elements nested within it. Backbone makes it easy to do so by defining the `$el` property and `$()` function. The `view.$el` property is equivalent to `$(view.el)` and `view.$(selector)` is equivalent to `$(view.el).find(selector)`. In our `TodosView` example's `render` method, we see `this.$el` used to set the HTML of the element and `this.$()` used to find subelements of class `'edit'`.

setElement

If you need to apply an existing Backbone view to a different DOM element `setElement` can be used for this purpose. Overriding `this.el` needs to both change the DOM reference and re-bind events to the new element (and unbind from the old).

`setElement` will create a cached `$el` reference for you, moving the delegated events for a view from the old element to the new one.

```

// We create two DOM elements representing buttons
// which could easily be containers or something else
var button1 = $('<button></button>');
var button2 = $('<button></button>');

// Define a new view

```

```

var View = Backbone.View.extend({
  events: {
    click: function(e) {
      console.log(view.el === e.target);
    }
  }
});

// Create a new instance of the view, applying it
// to button1
var view = new View({el: button1});

// Apply the view to button2 using setElement
view.setElement(button2);

button1.trigger('click');
button2.trigger('click'); // returns true

```

The “el” property represents the markup portion of the view that will be rendered; to get the view to actually render to the page, you need to add it as a new element or append it to an existing element.

```

// We can also provide raw markup to setElement
// as follows (just to demonstrate it can be done):
var view = new Backbone.View;
view.setElement('<p><a><b>test</b></a></p>');
view.$('a b').html(); // outputs "test"

```

Understanding render()

`render()` is an optional function that defines the logic for rendering a template. We’ll use Underscore’s micro-templating in these examples, but remember you can use other templating frameworks if you prefer. Our example will reference the following HTML markup:

```

<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
  <meta name="description" content="">
</head>
<body>
  <div id="todo">

```

```

</div>
<script type="text/template" id="item-template">
  <div>
    <input id="todo_complete" type="checkbox" <%= completed ? 'checked="checked"' : '' %>
    <%= title %>
  </div>
</script>
<script src="underscore-min.js"></script>
<script src="backbone-min.js"></script>
<script src="jquery-min.js"></script>
<script src="example.js"></script>
</body>
</html>

```

The `_.template` method in Underscore compiles JavaScript templates into functions which can be evaluated for rendering. In the `TodoView`, I'm passing the markup from the template with id `item-template` to `_.template()` to be compiled and stored in the `todoTpl` property when the view is created.

The `render()` method uses this template by passing it the `toJSON()` encoding of the attributes of the model associated with the view. The template returns its markup after using the model's title and completed flag to evaluate the expressions containing them. I then set this markup as the HTML content of the `e1` DOM element using the `$el` property.

Presto! This populates the template, giving you a data-complete set of markup in just a few short lines of code.

A common Backbone convention is to return `this` at the end of `render()`. This is useful for a number of reasons, including:

- Making views easily reusable in other parent views
- Creating a list of elements without rendering and painting each of them individually, only to be drawn once the entire list is populated.

Let's try to implement the latter of these. The `render` method of a simple `ListView` which doesn't use an `ItemView` for each item could be written:

```

var ListView = Backbone.View.extend({
  render: function(){
    this.$el.html(this.model.toJSON());
  }
});

```

Simple enough. Let's now assume a decision is made to construct the items using an `ItemView` to provide enhanced behaviour to our list. The `ItemView` could be written:

```

var ItemView = Backbone.View.extend({
  events: {},
  render: function(){
    this.$el.html(this.model.toJSON());
    return this;
  }
});

```

Note the usage of `return this;` at the end of `render`. This common pattern enables us to reuse the view as a sub-view. We can also use it to pre-render the view prior to rendering. Using this requires that we make a change to our `ListView`'s `render` method as follows:

```

var ListView = Backbone.View.extend({
  render: function(){

    // Assume our model exposes the items we will
    // display in our list
    var items = this.model.get('items');

    // Loop through each our items using the Underscore
    // _.each iterator
    _.each(items, function(item){

      // Create a new instance of the ItemView, passing
      // it a specific model item
      var itemView = new ItemView({ model: item });
      // The itemView's DOM element is appended after it
      // has been rendered. Here, the 'return this' is helpful
      // as the itemView renders its model. Later, we ask for
      // it's output ("el")
      this.$el.append( itemView.render().el );
    }, this);
  }
});

```

The events hash

The Backbone `events` hash allows us to attach event listeners to either `el`-relative custom selectors, or directly to `el` if no selector is provided. An event takes the form of a key-value pair `'eventName selector': 'callbackFunction'` and a number of DOM event-types are supported, including `click`, `submit`, `mouseover`, `dblclick` and more.

```
// A sample view
var TodoView = Backbone.View.extend({
  tagName: 'li',

  // with an events hash containing DOM events
  // specific to an item:
  events: {
    'click .toggle': 'toggleCompleted',
    'dblclick label': 'edit',
    'click .destroy': 'clear',
    'blur .edit': 'close'
  },

```

What isn't instantly obvious is that while Backbone uses jQuery's `.delegate()` underneath, it goes further by extending it so that `this` always refers to the current view object within callback functions. The only thing to really keep in mind is that any string callback supplied to the events attribute must have a corresponding function with the same name within the scope of your view.

The declarative, delegated jQuery events means that you don't have to worry about whether a particular element has been rendered to the DOM yet or not. Usually with jQuery you have to worry about "presence or absence in the DOM" all the time when binding events.

In our `TodoView` example, the `edit` callback is invoked when the user double-clicks a label element within the `el` element, `updateOnEnter` is called for each keypress in an element with class `'edit'`, and `close` executes when an element with class `'edit'` loses focus. Each of these callback functions can use `this` to refer to the `TodoView` object.

Note that you can also bind methods yourself using `_.bind(this.viewEvent, this)`, which is effectively what the value in each event's key-value pair is doing. Below we use `_.bind` to re-render our view when a model changes.

```
var TodoView = Backbone.View.extend({
  initialize: function() {
    this.model.bind('change', _.bind(this.render, this));
  }
});
```

`_.bind` only works on one method at a time, but supports currying and as it returns the bound function means that you can use `_.bind` on an anonymous function.

Collections

Collections are sets of Models and are created by extending `Backbone.Collection`.

Normally, when creating a collection you'll also want to define a property specifying the type of model that your collection will contain, along with any instance properties required.

In the following example, we create a `TodoCollection` that will contain our `Todo` models:

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo
});

var myTodo = new Todo({title: 'Read the whole book', id: 2});

// pass array of models on collection instantiation
var todos = new TodosCollection([myTodo]);
console.log("Collection size: " + todos.length); // Collection size: 1
```

Adding and Removing Models The preceding example populated the collection using an array of models when it was instantiated. After a collection has been created, models can be added and removed using the `add()` and `remove()` methods:

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo,
});

var a = new Todo({ title: 'Go to Jamaica.'}),
    b = new Todo({ title: 'Go to China.'}),
```

```

    c = new Todo({ title: 'Go to Disneyland.'});

var todos = new TodosCollection([a,b]);
console.log("Collection size: " + todos.length);
// Logs: Collection size: 2

todos.add(c);
console.log("Collection size: " + todos.length);
// Logs: Collection size: 3

todos.remove([a,b]);
console.log("Collection size: " + todos.length);
// Logs: Collection size: 1

todos.remove(c);
console.log("Collection size: " + todos.length);
// Logs: Collection size: 0

```

Note that `add()` and `remove()` accept both individual models and lists of models.

Retrieving Models There are a few different ways to retrieve a model from a collection. The most straight-forward is to use `Collection.get()` which accepts a single id as follows:

```

var myTodo = new Todo({title:'Read the whole book', id: 2});

// pass array of models on collection instantiation
var todos = new TodosCollection([myTodo]);

var todo2 = todos.get(2);

// Models, as objects, are passed by reference
console.log(todo2 === myTodo); // true

```

In client-server applications, collections contain models obtained from the server. Anytime you're exchanging data between the client and a server, you will need a way to uniquely identify models. In Backbone, this is done using the `id`, `cid`, and `idAttribute` properties.

Each model in Backbone has an `id`, which is a unique identifier that is either an integer or string (e.g., a UUID). Models also have a `cid` (client id) which is automatically generated by Backbone when the model is created. Either identifier can be used to retrieve a model from a collection.

The main difference between them is that the `cid` is generated by Backbone; it is helpful when you don't have a true id - this may be the case if your model has yet to be saved to the server or you aren't saving it to a database.

The `idAttribute` is the identifying attribute of the model returned from the server (i.e., the `id` in your database). This tells Backbone which data field from the server should be used to populate the `id` property (think of it as a mapper). By default, it assumes `id`, but this can be customized as needed. For instance, if your server sets a unique attribute on your model named “`userId`” then you would set `idAttribute` to “`userId`” in your model definition.

The value of a model’s `idAttribute` should be set by the server when the model is saved. After this point you shouldn’t need to set it manually, unless further control is required.

Internally, `Backbone.Collection` contains an array of models enumerated by their `id` property, if the model instances happen to have one. When `collection.get(id)` is called, this array is checked for existence of the model instance with the corresponding `id`.

```
// extends the previous example

var todoCid = todos.get(todo2.cid);

// As mentioned in previous example,
// models are passed by reference
console.log(todoCid === myTodo); // true
```

Listening for events As collections represent a group of items, we can listen for `add` and `remove` events which occur when models are added to or removed from a collection. Here’s an example:

```
var TodosCollection = new Backbone.Collection();

TodosCollection.on("add", function(todo) {
  console.log("I should " + todo.get("title") + ". Have I done it before? " + (todo.get("completed") === true));
});

TodosCollection.add([
  { title: 'go to Jamaica', completed: false },
  { title: 'go to China', completed: false },
  { title: 'go to Disneyland', completed: true }
]);

// The above logs:
// I should go to Jamaica. Have I done it before? No.
// I should go to China. Have I done it before? No.
// I should go to Disneyland. Have I done it before? Yeah!
```

In addition, we’re also able to bind to a `change` event to listen for changes to any of the models in the collection.

```

var TodosCollection = new Backbone.Collection();

// log a message if a model in the collection changes
TodosCollection.on("change:title", function(model) {
    console.log("Changed my mind! I should " + model.get('title'));
});

TodosCollection.add([
    { title: 'go to Jamaica.', completed: false, id: 3 },
]);

var myTodo = TodosCollection.get(3);

myTodo.set('title', 'go fishing');
// Logs: Changed my mind! I should go fishing

```

Resetting/Refreshing Collections Rather than adding or removing models individually, you might want to update an entire collection at once. `Collection.set()` takes an array of models and performs the necessary add, remove, and change operations required to update the collection.

```

var TodosCollection = new Backbone.Collection();

TodosCollection.add([
    { id: 1, title: 'go to Jamaica.', completed: false },
    { id: 2, title: 'go to China.', completed: false },
    { id: 3, title: 'go to Disneyland.', completed: true }
]);

// we can listen for add/change/remove events
TodosCollection.on("add", function(model) {
    console.log("Added " + model.get('title'));
});

TodosCollection.on("remove", function(model) {
    console.log("Removed " + model.get('title'));
});

TodosCollection.on("change:completed", function(model) {
    console.log("Completed " + model.get('title'));
});

TodosCollection.set([
    { id: 1, title: 'go to Jamaica.', completed: true },
    { id: 2, title: 'go to China.', completed: false },

```

```
    { id: 4, title: 'go to Disney World.', completed: false }
  ]);
```

```
// Above logs:
// Removed go to Disneyland.
// Completed go to Jamaica.
// Added go to Disney World.
```

If you need to simply replace the entire content of the collection then `Collection.reset()` can be used:

```
var TodosCollection = new Backbone.Collection();

// we can listen for reset events
TodosCollection.on("reset", function() {
  console.log("Collection reset.");
});

TodosCollection.add([
  { title: 'go to Jamaica.', completed: false },
  { title: 'go to China.', completed: false },
  { title: 'go to Disneyland.', completed: true }
]);

console.log('Collection size: ' + TodosCollection.length); // Collection size: 3

TodosCollection.reset([
  { title: 'go to Cuba.', completed: false }
]);
// Above logs 'Collection reset.'

console.log('Collection size: ' + TodosCollection.length); // Collection size: 1
```

Another useful tip is to use `reset` with no arguments to clear out a collection completely. This is handy when dynamically loading a new page of results where you want to blank out the current page of results.

```
myCollection.reset();
```

Note that using `Collection.reset()` doesn't fire any `add` or `remove` events. A `reset` event is fired instead as shown in the previous example. The reason you might want to use this is to perform super-optimized rendering in extreme cases where individual events are too expensive.

Underscore utility functions Backbone takes full advantage of its hard dependency on Underscore by making many of its utilities directly available on collections:

forEach: iterate over collections

```
var Todos = new Backbone.Collection();

Todos.add([
  { title: 'go to Belgium.', completed: false },
  { title: 'go to China.', completed: false },
  { title: 'go to Austria.', completed: true }
]);

// iterate over models in the collection
Todos.forEach(function(model){
  console.log(model.get('title'));
});
// Above logs:
// go to Belgium.
// go to China.
// go to Austria.
```

sortBy(): sort a collection on a specific attribute

```
// sort collection
var sortedByAlphabet = Todos.sortBy(function (todo) {
  return todo.get("title").toLowerCase();
});

console.log("- Now sorted: ");

sortedByAlphabet.forEach(function(model){
  console.log(model.get('title'));
});
// Above logs:
// go to Austria.
// go to Belgium.
// go to China.
```

map(): iterate through a collection, mapping each value through a transformation function

```
var count = 1;
console.log(Todos.map(function(model){
```

```

    return count++ + ". " + model.get('title');;
  }));
  // Above logs:
  //1. go to Belgium.
  //2. go to China.
  //3. go to Austria.

```

min()/max(): retrieve item with the min or max value of an attribute

```

Todos.max(function(model){
  return model.id;
}).id;

Todos.min(function(model){
  return model.id;
}).id;

```

pluck(): extract a specific attribute

```

var captions = Todos.pluck('caption');
// returns list of captions

```

filter(): filter a collection

Filter by an array of model IDs

```

var Todos = Backbone.Collection.extend({
  model: Todo,
  filterById: function(ids){
    return this.models.filter(
      function(c) {
        return _.contains(ids, c.id);
      })
  }
});

```

indexOf(): return the item at a particular index within a collection

```

Todos.indexOf(5);

```

any(): Confirm if any of the values in a collection pass an iterator truth test

```
Todos.any(function(model){
  return model.id === 100;
});
```

```
// or
Todos.some(function(model){
  return model.id === 100;
});
```

size(): return the size of a collection

```
Todos.size();
```

```
// equivalent to
Todos.length;
```

isEmpty(): determine whether a collection is empty

```
var isEmpty = Todos.isEmpty();
```

groupBy(): group a collection into groups of like items

```
var Todos = new Backbone.Collection();

Todos.add([
  { title: 'go to Belgium.', completed: false },
  { title: 'go to China.', completed: false },
  { title: 'go to Austria.', completed: true }
]);

// create groups of completed and incomplete models
var byCompleted = Todos.groupBy('completed');
var completed = new Backbone.Collection(byCompleted[true]);
console.log(completed.pluck('title'));
// logs: ["go to Austria."]
```

In addition, several of the Underscore operations on objects are available as methods on Models.

pick(): extract a set of attributes from a model

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
```

```

    completed: false
  }
});

var todo = new Todo({title: 'go to Austria.'});
console.log(todo.pick('title'));
// logs {title: "go to Austria"}

```

omit(): extract all attributes from a model except those listed

```

var todo = new Todo({title: 'go to Austria.'});
console.log(todo.omit('title'));
// logs {completed: false}

```

keys() and values(): get lists of attribute names and values

```

var todo = new Todo({title: 'go to Austria.'});
console.log(todo.keys());
// logs: ["title", "completed"]

console.log(todo.values());
//logs: ["go to Austria.", false]

```

pairs(): get list of attributes as [key, value] pairs

```

var todo = new Todo({title: 'go to Austria.'});
var pairs = todo.pairs();

console.log(pairs[0]);
// logs: ["title", "go to Austria."]
console.log(pairs[1]);
// logs: ["completed", false]

```

invert(): create object in which the values are keys and the attributes are values

```

var todo = new Todo({title: 'go to Austria.'});
console.log(todo.invert());

// logs: {go to Austria.: "title", false: "completed"}

```

The complete list of what Underscore can do can be found in its official [docs](#).

Chainable API Speaking of utility methods, another bit of sugar in Backbone is its support for Underscore's `chain()` method. Chaining is a common idiom in object-oriented languages; a chain is a sequence of method calls on the same object that are performed in a single statement. While Backbone makes Underscore's array manipulation operations available as methods of Collection objects, they cannot be directly chained since they return arrays rather than the original Collection.

Fortunately, the inclusion of Underscore's `chain()` method enables you to chain calls to these methods on Collections.

The `chain()` method returns an object that has all of the Underscore array operations attached as methods which return that object. The chain ends with a call to the `value()` method which simply returns the resulting array value. In case you haven't seen it before, the chainable API looks like this:

```
var collection = new Backbone.Collection([
  { name: 'Tim', age: 5 },
  { name: 'Ida', age: 26 },
  { name: 'Rob', age: 55 }
]);

var filteredNames = collection.chain() // start chain, returns wrapper around collection's methods
  .filter(function(item) { return item.get('age') > 10; }) // returns wrapped array excluding items with age < 10
  .map(function(item) { return item.get('name'); }) // returns wrapped array containing remaining items
  .value(); // terminates the chain and returns the resulting array

console.log(filteredNames); // logs: ['Ida', 'Rob']
```

Some of the Backbone-specific methods do return `this`, which means they can be chained as well:

```
var collection = new Backbone.Collection();

collection
  .add({ name: 'John', age: 23 })
  .add({ name: 'Harry', age: 33 })
  .add({ name: 'Steve', age: 41 });

var names = collection.pluck('name');

console.log(names); // logs: ['John', 'Harry', 'Steve']
```

RESTful Persistence

Thus far, all of our example data has been created in the browser. For most single page applications, the models are derived from a data set residing on

a server. This is an area in which Backbone dramatically simplifies the code you need to write to perform RESTful synchronization with a server through a simple API on its models and collections.

Fetching models from the server

`Collections.fetch()` retrieves a set of models from the server in the form of a JSON array by sending an HTTP GET request to the URL specified by the collection's `url` property (which may be a function). When this data is received, a `set()` will be executed to update the collection.

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo,
  url: '/todos'
});

var todos = new TodosCollection();
todos.fetch(); // sends HTTP GET to /todos
```

Saving models to the server

While Backbone can retrieve an entire collection of models from the server at once, updates to models are performed individually using the model's `save()` method. When `save()` is called on a model that was fetched from the server, it constructs a URL by appending the model's `id` to the collection's URL and sends an HTTP PUT to the server. If the model is a new instance that was created in the browser (i.e., it doesn't have an `id`) then an HTTP POST is sent to the collection's URL. `Collections.create()` can be used to create a new model, add it to the collection, and send it to the server in a single method call.

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo,
  url: '/todos'
```

```
});

var todos = new TodosCollection();
todos.fetch();

var todo2 = todos.get(2);
todo2.set('title', 'go fishing');
todo2.save(); // sends HTTP PUT to /todos/2

todos.create({title: 'Try out code samples'}); // sends HTTP POST to /todos and adds to col
```

As mentioned earlier, a model's `validate()` method is called automatically by `save()` and will trigger an `invalid` event on the model if validation fails.

Deleting models from the server

A model can be removed from the containing collection and the server by calling its `destroy()` method. Unlike `Collection.remove()` which only removes a model from a collection, `Model.destroy()` will also send an HTTP DELETE to the collection's URL.

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo,
  url: '/todos'
});

var todos = new TodosCollection();
todos.fetch();

var todo2 = todos.get(2);
todo2.destroy(); // sends HTTP DELETE to /todos/2 and removes from collection
```

Options

Each RESTful API method accepts a variety of options. Most importantly, all methods accept success and error callbacks which can be used to customize the handling of server responses. Specifying the `{patch: true}` option to `Model.save()` will cause it to use HTTP PATCH to send only the changed attributes to the server instead of the entire model. Similarly, passing the `{reset:`

`true`} option to `Collection.fetch()` will result in the collection being updated using `reset()` rather than `set()`.

See the Backbone.js documentation for full descriptions of the supported options.

Events

Events are a basic inversion of control, where instead of having a function call another function by name, the second function registers interest in being called whenever the first “event” occurs.

The part of your application that has to know how to call the other part of your app has been inverted. This is the core thing that makes it possible for your business logic to not have to know about how your user interface works and is the most powerful thing about the Backbone Events system.

Mastering events is one of the quickest ways to become more productive with Backbone, so let’s take a closer look at Backbone’s event model.

`Backbone.Events` is mixed into the other Backbone “classes”, including:

- Backbone
- Backbone.Model
- Backbone.Collection
- Backbone.Router
- Backbone.History
- Backbone.View

Note that `Backbone.Events` is mixed into the `Backbone` object. Since `Backbone` is globally visible, it can be used as a simple event bus:

```
Backbone.on('event', function() {console.log('Handled Backbone event');});
Backbone.trigger('event'); // logs: Handled Backbone event
```

on(), off(), and trigger() `Backbone.Events` can give any object the ability to bind and trigger custom events. We can mix this module into any object easily and there isn’t a requirement for events to be declared before being bound to a callback handler.

Example:

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);
```

```

// Add a custom event
ourObject.on('dance', function(msg){
  console.log('We triggered ' + msg);
});

// Trigger the custom event
ourObject.trigger('dance', 'our event');

```

If you're familiar with jQuery custom events or the concept of Publish/Subscribe, `Backbone.Events` provides a system that is very similar with `on` being analogous to `subscribe` and `trigger` being similar to `publish`.

`on` binds a callback function to an object, as we've done with `dance` in the above example. The callback is invoked whenever the event is triggered.

The official Backbone.js documentation recommends namespacing event names using colons if you end up using quite a few of these on your page. e.g.:

```

var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function dancing (msg) { console.log("We started " + msg); }

// Add namespaced custom events
ourObject.on("dance:tap", dancing);
ourObject.on("dance:break", dancing);

// Trigger the custom events
ourObject.trigger("dance:tap", "tap dancing. Yeah!");
ourObject.trigger("dance:break", "break dancing. Yeah!");

// This one triggers nothing as no listener listens for it
ourObject.trigger("dance", "break dancing. Yeah!");

```

A special `all` event is made available in case you would like notifications for every event that occurs on the object (e.g., if you would like to screen events in a single location). The `all` event can be used as follows:

```

var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

```

```
function dancing (msg) { console.log("We started " + msg); }

ourObject.on("all", function(eventName){
  console.log("The name of the event passed was " + eventName);
});

// This time each event will be caught with a catch 'all' event listener
ourObject.trigger("dance:tap", "tap dancing. Yeah!");
ourObject.trigger("dance:break", "break dancing. Yeah!");
ourObject.trigger("dance", "break dancing. Yeah!");
```

off removes callback functions that were previously bound to an object. Going back to our Publish/Subscribe comparison, think of it as an unsubscribe for custom events.

To remove the dance event we previously bound to ourObject, we would simply do:

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function dancing (msg) { console.log("We " + msg); }

// Add namespaced custom events
ourObject.on("dance:tap", dancing);
ourObject.on("dance:break", dancing);

// Trigger the custom events. Each will be caught and acted upon.
ourObject.trigger("dance:tap", "started tap dancing. Yeah!");
ourObject.trigger("dance:break", "started break dancing. Yeah!");

// Removes event bound to the object
ourObject.off("dance:tap");

// Trigger the custom events again, but one is logged.
ourObject.trigger("dance:tap", "stopped tap dancing."); // won't be logged as it's not list
ourObject.trigger("dance:break", "break dancing. Yeah!");
```

To remove all callbacks for the event we pass an event name (e.g., move) to the off() method on the object the event is bound to. If we wish to remove a specific callback, we can pass that callback as the second parameter:

```
var ourObject = {};
```

```

// Mixin
_.extend(ourObject, Backbone.Events);

function dancing (msg) { console.log("We are dancing. " + msg); }
function jumping (msg) { console.log("We are jumping. " + msg); }

// Add two listeners to the same event
ourObject.on("move", dancing);
ourObject.on("move", jumping);

// Trigger the events. Both listeners are called.
ourObject.trigger("move", "Yeah!");

// Removes specified listener
ourObject.off("move", dancing);

// Trigger the events again. One listener left.
ourObject.trigger("move", "Yeah, jump, jump!");

```

Finally, as we have seen in our previous examples, `trigger` triggers a callback for a specified event (or a space-separated list of events). e.g.:

```

var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function doAction (msg) { console.log("We are " + msg); }

// Add event listeners
ourObject.on("dance", doAction);
ourObject.on("jump", doAction);
ourObject.on("skip", doAction);

// Single event
ourObject.trigger("dance", 'just dancing.');
```

`ourObject.trigger("dance jump skip", 'very tired from so much action.');`

`trigger` can pass multiple arguments to the callback function:

```

var ourObject = {};

// Mixin

```

```

_.extend(ourObject, Backbone.Events);

function doAction (action, duration) {
  console.log("We are " + action + ' for ' + duration );
}

// Add event listeners
ourObject.on("dance", doAction);
ourObject.on("jump", doAction);
ourObject.on("skip", doAction);

// Passing multiple arguments to single event
ourObject.trigger("dance", 'dancing', "5 minutes");

// Passing multiple arguments to multiple events
ourObject.trigger("dance jump skip", 'on fire', "15 minutes");

```

listenTo() and stopListening() While `on()` and `off()` add callbacks directly to an observed object, `listenTo()` tells an object to listen for events on another object, allowing the listener to keep track of the events for which it is listening. `stopListening()` can subsequently be called on the listener to tell it to stop listening for events:

```

var a = _.extend({}, Backbone.Events);
var b = _.extend({}, Backbone.Events);
var c = _.extend({}, Backbone.Events);

// add listeners to A for events on B and C
a.listenTo(b, 'anything', function(event){ console.log("anything happened"); });
a.listenTo(c, 'everything', function(event){ console.log("everything happened"); });

// trigger an event
b.trigger('anything'); // logs: anything happened

// stop listening
a.stopListening();

// A does not receive these events
b.trigger('anything');
c.trigger('everything');

```

`stopListening()` can also be used to selectively stop listening based on the event, model, or callback handler.

If you use `on` and `off` and remove views and their corresponding models at the same time, there are generally no problems. But a problem arises when you

remove a view that had registered to be notified about events on a model, but you don't remove the model or call `off` to remove the view's event handler. Since the model has a reference to the view's callback function, the JavaScript garbage collector cannot remove the view from memory. This is called a "ghost view" and is a form of memory leak which is common since the models generally tend to outlive the corresponding views during an application's lifecycle. For details on the topic and a solution, check this [excellent article](#) by Derick Bailey.

Practically, every `on` called on an object also requires an `off` to be called in order for the garbage collector to do its job. `listenTo()` changes that, allowing Views to bind to Model notifications and unbind from all of them with just one call - `stopListening()`.

The default implementation of `View.remove()` makes a call to `stopListening()`, ensuring that any listeners bound using `listenTo()` are unbound before the view is destroyed.

```
var view = new Backbone.View();
var b = _.extend({}, Backbone.Events);

view.listenTo(b, 'all', function(){ console.log(true); });
b.trigger('anything');

view.listenTo(b, 'all', function(){ console.log(false); });
view.remove(); // stopListening() implicitly called
b.trigger('anything');
// logs: true
```

Events and Views Within a View, there are two types of events you can listen for: DOM events and events triggered using the Event API. It is important to understand the differences in how views bind to these events and the context in which their callbacks are invoked.

DOM events can be bound to using the View's `events` property or using `jQuery.on()`. Within callbacks bound using the `events` property, `this` refers to the View object; whereas any callbacks bound directly using `jQuery` will have `this` set to the handling DOM element by `jQuery`. All DOM event callbacks are passed an `event` object by `jQuery`. See `delegateEvents()` in the Backbone documentation for additional details.

Event API events are bound as described in this section. If the event is bound using `on()` on the observed object, a context parameter can be passed as the third argument. If the event is bound using `listenTo()` then within the callback `this` refers to the listener. The arguments passed to Event API callbacks depends on the type of event. See the Catalog of Events in the Backbone documentation for details.

The following example illustrates these differences:


```

<div id="todo">
  <input type='checkbox' />
</div>

var View = Backbone.View.extend({

  el: '#todo',

  // bind to DOM event using events property
  events: {
    'click [type="checkbox"]': 'clicked',
  },

  initialize: function () {
    // bind to DOM event using jQuery
    this.$el.click(this.jqueryClicked);

    // bind to API event
    this.on('apiEvent', this.callback);
  },

  // 'this' is view
  clicked: function(event) {
    console.log("events handler for " + this.el.outerHTML);
    this.trigger('apiEvent', event.type);
  },

  // 'this' is handling DOM element
  jqueryClicked: function(event) {
    console.log("jQuery handler for " + this.outerHTML);
  },

  callback: function(eventType) {
    console.log("event type was " + eventType);
  }

});

var view = new View();

```

Routers

In Backbone, routers provide a way for you to connect URLs (either hash fragments, or real) to parts of your application. Any piece of your application

that you want to be bookmarkable, shareable, and back-button-able, needs a URL.

Some examples of routes using the hash mark may be seen below:

```
http://example.com/#about
http://example.com/#search/seasonal-horns/page2
```

An application will usually have at least one route mapping a URL route to a function that determines what happens when a user reaches that route. This relationship is defined as follows:

```
'route' : 'mappedFunction'
```

Let's define our first router by extending `Backbone.Router`. For the purposes of this guide, we're going to continue pretending we're creating a complex todo application (something like a personal organizer/planner) that requires a complex `TodoRouter`.

Note the inline comments in the code example below as they continue our lesson on routers.

```
var TodoRouter = Backbone.Router.extend({
  /* define the route and function maps for this router */
  routes: {
    "about" : "showAbout",
    /* Sample usage: http://example.com/#about */

    "todo/:id" : "getTodo",
    /* This is an example of using a ":param" variable which allows us to match
    any of the components between two URL slashes */
    /* Sample usage: http://example.com/#todo/5 */

    "search/:query" : "searchTodos",
    /* We can also define multiple routes that are bound to the same map function,
    in this case searchTodos(). Note below how we're optionally passing in a
    reference to a page number if one is supplied */
    /* Sample usage: http://example.com/#search/job */

    "search/:query/p:page" : "searchTodos",
    /* As we can see, URLs may contain as many ":param"s as we wish */
    /* Sample usage: http://example.com/#search/job/p1 */

    "todos/:id/download/*documentPath" : "downloadDocument",
    /* This is an example of using a *splat. Splats are able to match any number of
```

```

    URL components and can be combined with ":param"s*/
    /* Sample usage: http://example.com/#todos/5/download/files/Meeting_schedule.doc */

    /* If you wish to use splats for anything beyond default routing, it's probably a good
    idea to leave them at the end of a URL otherwise you may need to apply regular
    expression parsing on your fragment */

    "*other"      : "defaultRoute"
    /* This is a default route that also uses a *splat. Consider the
    default route a wildcard for URLs that are either not matched or where
    the user has incorrectly typed in a route path manually */
    /* Sample usage: http://example.com/# <anything> */
  },

  showAbout: function(){
  },

  getTodo: function(id){
    /*
    Note that the id matched in the above route will be passed to this function
    */
    console.log("You are trying to reach todo " + id);
  },

  searchTodos: function(query, page){
    var page_number = page || 1;
    console.log("Page number: " + page_number + " of the results for todos containing " + query);
  },

  downloadDocument: function(id, path){
  },

  defaultRoute: function(other){
    console.log('Invalid. You attempted to reach:' + other);
  }
});

/* Now that we have a router setup, we need to instantiate it */

var myTodoRouter = new TodoRouter();

```

Backbone offers an opt-in for HTML5 pushState support via `window.history.pushState`. This permits you to define routes such as `http://backbonejs.org/just/an/example`. This will be supported with automatic degradation when a user's browser doesn't support pushState. Note that it is vastly preferred that you're capable of also supporting pushState on the server side, although it is a little more

difficult to implement.

Is there a limit to the number of routers I should be using?

Andrew de Andrade has pointed out that DocumentCloud, the creators of Backbone, usually only use a single router in most of their applications. You're very likely to not require more than one or two routers in your own projects; the majority of your application routing can be kept organized in a single router without it getting unwieldy.

Backbone.history Next, we need to initialize `Backbone.history` as it handles `hashchange` events in our application. This will automatically handle routes that have been defined and trigger callbacks when they've been accessed.

The `Backbone.history.start()` method will simply tell Backbone that it's okay to begin monitoring all `hashchange` events as follows:

```
var TodoRouter = Backbone.Router.extend({
  /* define the route and function maps for this router */
  routes: {
    "about" : "showAbout",
    "search/:query" : "searchTodos",
    "search/:query/p:page" : "searchTodos"
  },

  showAbout: function(){},

  searchTodos: function(query, page){
    var page_number = page || 1;
    console.log("Page number: " + page_number + " of the results for todos containing the word " + query);
  }
});

var myTodoRouter = new TodoRouter();

Backbone.history.start();

// Go to and check console:
// http://localhost/#search/job/p3    logs: Page number: 3 of the results for todos containing the word job
// http://localhost/#search/job       logs: Page number: 1 of the results for todos containing the word job
// etc.
```

Note: To run the last example, you'll need to create a local development environment and test project, which we will cover later on in the book.

If you would like to update the URL to reflect the application state at a particular point, you can use the router's `.navigate()` method. By default, it simply updates your URL fragment without triggering the `hashchange` event:

```

// Let's imagine we would like a specific fragment (edit) once a user opens a single todo
var TodoRouter = Backbone.Router.extend({
  routes: {
    "todo/:id": "viewTodo",
    "todo/:id/edit": "editTodo"
    // ... other routes
  },

  viewTodo: function(id){
    console.log("View todo requested.");
    this.navigate("todo/" + id + '/edit'); // updates the fragment for us, but doesn't trigger
  },

  editTodo: function(id) {
    console.log("Edit todo opened.");
  }
});

var myTodoRouter = new TodoRouter();

Backbone.history.start();

// Go to: http://localhost/#todo/4
//
// URL is updated to: http://localhost/#todo/4/edit
// but editTodo() function is not invoked even though location we end up is mapped to it.
//
// logs: View todo requested.

```

It is also possible for Router.navigate() to trigger the route as well as update the URL fragment.

```

var TodoRouter = Backbone.Router.extend({
  routes: {
    "todo/:id": "viewTodo",
    "todo/:id/edit": "editTodo"
    // ... other routes
  },

  viewTodo: function(id){
    console.log("View todo requested.");
    this.navigate("todo/" + id + '/edit', true); // updates the fragment and triggers the route
  },

  editTodo: function(id) {

```

```

        console.log("Edit todo opened.");
    }
});

var myTodoRouter = new TodoRouter();

Backbone.history.start();

// Go to: http://localhost/#todo/4
//
// URL is updated to: http://localhost/#todo/4/edit
// and this time editTodo() function is invoked.
//
// logs:
// View todo requested.
// Edit todo opened.

```

Backbone's Sync API

We previously discussed how Backbone supports RESTful persistence via its `fetch()` and `create()` methods on Collections and `save()`, and `delete()` methods on Models. Now we are going to take a closer look at Backbone's `sync` method which underlies these operations.

The `Backbone.sync` method is an integral part of Backbone.js. It assumes a jQuery-like `$.ajax()` method, so HTTP parameters are organized based on jQuery's API. Since some legacy servers may not support JSON-formatted requests and HTTP PUT and DELETE operations, Backbone can be configured to emulate these capabilities using the two configuration variables shown below with their default values:

```

Backbone.emulateHTTP = false; // set to true if server cannot handle HTTP PUT or HTTP DELETE
Backbone.emulateJSON = false; // set to true if server cannot handle application/json requests

```

The `Backbone.emulateHTTP` variable should be set to true if extended HTTP methods are not supported by the server. The `Backbone.emulateJSON` variable should be set to true if the server does not understand the MIME type for JSON.

`Backbone.sync` is called every time Backbone tries to read, save, or delete models. It uses jQuery or Zepto's `$.ajax()` implementations to make these RESTful requests, however this can be overridden as per your needs.

Overriding Backbone.sync

The `sync` function may be overridden globally as `Backbone.sync`, or at a finer-grained level, by adding a `sync` function to a Backbone collection or to an individual model.

Since all persistence is handled by the `Backbone.sync` function, an alternative persistence layer can be used by simply overriding `Backbone.sync` with a function that has the same signature:

```
Backbone.sync = function(method, model, options) {  
};
```

The `methodMap` below is used by the standard `sync` implementation to map the `method` parameter to an HTTP operation and illustrates the type of action required by each `method` argument:

```
var methodMap = {  
  'create': 'POST',  
  'update': 'PUT',  
  'patch': 'PATCH',  
  'delete': 'DELETE',  
  'read': 'GET'  
};
```

If we wanted to replace the standard `sync` implementation with one that simply logged the calls to `sync`, we could do this:

```
var id_counter = 1;  
Backbone.sync = function(method, model) {  
  console.log("I've been passed " + method + " with " + JSON.stringify(model));  
  if(method === 'create'){ model.set('id', id_counter++); }  
};
```

Note that we assign a unique `id` to any created models.

The `Backbone.sync` method is intended to be overridden to support other persistence backends. The built-in method is tailored to a certain breed of RESTful JSON APIs - Backbone was originally extracted from a Ruby on Rails application, which uses HTTP methods like `PUT` in the same way.

The `sync` method is called with three parameters:

- `method`: One of `create`, `update`, `patch`, `delete`, or `read`
- `model`: The Backbone model object
- `options`: May include success and error methods

Implementing a new `sync` method can use the following pattern:

```

Backbone.sync = function(method, model, options) {

  function success(result) {
    // Handle successful results from MyAPI
    if (options.success) {
      options.success(result);
    }
  }

  function error(result) {
    // Handle error results from MyAPI
    if (options.error) {
      options.error(result);
    }
  }

  options || (options = {});

  switch (method) {
    case 'create':
      return MyAPI.create(model, success, error);

    case 'update':
      return MyAPI.update(model, success, error);

    case 'patch':
      return MyAPI.patch(model, success, error);

    case 'delete':
      return MyAPI.destroy(model, success, error);

    case 'read':
      if (model.attributes[model.idAttribute]) {
        return MyAPI.find(model, success, error);
      } else {
        return MyAPI.findAll(model, success, error);
      }
  }
};

```

This pattern delegates API calls to a new object (MyAPI), which could be a Backbone-style class that supports events. This can be safely tested separately, and potentially used with libraries other than Backbone.

There are quite a few sync implementations out there. The following examples are all available on GitHub:

- Backbone localStorage: persists to the browser's local storage
- Backbone offline: supports working offline
- Backbone Redis: uses Redis key-value store
- backbone-parse: integrates Backbone with Parse.com
- backbone-websql: stores data to WebSQL
- Backbone Caching Sync: uses local storage as cache for other sync implementations

Dependencies

The official Backbone.js [documentation](#) states:

Backbone's only hard dependency is either Underscore.js ($\geq 1.4.3$) or Lo-Dash. For RESTful persistence, history support via Backbone.Router and DOM manipulation with Backbone.View, include json2.js, and either jQuery ($\geq 1.7.0$) or Zepto.

What this translates to is that if you require working with anything beyond models, you will need to include a DOM manipulation library such as jQuery or Zepto. Underscore is primarily used for its utility methods (which Backbone relies upon heavily) and json2.js for legacy browser JSON support if Backbone.sync is used.

Summary

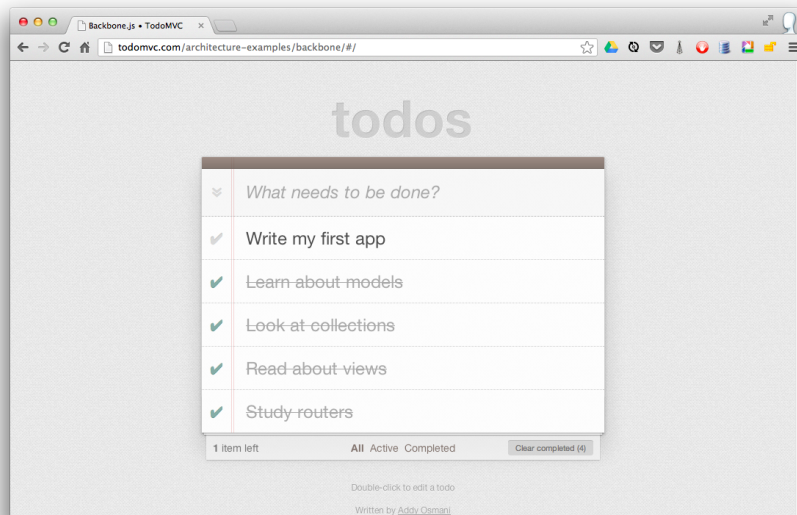
In this chapter we have introduced you to the components you will be using to build applications with Backbone: Models, Views, Collections, and Routers. We've also explored the Events mix-in that Backbone uses to enhance all components with publish-subscribe capabilities and seen how it can be used with arbitrary objects. Finally, we saw how Backbone leverages the Underscore.js and jQuery/Zepto APIs to add rich manipulation and persistence features to Backbone Collections and Models.

Backbone has many operations and options beyond those we have covered here and is always evolving, so be sure to visit the official [documentation](#) for more details and the latest features. In the next chapter you will start to get your hands dirty as we walk you through implementation of your first Backbone application.

Exercise 1: Todos - Your First Backbone.js App

Now that we've covered fundamentals, let's write our first Backbone.js application. We'll build the Backbone Todo List application exhibited on [TodoMVC.com](#).

Building a Todo List is a great way to learn Backbone's conventions. It's a relatively simple application, yet technical challenges surrounding binding, persisting model data, routing, and template rendering provide opportunities to illustrate some core Backbone features.



Let's consider the application's architecture at a high level. We'll need:

- a **Todo** model to describe individual todo items
- a **TodoList** collection to store and persist todos
- a way of creating todos
- a way to display a listing of todos
- a way to edit existing todos
- a way to mark a todo as completed
- a way to delete todos
- a way to filter the items that have been completed or are remaining

Essentially, these features are classic **CRUD** methods. Let's get started!

Static HTML

We'll place all of our HTML in a single file named `index.html`.

Header and Scripts First, we'll set up the header and the basic application dependencies: [jQuery](#), [Underscore](#), [Backbone.js](#) and the [Backbone LocalStorage adapter](#).

```

<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>Backbone.js • TodoMVC</title>
  <link rel="stylesheet" href="assets/base.css">
</head>
<body>
  <script type="text/template" id="item-template"></script>
  <script type="text/template" id="stats-template"></script>
  <script src="js/lib/jquery.min.js"></script>
  <script src="js/lib/underscore-min.js"></script>
  <script src="js/lib/backbone-min.js"></script>
  <script src="js/lib/backbone.localStorage.js"></script>
  <script src="js/models/todo.js"></script>
  <script src="js/collections/todos.js"></script>
  <script src="js/views/todos.js"></script>
  <script src="js/views/app.js"></script>
  <script src="js/routers/router.js"></script>
  <script src="js/app.js"></script>
</body>
</html>

```

In addition to the aforementioned dependencies, note that a few other application-specific files are also loaded. These are organized into folders representing their application responsibilities: models, views, collections, and routers. An `app.js` file is present to house central initialization code.

Note: If you want to follow along, create a directory structure as demonstrated in `index.html`:

1. Place the `index.html` in a top-level directory.
2. Download jQuery, Underscore, Backbone, and Backbone LocalStorage from their respective web sites and place them under `js/lib`
3. Create the directories `js/models`, `js/collections`, `js/views`, and `js/routers`

You will also need [base.css](#) and [bg.png](#), which should live in an `assets` directory. And remember that you can see a demo of the final application at [TodoMVC.com](#).

We will be creating the application JavaScript files during the tutorial. Don't worry about the two 'text/template' script elements - we will replace those soon!

Application HTML Now let's populate the body of `index.html`. We'll need an `<input>` for creating new todos, a `<ul id="todo-list" />` for listing the

actual todos, and a footer where we can later insert statistics and links for performing operations such as clearing completed todos. We'll add the following markup immediately inside our body tag before the script elements:

```
<section id="todoapp">
  <header id="header">
    <h1>todos</h1>
    <input id="new-todo" placeholder="What needs to be done?" autofocus>
  </header>
  <section id="main">
    <input id="toggle-all" type="checkbox">
    <label for="toggle-all">Mark all as complete</label>
    <ul id="todo-list"></ul>
  </section>
  <footer id="footer"></footer>
</section>
<div id="info">
  <p>Double-click to edit a todo</p>
  <p>Written by <a href="https://github.com/addyosmani">Addy Osmani</a></p>
  <p>Part of <a href="http://todomvc.com">TodoMVC</a></p>
</div>
```

Templates To complete index.html, we need to add the templates which we will use to dynamically create HTML by injecting model data into their placeholders. One way of including templates in the page is by using custom script tags. These don't get evaluated by the browser, which just interprets them as plain text. Underscore micro-templating can then access the templates, rendering fragments of HTML.

We'll start by filling in the #item-template which will be used to display individual todo items.

```
<!-- index.html -->

<script type="text/template" id="item-template">
  <div class="view">
    <input class="toggle" type="checkbox" <%= completed ? 'checked' : '' %>>
    <label><%- title %></label>
    <button class="destroy"></button>
  </div>
  <input class="edit" value="<%- title %>">
</script>
```

The template tags in the above markup, such as <%= and <%-, are specific to Underscore.js and are documented on the Underscore site. In your own

applications, you have a choice of template libraries, such as Mustache or Handlebars. Use whichever you prefer, Backbone doesn't mind.

We also need to define the `#stats-template` template which we will use to populate the footer.

```
<!-- index.html -->

<script type="text/template" id="stats-template">
  <span id="todo-count"><strong><%= remaining %></strong> <%= remaining === 1 ? 'item' : 'items' %></span>
  <ul id="filters">
    <li>
      <a class="selected" href="#/">All</a>
    </li>
    <li>
      <a href="#/active">Active</a>
    </li>
    <li>
      <a href="#/completed">Completed</a>
    </li>
  </ul>
  <% if (completed) { %>
    <button id="clear-completed">Clear completed (<%= completed %>)</button>
  <% } %>
</script>
```

The `#stats-template` displays the number of remaining incomplete items and contains a list of hyperlinks which will be used to perform actions when we implement our router. It also contains a button which can be used to clear all of the completed items.

Now that we have all the HTML that we will need, we'll start implementing our application by returning to the fundamentals: a `Todo` model.

Todo model

The `Todo` model is remarkably straightforward. First, a `todo` has two attributes: a `title` stores a `todo` item's title and a `completed` status indicates if it's complete. These attributes are passed as defaults, as shown below:

```
// js/models/todo.js

var app = app || {};
```

```

// Todo Model
// -----
// Our basic **Todo** model has 'title', 'order', and 'completed' attributes.

app.Todo = Backbone.Model.extend({

  // Default attributes ensure that each todo created has 'title' and 'completed' keys.
  defaults: {
    title: '',
    completed: false
  },

  // Toggle the 'completed' state of this todo item.
  toggle: function() {
    this.save({
      completed: !this.get('completed')
    });
  }

});

```

Second, the Todo model has a `toggle()` method through which a Todo item's completion status can be set and simultaneously persisted.

Todo collection

Next, a `TodoList` collection is used to group our models. The collection uses the `LocalStorage` adapter to override Backbone's default `sync()` operation with one that will persist our Todo records to HTML5 Local Storage. Through local storage, they're saved between page requests.

```

// js/collections/todos.js

var app = app || {};

// Todo Collection
// -----

// The collection of todos is backed by *localStorage* instead of a remote
// server.
var TodoList = Backbone.Collection.extend({

  // Reference to this collection's model.

```

```

model: app.Todo,

// Save all of the todo items under the "todos-backbone" namespace.
localStorage: new Backbone.LocalStorage('todos-backbone'),

// Filter down the list of all todo items that are finished.
completed: function() {
  return this.filter(function( todo ) {
    return todo.get('completed');
  });
},

// Filter down the list to only todo items that are still not finished.
remaining: function() {
  return this.without.apply( this, this.completed() );
},

// We keep the Todos in sequential order, despite being saved by unordered
// GUID in the database. This generates the next order number for new items.
nextOrder: function() {
  if ( !this.length ) {
    return 1;
  }
  return this.last().get('order') + 1;
},

// Todos are sorted by their original insertion order.
comparator: function( todo ) {
  return todo.get('order');
}
});

// Create our global collection of **Todos**.
app.Todos = new TodoList();

```

The collection's `completed()` and `remaining()` methods return an array of finished and unfinished todos, respectively.

A `nextOrder()` method implements a sequence generator while a `comparator()` sorts items by their insertion order.

Note: `this.filter`, `this.without` and `this.last` are Underscore methods that are mixed in to `Backbone.Collection` so that the reader knows how to find out more about them.

Application View

Let's examine the core application logic which resides in the views. Each view supports functionality such as edit-in-place, and therefore contains a fair amount of logic. To help organize this logic, we'll use the element controller pattern. The element controller pattern consists of two views: one controls a collection of items while the other deals with each individual item.

In our case, an **AppView** will handle the creation of new todos and rendering of the initial todo list. Instances of **TodoView** will be associated with each individual Todo record. Todo instances can handle editing, updating, and destroying their associated todo.

To keep things short and simple, we won't be implementing all of the application's features in this tutorial, we'll just cover enough to get you started. Even so, there is a lot for us to cover in **AppView**, so we'll split our discussion into two sections.

```
// js/views/app.js

var app = app || {};

// The Application
// -----

// Our overall **AppView** is the top-level piece of UI.
app.AppView = Backbone.View.extend({

  // Instead of generating a new element, bind to the existing skeleton of
  // the App already present in the HTML.
  el: '#todoapp',

  // Our template for the line of statistics at the bottom of the app.
  statsTemplate: _.template( $('#stats-template').html() ),

  // At initialization we bind to the relevant events on the 'Todos'
  // collection, when items are added or changed.
  initialize: function() {
    this.allCheckbox = this.$('#toggle-all')[0];
    this.$input = this.$('#new-todo');
    this.$footer = this.$('#footer');
    this.$main = this.$('#main');

    this.listenTo(app.Todos, 'add', this.addOne);
    this.listenTo(app.Todos, 'reset', this.addAll);
  },
```



```

    // Add a single todo item to the list by creating a view for it, and
    // appending its element to the '<ul>'.
    addOne: function( todo ) {
        var view = new app.TodoView({ model: todo });
        $('#todo-list').append( view.render().el );
    },

    // Add all items in the **Todos** collection at once.
    addAll: function() {
        this.$('#todo-list').html('');
        app.Todos.each(this.addOne, this);
    }

});

```

A few notable features are present in our initial version of `AppView`, including a `statsTemplate`, an `initialize` method that's implicitly called on instantiation, and several view-specific methods.

An `el` (element) property stores a selector targeting the DOM element with an ID of `todoapp`. In the case of our application, `el` refers to the matching `<section id="todoapp" />` element in `index.html`.

The call to `_.template` uses Underscore's micro-templating to construct a `statsTemplate` object from our `#stats-template`. We will use this template later when we render our view.

Now let's take a look at the `initialize` function. First, it's using jQuery to cache the elements it will be using into local properties (recall that `this.$()` finds elements relative to `this.$el`). Then it's binding to two events on the `Todos` collection: `add` and `reset`. Since we're delegating handling of updates and deletes to the `TodoView` view, we don't need to worry about those here. The two pieces of logic are:

- When an `add` event is fired the `addOne()` method is called and passed the new model. `addOne()` creates an instance of `TodoView` view, renders it, and appends the resulting element to our `Todo` list.
- When a `reset` event occurs (i.e., we update the collection in bulk as happens when the `Todos` are loaded from Local Storage), `addAll()` is called, which iterates over all of the `Todos` currently in our collection and fires `addOne()` for each item.

Note that we were able to use `this` within `addAll()` to refer to the view because `listenTo()` implicitly set the callback's context to the view when it created the binding.

Now, let's add some more logic to complete our AppView!

```
// js/views/app.js

var app = app || {};

// The Application
// -----

// Our overall **AppView** is the top-level piece of UI.
app.AppView = Backbone.View.extend({

  // Instead of generating a new element, bind to the existing skeleton of
  // the App already present in the HTML.
  el: '#todoapp',

  // Our template for the line of statistics at the bottom of the app.
  statsTemplate: _.template( $('#stats-template').html() ),

  // Delegated events for creating new items, and clearing completed ones.
  events: {
    'keypress #new-todo': 'createOnEnter',
    'click #clear-completed': 'clearCompleted',
    'click #toggle-all': 'toggleAllComplete'
  },

  // At initialization we bind to the relevant events on the 'Todos'
  // collection, when items are added or changed. Kick things off by
  // loading any preexisting todos that might be saved in *localStorage*.
  initialize: function() {
    this.allCheckbox = this.$('#toggle-all')[0];
    this.$input = this.$('#new-todo');
    this.$footer = this.$('#footer');
    this.$main = this.$('#main');

    this.listenTo(app.Todos, 'add', this.addOne);
    this.listenTo(app.Todos, 'reset', this.addAll);
    this.listenTo(app.Todos, 'change:completed', this.filterOne);
    this.listenTo(app.Todos, 'filter', this.filterAll);
    this.listenTo(app.Todos, 'all', this.render);

    app.Todos.fetch();
  },

  // Re-rendering the App just means refreshing the statistics -- the rest
```

```

// of the app doesn't change.
render: function() {
    var completed = app.Todos.completed().length;
    var remaining = app.Todos.remaining().length;

    if ( app.Todos.length ) {
        this.$main.show();
        this.$footer.show();

        this.$footer.html(this.statsTemplate({
            completed: completed,
            remaining: remaining
        }));

        this.$('#filters li a')
            .removeClass('selected')
            .filter('[href="#/' + ( app.TODOFilter || '' ) + '"]')
            .addClass('selected');
    } else {
        this.$main.hide();
        this.$footer.hide();
    }

    this.allCheckbox.checked = !remaining;
},

// Add a single todo item to the list by creating a view for it, and
// appending its element to the '<ul>'.
addOne: function( todo ) {
    var view = new app.TODOView({ model: todo });
    $('#todo-list').append( view.render().el );
},

// Add all items in the **Todos** collection at once.
addAll: function() {
    this.$('#todo-list').html('');
    app.Todos.each(this.addOne, this);
},

filterOne : function (todo) {
    todo.trigger('visible');
},

filterAll : function () {
    app.Todos.each(this.filterOne, this);
},

```

```

// Generate the attributes for a new Todo item.
newAttributes: function() {
  return {
    title: this.$input.val().trim(),
    order: app.Todos.nextOrder(),
    completed: false
  };
},

// If you hit return in the main input field, create new Todo model,
// persisting it to localStorage.
createOnEnter: function( event ) {
  if ( event.which !== ENTER_KEY || !this.$input.val().trim() ) {
    return;
  }

  app.Todos.create( this.newAttributes() );
  this.$input.val('');
},

// Clear all completed todo items, destroying their models.
clearCompleted: function() {
  _.invoke(app.Todos.completed(), 'destroy');
  return false;
},

toggleAllComplete: function() {
  var completed = this.allCheckbox.checked;

  app.Todos.each(function( todo ) {
    todo.save({
      'completed': completed
    });
  });
});
});

```

We have added the logic for creating new todos, editing them, and filtering them based on their completed status.

- **events**: We've defined an **events** hash containing declarative callbacks for our DOM events. It binds those events to the following methods:
- **createOnEnter()**: Creates a new Todo model and persists it in local-Storage when a user hits enter inside the `<input/>` field. Also resets the

main `<input/>` field value to prepare it for the next entry. The model is populated by `newAttributes()`, which returns an object literal composed of the title, order, and completed state of the new item. Note that `this` is referring to the view and not the DOM element since the callback was bound using the `events` hash.

- `clearCompleted()`: Removes the items in the todo list that have been marked as completed when the user clicks the clear-completed checkbox (this checkbox will be in the footer populated by the `#stats-template`).
- `toggleAllComplete()`: Allows a user to mark all of the items in the todo list as completed by clicking the toggle-all checkbox.
- `initialize()`: We've bound callbacks to several additional events:
- We've bound a `filterOne()` callback on the Todos collection for a `change:completed` event. This listens for changes to the completed flag for any model in the collection. The affected todo is passed to the callback which triggers a custom `visible` event on the model.
- We've bound a `filterAll()` callback for a `filter` event, which works a little similar to `addOne()` and `addAll()`. It's responsibility is to toggle which todo items are visible based on the filter currently selected in the UI (all, completed or remaining) via calls to `filterOne()`.
- We've used the special `all` event to bind any event triggered on the Todos collection to the view's render method (discussed below).

The `initialize()` method completes by fetching the previously saved todos from `localStorage`.

- `render()`: Several things are happening in our `render()` method:
- The `#main` and `#footer` sections are displayed or hidden depending on whether there are any todos in the collection.
- The footer is populated with the HTML produced by instantiating the `statsTemplate` with the number of completed and remaining todo items.
- The HTML produced by the preceding step contains a list of filter links. The value of `app.TODO_FILTER`, which will be set by our router, is being used to apply the class 'selected' to the link corresponding to the currently selected filter. This will result in conditional CSS styling being applied to that filter.
- The `allCheckbox` is updated based on whether there are remaining todos.

Individual Todo View

Now let's look at the `TodoView` view. This will be in charge of individual Todo records, making sure the view updates when the todo does. To enable this

functionality, we will add event listeners to the view that listen for events on an individual todo's HTML representation.

```
// js/views/todos.js

var app = app || {};

// Todo Item View
// -----

// The DOM element for a todo item...
app.TodoView = Backbone.View.extend({

  //... is a list tag.
  tagName: 'li',

  // Cache the template function for a single item.
  template: _.template( $('#item-template').html() ),

  // The DOM events specific to an item.
  events: {
    'dblclick label': 'edit',
    'keypress .edit': 'updateOnEnter',
    'blur .edit': 'close'
  },

  // The TodoView listens for changes to its model, re-rendering. Since there's
  // a one-to-one correspondence between a **Todo** and a **TodoView** in this
  // app, we set a direct reference on the model for convenience.
  initialize: function() {
    this.listenTo(this.model, 'change', this.render);
  },

  // Re-renders the titles of the todo item.
  render: function() {
    this.$el.html( this.template( this.model.toJSON() ) );
    this.$input = this.$('.edit');
    return this;
  },

  // Switch this view into "editing" mode, displaying the input field.
  edit: function() {
    this.$el.addClass('editing');
    this.$input.focus();
  },
});
```

```

// Close the "editing" mode, saving changes to the todo.
close: function() {
    var value = this.$input.val().trim();

    if ( value ) {
        this.model.save({ title: value });
    }

    this.$el.removeClass('editing');
},

// If you hit 'enter', we're through editing the item.
updateOnEnter: function( e ) {
    if ( e.which === ENTER_KEY ) {
        this.close();
    }
}
});

```

In the `initialize()` constructor, we set up a listener that monitors a todo model's `change` event. As a result, when the todo gets updated, the application will re-render the view and visually reflect its changes. Note that the model passed in the arguments hash by our AppView is automatically available to us as `this.model`.

In the `render()` method, we render our Underscore.js `#item-template`, which was previously compiled into `this.template` using Underscore's `_.template()` method. This returns an HTML fragment that replaces the content of the view's element (an `li` element was implicitly created for us based on the `tagName` property). In other words, the rendered template is now present under `this.el` and can be appended to the todo list in the user interface. `render()` finishes by caching the input element within the instantiated template into `this.input`.

Our events hash includes three callbacks:

- `edit()`: changes the current view into editing mode when a user double-clicks on an existing item in the todo list. This allows them to change the existing value of the item's title attribute.
- `updateOnEnter()`: checks that the user has hit the return/enter key and executes the `close()` function.
- `close()`: trims the value of the current text in our `<input/>` field, ensuring that we don't process it further if it does not contain any text (e.g. ""). If a valid value has been provided, we save the changes to the current todo model and close editing mode by removing the corresponding CSS class.

Startup

So now we have two views: **AppView** and **TodoView**. The former needs to be instantiated on page load so its code gets executed. This can be accomplished through jQuery's `ready()` utility, which will execute a function when the DOM is loaded.

```
// js/app.js

var app = app || {};
var ENTER_KEY = 13;

$(function() {

    // Kick things off by creating the App.
    new app.AppView();

});
```

In action

Let's pause and ensure that the work we've done so far functions as intended.

If you are following along, open `file:/*path*/index.html` in your web browser and monitor its console. If all is well, you shouldn't see any JavaScript errors other than regarding the `router.js` file that we haven't created yet. The todo list should be blank as we haven't yet created any todos. Plus, there is some additional work we'll need to do before the user interface fully functions.

However, a few things can be tested through the JavaScript console.

In the console, add a new todo item: `window.app.Todos.create({ title: 'My first Todo item' });` and hit return.

If all is functioning properly, this should log the new todo we've just added to the todos collection. The newly created todo is also saved to Local Storage and will be available on page refresh.

`window.app.Todos.create()` executes a collection method (`Collection.create(attributes, [options])`) which instantiates a new model item of the type passed into the collection definition, in our case `app.Todo`:

```
// from our js/collections/todos.js

var TodoList = Backbone.Collection.extend({
```




```
    model: app.Todo // the model type used by collection.create() to instantiate new models
    ...
  });
```

Run the following in the console to check it out:

```
var secondTodo = window.app.Todos.create({ title: 'My second Todo item'});
secondTodo instanceof app.Todo // returns true
```

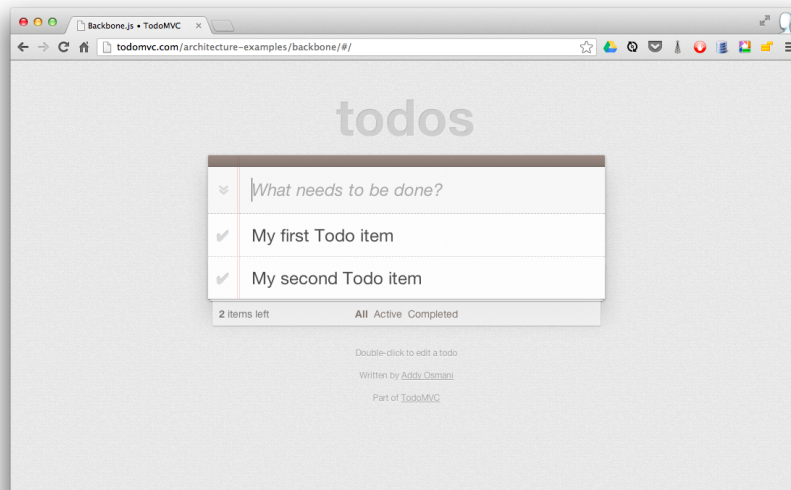
Now refresh the page and we should be able to see the fruits of our labour.

The todos added through the console should still appear in the list since they are populated from the Local Storage. Also, we should be able to create a new todo by typing a title and pressing enter.

Excellent, we're making great progress, but what about completing and deleting todos?

Completing & deleting todos

The next part of our tutorial is going to cover completing and deleting todos. These two actions are specific to each Todo item, so we need to add this functionality to the `TodoView` view. We will do so by adding `togglecompleted()` and `clear()` methods along with corresponding entries in the `events` hash.



```
// js/view/todos.js

var app = app || {};

// Todo Item View
// -----

// The DOM element for a todo item...
app.TodoView = Backbone.View.extend({

  //... is a list tag.
  tagName: 'li',

  // Cache the template function for a single item.
  template: _.template( $('#item-template').html() ),

  // The DOM events specific to an item.
  events: {
    'click .toggle': 'toggleCompleted', // NEW
    'dblclick label': 'edit',
    'click .destroy': 'clear',           // NEW
    'keypress .edit': 'updateOnEnter',
    'blur .edit': 'close'
  },
});
```

```

// The TodoView listens for changes to its model, re-rendering. Since there's
// a one-to-one correspondence between a Todo and a TodoView in this
// app, we set a direct reference on the model for convenience.
initialize: function() {
    this.listenTo(this.model, 'change', this.render);
    this.listenTo(this.model, 'destroy', this.remove); // NEW
    this.listenTo(this.model, 'visible', this.toggleVisible); // NEW
},

// Re-render the titles of the todo item.
render: function() {
    this.$el.html( this.template( this.model.toJSON() ) );

    this.$el.toggleClass( 'completed', this.model.get('completed') ); // NEW
    this.toggleVisible(); // NEW

    this.$input = this.$('.edit');
    return this;
},

// NEW - Toggles visibility of item
toggleVisible : function () {
    this.$el.toggleClass( 'hidden', this.isHidden());
},

// NEW - Determines if item should be hidden
isHidden : function () {
    var isCompleted = this.model.get('completed');
    return ( // hidden cases only
        (!isCompleted && app.TODOFilter === 'completed')
        || (isCompleted && app.TODOFilter === 'active')
    );
},

// NEW - Toggle the "completed" state of the model.
togglecompleted: function() {
    this.model.toggle();
},

// Switch this view into "editing" mode, displaying the input field.
edit: function() {
    this.$el.addClass('editing');
    this.$input.focus();
},

// Close the "editing" mode, saving changes to the todo.

```

```

close: function() {
    var value = this.$input.val().trim();

    if ( value ) {
        this.model.save({ title: value });
    } else {
        this.clear(); // NEW
    }

    this.$el.removeClass('editing');
},

// If you hit 'enter', we're through editing the item.
updateOnEnter: function( e ) {
    if ( e.which === ENTER_KEY ) {
        this.close();
    }
},

// NEW - Remove the item, destroy the model from *localStorage* and delete its view.
clear: function() {
    this.model.destroy();
}
});

```

The key part of this is the two event handlers we've added, a `togglecompleted` event on the todo's checkbox, and a click event on the todo's `<button class="destroy" />` button.

Let's look at the events that occur when we click the checkbox for a todo item:

1. The `togglecompleted()` function is invoked which calls `toggle()` on the todo model.
2. `toggle()` toggles the completed status of the todo and calls `save()` on the model.
3. The save generates a `change` event on the model which is bound to our `TodoView`'s `render()` method. We've added a statement in `render()` which toggles the completed class on the element depending on the model's completed state. The associated CSS changes the color of the title text and strikes a line through it when the todo is completed.
4. The save also results in a `change:completed` event on the model which is handled by the `AppView`'s `filterOne()` method. If we look back at the `AppView`, we see that `filterOne()` will trigger a `visible` event on the model. This is used in conjunction with the filtering in our routes and collections so that we only display an item if its completed state falls in line with the current filter. In our update to the `TodoView`, we bound the

model's visible event to the `toggleVisible()` method. This method uses the new `isHidden()` method to determine if the todo should be visible and updates it accordingly.

Now let's look at what happens when we click on a todo's destroy button:

1. The `clear()` method is invoked which calls `destroy()` on the todo model.
2. The todo is deleted from local storage and a `destroy` event is triggered.
3. In our update to the `TodoView`, we bound the model's `destroy` event to the view's inherited `remove()` method. This method deletes the view and automatically removes the associated element from the DOM. Since we used `listenTo()` to bind the view's listeners to its model, `remove()` also unbinds the listening callbacks from the model ensuring that a memory leak does not occur.
4. `destroy()` also removes the model from the `Todos` collection, which triggers a `remove` event on the collection.
5. Since the `AppView` has its `render()` method bound to **all** events on the `Todos` collection, that view is rendered and the stats in the footer are updated.

That's all there is to it!

If you want to see an example of those, see the [complete source](#).

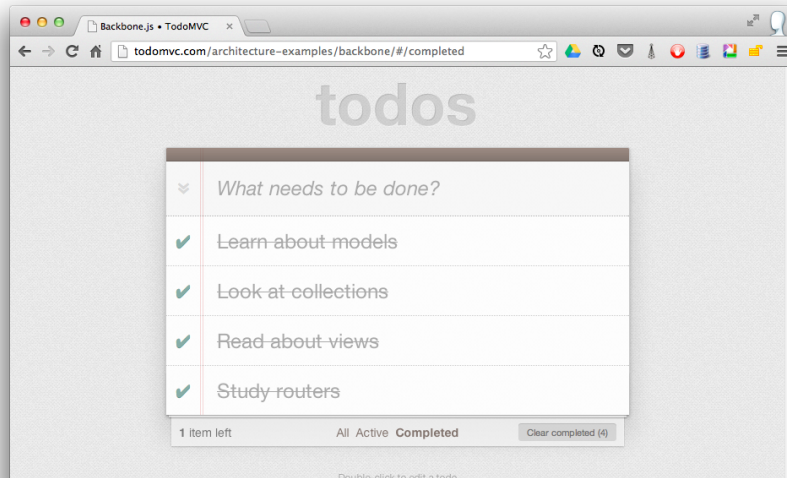
Todo routing

Finally, we move on to routing, which will allow us to easily filter the list of items that are active as well as those which have been completed. We'll be supporting the following routes:

```
#!/ (all - default)
#/active
#/completed
```

When the route changes, the todo list will be filtered on a model level and the selected class on the filter links in the footer will be toggled as described above. When an item is updated while a filter is active it will be updated accordingly (e.g., if the filter is active and the item is checked, it will be hidden). The active filter is persisted on reload.

```
// js/routers/router.js
```



```
// Todo Router
// -----

var Workspace = Backbone.Router.extend({
  routes:{
    '*filter': 'setFilter'
  },

  setFilter: function( param ) {
    // Set the current filter to be used
    app.TODOFilter = param.trim() || '';

    // Trigger a collection filter event, causing hiding/unhiding
    // of Todo view items
    app.Todos.trigger('filter');
  }
});

app.TODORouter = new Workspace();
Backbone.history.start();
```

Our router uses a `*splat` to set up a default route which passes the string after `'#/'` in the URL to `setFilter()` which sets `window.app.TODOFilter` to that string.

As we can see in the line `window.app.Todos.trigger('filter')`, once the filter has been set, we simply trigger 'filter' on our Todos collection to toggle which items are visible and which are hidden. Recall that our AppView's `filterAll()` method is bound to the collection's filter event and that any event on the collection will cause the AppView to re-render.

Finally, we create an instance of our router and call `Backbone.history.start()` to route the initial URL during page load.

Summary

We've now built our first complete Backbone.js application. The latest version of the full app can be viewed online at any time and the sources are readily available via [TodoMVC](#).

Later on in the book, we'll learn how to further modularize this application using RequireJS, swap out our persistence layer to a database back-end, and finally unit test the application with a few different testing frameworks.

Exercise 2: Book Library - Your First RESTful Backbone.js App

While our first application gave us a good taste of how Backbone.js applications are made, most real-world applications will want to communicate with a back-end of some sort. Let's reinforce what we have already learned with another example, but this time we will also create a RESTful API for our application to talk to.

In this exercise we will build a library application for managing digital books using Backbone. For each book we will store the title, author, date of release, and some keywords. We'll also show a picture of the cover.

Setting up

First we need to create a folder structure for our project. To keep the front-end and back-end separate, we will create a folder called *site* for our client in the project root. Within it we will create *css*, *img*, and *js* directories.

As with the last example we will split our JavaScript files by their function, so under the *js* directory create folders named *lib*, *models*, *collections*, and *views*. Your directory hierarchy should look like this:

```
site/  
  css/  
  img/
```

```
js/  
  collections/  
  lib/  
  models/  
  views/
```

Download the Backbone, Underscore, and jQuery libraries and copy them to your js/lib folder. We need a placeholder image for the book covers. Save this image to your site/img folder:



Just like before we need to load all of our dependencies in the site/index.html file:

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="UTF-8"/>  
    <title>Backbone.js Library</title>  
    <link rel="stylesheet" href="css/screen.css">  
  </head>  
  <body>  
    <script src="js/lib/jquery.min.js"></script>  
    <script src="js/lib/underscore-min.js"></script>  
    <script src="js/lib/backbone-min.js"></script>  
    <script src="js/models/book.js"></script>  
    <script src="js/collections/library.js"></script>  
    <script src="js/views/book.js"></script>  
    <script src="js/views/library.js"></script>  
    <script src="js/app.js"></script>  
  </body>  
</html>
```


We should also add in the HTML for the user interface. We'll want a form for adding a new book so add the following immediately inside the `body` element:

```
<div id="books">
  <form id="addBook" action="#">
    <div>
      <label for="coverImage">CoverImage: </label><input id="coverImage" type="file" />
      <label for="title">Title: </label><input id="title" type="text" />
      <label for="author">Author: </label><input id="author" type="text" />
      <label for="releaseDate">Release date: </label><input id="releaseDate" type="text" />
      <label for="keywords">Keywords: </label><input id="keywords" type="text" />
      <button id="add">Add</button>
    </div>
  </form>
</div>
```

and we'll need a template for displaying each book which should be placed before the `script` tags:

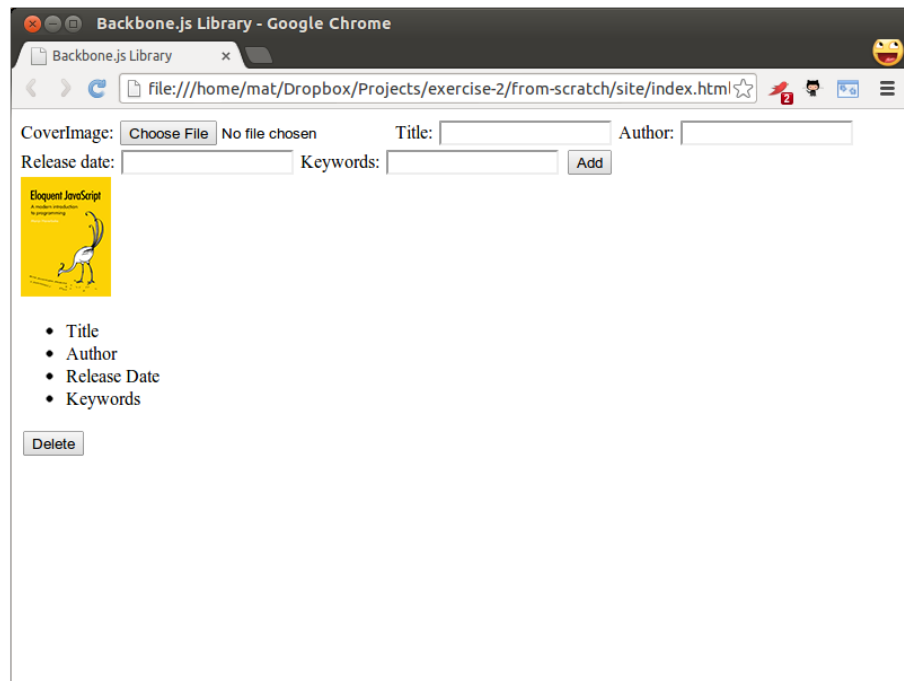
```
<script id="bookTemplate" type="text/template">
  
  <ul>
    <li><%= title %></li>
    <li><%= author %></li>
    <li><%= releaseDate %></li>
    <li><%= keywords %></li>
  </ul>

  <button class="delete">Delete</button>
</script>
```

To see what this will look like with some data in it, go ahead and add a manually filled-in book to the `books` div.

```
<div class="bookContainer">
  
  <ul>
    <li>Title</li>
    <li>Author</li>
    <li>Release Date</li>
    <li>Keywords</li>
  </ul>

  <button class="delete">Delete</button>
</div>
```



Open this file in a browser and it should look something like this:

Not so great. This is not a CSS tutorial, but we still need to do some formatting. Create a file named `screen.css` in your `site/css` folder:

```
body {
    background-color: #eee;
}

.bookContainer {
    outline: 1px solid #aaa;
    width: 350px;
    height: 130px;
    background-color: #fff;
    float: left;
    margin: 5px;
}

.bookContainer img {
    float: left;
    margin: 10px;
}
```

```

.bookContainer ul {
    list-style-type: none;
    margin-bottom: 0;
}

.bookContainer button {
    float: right;
    margin: 10px;
}

#addBook label {
    width: 100px;
    margin-right: 10px;
    text-align: right;
    line-height: 25px;
}

#addBook label, #addBook input {
    display: block;
    margin-bottom: 10px;
    float: left;
}

#addBook label[for="title"], #addBook label[for="releaseDate"] {
    clear: both;
}

#addBook button {
    display: block;
    margin: 5px 20px 10px 10px;
    float: right;
    clear: both;
}

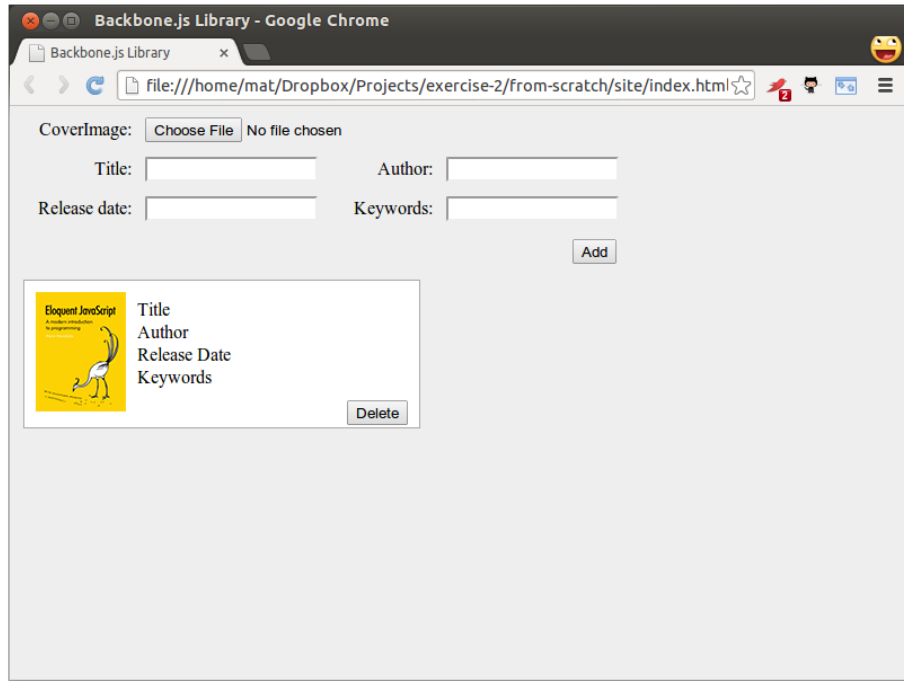
#addBook div {
    width: 550px;
}

#addBook div:after {
    content: "";
    display: block;
    height: 0;
    visibility: hidden;
    clear: both;
    font-size: 0;
    line-height: 0;
}

```

}

Now it looks a bit better:



So this is what we want the final result to look like, but with more books. Go ahead and copy the bookContainer div a few more times if you would like to see what it looks like. Now we are ready to start developing the actual application.

Creating the Model, Collection, Views, and App First, we'll need a model of a book and a collection to hold the list. These are both very simple, with the model only declaring some defaults:

```
// site/js/models/book.js

var app = app || {};

app.Book = Backbone.Model.extend({
  defaults: {
    coverImage: 'img/placeholder.png',
    title: 'No title',
    author: 'Unknown',
    releaseDate: 'Unknown',
```

```

        keywords: 'None'
    }
});

// site/js/collections/library.js

var app = app || {};

app.Library = Backbone.Collection.extend({
    model: app.Book
});

```

Next, in order to display books we'll need a view:

```

// site/js/views/book.js

var app = app || {};

app.BookView = Backbone.View.extend({
    tagName: 'div',
    className: 'bookContainer',
    template: $( '#bookTemplate' ).html(),

    render: function() {
        //tpl is a function that takes a JSON object and returns html
        var tpl = _.template( this.template );

        //this.el is what we defined in tagName. use $el to get access to jQuery html() fun
        this.$el.html( tpl( this.model.toJSON() ) );

        return this;
    }
});

```

We'll also need a view for the list itself:

```

// site/js/views/library.js

var app = app || {};

app.LibraryView = Backbone.View.extend({
    el: $( '#books' ),

    initialize: function( initialBooks ) {

```

```

        this.collection = new app.Library( initialBooks );
        this.render();
    },

    // render library by rendering each book in its collection
    render: function() {
        this.collection.each(function( item ) {
            this.renderBook( item );
        }, this );
    },

    // render a book by creating a BookView and appending the
    // element it renders to the library's element
    renderBook: function( item ) {
        var bookView = new app.BookView({
            model: item
        });
        this.$el.append( bookView.render().el );
    }
});

```

Note that in the initialize function we accept an array of data that we pass to the app.Library constructor. We'll use this to populate our collection with some sample data so that we can see everything is working correctly. Finally, we have the entry point for our code, along with the sample data:

```

// site/js/app.js

var app = app || {};

$(function() {
    var books = [
        { title: 'JavaScript: The Good Parts', author: 'Douglas Crockford', releaseDate: '2005', keywords: 'javascript, good parts' },
        { title: 'The Little Book on CoffeeScript', author: 'Alex MacCaw', releaseDate: '2010', keywords: 'coffeescript, little book' },
        { title: 'Scala for the Impatient', author: 'Cay S. Horstmann', releaseDate: '2012', keywords: 'scala, impatient' },
        { title: 'American Psycho', author: 'Bret Easton Ellis', releaseDate: '1991', keywords: 'american psycho, bret easton ellis' },
        { title: 'Eloquent JavaScript', author: 'Marijn Haverbeke', releaseDate: '2011', keywords: 'eloquent javascript, marijn haverbeke' }
    ];

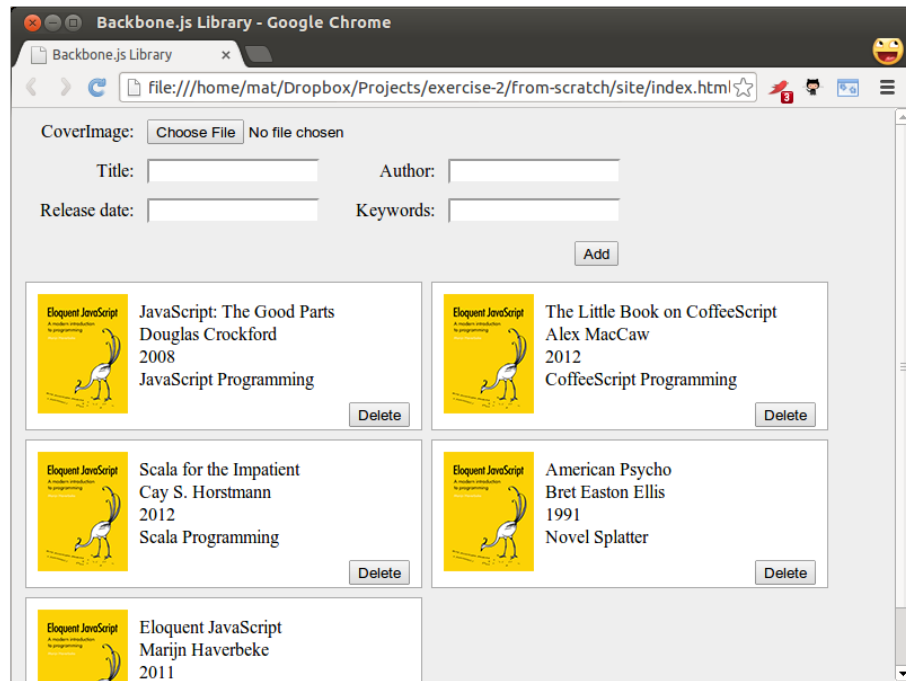
    new app.LibraryView( books );
});

```

Our app just passes the sample data to a new instance of app.LibraryView that it creates. Since the initialize() constructor in LibraryView invokes the view's render() method, all the books in the library will be displayed. Since we are

passing our entry point as a callback to jQuery (in the form of its \$ alias), the function will execute when the DOM is ready.

If you view index.html in a browser you should see something like this:



This is a complete Backbone application, though it doesn't yet do anything interesting.

Wiring in the interface

Now we'll add some functionality to the useless form at the top and the delete buttons on each book.

Adding models

When the user clicks the add button we want to take the data in the form and use it to create a new model. In the LibraryView we need to add an event handler for the click event:

```
events:{  
  'click #add': 'addBook'  
},
```

```

addBook: function( e ) {
  e.preventDefault();

  var formData = {};

  $( '#addBook div' ).children( 'input' ).each( function( i, el ) {
    if( $( el ).val() != '' )
    {
      formData[ el.id ] = $( el ).val();
    }
  });

  this.collection.add( new app.Book( formData ) );
},

```

We select all the input elements of the form that have a value and iterate over them using jQuery's each. Since we used the same names for ids in our form as the keys on our Book model we can simply store them directly in the formData object. We then create a new Book from the data and add it to the collection. We skip fields without a value so that the defaults will be applied.

Backbone passes an event object as a parameter to the event-handling function. This is useful for us in this case since we don't want the form to actually submit and reload the page. Adding a call to `preventDefault` on the event in the `addBook` function takes care of this for us.

Now we just need to make the view render again when a new model is added. To do this, we put

```

this.listenTo( this.collection, 'add', this.renderBook );

```

in the initialize function of LibraryView.

Now you should be ready to take the application for a spin.

You may notice that the file input for the cover image isn't working, but that is left as an exercise to the reader.

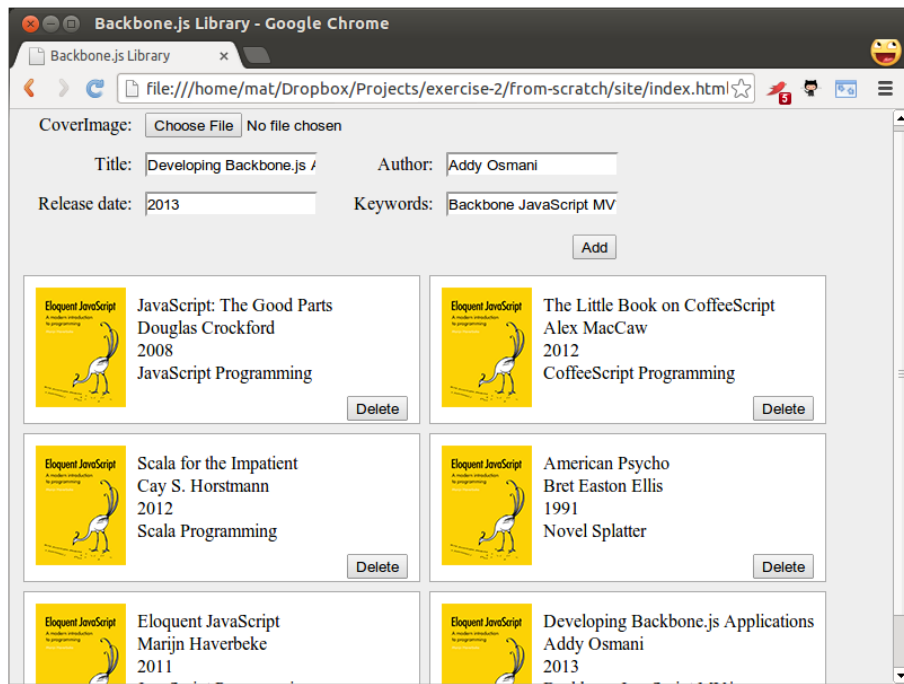
Removing models

Next, we need to wire up the delete button. Set up the event handler in the BookView:

```

events: {
  'click .delete': 'deleteBook'
}

```

```

},

deleteBook: function() {
    //Delete model
    this.model.destroy();

    //Delete view
    this.remove();
},

```

You should now be able to add and remove books from the library.

Creating the back-end

Now we need to make a small detour and set up a server with a REST api. Since this is a JavaScript book we will use JavaScript to create the server using node.js. If you are more comfortable in setting up a REST server in another language, this is the API you need to conform to:

url	HTTP Method	Operation
/api/books	GET	Get an array of all books

/api/books/:id	GET	Get the book with id of :id
/api/books	POST	Add a new book and return the book with an id attribute added
/api/books/:id	PUT	Update the book with id of :id
/api/books/:id	DELETE	Delete the book with id of :id

The outline for this section looks like this:

- Install node.js, npm, and MongoDB
- Install node modules
- Create a simple web server
- Connect to the database
- Create the REST API

Install node.js, npm, and MongoDB

Download and install node.js from nodejs.org. The node package manager (npm) will be installed as well.

Download and install MongoDB from mongodb.org. There are detailed installation guides [on the website](#).

Install node modules

Create a file called `package.json` in the root of your project. It should look like

```
{
  "name": "backbone-library",
  "version": "0.0.1",
  "description": "A simple library application using the Backbone framework",
  "dependencies": {
    "express": "~3.1.0",
    "path": "~0.4.9",
    "mongoose": "~3.5.5"
  }
}
```

Amongst other things, this file tells npm what the dependencies are for our project. On the command line, from the root of your project, type:

```
npm install
```

You should see npm fetch the dependencies that we listed in our `package.json` and save them within a folder called `node_modules`.

Your folder structure should look something like this:

```
node_modules/  
  .bin/  
  express/  
  mongoose/  
  path/  
site/  
  css/  
  img/  
  js/  
  index.html  
package.json
```

Create a simple web server

Create a file named server.js in the project root containing the following code:

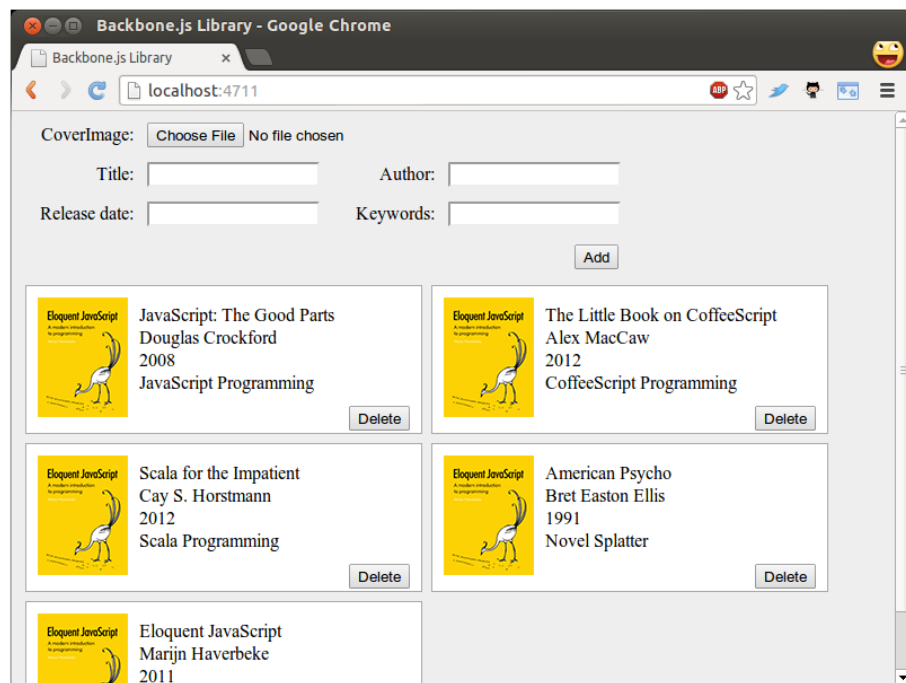
```
// Module dependencies.  
var application_root = __dirname,  
    express = require( 'express' ), //Web framework  
    path = require( 'path' ), //Utilities for dealing with file paths  
    mongoose = require( 'mongoose' ); //MongoDB integration  
  
//Create server  
var app = express();  
  
// Configure server  
app.configure( function() {  
    //parses request body and populates request.body  
    app.use( express.bodyParser() );  
  
    //checks request.body for HTTP method overrides  
    app.use( express.methodOverride() );  
  
    //perform route lookup based on url and HTTP method  
    app.use( app.router );  
  
    //Where to serve static content  
    app.use( express.static( path.join( application_root, 'site' ) ) );  
  
    //Show all errors in development  
    app.use( express.errorHandler({ dumpExceptions: true, showStack: true }));  
});  
  
//Start server  
var port = 4711;
```

```
app.listen( port, function() {
  console.log( 'Express server listening on port %d in %s mode', port, app.settings.env );
});
```

We start off by loading the modules required for this project: Express for creating the HTTP server, Path for dealing with file paths, and mongoose for connecting with the database. We then create an Express server and configure it using an anonymous function. This is a pretty standard configuration and for our application we don't actually need the `methodOverride` part. It is used for issuing PUT and DELETE HTTP requests directly from a form, since forms normally only support GET and POST. Finally, we start the server by running the `listen` function. The port number used, in this case 4711, could be any free port on your system. I simply used 4711 since it is unlikely to have been used by anything else. We are now ready to run our first server:

```
node server.js
```

If you open a browser on `http://localhost:4711` you should see something like this:

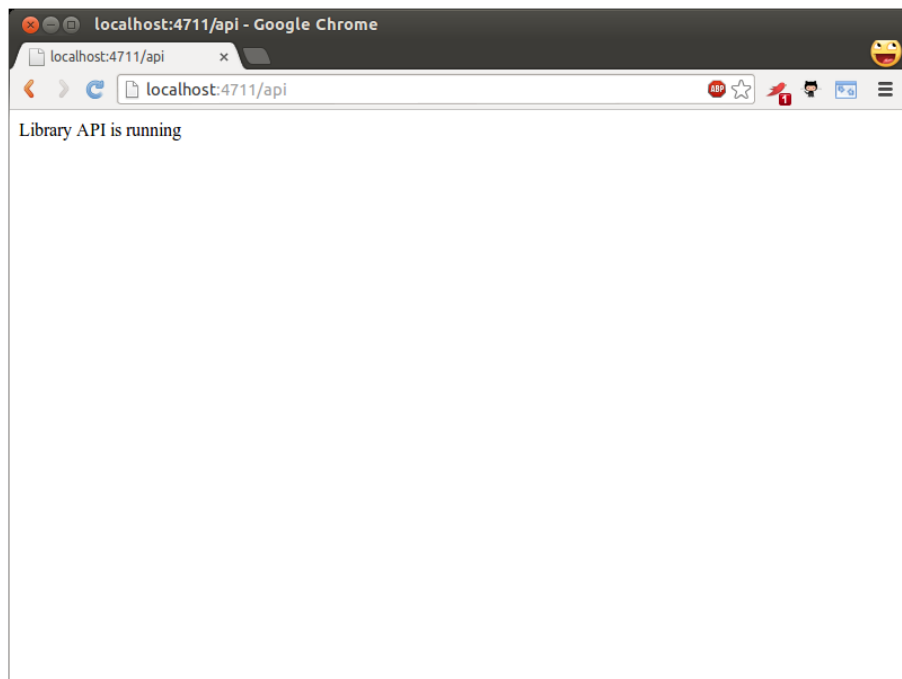


This is where we left off in Part 2, but we are now running on a server instead of directly from the files. Great job! We can now start defining routes (URLs) that

the server should react to. This will be our REST API. Routes are defined by using `app` followed by one of the HTTP verbs `get`, `put`, `post`, and `delete`, which corresponds to Create, Read, Update and Delete. Let us go back to `server.js` and define a simple route:

```
// Routes
app.get( '/api', function( request, response ) {
  response.send( 'Library API is running' );
});
```

The `get` function takes a URL as the first parameter and a function as the second. The function will be called with `request` and `response` objects. Now you can restart node and go to our specified URL:



Connect to the database

Fantastic. Now, since we want to store our data in MongoDB, we need to define a schema. Add this to `server.js`:

```
//Connect to database
mongoose.connect( 'mongodb://localhost/library_database' );
```

```

//Schemas
var Book = new mongoose.Schema({
  title: String,
  author: String,
  releaseDate: Date
});

//Models
var BookModel = mongoose.model( 'Book', Book );

```

As you can see, schema definitions are quite straight forward. They can be more advanced, but this will do for us. I also extracted a model (BookModel) from Mongo. This is what we will be working with. Next up, we define a GET operation for the REST API that will return all books:

```

//Get a list of all books
app.get( '/api/books', function( request, response ) {
  return BookModel.find( function( err, books ) {
    if( !err ) {
      return response.send( books );
    } else {
      return console.log( err );
    }
  });
});

```

The find function of Model is defined like this: `function find (conditions, fields, options, callback)` – but since we want a function that returns all books we only need the callback parameter. The callback will be called with an error object and an array of found objects. If there was no error we return the array of objects to the client using the `send` function of the response object, otherwise we log the error to the console.

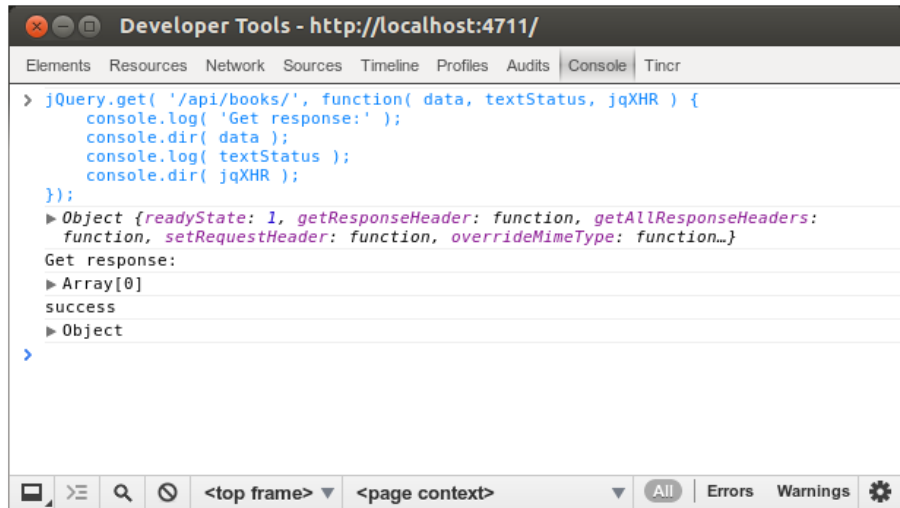
To test our API we need to do a little typing in a JavaScript console. Restart node and go to localhost:4711 in your browser. Open up the JavaScript console. If you are using Google Chrome, go to View->Developer->JavaScript Console. If you are using Firefox, install Firebug and go to View->Firebug. Most other browsers will have a similar console. In the console type the following:

```

jQuery.get( '/api/books/', function( data, textStatus, jqXHR ) {
  console.log( 'Get response:' );
  console.dir( data );
  console.log( textStatus );
  console.dir( jqXHR );
});

```

... and press enter and you should get something like this:



Here I used jQuery to make the call to our REST API, since it was already loaded on the page. The returned array is obviously empty, since we have not put anything into the database yet. Let's go and create a POST route that enables adding new items in server.js:

```
//Insert a new book
app.post( '/api/books', function( request, response ) {
  var book = new BookModel({
    title: request.body.title,
    author: request.body.author,
    releaseDate: request.body.releaseDate
  });
  book.save( function( err ) {
    if( !err ) {
      return console.log( 'created' );
    } else {
      return console.log( err );
    }
  });
  return response.send( book );
});
```

We start by creating a new BookModel, passing an object with title, author, and releaseDate attributes. The data are collected from request.body. This means that anyone calling this operation in the API needs to supply a JSON object

containing the title, author, and releaseDate attributes. Actually, the caller can omit any or all attributes since we have not made any of them mandatory.

We then call the save function on the BookModel passing in a callback in the same way as with the previous get route. Finally, we return the saved BookModel. The reason we return the BookModel and not just “success” or similar string is that when the BookModel is saved it will get an `__id` attribute from MongoDB, which the client needs when updating or deleting a specific book. Let’s try it out again. Restart node and go back to the console and type:

```
jQuery.post( '/api/books', {
  'title': 'JavaScript the good parts',
  'author': 'Douglas Crockford',
  'releaseDate': new Date( 2008, 4, 1 ).getTime()
}, function(data, textStatus, jqXHR) {
  console.log( 'Post response:' );
  console.dir( data );
  console.log( textStatus );
  console.dir( jqXHR );
});
```

..and then

```
jQuery.get( '/api/books/', function( data, textStatus, jqXHR ) {
  console.log( 'Get response:' );
  console.dir( data );
  console.log( textStatus );
  console.dir( jqXHR );
});
```

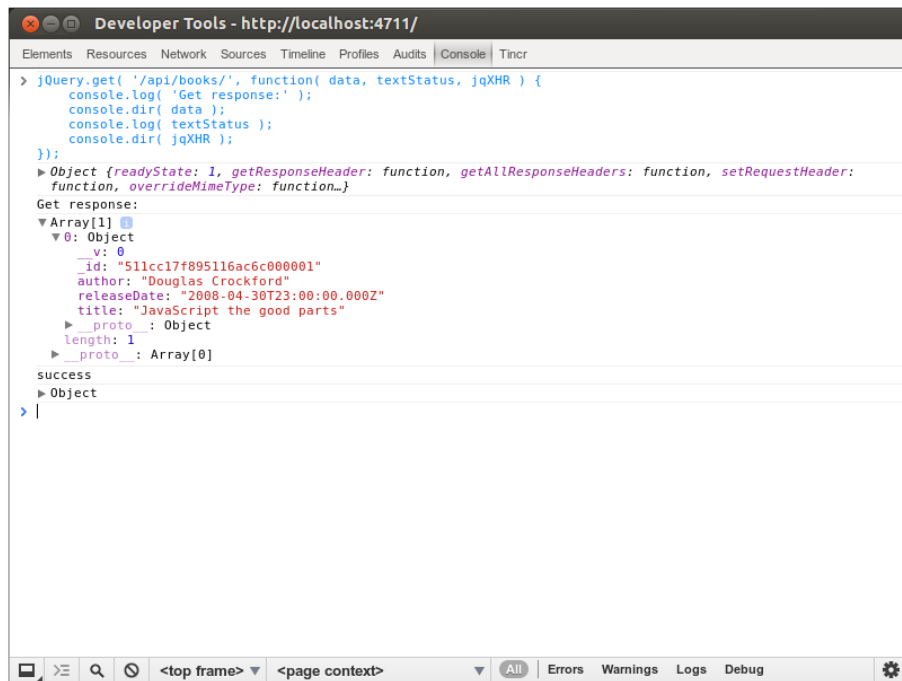
You should now get a one-element array back from our server. You may wonder about this line:

```
'releaseDate': new Date(2008, 4, 1).getTime()
```

MongoDB expects dates in UNIX time format (milliseconds from the start of Jan 1st 1970 UTC), so we have to convert dates before posting. The object we get back however, contains a JavaScript Date object. Also note the `__id` attribute of the returned object.

Let’s move on to creating a GET request that retrieves a single book in server.js:

```
//Get a single book by id
app.get( '/api/books/:id', function( request, response ) {
  return BookModel.findById( request.params.id, function( err, book ) {
```

```

    if( !err ) {
        return response.send( book );
    } else {
        return console.log( err );
    }
  });
});

```

Here we use colon notation (:id) to tell Express that this part of the route is dynamic. We also use the `findById` function on `BookModel` to get a single result. If you restart node, you can get a single book by adding the id previously returned to the URL like this:

```

jQuery.get( '/api/books/4f95a8cb1baa9b8a1b000006', function( data, textStatus, jqXHR ) {
  console.log( 'Get response:' );
  console.dir( data );
  console.log( textStatus );
  console.dir( jqXHR );
});

```

Let's create the PUT (update) function next:

```

//Update a book
app.put( '/api/books/:id', function( request, response ) {
  console.log( 'Updating book ' + request.body.title );
  return BookModel.findById( request.params.id, function( err, book ) {
    book.title = request.body.title;
    book.author = request.body.author;
    book.releaseDate = request.body.releaseDate;

    return book.save( function( err ) {
      if( !err ) {
        console.log( 'book updated' );
      } else {
        console.log( err );
      }
      return response.send( book );
    });
  });
});

```

This is a little larger than previous ones, but is also pretty straight forward – we find a book by id, update its properties, save it, and send it back to the client.

To test this we need to use the more general jQuery ajax function. Again, in these examples you will need to replace the id property with one that matches an item in your own database:

```

jQuery.ajax({
  url: '/api/books/4f95a8cb1baa9b8a1b000006',
  type: 'PUT',
  data: {
    'title': 'JavaScript The good parts',
    'author': 'The Legendary Douglas Crockford',
    'releaseDate': new Date( 2008, 4, 1 ).getTime()
  },
  success: function( data, textStatus, jqXHR ) {
    console.log( 'Post response:' );
    console.dir( data );
    console.log( textStatus );
    console.dir( jqXHR );
  }
});

```

Finally we create the delete route:

```

//Delete a book
app.delete( '/api/books/:id', function( request, response ) {

```

```

        console.log( 'Deleting book with id: ' + request.params.id );
        return BookModel.findById( request.params.id, function( err, book ) {
            return book.remove( function( err ) {
                if( !err ) {
                    console.log( 'Book removed' );
                    return response.send( '' );
                } else {
                    console.log( err );
                }
            });
        });
    });
});

```

...and try it out:

```

jQuery.ajax({
    url: '/api/books/4f95a5251baa9b8a1b000001',
    type: 'DELETE',
    success: function( data, textStatus, jqXHR ) {
        console.log( 'Post response:' );
        console.dir( data );
        console.log( textStatus );
        console.dir( jqXHR );
    }
});

```

So now our REST API is complete – we have support for all four HTTP verbs. What's next? Well, until now I have left out the keywords part of our books. This is a bit more complicated since a book could have several keywords and we don't want to represent them as a string, but rather an array of strings. To do that we need another schema. Add a Keywords schema right above our Book schema:

```

//Schemas
var Keywords = new mongoose.Schema({
    keyword: String
});

```

To add a sub schema to an existing schema we use brackets notation like so:

```

var Book = new mongoose.Schema({
    title: String,
    author: String,
    releaseDate: Date,
    keywords: [ Keywords ]
});

```

// NEW

Also update POST and PUT:

```
//Insert a new book
app.post( '/api/books', function( request, response ) {
    var book = new BookModel({
        title: request.body.title,
        author: request.body.author,
        releaseDate: request.body.releaseDate,
        keywords: request.body.keywords // NEW
    });
    book.save( function( err ) {
        if( !err ) {
            return console.log( 'created' );
        } else {
            return console.log( err );
        }
    });
    return response.send( book );
});

//Update a book
app.put( '/api/books/:id', function( request, response ) {
    console.log( 'Updating book ' + request.body.title );
    return BookModel.findById( request.params.id, function( err, book ) {
        book.title = request.body.title;
        book.author = request.body.author;
        book.releaseDate = request.body.releaseDate;
        book.keywords = request.body.keywords; // NEW

        return book.save( function( err ) {
            if( !err ) {
                console.log( 'book updated' );
            } else {
                console.log( err );
            }
        });
        return response.send( book );
    });
});
```

There we are, that should be all we need, now we can try it out in the console:

```
jQuery.post( '/api/books', {
    'title': 'Secrets of the JavaScript Ninja',
    'author': 'John Resig',
```

```

    'releaseDate': new Date( 2008, 3, 12 ).getTime(),
    'keywords':[
      { 'keyword': 'JavaScript' },
      { 'keyword': 'Reference' }
    ]
  }, function( data, textStatus, jqXHR ) {
    console.log( 'Post response:' );
    console.dir( data );
    console.log( textStatus );
    console.dir( jqXHR );
  });

```

You now have a fully functional REST server that we can hook into from our front-end.

Talking to the server

In this part we will cover connecting our Backbone application to the server through the REST API.

As we mentioned in chapter 3 *Backbone Basics*, we can retrieve models from a server using `collection.fetch()` by setting `collection.url` to be the URL of the API endpoint. Let's update the Library collection to do that now:

```

var app = app || {};

app.Library = Backbone.Collection.extend({
  model: app.Book,
  url: '/api/books'      // NEW
});

```

This results in the default implementation of `Backbone.sync` assuming that the API looks like this:

url	HTTP Method	Operation
/api/books	GET	Get an array of all books
/api/books/:id	GET	Get the book with id of :id
/api/books	POST	Add a new book and return the book with an id attribute added
/api/books/:id	PUT	Update the book with id of :id
/api/books/:id	DELETE	Delete the book with id of :id

To have our application retrieve the Book models from the server on page load we need to update the LibraryView. The Backbone documentation recommends inserting all models when the page is generated on the server side, rather than

fetching them from the client side once the page is loaded. Since this chapter is trying to give you a more complete picture of how to communicate with a server, we will go ahead and ignore that recommendation. Go to the `LibraryView` declaration and update the `initialize` function as follows:

```
initialize: function() {
  this.collection = new app.Library();
  this.collection.fetch(); // NEW
  this.render();

  this.listenTo( this.collection, 'add', this.renderBook );
  this.listenTo( this.collection, 'reset', this.render ); // NEW
},
```

Now that we are populating our `Library` from the database using `this.collection.fetch()`, the `initialize()` function no longer takes a set of sample data as an argument and doesn't pass anything to the `app.Library` constructor. You can now remove the sample data from `site/js/app.js`, which should reduce it to a single statement which creates the `LibraryView`:

```
// site/js/app.js

var app = app || {};

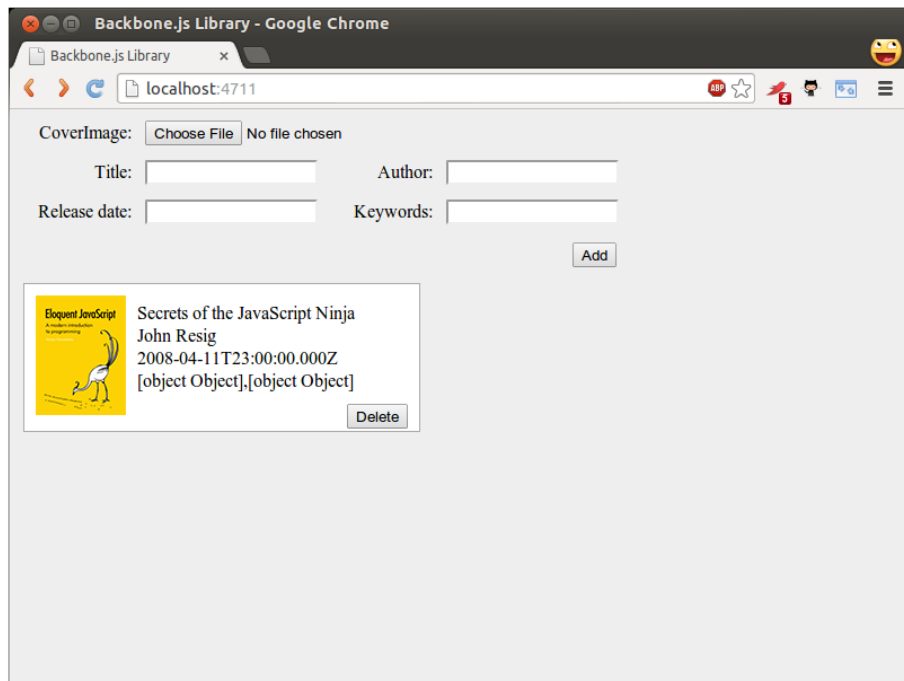
$(function() {
  new app.LibraryView();
});
```

We have also added a listener on the `reset` event. We need to do this since the models are fetched asynchronously after the page is rendered. When the fetch completes, Backbone fires the `reset` event and our listener re-renders the view. If you reload the page now you should see all books that are stored on the server:

As you can see the date and keywords look a bit weird. The date delivered from the server is converted into a JavaScript `Date` object and when applied to the underscore template it will use the `toString()` function to display it. There isn't very good support for formatting dates in JavaScript so we will use the `dateFormat` jQuery plugin to fix this. Go ahead and download it from [here](#) and put it in your `site/js/lib` folder. Update the book template so that the date is displayed with:

```
<li><%= $.format.date( new Date( releaseDate ), 'MMMM yyyy' ) %></li>
```

and add a script element for the plugin



```
<script src="js/lib/jquery-dateFormat-1.0.js"></script>
```

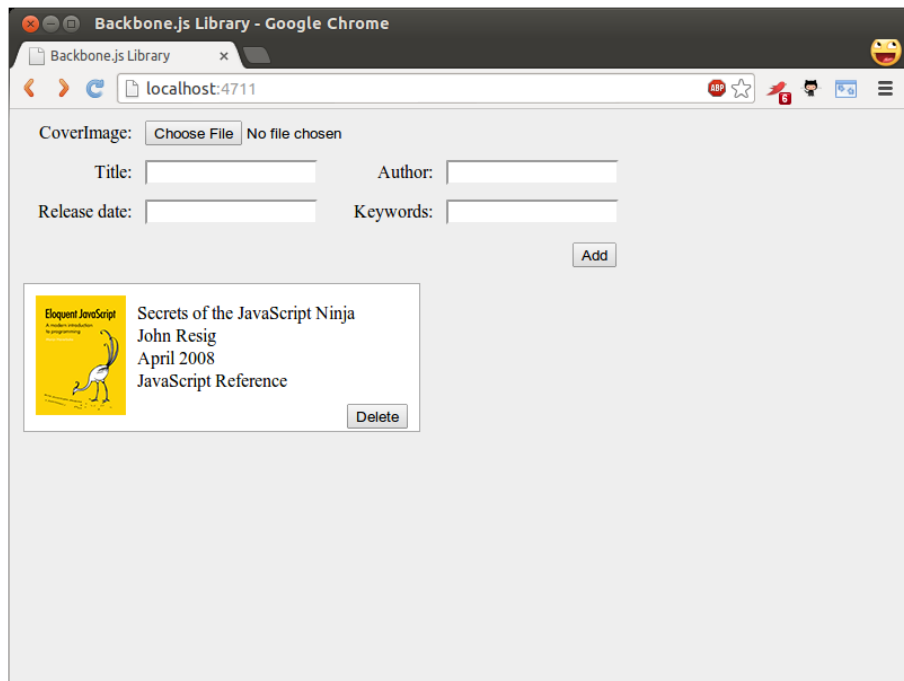
Now the date on the page should look a bit better. How about the keywords? Since we are receiving the keywords in an array we need to execute some code that generates a string of separated keywords. To do that we can omit the equals character in the template tag which will let us execute code that doesn't display anything:

```
<li><% _.each( keywords, function( keyobj ) {> <%= keyobj.keyword %><% } ); %></li>
```

Here I iterate over the keywords array using the Underscore `each` function and print out every single keyword. Note that I display the keyword using the `<%=` tag. This will display the keywords with spaces between them.

Reloading the page again should look quite decent:

Now go ahead and delete a book and then reload the page: Tadaa! the deleted book is back! Not cool, why is this? This happens because when we get the BookModels from the server they have an `_id` attribute (notice the underscore), but Backbone expects an `id` attribute (no underscore). Since no `id` attribute is present, Backbone sees this model as new and deleting a new model doesn't need any synchronization.



To fix this we can use the parse function of Backbone.Model. The parse function lets you edit the server response before it is passed to the Model constructor. Add a parse method to the Book model:

```
parse: function( response ) {  
    response.id = response._id;  
    return response;  
}
```

Simply copy the value of `_id` to the needed `id` attribute. If you reload the page you will see that models are actually deleted on the server when you press the delete button.

Another, simpler way of making Backbone recognize `_id` as its unique identifier is to set the `idAttribute` of the model to `_id`.

If you now try to add a new book using the form you'll notice that it is a similar story to delete – models won't get persisted on the server. This is because `Backbone.Collection.add` doesn't automatically sync, but it is easy to fix. In the `LibraryView` we find in `views/library.js` change the line reading:

```
this.collection.add( new Book( formData ) );
```


...to:

```
this.collection.create( formData );
```

Now newly created books will get persisted. Actually, they probably won't if you enter a date. The server expects a date in UNIX timestamp format (milliseconds since Jan 1, 1970). Also, any keywords you enter won't be stored since the server expects an array of objects with the attribute 'keyword'.

We'll start by fixing the date issue. We don't really want the users to manually enter a date in a specific format, so we'll use the standard datepicker from jQuery UI. Go ahead and create a custom jQuery UI download containing datepicker from [here](#). Add the css theme to site/css/ and the JavaScript to site/js/lib. Link to them in index.html:

```
<link rel="stylesheet" href="css/cupertino/jquery-ui-1.10.0.custom.css">
```

"cupertino" is the name of the style I chose when downloading jQuery UI.

The JavaScript file must be loaded after jQuery.

```
<script src="js/lib/jquery.min.js"></script>
<script src="js/lib/jquery-ui-1.10.0.custom.min.js"></script>
```

Now in app.js, bind a datepicker to our releaseDate field:

```
var app = app || {};

$(function() {
  $( '#releaseDate' ).datepicker();
  new app.LibraryView();
});
```

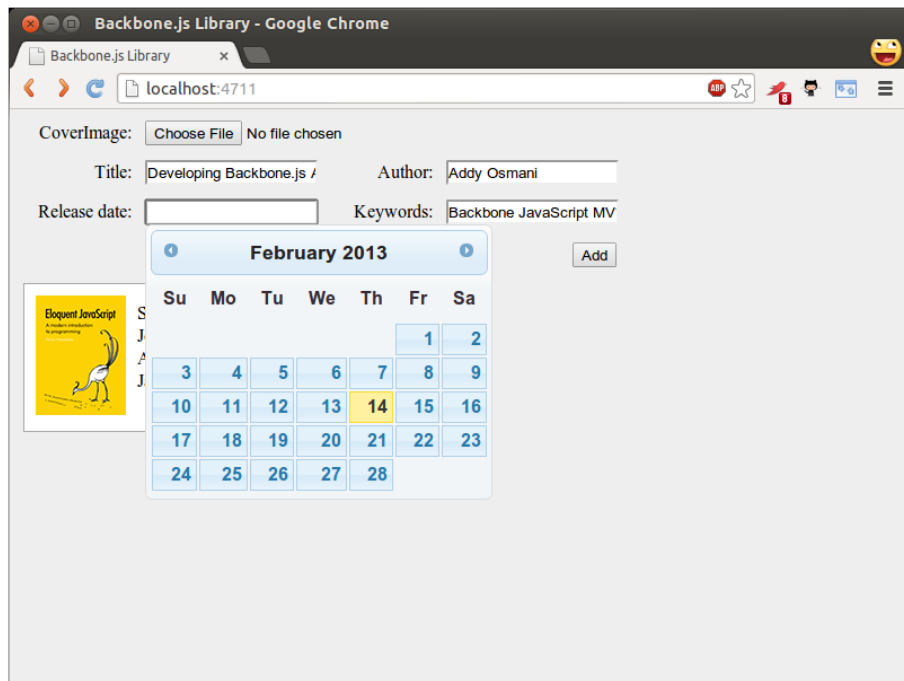
You should now be able to pick a date when clicking in the releaseDate field:

Finally, we have to make sure that the form input is properly transformed into our storage format. Change the addBook function in LibraryView to:

```
addBook: function( e ) {
  e.preventDefault();

  var formData = {};

  $( '#addBook div' ).children( 'input' ).each( function( i, el ) {
    if( $( el ).val() != '' )
```



```

{
  if( el.id === 'keywords' ) {
    formData[ el.id ] = [];
    _.each( $( el ).val().split( ' ' ), function( keyword ) {
      formData[ el.id ].push( { 'keyword': keyword } );
    });
  } else if( el.id === 'releaseDate' ) {
    formData[ el.id ] = $( '#releaseDate' ).datepicker( 'getDate' ).getTime();
  } else {
    formData[ el.id ] = $( el ).val();
  }
}
});

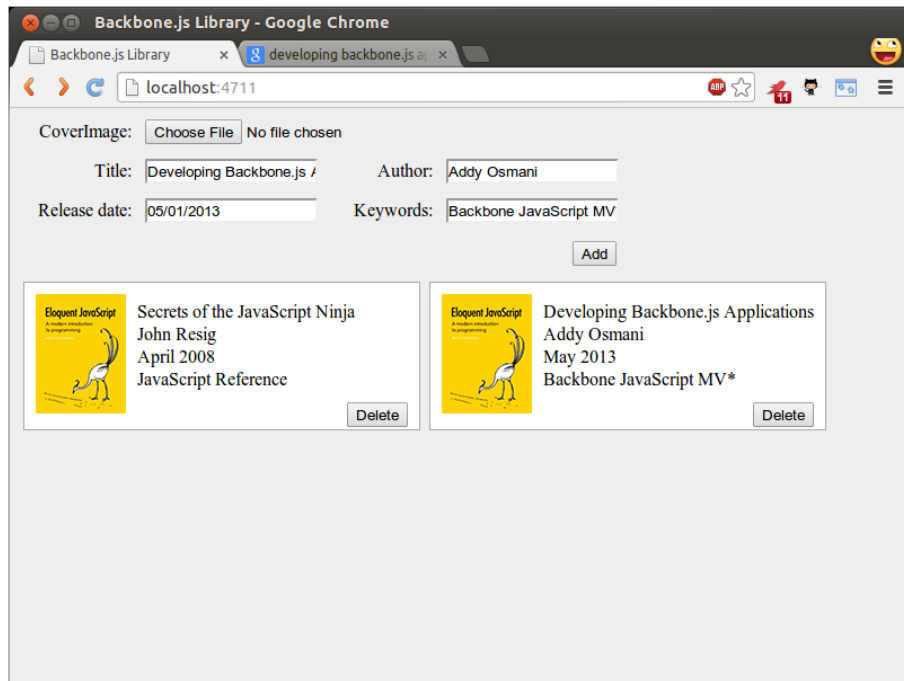
this.collection.create( formData );
},

```

Our change adds two checks to the form input fields. First, we're checking if the current element is the keywords input field, in which case we're splitting the string on each space and creating an array of keyword objects.

Then we're checking if the current element is the releaseDate input field, in which case we're calling `datepicker("getDate")` which returns a Date object.

We then use the `getTime` function on that to get the time in milliseconds.
Now you should be able to add new books with both a release date and keywords!



Summary

In this chapter we made our application persistent by binding it to a server using a REST API. We also looked at some problems that might occur when serializing and deserializing data and their solutions. We looked at the `dateFormat` and the `datepicker` jQuery plugins and how to do some more advanced things in our Underscore templates. The code is available [here](#).

Backbone Extensions

MarionetteJS (Backbone.Marionette)

By Derick Bailey & Addy Osmani

As we've seen, Backbone provides a great set of building blocks for our JavaScript applications. It gives us the core constructs that are needed to build small to mid-sized apps, organize jQuery DOM events, or create single page apps that

support mobile devices and large scale enterprise needs. But Backbone is not a complete framework. It's a set of building blocks that leaves much of the application design, architecture, and scalability to the developer, including memory management, view management, and more.

[MarionetteJS](#) (a.k.a Backbone.Marionette) provides many of the features that the non-trivial application developer needs, above what Backbone itself provides. It is a composite application library that aims to simplify the construction of large scale applications. It does this by providing a collection of common design and implementation patterns found in the applications that the creator, [Derick Bailey](#), and many other [contributors](#) have been using to build Backbone apps.

Marionette's key benefits include:

- Scaling applications out with modular, event driven architecture
- Sensible defaults, such as using Underscore templates for view rendering
- Easy to modify to make it work with your application's specific needs
- Reducing boilerplate for views, with specialized view types
- Build on a modular architecture with an Application and modules that attach to it
- Compose your application's visuals at runtime, with Region and Layout
- Nested views and layouts within visual regions
- Built-in memory management and zombie killing in views, regions, and layouts
- Built-in event clean up with the EventBinder
- Event-driven architecture with the EventAggregator
- Flexible, "as-needed" architecture allowing you to pick and choose what you need
- And much, much more

Marionette follows a similar philosophy to Backbone in that it provides a suite of components that can be used independently of each other, or used together to create a significant advantages for us as developers. But it steps above the structural components of Backbone and provides an application layer, with more than a dozen components and building blocks.

Marionette's components range greatly in the features they provide, but they all work together to create a composite application layer that can both reduce boilerplate code and provide a much needed application structure. Its core components include various and specialized view types that take the boilerplate out of rendering common Backbone.Model and Backbone.Collection scenarios; an Application object and Module architecture to scale applications across sub-applications, features and files; integration of a command pattern, event aggregator, and request/response mechanism; and many more object types that can be extended in a myriad of ways to create an architecture that facilitates an application's specific needs.

In spite of the large number of constructs that Marionette provides, though, you're not required to use all of it just because you want to use some of it. Much like Backbone itself, you can pick and choose which features you want to use and when. This allows you to work with other Backbone frameworks and plugins very easily. It also means that you are not required to engage in an all-or-nothing migration to begin using Marionette.

Boilerplate Rendering Code

Consider the code that it typically requires to render a view with Backbone and Underscore template. We need a template to render, which can be placed in the DOM directly, and we need the JavaScript that defines a view that uses the template and populates it with data from a model.

```
<script type="text/html" id="my-view-template">
  <div class="row">
    <label>First Name:</label>
    <span><%= firstName %></span>
  </div>
  <div class="row">
    <label>Last Name:</label>
    <span><%= lastName %></span>
  </div>
  <div class="row">
    <label>Email:</label>
    <span><%= email %></span>
  </div>
</script>

var MyView = Backbone.View.extend({
  template: $('#my-view-template').html(),

  render: function(){

    // compile the Underscore.js template
    var compiledTemplate = _.template(this.template);

    // render the template with the model data
    var data = this.model.toJSON();
    var html = compiledTemplate(data);

    // populate the view with the rendered html
    this.$el.html(html);
  }
});
```

Once this is in place, you need to create an instance of your view and pass your model into it. Then you can take the view's `el` and append it to the DOM in order to display the view.

```
var Derick = new Person({
  firstName: 'Derick',
  lastName: 'Bailey',
  email: 'derickbailey@example.com'
});

var myView = new MyView({
  model: Derick
})

myView.render();

$('#content').html(myView.el)
```

This is a standard set up for defining, building, rendering, and displaying a view with Backbone. This is also what we call “boilerplate code” - code that is repeated over and over and over again, across every project and every implementation with the same functionality. It gets to be tedious and repetitious very quickly.

Enter Marionette's `ItemView` - a simple way to reduce the boilerplate of defining a view.

Reducing Boilerplate With `Marionette.ItemView`

All of Marionette's view types - with the exception of `Marionette.View` - include a built-in `render` method that handles the core rendering logic for you. We can take advantage of this by changing the `MyView` instance to inherit from one of these rather than `Backbone.View`. Instead of having to provide our own `render` method for the view, we can let Marionette render it for us. We'll still use the same Underscore.js template and rendering mechanism, but the implementation of this is hidden behind the scenes. Thus, we can reduce the amount of code needed for this view.

```
var MyView = Marionette.ItemView.extend({
  template: '#my-view-template'
});
```

And that's it - that's all you need to get the exact same behaviour as the previous view implementation. Just replace `Backbone.View.extend` with `Marionette.ItemView.extend`, then get rid of the `render` method. You can still create the view instance with a `model`, call the `render` method on the view

instance, and display the view in the DOM the same way that we did before. But the view definition has been reduced to a single line of configuration for the template.

Memory Management

In addition to the reduction of code needed to define a view, Marionette includes some advanced memory management in all of its views, making the job of cleaning up a view instance and its event handlers easy.

Consider the following view implementation:

```
var ZombieView = Backbone.View.extend({
  template: '#my-view-template',

  initialize: function(){

    // bind the model change to re-render this view
    this.model.on('change', this.render, this);

  },

  render: function(){

    // This alert is going to demonstrate a problem
    alert('We're rendering the view');

  }
});
```

If we create two instances of this view using the same variable name for both instances, and then change a value in the model, how many times will we see the alert box?

```
var Person = Backbone.Model.extend({
  defaults: {
    "firstName": "Jeremy",
    "lastName": "Ashkenas",
    "email": "jeremy@example.com"
  }
});

var Derick = new Person({
  firstName: 'Derick',
```

```

    lastName: 'Bailey',
    email: 'derick@example.com'
  });

  // create the first view instance
  var zombieView = new ZombieView({
    model: Derick
  });

  // create a second view instance, re-using
  // the same variable name to store it
  zombieView = new ZombieView({
    model: Derick
  });

  Derick.set('email', 'derickbailey@example.com');

```

Since we're re-using the same `zombieView` variable for both instances, the first instance of the view will fall out of scope immediately after the second is created. This allows the JavaScript garbage collector to come along and clean it up, which should mean the first view instance is no longer active and no longer going to respond to the model's "change" event.

But when we run this code, we end up with the alert box showing up twice!

The problem is caused by the model event binding in the view's `initialize` method. Whenever we pass `this.render` as the callback method to the model's `on` event binding, the model itself is being given a direct reference to the view instance. Since the model is now holding a reference to the view instance, replacing the `zombieView` variable with a new view instance is not going to let the original view fall out of scope. The model still has a reference, therefore the view is still in scope.

Since the original view is still in scope, and the second view instance is also in scope, changing data on the model will cause both view instances to respond.

Fixing this is easy, though. You just need to call `off` when the view is done with its work and ready to be closed. To do this, add a `close` method to the view.

```

var ZombieView = Backbone.View.extend({
  template: '#my-view-template',

  initialize: function(){
    // bind the model change to re-render this view
    this.listenTo(this.model, 'change', this.render);
  },

```



```

close: function(){
  // unbind the events that this view is listening to
  this.stopListening();
},

render: function(){

  // This alert is going to demonstrate a problem
  alert('We're rendering the view');

}
});

```

Then call `close` on the first instance when it is no longer needed, and only one view instance will remain alive. For more information about the `listenTo` and `stopListening` functions, see [the Backbone documentation](#), and my blog post on [Managing Events As Relationships, Not Just Resources](#).

```

var Jeremy = new Person({
  firstName: 'Jeremy',
  lastName: 'Ashkenas',
  email: 'jeremy@example.com'
});

// create the first view instance
var zombieView = new ZombieView({
  model: Person
})
zombieView.close(); // double-tap the zombie

// create a second view instance, re-using
// the same variable name to store it
zombieView = new ZombieView({
  model: Person
})

Person.set('email', 'jeremyashkenas@example.com');

```

Now we only see one alert box when this code runs.

Rather than having to manually remove these event handlers, though, we can let Marionette do it for us.

```

var ZombieView = Marionette.ItemView.extend({
  template: '#my-view-template',

```

```

initialize: function(){

    // bind the model change to re-render this view
    this.listenTo(this.model, 'change', this.render);

},

render: function(){

    // This alert is going to demonstrate a problem
    alert('We're rendering the view');

}
});

```

Notice in this case we are using a method called `listenTo`. This method comes from `Backbone.Events`, and is available in all objects that mix in `Backbone.Events` - including most `Marionette` objects. The `listenTo` method signature is similar to that of the `on` method, with the exception of passing the object that triggers the event as the first parameter.

`Marionette`'s views also provide a `close` event, in which the event bindings that are set up with the `listenTo` are automatically removed. This means we no longer need to define a `close` method directly, and when we use the `listenTo` method, we know that our events will be removed and our views will not turn into zombies.

But how do we automate the call to `close` on a view, in the real application? When and where do we call that? Enter the `Marionette.Region` - an object that manages the lifecycle of an individual view.

Region Management

After a view is created, it typically needs to be placed in the DOM so that it becomes visible. This is usually done with a jQuery selector and setting the `html()` of the resulting object:

```

var Joe = new Person({
  firstName: 'Joe',
  lastName: 'Bob',
  email: 'joebob@example.com'
});

var myView = new MyView({
  model: Joe

```

```

})

myView.render();

// show the view in the DOM
$('#content').html(myView.el)

```

This, again, is boilerplate code. We shouldn't have to manually call **render** and manually select the DOM elements to show the view. Furthermore, this code doesn't lend itself to closing any previous view instance that might be attached to the DOM element we want to populate. And we've seen the danger of zombie views already.

To solve these problems, Marionette provides a **Region** object - an object that manages the lifecycle of individual views, displayed in a particular DOM element.

```

// create a region instance, telling it which DOM element to manage
var myRegion = new Marionette.Region({
  el: '#content'
});

// show a view in the region
var view1 = new MyView({ /* ... */ });
myRegion.show(view1);

// somewhere else in the code,
// show a different view
var view2 = new MyView({ /* ... */ });
myRegion.show(view2);

```

There are several things to note, here. First, we're telling the region what DOM element to manage by specifying an **el** in the region instance. Second, we're no longer calling the **render** method on our views. And lastly, we're not calling **close** on our view, either, though this is getting called for us.

When we use a region to manage the lifecycle of our views, and display the views in the DOM, the region itself handles these concerns. By passing a view instance into the **show** method of the region, it will call the render method on the view for us. It will then take the resulting **el** of the view and populate the DOM element.

The next time we call the **show** method of the region, the region remembers that it is currently displaying a view. The region calls the **close** method on the view, removes it from the DOM, and then proceeds to run the render & display code for the new view that was passed in.

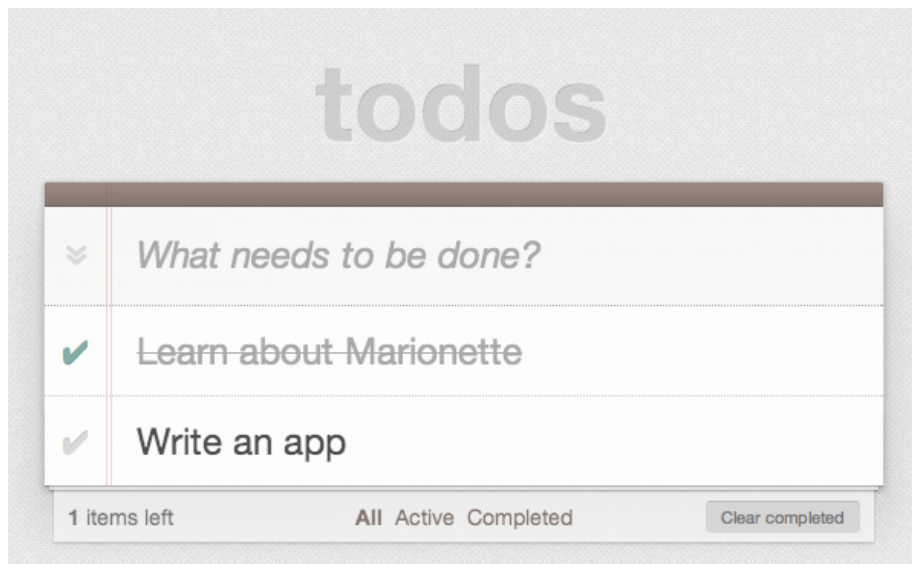
Since the region handles calling `close` for us, and we're using the `listenTo` event binder in our view instance, we no longer have to worry about zombie views in our application.

Regions are not limited to just Marionette views, though. Any valid Backbone.View can be managed by a Marionette.Region. If your view happens to have a `close` method, it will be called when the view is closed. If not, the Backbone.View built-in method, `remove`, will be called instead. If neither of these methods exist, nothing will be called and you will be responsible for cleaning up the events and other bits, yourself.

Marionette Todo app

Having learned about Marionette's high-level concepts, let's explore refactoring the Todo application we created in our first exercise to use it. The complete code for this application can be found in Derick's TodoMVC [fork](#).

Our final implementation will be visually and functionally equivalent to the original app, as seen below.



First, we define an application object representing our base TodoMVC app. This will contain initialization code and define the default layout regions for our app.

TodoMVC.js:

```
var TodoMVC = new Marionette.Application();

TodoMVC.addRegions({
```

```

    header : '#header',
    main   : '#main',
    footer : '#footer'
  });

  TodoMVC.on('initialize:after', function(){
    Backbone.history.start();
  });

```

Regions are used to manage the content that's displayed within specific elements, and the `addRegions` method on the `TodoMVC` object is just a shortcut for creating `Region` objects. We supply a jQuery selector for each region to manage (e.g., `#header`, `#main`, and `#footer`) and then tell the region to show various Backbone views within that region.

Once the application object has been initialized, we call `Backbone.history.start()` to route the initial URL.

Next, we define our Layouts. A layout is a specialized type of view that directly extends `Marionette.ItemView`. This means it's intended to render a single template and may or may not have a model (or `item`) associated with the template.

One of the main differences between a Layout and an `ItemView` is that the layout contains regions. When defining a Layout, we supply it with both a `template` and the regions that the template contains. After rendering the layout, we can display other views within the layout using the regions that were defined.

In our `TodoMVC` Layout module below, we define Layouts for:

- Header: where we can create new Todos
- Footer: where we summarize how many Todos are remaining/have been completed

This captures some of the view logic that was previously in our `AppView` and `TodoView`.

Note that Marionette modules (such as the below) offer a simple module system which is used to create privacy and encapsulation in Marionette apps. These certainly don't have to be used however, and later on in this section we'll provide links to alternative implementations using RequireJS + AMD instead.

TodoMVC.Layout.js:

```

TodoMVC.module('Layout', function(Layout, App, Backbone, Marionette, $, _){

  // Layout Header View
  // -----

```

```

Layout.Header = Marionette.ItemView.extend({
  template : '#template-header',

  // UI bindings create cached attributes that
  // point to jQuery selected objects
  ui : {
    input : '#new-todo'
  },

  events : {
    'keypress #new-todo': 'onInputKeypress'
  },

  onInputKeypress : function(evt) {
    var ENTER_KEY = 13;
    var todoText = this.ui.input.val().trim();

    if ( evt.which === ENTER_KEY && todoText ) {
      this.collection.create({
        title : todoText
      });
      this.ui.input.val('');
    }
  }
});

```

```

// Layout Footer View
// -----

```

```

Layout.Footer = Marionette.Layout.extend({
  template : '#template-footer',

  // UI bindings create cached attributes that
  // point to jQuery selected objects
  ui : {
    count   : '#todo-count strong',
    filters : '#filters a'
  },

  events : {
    'click #clear-completed' : 'onClearClick'
  },

  initialize : function() {

```

```

        this.listenTo(App.vent, 'todoList:filter', this.updateFilterSelection);
        this.listenTo(this.collection, 'all', this.updateCount);
    },

    onRender : function() {
        this.updateCount();
    },

    updateCount : function() {
        var count = this.collection.getActive().length;
        this.ui.count.html(count);

        if (count === 0) {
            this.$el.parent().hide();
        } else {
            this.$el.parent().show();
        }
    },

    updateFilterSelection : function(filter) {
        this.ui.filters
            .removeClass('selected')
            .filter('[href="#" + filter + "']')
            .addClass('selected');
    },

    onClearClick : function() {
        var completed = this.collection.getCompleted();
        completed.forEach(function destroy(todo) {
            todo.destroy();
        });
    }
});
});

```

Next, we tackle application routing and workflow, such as controlling Layouts in the page which can be shown or hidden.

Recall how Backbone routes trigger methods within the Router as shown below in our original Workspace router from our first exercise:

```

var Workspace = Backbone.Router.extend({
    routes:{
        '*filter': 'setFilter'
    },

```

```

    setFilter: function( param ) {
        // Set the current filter to be used
        app.TODOFilter = param.trim() || '';

        // Trigger a collection filter event, causing hiding/unhiding
        // of Todo view items
        app.Todos.trigger('filter');
    }
});

```

Marionette uses the concept of an AppRouter to simplify routing. This reduces the boilerplate for handling route events and allows routers to be configured to call methods on an object directly. We configure our AppRouter using `appRoutes` which replaces the `'*filter': 'setFilter'` route defined in our original router and invokes a method on our Controller.

The `TodoList` Controller, also found in this next code block, handles some of the remaining visibility logic originally found in `AppView` and `TodoView`, albeit using very readable Layouts.

TodoMVC.TodoList.js:

```

TodoMVC.module('TodoList', function(TodoList, App, Backbone, Marionette, $, _){

    // TodoList Router
    // -----
    //
    // Handle routes to show the active vs complete todo items

    TodoList.Router = Marionette.AppRouter.extend({
        appRoutes : {
            '*filter': 'filterItems'
        }
    });

    // TodoList Controller (Mediator)
    // -----
    //
    // Control the workflow and logic that exists at the application
    // level, above the implementation detail of views and models

    TodoList.Controller = function(){
        this.todoList = new App.Todos.TodoList();
    };

```



```

_.extend(TodoList.Controller.prototype, {

    // Start the app by showing the appropriate views
    // and fetching the list of todo items, if there are any
    start: function(){
        this.showHeader(this.todoList);
        this.showFooter(this.todoList);
        this.showTodoList(this.todoList);

        this.todoList.fetch();
    },

    showHeader: function(todoList){
        var header = new App.Layout.Header({
            collection: todoList
        });
        App.header.show(header);
    },

    showFooter: function(todoList){
        var footer = new App.Layout.Footer({
            collection: todoList
        });
        App.footer.show(footer);
    },

    showTodoList: function(todoList){
        App.main.show(new TodoList.Views.ListView({
            collection : todoList
        }));
    },

    // Set the filter to show complete or all items
    filterItems: function(filter){
        App.vent.trigger('todoList:filter', filter.trim() || '');
    }
});

// TodoList Initializer
// -----
//
// Get the TodoList up and running by initializing the mediator
// when the application is started, pulling in all of the
// existing Todo items and displaying them.

TodoList.addInitializer(function(){

```

```

    var controller = new TodoList.Controller();
    new TodoList.Router({
        controller: controller
    });

    controller.start();

});

});

```

Controllers In this particular app, note that Controllers don't add a great deal to the overall workflow. In general, Marionette's philosophy on routers is that they should be an afterthought in the implementation of applications. Quite often, we've seen developers abuse Backbone's routing system by making it the sole controller of the entire application workflow and logic.

This inevitably leads to mashing every possible combination of code into the router methods - view creation, model loading, coordinating different parts of the app, etc. Developers such as Derick view this as a violation of the [single-responsibility principle](#) (SRP) and separation of concerns.

Backbone's router and history exist to deal with a specific aspect of browsers - managing the forward and back buttons. Marionette's philosophy is that it should be limited to that, with the code that gets executed by the navigation being somewhere else. This allows the application to be used with or without a router. We can call a controller's "show" method from a button click, from an application event handler, or from a router, and we will end up with the same application state no matter how we called that method.

Derick has written extensively about his thoughts on this topic, which you can read more about on his blog:

- <http://lostechies.com/derickbailey/2011/12/27/the-responsibilities-of-the-various-pieces-of-backbone-js/>
- <http://lostechies.com/derickbailey/2012/01/02/reducing-backbone-routers-to-nothing-more-than-configuration/>
- <http://lostechies.com/derickbailey/2012/02/06/3-stages-of-a-backbone-applications-startup/>

CompositeView Our next task is defining the actual views for individual Todo items and lists of items in our TodoMVC application. For this, we make use of Marionette's **CompositeViews**. The idea behind a CompositeView is that it represents a visualization of a composite or hierarchical structure of leaves (or nodes) and branches.

Think of these views as being a hierarchy of parent-child models, and recursive by default. The same CompositeView type will be used to render each item in a

collection that is handled by the composite view. For non-recursive hierarchies, we are able to override the item view by defining an `itemView` attribute.

For our Todo List Item View, we define it as an `ItemView`, then our Todo List View is a `CompositeView` where we override the `itemView` setting and tell it to use the Todo List item View for each item in the collection.

TodoMVC.TodoList.Views.js

```
TodoMVC.module('TodoList.Views', function(Views, App, Backbone, Marionette, $, _){

  // Todo List Item View
  // -----
  //
  // Display an individual todo item, and respond to changes
  // that are made to the item, including marking completed.

  Views.ItemView = Marionette.ItemView.extend({
    tagName : 'li',
    template : '#template-todoItemView',

    ui : {
      edit : '.edit'
    },

    events : {
      'click .destroy' : 'destroy',
      'dblclick label' : 'onEditClick',
      'keypress .edit' : 'onEditKeypress',
      'click .toggle' : 'toggle'
    },

    initialize : function() {
      this.listenTo(this.model, 'change', this.render);
    },

    onRender : function() {
      this.$el.removeClass('active completed');
      if (this.model.get('completed')) this.$el.addClass('completed');
      else this.$el.addClass('active');
    },

    destroy : function() {
      this.model.destroy();
    },

    toggle : function() {
```

```

        this.model.toggle().save();
    },

    onEditClick : function() {
        this.$el.addClass('editing');
        this.ui.edit.focus();
    },

    onEditKeypress : function(evt) {
        var ENTER_KEY = 13;
        var todoText = this.ui.edit.val().trim();

        if ( evt.which === ENTER_KEY && todoText ) {
            this.model.set('title', todoText).save();
            this.$el.removeClass('editing');
        }
    }
});

// Item List View
// -----
//
// Controls the rendering of the list of items, including the
// filtering of active vs completed items for display.

Views.ListView = Marionette.CompositeView.extend({
    template : '#template-todoListCompositeView',
    itemView : Views.ItemView,
    itemViewContainer : '#todo-list',

    ui : {
        toggle : '#toggle-all'
    },

    events : {
        'click #toggle-all' : 'onToggleAllClick'
    },

    initialize : function() {
        this.listenTo(this.collection, 'all', this.update);
    },

    onRender : function() {
        this.update();
    },
});

```

```

    update : function() {
      function reduceCompleted(left, right) { return left && right.get('completed'); }
      var allCompleted = this.collection.reduce(reduceCompleted, true);
      this.ui.toggle.prop('checked', allCompleted);

      if (this.collection.length === 0) {
        this.$el.parent().hide();
      } else {
        this.$el.parent().show();
      }
    },

    onToggleAllClick : function(evt) {
      var isChecked = evt.currentTarget.checked;
      this.collection.each(function(todo){
        todo.save({'completed': isChecked});
      });
    }
  });

  // Application Event Handlers
  // -----
  //
  // Handler for filtering the list of items by showing and
  // hiding through the use of various CSS classes

  App.vent.on('todoList:filter', function(filter) {
    filter = filter || 'all';
    $('#todoapp').attr('class', 'filter-' + filter);
  });

});

```

At the end of the last code block, you will also notice an event handler using `vent`. This is an event aggregator that allows us to handle `filterItem` triggers from our `TodoList` controller.

Finally, we define the model and collection for representing our `Todo` items. These are semantically not very different from the original versions we used in our first exercise and have been re-written to better fit in with Derick's preferred style of coding.

Todos.js:

```

TodoMVC.module('Todos', function(Todos, App, Backbone, Marionette, $, _){

```

```

// Todo Model
// -----

Todos.Todo = Backbone.Model.extend({
  localStorage: new Backbone.LocalStorage('todos-backbone'),

  defaults: {
    title      : '',
    completed  : false,
    created    : 0
  },

  initialize : function() {
    if (this.isNew()) this.set('created', Date.now());
  },

  toggle : function() {
    return this.set('completed', !this.isCompleted());
  },

  isCompleted: function() {
    return this.get('completed');
  }
});

// Todo Collection
// -----

Todos.TodoList = Backbone.Collection.extend({
  model: Todos.Todo,

  localStorage: new Backbone.LocalStorage('todos-backbone'),

  getCompleted: function() {
    return this.filter(this._isCompleted);
  },

  getActive: function() {
    return this.reject(this._isCompleted);
  },

  comparator: function( todo ) {
    return todo.get('created');
  },

  _isCompleted: function(todo){

```

```

        return todo.isCompleted();
    }
});

});

```

We finally kick-start everything off in our application index file, by calling `start` on our main application object:

Initialization:

```

$(function(){
    // Start the TodoMVC app (defined in js/TodoMVC.js)
    TodoMVC.start();
});

```

And that's it!

Is the Marionette implementation of the Todo app more maintainable?

Derick feels that maintainability largely comes down to modularity, separating responsibilities (Single Responsibility Principle and Separation of Concerns) by using patterns to keep concerns from being mixed together. It can, however, be difficult to simply extract things into separate modules for the sake of extraction, abstraction, or dividing the concept down into its simplest parts.

The Single Responsibility Principle (SRP) tells us quite the opposite - that we need to understand the context in which things change. What parts always change together, in *this* system? What parts can change independently? Without knowing this, we won't know what pieces should be broken out into separate components and modules versus put together into the same module or object.

The way Derick organizes his apps into modules is by creating a breakdown of concepts at each level. A higher level module is a higher level of concern - an aggregation of responsibilities. Each responsibility is broken down into an expressive API set that is implemented by lower level modules (Dependency Inversion Principle). These are coordinated through a mediator - which he typically refers to as the Controller in a module.

The way Derick organizes his files also plays directly into maintainability and he has also written posts about the importance of keeping a sane application folder structure that I recommend reading:

- <http://lostechies.com/derickbailey/2012/02/02/javascript-file-folder-structures-just-pick-one/>
- <http://hilojs.codeplex.com/discussions/362875#post869640>

Marionette And Flexibility

Marionette is a flexible framework, much like Backbone itself. It offers a wide variety of tools to help create and organize an application architecture on top of Backbone, but like Backbone itself, it doesn't dictate that you have to use all of its pieces in order to use any of them.

The flexibility and versatility in Marionette is easiest to understand by examining three variations of TodoMVC implemented with it that have been created for comparison purposes:

- [Simple](#) - by Jarrod Overson
- [RequireJS](#) - also by Jarrod
- [Marionette modules](#) - by Derick Bailey

The simple version: This version of TodoMVC shows some raw use of Marionette's various view types, an application object, and the event aggregator. The objects that are created are added directly to the global namespace and are fairly straightforward. This is a great example of how Marionette can be used to augment existing code without having to re-write everything around Marionette.

The RequireJS version: Using Marionette with RequireJS helps to create a modularized application architecture - a tremendously important concept in scaling JavaScript applications. RequireJS provides a powerful set of tools that can be leveraged to great advantage, making Marionette even more flexible than it already is.

The Marionette module version: RequireJS isn't the only way to create a modularized application architecture, though. For those that wish to build applications in modules and namespaces, Marionette provides a built-in module and namespacing structure. This example application takes the simple version of the application and re-writes it into a namespaced application architecture, with an application controller (mediator / workflow object) that brings all of the pieces together.

Marionette certainly provides its share of opinions on how a Backbone application should be architected. The combination of modules, view types, event aggregator, application objects, and more, can be used to create a very powerful and flexible architecture based on these opinions.

But as you can see, Marionette isn't a completely rigid, "my way or the highway" framework. It provides many elements of an application foundation that can be mixed and matched with other architectural styles, such as AMD or namespacing, or provide simple augmentation to existing projects by reducing boilerplate code for rendering views.

This flexibility creates a much greater opportunity for Marionette to provide value to you and your projects, as it allows you to scale the use of Marionette with your application's needs.

And So Much More

This is just the tip of the proverbial iceberg for Marionette, even for the `ItemView` and `Region` objects that we've explored. There is far more functionality, more features, and more flexibility and customizability that can be put to use in both of these objects. Then we have the other dozen or so components that Marionette provides, each with their own set of behaviors built in, customization and extension points, and more.

To learn more about Marionette's components, the features they provide and how to use them, check out the Marionette documentation, links to the wiki, to the source code, the project core contributors, and much more at <http://marionettejs.com>.

Thorax

By Ryan Eastridge & Addy Osmani

Part of Backbone's appeal is that it provides structure but is generally unopinionated, in particular when it comes to views. Thorax makes an opinionated decision to use Handlebars as its templating solution. Some of the patterns found in Marionette are found in Thorax as well. Marionette exposes most of these patterns as JavaScript APIs while in Thorax they are often exposed as template helpers. This chapter assumes the reader has knowledge of Handlebars.

Thorax was created by Ryan Eastridge and Kevin Decker to create Walmart's mobile web application. This chapter is limited to Thorax's templating features and patterns implemented in Thorax that you can utilize in your application regardless of whether you choose to adopt Thorax. To learn more about other features implemented in Thorax and to download boilerplate projects visit the [Thorax website](#).

Hello World

`Thorax.View` differs from `Backbone.View` in that there is no `options` object. All arguments passed to the constructor become properties of the view, which in turn become available to the `template`:

```
var view = new Thorax.View({
  greeting: 'Hello',
  template: Handlebars.compile('{{greeting}} World!')
});
view.appendTo('body');
```

In most examples in this chapter a `template` property will be specified. In larger projects including the boilerplate projects provided on the Thorax website a `name` property would instead be used and a `template` of the same file name in your project would automatically be assigned to the view.

If a `model` is set on a view, its attributes also become available to the template:

```
var view = new Thorax.View({
  model: new Thorax.Model({key: 'value'}),
  template: Handlebars.compile('{{key}}')
});
```

Embedding child views

The view helper allows you to embed other views within a view. Child views can be specified as properties of the view:

```
var parent = new Thorax.View({
  child: new Thorax.View(...),
  template: Handlebars.compile('{{view child}}')
});
```

Or the name of a child view to initialize (and any optional properties to pass). In this case the child view must have previously been created with `extend` and given a `name` property:

```
var ChildView = Thorax.View.extend({
  name: 'child',
  template: ...
});

var parent = new Thorax.View({
  template: Handlebars.compile('{{view "child" key="value"}}')
});
```

The view helper may also be used as a block helper, in which case the block will be assigned as the `template` property of the child view:

```
{{#view child}}
  child will have this block
  set as its template property
{{/view}}
```

Handlebars is string based, while `Backbone.View` instances have a DOM `el`. Since we are mixing metaphors, the embedding of views works via a placeholder mechanism where the `view` helper in this case adds the view passed to the helper to a hash of `children`, then injects placeholder HTML into the template such as:

```
<div data-view-placeholder-cid="view2"></div>
```

Then once the parent view is rendered, we walk the DOM in search of all the placeholders we created, replacing them with the child views' `els`:

```
this.$el.find('[data-view-placeholder-cid]').forEach(function(el) {
  var cid = el.getAttribute('data-view-placeholder-cid'),
      view = this.children[cid];
  view.render();
  $(el).replaceWith(view.el);
}, this);
```

View helpers

One of the most useful constructs in Thorax is `Handlebars.registerViewHelper` (not to be confused with `Handlebars.registerHelper`). This method will register a new block helper that will create and embed a `HelperView` instance with its `template` set to the captured block. A `HelperView` instance is different from that of a regular child view in that its context will be that of the parent's in the template. Like other child views it will have a `parent` property set to that of the declaring view. Many of the built-in helpers in Thorax including the collection helper are created in this manner.

A simple example would be an `on` helper that re-rendered the generated `HelperView` instance each time an event was triggered on the declaring / parent view:

```
Handlebars.registerViewHelper('on', function(eventName, helperView) {
  helperView.parent.on(eventName, function() {
    helperView.render();
  });
});
```

An example use of this would be to have a counter that would increment each time a button was clicked. This example makes use of the `button` helper in Thorax which simply makes a button that calls a method when clicked:

```
{{#on "incremented"}}{{i}}{/on}}
{{#button trigger="incremented"}}Add{{/button}}
```

And the corresponding view class:

```
new Thorax.View({
  events: {
    incremented: function() {
      ++this.i;
    }
  },
  initialize: function() {
    this.i = 0;
  },
  template: ...
});
```

collection helper

The `collection` helper creates and embeds a `CollectionView` instance, creating a view for each item in a collection, updating when items are added, removed, or changed in the collection. The simplest usage of the helper would look like:

```
{{#collection kittens}}
  <li>{{name}}</li>
{{/collection}}
```

And the corresponding view:

```
new Thorax.View({
  kittens: new Thorax.Collection(...),
  template: ...
});
```

The block in this case will be assigned as the `template` for each item view created, and the context will be the `attributes` of the given model. This helper accepts options that can be arbitrary HTML attributes, a `tag` option to specify the type of tag containing the collection, or any of the following:

- `item-template` - A template to display for each model. If a block is specified it will become the item-template
- `item-view` - A view class to use when each item view is created
- `empty-template` - A template to display when the collection is empty. If an inverse / else block is specified it will become the empty-template
- `empty-view` - A view to display when the collection is empty

Options and blocks can be used in combination, in this case creating a `KittenView` class with a `template` set to the captured block for each kitten in the collection:

```
{{#collection kittens item-view="KittenView" tag="ul"}}
  <li>{{name}}</li>
{{else}}
  <li>No kittens!</li>
{{/collection}}
```

Note that multiple collections can be used per view, and collections can be nested. This is useful when there are models that contain collections that contain models that contain...

```
{{#collection kittens}}
  <h2>{{name}}</h2>
  <p>Kills:</p>
  {{#collection miceKilled tag="ul"}}
    <li>{{name}}</li>
  {{/collection}}
{{/collection}}
```

Custom HTML data attributes

Thorax makes heavy use of custom HTML data attributes to operate. While some make sense only within the context of Thorax, several are quite useful to have in any Backbone project for writing other functions against, or for general debugging. In order to add some to your views in non-Thorax projects, override the `setElement` method in your base view class:

```
MyApplication.View = Backbone.View.extend({
  setElement: function() {
    var response = Backbone.View.prototype.setElement.apply(this, arguments);
    this.name && this.$el.attr('data-view-name', this.name);
    this.$el.attr('data-view-cid', this.cid);
    this.collection && this.$el.attr('data-collection-cid', this.collection.cid);
    this.model && this.$el.attr('data-model-cid', this.model.cid);
    return response;
  }
});
```

In addition to making your application more immediately comprehensible in the inspector, it's now possible to extend jQuery / Zepto with functions to lookup the closest view, model or collection to a given element. In order to make it

work you have to save references to each view created in your base view class by overriding the `_configure` method:

```
MyApplication.View = Backbone.View.extend({
  _configure: function() {
    Backbone.View.prototype._configure.apply(this, arguments);
    Thorax._viewsIndexedByCid[this.cid] = this;
  },
  dispose: function() {
    Backbone.View.prototype.dispose.apply(this, arguments);
    delete Thorax._viewsIndexedByCid[this.cid];
  }
});
```

Then we can extend jQuery / Zepto:

```
$.fn.view = function() {
  var el = $(this).closest('[data-view-cid]');
  return el && Thorax._viewsIndexedByCid[el.attr('data-view-cid')];
};

$.fn.model = function(view) {
  var $this = $(this),
      modelElement = $this.closest('[data-model-cid]'),
      modelCid = modelElement && modelElement.attr('data-model-cid');
  if (modelCid) {
    var view = $this.view();
    return view && view.model;
  }
  return false;
};
```

Now instead of storing references to models randomly throughout your application to lookup when a given DOM event occurs you can use `$(element).model()`. In Thorax, this can particularly useful in conjunction with the `collection` helper which generates a view class (with a `model` property) for each `model` in the collection. An example template:

```
{{#collection kittens tag="ul"}}
  <li>{{name}}</li>
{{/collection}}
```

And the corresponding view class:

```

Thorax.View.extend({
  events: {
    'click li': function(event) {
      var kitten = $(event.target).model();
      console.log('Clicked on ' + kitten.get('name'));
    }
  },
  kittens: new Thorax.Collection(...),
  template: ...
});

```

A common anti-pattern in Backbone applications is to assign a `className` to a single view class. Consider using the `data-view-name` attribute as a CSS selector instead, saving CSS classes for things that will be used multiple times:

```

[data-view-name="child"] {

}

```

Thorax Resources

No Backbone related tutorial would be complete without a todo application. A [Thorax implementation of TodoMVC](#) is available, in addition to this far simpler example composed of this single Handlebars template:

```

{{#collection todos tag="ul"}}
  <li{{#if done}} class="done"{{/if}}>
    <input type="checkbox" name="done"{{#if done}} checked="checked"{{/if}}>
    <span>{{item}}</span>
  </li>
{{/collection}}
<form>
  <input type="text">
  <input type="submit" value="Add">
</form>

```

and the corresponding JavaScript:

```

var todosView = Thorax.View({
  todos: new Thorax.Collection(),
  events: {
    'change input[type="checkbox"]': function(event) {
      var target = $(event.target);
      target.model().set({done: !!target.attr('checked')});
    }
  }
});

```

```

    },
    'submit form': function(event) {
        event.preventDefault();
        var input = this.$('input[type="text"]');
        this.todos.add({item: input.val()});
        input.val('');
    }
},
template: '...'
});
todosView.appendTo('body');

```

To see Thorax in action on a large scale website visit walmart.com on any Android or iOS device. For a complete list of resources visit the [Thorax website](#).

Common Problems & Solutions

In this section, we will review a number of common problems developers often experience once they've started to work on relatively non-trivial projects using Backbone.js, as well as present potential solutions.

Perhaps the most frequent of these questions surround how to do more with Views. If you are interested in discovering how to work with nested Views, learn about view disposal and inheritance, this section will hopefully have you covered.

View Nesting Problem

What is the best approach for rendering and appending Subviews in Backbone.js?

Solution 1

Since pages are composed of nested elements and Backbone views correspond to elements within the page, nesting views is an intuitive approach to managing a hierarchy of elements.

The best way to combine views is simply using:

```
this.$('.someContainer').append(innerView.el);
```

which just relies on jQuery. We could use this in a real example as follows:


```

...
initialize : function () {
    //...
},

render : function () {

    this.$el.empty();

    this.innerView1 = new Subview({options});
    this.innerView2 = new Subview({options});

    this.$('.inner-view-container')
        .append(this.innerView1.el)
        .append(this.innerView2.el);
}

```

Solution 2

Beginners sometimes also try using `setElement` to solve this problem, however keep in mind that using this method is an easy way to shoot yourself in the foot. Try to avoid if possible:

```

// Where we have previously defined a View, SubView
// in a parent View we could do:

...
initialize : function () {

    this.innerView1 = new Subview({options});
    this.innerView2 = new Subview({options});
},

render : function () {

    this.$el.html(this.template());

    this.innerView1.setElement('.some-element1').render();
    this.innerView2.setElement('.some-element2').render();
}

```

Here we are creating subviews in the parent view's `initialize()` method and rendering the subviews in the parent's `render()` method. The elements managed by the subviews exist in the parent's template and the `View.setElement()` method is used to re-assign the element associated with each subview.

`setElement()` changes a view's element, including re-delegating event handlers by removing them from the old element and binding them to the new element. Note that `setElement()` returns the view, allowing us to chain the call to `render()`.

This works and has some positive qualities: you don't need to worry about maintaining the order of your DOM elements when appending, views are initialized early, and the `render()` method doesn't need to take on too many responsibilities at once.

Unfortunately, downsides are that you can't set the `tagName` property of subviews and events need to be re-delegated. The first solution doesn't suffer from this problem.

Solution 3

One more possible solution to this problem could be written:

```
var OuterView = Backbone.View.extend({
  initialize: function() {
    this.inner = new InnerView();
  },

  render: function() {
    this.$el.html(template); // or this.$el.empty() if you have no template
    this.$el.append(this.inner.$el);
    this.inner.render();
  }
});

var InnerView = Backbone.View.extend({
  render: function() {
    this.$el.html(template);
    this.delegateEvents();
  }
});
```

This tackles a few specific design decisions:

- The order in which you append the sub-elements matters
- The `OuterView` doesn't contain the HTML elements to be set in the `InnerView(s)`, meaning that we can still specify `tagName` in the `InnerView`
- `render()` is called after the `InnerView` element has been placed into the DOM. This is useful if your `InnerView's` `render()` method is sizing itself on the page based on the dimensions of another element. This is a common use case.

Note that `InnerView` needs to call `View.delegateEvents()` to bind its event handlers to its new DOM since it is replacing the content of its element.

Solution 4

A better solution, which is more clean but has the potential to affect performance is:

```
var OuterView = Backbone.View.extend({
  initialize: function() {
    this.render();
  },

  render: function() {
    this.$el.html(template); // or this.$el.empty() if you have no template
    this.inner = new InnerView();
    this.$el.append(this.inner.$el);
  }
});

var InnerView = Backbone.View.extend({
  initialize: function() {
    this.render();
  },

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

If multiple views need to be nested at particular locations in a template, a hash of child views indexed by child view `cids` should be created. In the template, use a custom HTML attribute named `data-view-cid` to create placeholder elements for each view to embed. Once the template has been rendered and its output appended to the parent view's `$el`, each placeholder can be queried for and replaced with the child view's `el`.

A sample implementation containing a single child view could be written:

```
var OuterView = Backbone.View.extend({
  initialize: function() {
    this.children = {};
    var child = new Backbone.View();
    this.children[child.cid] = child;
  },
```

```

render: function() {
  this.$el.html('<div data-view-cid="' + this.child.cid + '"></div>');
  _.each(this.children, function(view, cid) {
    this.$('[data-view-cid="' + cid + '"]').replaceWith(view.el);
  }, this);
}
};

```

The use of `cids` (client ids) here is useful because it illustrates separating a model and its views by having views referenced by their instances and not their attributes. It's quite common to ask for all views that satisfy an attribute on their models, but if you have recursive subviews or repeated views (a common occurrence), you can't simply ask for views by attributes. That is, unless you specify additional attributes that separate duplicates. Using `cids` solves this problem as it allows for direct references to views.

Generally speaking, more developers opt for Solution 1 or 5 as:

- The majority of their views may already rely on being in the DOM in their `render()` method
- When the `OuterView` is re-rendered, views don't have to be re-initialized where re-initialization has the potential to cause memory leaks and issues with existing bindings

The Backbone extensions `Marionette` and `Thorax` provide logic for nesting views, and rendering collections where each item has an associated view. `Marionette` provides APIs in JavaScript while `Thorax` provides APIs via Handlebars template helpers. We will examine both of these in an upcoming chapter.

(Thanks to [Lukas](#) and [Ian Taylor](#) for these tips).

What is the best way to manage models in nested Views? In order to reach attributes on related models in a nested setup, models require some prior knowledge of each other, something which Backbone doesn't implicitly handle out of the box.

One approach is to make sure each child model has a 'parent' attribute. This way you can traverse the nesting first up to the parent and then down to any siblings that you know of. So, assuming we have models `modelA`, `modelB` and `modelC`:

```

// When initializing modelA, I would suggest setting a link to the parent
// model when doing this, like this:

```

```

ModelA = Backbone.Model.extend({

    initialize: function(){
        this.modelB = new modelB();
        this.modelB.parent = this;
        this.modelC = new modelC();
        this.modelC.parent = this;
    }
}

```

This allows you to reach the parent model in any child model function through `this.parent`.

Now, we have already discussed a few options for how to construct nested Views using Backbone. For the sake of simplicity, let us imagine that we are creating a new child view `ViewB` from within the `initialize()` method of `ViewA` below. `ViewB` can reach out over the `ViewA` model and listen out for changes on any of its nested models.

See inline for comments on exactly what each step is enabling:

```

// Define View A
ViewA = Backbone.View.extend({

    initialize: function(){
        // Create an instance of View B
        this.viewB = new ViewB();

        // Create a reference back to this (parent) view
        this.viewB.parentView = this;

        // Append ViewB to ViewA
        $(this.el).append(this.viewB.el);
    }
});

// Define View B
ViewB = Backbone.View.extend({

    //...,

    initialize: function(){
        // Listen for changes to the nested models in our parent ViewA
        this.listenTo(this.model.parent.modelB, "change", this.render);
    }
});

```

```

        this.listenTo(this.model.parent.modelC, "change", this.render);

        // We can also call any method on our parent view if it is defined
        // $(this.parentView.el).shake();
    }

});

// Create an instance of ViewA with ModelA
// viewA will create its own instance of ViewB
// from inside the initialize() method
var viewA = new ViewA({ model: ModelA });

```

Rendering Parent View from Child Problem

How would one render a Parent View from one of its Children?

Solution

In a scenario where you have a view containing another view, such as a photo gallery containing a larger view modal, you may find that you need to render or re-render the parent view from the child. The good news is that solving this problem is quite straight-forward.

The simplest solution is to just use `this.parentView.render();`.

If however inversion of control is desired, events may be used to provide an equally valid solution.

Say we wish to begin rendering when a particular event has occurred. For the sake of example, let us call this event 'somethingHappened'. The parent view can bind notifications on the child view to know when the event has occurred. It can then render itself.

In the parent view:

```

// Parent initialize
this.listenTo(this.childView, 'somethingHappened', this.render);

// Parent removal
this.stopListening(this.childView, 'somethingHappened');

```

In the child view:

```

// After the event has occurred
this.trigger('somethingHappened');

```

The child will trigger a “somethingHappened” event and the parent’s render function will be called.

(Thanks to Tal [Bereznitskey](#) for this tip)

Disposing View hierarchies Problem

In the last question, we looked at how to effectively dispose of views to decrease memory usage.

Where your application is setup with multiple Parent and Child Views, it is also common to desire removing any DOM elements associated with such views as well as unbinding any event handlers tied to child elements when you no longer require them.

Solution

The solution in the last question should be enough to handle this use case, but if you require a more explicit example that handles children, we can see one below:

```
Backbone.View.prototype.close = function() {
  if (this.onClose) {
    this.onClose();
  }
  this.remove();
  this.unbind();
};

NewView = Backbone.View.extend({
  initialize: function() {
    this.childViews = [];
  },
  renderChildren: function(item) {
    var itemView = new NewChildView({ model: item });
    $(this.el).prepend(itemView.render());
    this.childViews.push(itemView);
  },
  onClose: function() {
    _(this.childViews).each(function(view) {
      view.close();
    });
  }
});

NewChildView = Backbone.View.extend({
  tagName: 'li',
  render: function() {
```

```
    }  
  });
```

Here, a `close()` method for views is implemented which disposes of a view when it is no longer needed or needs to be reset.

In most cases, the view removal should not affect any associated models. For example, if you are working on a blogging application and you remove a view with comments, perhaps another view in your app shows a selection of comments and resetting the collection would affect those views as well.

(Thanks to [dura](#) for this tip)

Note: You may also be interested in reading the about Marionette Composite Views in the Extensions part of the book.

Rendering View hierarchies Problem

Let us say you have a Collection, where each item in the Collection could itself be a Collection. You can render each item in the Collection, and indeed can render any items which themselves are Collections. The problem you might have is how to render HTML that reflects the hierarchical nature of the data structure.

Solution

The most straight-forward way to approach this problem is to use a framework like Derick Bailey's [Backbone.Marionette](#). In this framework is a type of view called a `CompositeView`.

The basic idea of a `CompositeView` is that it can render a model and a collection within the same view.

It can render a single model with a template. It can also take a collection from that model and for each model in that collection, render a view. By default it uses the same composite view type that you've defined to render each of the models in the collection. All you have to do is tell the view instance where the collection is, via the `initialize` method, and you'll get a recursive hierarchy rendered.

There is a working demo of this in action available [online](#).

And you can get the source code and documentation for [Marionette](#) too.

Better Model Property Validation Problem

As we learned earlier in the book, the `validate` method on a Model is called by `set` (when the `validate` option is set) and `save`, and is passed the model attributes updated with the values passed to these methods.

By default, when we define a custom `validate` method, Backbone passes all of a Model's attributes through this validation each time, regardless of which model attributes are being set.

This means that it can be a challenge to determine which specific fields are being set or validated without being concerned about the others that aren't being set at the same time.

Solution

The most optimal solution to this problem probably isn't to stick validation in your model attributes. Instead, have a function specifically designed for validating that particular form. There are many good JavaScript form validation libraries out there. If you want to stick it on your model, just make it a class function:

```
User.validate = function(formElement) {  
  //...  
};
```

To illustrate this problem better, let us look at a typical registration form use case that:

- Validates form fields using the blur event
- Validates each field regardless of whether other model attributes (i.e., other form data) are valid or not.

Here is one example of a desired use case:

We have a form where a user focuses and blurs first name, last name, and email HTML input boxes without entering any data. A “this field is required” message should be presented next to each form field.

HTML:

```
<!doctype html>  
<html>  
<head>  
  <meta charset=utf-8>  
  <title>Form Validation - Model#validate</title>  
  <script src='http://code.jquery.com/jquery.js'></script>  
  <script src='http://underscorejs.org/underscore.js'></script>  
  <script src='http://backbonejs.org/backbone.js'></script>  
</head>  
<body>  
  <form>  
    <label>First Name</label>
```

```

    <input name='firstname'>
    <span data-msg='firstname'></span>
    <br>
    <label>Last Name</label>
    <input name='lastname'>
    <span data-msg='lastname'></span>
    <br>
    <label>Email</label>
    <input name='email'>
    <span data-msg='email'></span>
  </form>
</body>
</html>

```

Some simple validation that could be written using the current Backbone `validate` method to work with this form could be implemented using something like:

```

validate: function(attrs) {

    if(!attrs.firstname) return 'first name is empty';
    if(!attrs.lastname) return 'last name is empty';
    if(!attrs.email) return 'email is empty';

}

```

Unfortunately, this method would trigger a first name error each time any of the fields were blurred and only an error message next to the first name field would be presented.

One potential solution to the problem is to validate all fields and return all of the errors:

```

validate: function(attrs) {
    var errors = {};

    if (!attrs.firstname) errors.firstname = 'first name is empty';
    if (!attrs.lastname) errors.lastname = 'last name is empty';
    if (!attrs.email) errors.email = 'email is empty';

    if (!_.isEmpty(errors)) return errors;
}

```

This can be adapted into a complete solution that defines a Field model for each input in our form and works within the parameters of our use case as follows:

```

$(function($) {

    var User = Backbone.Model.extend({
        validate: function(attrs) {
            var errors = this.errors = {};

            if (!attrs.firstname) errors.firstname = 'firstname is required';
            if (!attrs.lastname) errors.lastname = 'lastname is required';
            if (!attrs.email) errors.email = 'email is required';

            if (!_isEmpty(errors)) return errors;
        }
    });

    var Field = Backbone.View.extend({
        events: {blur: 'validate'},
        initialize: function() {
            this.name = this.$el.attr('name');
            this.$msg = $('[data-msg=' + this.name + ']');
        },
        validate: function() {
            this.model.set(this.name, this.$el.val());
            this.$msg.text(this.model.errors[this.name] || '');
        }
    });

    var user = new User;

    $('input').each(function() {
        new Field({el: this, model: user});
    });

});

```

This works great as the solution checks the validation for each attribute individually and sets the message for the correct blurred field. A [demo](#) of the above by [@braddunbar](#) is also available.

Unfortunately, this solution does perform validation on all fields every time, even though we are only displaying errors for the field that has changed. If we have multiple client-side validation methods, we may not want to have to call each validation method on every attribute every time, so this solution might not be ideal for everyone.

A potentially better alternative to the above is to use [@gfranko's Backbone.validateAll](#) plugin, specifically created to validate specific Model properties

(or form fields) without worrying about the validation of any other Model properties (or form fields).

Here is how we would setup a partial User Model and validate method using this plugin for our use case:

```
// Create a new User Model
var User = Backbone.Model.extend({

  // RegEx Patterns
  patterns: {

    specialCharacters: '[^a-zA-Z 0-9]+',

    digits: '[0-9]',

    email: '^([a-zA-Z0-9._-]+@[a-zA-Z0-9][a-zA-Z0-9.-]*[.]{1}[a-zA-Z]{2,6})$',

  },

  // Validators
  validators: {

    minLength: function(value, minLength) {
      return value.length >= minLength;
    },

    maxLength: function(value, maxLength) {
      return value.length <= maxLength;
    },

    isEmail: function(value) {
      return User.prototype.validators.pattern(value, User.prototype.patterns.email);
    },

    hasSpecialCharacter: function(value) {
      return User.prototype.validators.pattern(value, User.prototype.patterns.specialCharacters);
    },

    ...

  },

  // We can determine which properties are getting validated by
  // checking to see if properties are equal to null
});
```

```

validate: function(attrs) {

    var errors = this.errors = {};

    if(attrs.firstname != null) {
        if (!attrs.firstname) {
            errors.firstname = 'firstname is required';
            console.log('first name isEmpty validation called');
        }

        else if(!this.validators.minLength(attrs.firstname, 2))
            errors.firstname = 'firstname is too short';
        else if(!this.validators.maxLength(attrs.firstname, 15))
            errors.firstname = 'firstname is too large';
        else if(this.validators.hasSpecialCharacter(attrs.firstname)) errors.firstname = 'firstname has special character'

    }

    if(attrs.lastname != null) {

        if (!attrs.lastname) {
            errors.lastname = 'lastname is required';
            console.log('last name isEmpty validation called');
        }

        else if(!this.validators.minLength(attrs.lastname, 2))
            errors.lastname = 'lastname is too short';
        else if(!this.validators.maxLength(attrs.lastname, 15))
            errors.lastname = 'lastname is too large';
        else if(this.validators.hasSpecialCharacter(attrs.lastname)) errors.lastname = 'lastname has special character'

    }

}

```

This allows the logic inside of our validate methods to determine which form fields are currently being set/validated, and ignore the model properties that are not being set.

It's fairly straight-forward to use as well. We can simply define a new Model instance and then set the data on our model using the `validateAll` option to use the behavior defined by the plugin:

```

var user = new User();
user.set({ 'firstname': 'Greg' }, {validate: true, validateAll: false});

```

That's it!

The Backbone.validateAll logic doesn't override the default Backbone logic by default and so it's perfectly capable of being used for scenarios where you might care more about field-validation [performance](#) as well as those where you don't. Both solutions presented in this section should work fine however.

Multiple Backbone versions Problem

In some instances it may be necessary to use multiple versions of Backbone in the same project.

Solution

Like most client-side projects, Backbone's code is wrapped in an immediately-invoked function expression:

```
(function(){  
    // Backbone.js  
}).call(this);
```

Several things happen during this configuration stage. A Backbone `namespace` is created, and multiple versions of Backbone on the same page are supported through the `noConflict` mode:

```
var root = this;  
var previousBackbone = root.Backbone;  
  
Backbone.noConflict = function() {  
    root.Backbone = previousBackbone;  
    return this;  
};
```

Multiple versions of Backbone can be used on the same page by calling `noConflict` like this:

```
var Backbone19 = Backbone.noConflict();  
// Backbone19 refers to the most recently loaded version,  
// and 'window.Backbone' will be restored to the previously  
// loaded version
```

Building Model and View hierarchies Problem

How does inheritance work with Backbone? How can I share code between similar models and views? How can I call methods that have been overridden?

Solution

For its inheritance, Backbone internally uses an `inherits` function inspired by `goog.inherits`, Google's implementation from the Closure Library. It's basically a function to correctly setup the prototype chain.

```
var inherits = function(parent, protoProps, staticProps) {  
  ...  
}
```

The only major difference here is that Backbone's API accepts two objects containing `instance` and `static` methods.

Following on from this, for inheritance purposes all of Backbone's objects contain an `extend` method as follows:

```
Model.extend = Collection.extend = Router.extend = View.extend = extend;
```

Most development with Backbone is based around inheriting from these objects, and they're designed to mimic a classical object-oriented implementation.

The above isn't quite the same as ECMAScript 5's `Object.create`, as it's actually copying properties (methods and values) from one object to another. As this isn't enough to support Backbone's inheritance and class model, the following steps are performed:

- The instance methods are checked to see if there's a constructor property. If so, the class's constructor is used, otherwise the parent's constructor is used (i.e., `Backbone.Model`)
- Underscore's `extend` method is called to add the parent class's methods to the new child class
- The `prototype` property of a blank constructor function is assigned with the parent's prototype, and a new instance of this is set to the child's `prototype` property
- Underscore's `extend` method is called twice to add the static and instance methods to the child class
- The child's prototype's constructor and a `__super__` property are assigned
- This pattern is also used for classes in CoffeeScript, so Backbone classes are compatible with CoffeeScript classes.

`extend` can be used for a great deal more and developers who are fans of mixins will like that it can be used for this too. You can define functionality on any custom object, and then quite literally copy & paste all of the methods and attributes from that object to a Backbone one:

For example:

```

var MyMixin = {
  foo: 'bar',
  sayFoo: function(){alert(this.foo);}
};

var MyView = Backbone.View.extend({
  // ...
});

_.extend(MyView.prototype, MyMixin);

var myView = new MyView();
myView.sayFoo(); //=> 'bar'

```

We can take this further and also apply it to View inheritance. The following is an example of how to extend one View using another:

```

var Panel = Backbone.View.extend({
});

var PanelAdvanced = Panel.extend({
});

```

Calling Overridden Methods

However, if you have an `initialize()` method in `Panel`, then it won't be called if you also have an `initialize()` method in `PanelAdvanced`, so you would have to call `Panel`'s `initialize` method explicitly:

```

var Panel = Backbone.View.extend({
  initialize: function(options){
    console.log('Panel initialized');
    this.foo = 'bar';
  }
});

var PanelAdvanced = Panel.extend({
  initialize: function(options){
    Panel.prototype.initialize.call(this, [options]);
    console.log('PanelAdvanced initialized');
    console.log(this.foo); // Log: bar
  }
});

// We can also inherit PanelAdvanced if needed

```



```

var PanelAdvancedExtra = PanelAdvanced.extend({
  initialize: function(options){
    PanelAdvanced.prototype.initialize.call(this, [options]);
    console.log('PanelAdvancedExtra initialized');
  }
});

new Panel();
new PanelAdvanced();
new PanelAdvancedExtra();

```

This isn't the most elegant of solutions because if you have a lot of Views that inherit from Panel, then you'll have to remember to call Panel's initialize from all of them.

It's worth noting that if Panel doesn't have an initialize method now but you choose to add it in the future, then you'll need to go to all of the inherited classes in the future and make sure they call Panel's initialize.

So here's an alternative way to define Panel so that your inherited views don't need to call Panel's initialize method:

```

var Panel = function (options) {
  // put all of Panel's initialization code here
  console.log('Panel initialized');
  this.foo = 'bar';

  Backbone.View.apply(this, [options]);
};

_.extend(Panel.prototype, Backbone.View.prototype, {
  // put all of Panel's methods here. For example:
  sayHi: function () {
    console.log('hello from Panel');
  }
});

Panel.extend = Backbone.View.extend;

// other classes then inherit from Panel like this:
var PanelAdvanced = Panel.extend({
  initialize: function (options) {
    console.log('PanelAdvanced initialized');
    console.log(this.foo);
  }
});

```

```
var panelAdvanced = new PanelAdvanced(); //Logs: Panel initialized, PanelAdvanced initialized
panelAdvanced.sayHi(); // Logs: hello from Panel
```

When used appropriately, Underscore's `extend` method can save a great deal of time and effort writing redundant code.

(Thanks to [Alex Young](#), [Derick Bailey](#) and [JohnnyO](#) for the heads up about these tips).

Backbone-Super

[Backbone-Super](#) by Lukas Olson adds a `__super` method to `Backbone.Model` using [John Resig's Inheritance script](#). Rather than using `Backbone.Model.prototype.set.call` as per the Backbone.js documentation, `__super` can be called instead:

```
// This is how we normally do it
var OldFashionedNote = Backbone.Model.extend({
  set: function(attributes, options) {
    // Call parent's method
    Backbone.Model.prototype.set.call(this, attributes, options);
    // some custom code here
    // ...
  }
});
```

After including this plugin, you can do the same thing with the following syntax:

```
// This is how we can do it after using the Backbone-super plugin
var Note = Backbone.Model.extend({
  set: function(attributes, options) {
    // Call parent's method
    this.__super(attributes, options);
    // some custom code here
    // ...
  }
});
```

Event Aggregators And Mediators Problem

How do I channel multiple event sources through a single object?*

Solution

Using an Event Aggregator. It's common for developers to think of Mediators when faced with this problem, so let's explore what an Event Aggregator is, what the Mediator pattern is and how they differ.

Design patterns often differ only in semantics and intent. That is, the language used to describe the pattern is what sets it apart, more than an implementation of that specific pattern. It often comes down to squares vs rectangles vs polygons. You can create the same end result with all three, given the constraints of a square are still met – or you can use polygons to create an infinitely larger and more complex set of things.

When it comes to the Mediator and Event Aggregator patterns, there are some times where it may look like the patterns are interchangeable due to implementation similarities. However, the semantics and intent of these patterns are very different. And even if the implementations both use some of the same core constructs, I believe there is a distinct difference between them. I also believe they should not be interchanged or confused in communication because of the differences.

Event Aggregator The core idea of the Event Aggregator, according to Martin Fowler, is to channel multiple event sources through a single object so that other objects needing to subscribe to the events don't need to know about every event source.

Backbone's Event Aggregator

The easiest event aggregator to show is that of Backbone.js – it's built in to the Backbone object directly.

```
var View1 = Backbone.View.extend({
  // ...

  events: {
    "click .foo": "doIt"
  },

  doIt: function(){
    // trigger an event through the event aggregator
    Backbone.trigger("some:event");
  }
});

var View2 = Backbone.View.extend({
  // ...

  initialize: function(){
    // subscribe to the event aggregator's event
    Backbone.on("some:event", this.doStuff, this);
  },
});
```

```

doStuff: function(){
  // ...
}
})

```

In this example, the first view is triggering an event when a DOM element is clicked. The event is triggered through Backbone’s built-in event aggregator – the Backbone object. Of course, it’s trivial to create your own event aggregator in Backbone, and there are some key things that we need to keep in mind when using an event aggregator, to keep our code simple.

jQuery’s Event Aggregator

Did you know that jQuery has a built-in event aggregator? They don’t call it this, but it’s in there and it’s scoped to DOM events. It also happens to look like Backbone’s event aggregator:

```

$("#mainArticle").on("click", function(e){

  // handle the event that any element underneath of our #mainArticle element

});

```

This code sets up an event handler function that waits for an unknown number of event sources to trigger a “click” event, and it allows any number of listeners to attach to the events of those event publishers. jQuery just happens to scope this event aggregator to the DOM.

Mediator A Mediator is an object that coordinates interactions (logic and behavior) between multiple objects. It makes decisions on when to call which objects, based on the actions (or in-action) of other objects and input.

A Mediator For Backbone

Backbone doesn’t have the idea of a mediator built in to it like a lot of other MV* frameworks do. But that doesn’t mean you can’t write one in 1 line of code:

```

var mediator = {};

```

Yes, of course this is just an object literal in JavaScript. Once again, we’re talking about semantics here. The purpose of the mediator is to control the workflow between objects and we really don’t need anything more than an object literal to do this.

```

var orgChart = {

  addNewEmployee: function(){

    // getEmployeeDetail provides a view that users interact with
    var employeeDetail = this.getEmployeeDetail();

    // when the employee detail is complete, the mediator (the 'orgchart' object)
    // decides what should happen next
    employeeDetail.on("complete", function(employee){

      // set up additional objects that have additional events, which are used
      // by the mediator to do additional things
      var managerSelector = this.selectManager(employee);
      managerSelector.on("save", function(employee){
        employee.save();
      });

    });
  },

  // ...
}

```

This example shows a very basic implementation of a mediator object with Backbone based objects that can trigger and subscribe to events. I've often referred to this type of object as a “workflow” object in the past, but the truth is that it is a mediator. It is an object that handles the workflow between many other objects, aggregating the responsibility of that workflow knowledge in to a single object. The result is workflow that is easier to understand and maintain.

Similarities And Differences There are, without a doubt, similarities between the event aggregator and mediator examples that I've shown here. The similarities boil down to two primary items: events and third-party objects. These differences are superficial at best, though. When we dig in to the intent of the pattern and see that the implementations can be dramatically different, the nature of the patterns become more apparent.

Events

Both the event aggregator and mediator use events, in the above examples. An event aggregator obviously deals with events – it's in the name after all. The mediator only uses events because it makes life easy when dealing with Backbone, though. There is nothing that says a mediator must be built with events. You can build a mediator with callback methods, by handing the mediator reference to the child object, or by any of a number of other means.

The difference, then, is why these two patterns are both using events. The event aggregator, as a pattern, is designed to deal with events. The mediator, though, only uses them because it's convenient.

Third-Party Objects

Both the event aggregator and mediator, by design, use a third-party object to facilitate things. The event aggregator itself is a third-party to the event publisher and the event subscriber. It acts as a central hub for events to pass through. The mediator is also a third party to other objects, though. So where is the difference? Why don't we call an event aggregator a mediator? The answer largely comes down to where the application logic and workflow is coded.

In the case of an event aggregator, the third party object is there only to facilitate the pass-through of events from an unknown number of sources to an unknown number of handlers. All workflow and business logic that needs to be kicked off is put directly in to the the object that triggers the events and the objects that handle the events.

In the case of the mediator, though, the business logic and workflow is aggregated in to the mediator itself. The mediator decides when an object should have it's methods called and attributes updated based on factors that the mediator knows about. It encapsulates the workflow and process, coordinating multiple objects to produce the desired system behaviour. The individual objects involved in this workflow each know how to perform their own task. But it's the mediator that tells the objects when to perform the tasks by making decisions at a higher level than the individual objects.

An event aggregator facilitates a "fire and forget" model of communication. The object triggering the event doesn't care if there are any subscribers. It just fires the event and moves on. A mediator, though, might use events to make decisions, but it is definitely not "fire and forget". A mediator pays attention to a known set of input or activities so that it can facilitate and coordinate additional behavior with a known set of actors (objects).

Relationships: When To Use Which Understanding the similarities and differences between an event aggregator and mediator is important for semantic reasons. It's equally as important to understand when to use which pattern, though. The basic semantics and intent of the patterns does inform the question of when, but actual experience in using the patterns will help you understand the more subtle points and nuanced decisions that have to be made.

Event Aggregator Use

In general, an event aggregator is uses when you either have too many objects to listen to directly, or you have objects that are unrelated entirely.

When two objects have a direct relationship already – say, a parent view and child view – then there might be little benefit in using an event aggregator.

Have the child view trigger an event and the parent view can handle the event. This is most commonly seen in Backbone's Collection and Model, where all Model events are bubbled up to and through its parent Collection. A Collection often uses model events to modify the state of itself or other models. Handling "selected" items in a collection is a good example of this.

jQuery's `on` method as an event aggregator is a great example of too many objects to listen to. If you have 10, 20 or 200 DOM elements that can trigger a "click" event, it might be a bad idea to set up a listener on all of them individually. This could quickly deteriorate performance of the application and user experience. Instead, using jQuery's `on` method allows us to aggregate all of the events and reduce the overhead of 10, 20, or 200 event handlers down to 1.

Indirect relationships are also a great time to use event aggregators. In Backbone applications, it is very common to have multiple view objects that need to communicate, but have no direct relationship. For example, a menu system might have a view that handles the menu item clicks. But we don't want the menu to be directly tied to the content views that show all of the details and information when a menu item is clicked. Having the content and menu coupled together would make the code very difficult to maintain, in the long run. Instead, we can use an event aggregator to trigger "menu:click:foo" events, and have a "foo" object handle the click event to show its content on the screen.

Mediator Use

A mediator is best applied when two or more objects have an indirect working relationship, and business logic or workflow needs to dictate the interactions and coordination of these objects.

A wizard interface is a good example of this, as shown with the "orgChart" example, above. There are multiple views that facilitate the entire workflow of the wizard. Rather than tightly coupling the view together by having them reference each other directly, we can decouple them and more explicitly model the workflow between them by introducing a mediator.

The mediator extracts the workflow from the implementation details and creates a more natural abstraction at a higher level, showing us at a much faster glance what that workflow is. We no longer have to dig in to the details of each view in the workflow, to see what the workflow actually is.

Event Aggregator And Mediator Together The crux of the difference between an event aggregator and a mediator, and why these pattern names should not be interchanged with each other, is illustrated best by showing how they can be used together. The menu example for an event aggregator is the perfect place to introduce a mediator as well.

Clicking a menu item may trigger a series of changes throughout an application. Some of these changes will be independent of others, and using an event aggregator for this makes sense. Some of these changes may be internally related to

each other, though, and may use a mediator to enact those changes. A mediator, then, could be set up to listen to the event aggregator. It could run its logic and process to facilitate and coordinate many objects that are related to each other, but unrelated to the original event source.

```
var MenuItem = Backbone.View.extend({

  events: {
    "click .thatThing": "clickedIt"
  },

  clickedIt: function(e){
    e.preventDefault();

    // assume this triggers "menu:click:foo"
    Backbone.trigger("menu:click:" + this.model.get("name"));
  }

});

// ... somewhere else in the app

var MyWorkflow = function(){
  Backbone.on("menu:click:foo", this.doStuff, this);
};

MyWorkflow.prototype.doStuff = function(){
  // instantiate multiple objects here.
  // set up event handlers for those objects.
  // coordinate all of the objects in to a meaningful workflow.
};
```

In this example, when the MenuItem with the right model is clicked, the "menu:click:foo" event will be triggered. An instance of the "MyWorkflow" object, assuming one is already instantiated, will handle this specific event and will coordinate all of the objects that it knows about, to create the desired user experience and workflow.

An event aggregator and a mediator have been combined to create a much more meaningful experience in both the code and the application itself. We now have a clean separation between the menu and the workflow through an event aggregator. And we are still keeping the workflow itself clean and maintainable through the use of a mediator.

Pattern Language: Semantics There is one overriding point to make in all of this discussion: semantics. Communicating intent and semantics through

the use of named patterns is only viable and only valid when all parties in a communication medium understand the language in the same way.

If I say “apple”, what am I talking about? Am I talking about a fruit? Or am I talking about a technology and consumer products company? As Sharon Cichelli says: “semantics will continue to be important, until we learn how to communicate in something other than language”.

Modular Development

Introduction

When we say an application is modular, we generally mean it’s composed of a set of highly decoupled, distinct pieces of functionality stored in modules. As you probably know, loose coupling facilitates easier maintainability of apps by removing dependencies where possible. When this is implemented efficiently, it’s quite easy to see how changes to one part of a system may affect another.

Unlike some more traditional programming languages, the current iteration of JavaScript (ECMA-262) doesn’t provide developers with the means to import such modules of code in a clean, organized manner.

Instead, developers are left to fall back on variations of the module or object literal patterns combined with script tags or a script loader. With many of these, module scripts are strung together in the DOM with namespaces being described by a single global object where it’s still possible to have name collisions. There’s also no clean way to handle dependency management without some manual effort or third party tools.

Whilst native solutions to these problems may be arriving via [ES6](#) (the next version of the official JavaScript specification) [modules proposal](#), the good news is that writing modular JavaScript has never been easier and you can start doing it today.

In this next part of the book, we’re going to look at how to use AMD modules and RequireJS to cleanly wrap units of code in your application into manageable modules. We’ll also cover an alternate approach called Lumbar which uses routes to determine when modules are loaded.

Organizing modules with RequireJS and AMD

Partly Contributed by [Jack Franklin](#)

[RequireJS](#) is a popular script loader written by James Burke - a developer who has been quite instrumental in helping shape the AMD module format, which we’ll discuss shortly. Amongst other things RequireJS helps you to load multiple

script files, define modules with or without dependencies, and load in non-script dependencies such as text files.

Maintainability problems with multiple script files

You might be thinking that there is little benefit to RequireJS. After all, you can simply load in your JavaScript files through multiple `<script>` tags, which is very straightforward. However, doing it that way has a lot of drawbacks, including increasing the HTTP overhead.

Every time the browser loads in a file you've referenced in a `<script>` tag, it makes an HTTP request to load the file's contents. It has to make a new HTTP request for each file you want to load, which causes problems.

- Browsers are limited in how many parallel requests they can make, so often it's slow to load multiple files, as it can only do a certain number at a time. This number depends on the user's settings and browser, but is usually around 4-8. When working on Backbone applications it's good to split your app into multiple JS files, so it's easy to hit that limit quickly. This can be negated by minifying your code into one file as part of a build process, but does not help with the next point.
- Scripts are loaded synchronously. This means that the browser cannot continue page rendering while the script is loading, .

What tools like RequireJS do is load scripts asynchronously. This means we have to adjust our code slightly, you can't just swap out `<script>` elements for a small piece of RequireJS code, but the benefits are very worthwhile:

- Loading the scripts asynchronously means the load process is non-blocking. The browser can continue to render the rest of the page as the scripts are being loaded, speeding up the initial load time.
- We can load modules in more intelligently, having more control over when they are loaded and ensuring that modules which have dependencies are loaded in the right order.

Need for better dependency management

Dependency management is a challenging subject, in particular when writing JavaScript in the browser. The closest thing we have to dependency management by default is simply making sure we order our `<script>` tags such that code that depends on code in another file is loaded after the file it depends on. This is not a good approach. As I've already discussed, loading multiple files in that way is bad for performance; needing them to be loaded in a certain order is very brittle.

Being able to load code on an as-needed basis is something RequireJS is very good at. Rather than load all our JavaScript code in during initial page load, a better approach is to dynamically load modules when that code is required. This avoids loading all the code when the user first hits your application, consequently speeding up initial load times.

Think about the GMail web client for a moment. When a user initially loads the page on their first visit, Google can simply hide widgets such as the chat module until the user has indicated (by clicking ‘expand’) that they wish to use it. Through dynamic dependency loading, Google could load up the chat module at that time, rather than forcing all users to load it when the page first initializes. This can improve performance and load times and can definitely prove useful when building larger applications. As the codebase for an application grows this becomes even more important.

The important thing to note here is that while it’s absolutely fine to develop applications without a script loader, there are significant benefits to utilizing tools like RequireJS in your application.

Asynchronous Module Definition (AMD)

RequireJS implements the [AMD Specification](#) which defines a method for writing modular code and managing dependencies. The RequireJS website also has a section [documenting the reasons behind implementing AMD](#):

The AMD format comes from wanting a module format that was better than today’s “write a bunch of script tags with implicit dependencies that you have to manually order” and something that was easy to use directly in the browser. Something with good debugging characteristics that did not require server-specific tooling to get started.

Writing AMD modules with RequireJS

As discussed above, the overall goal for the AMD format is to provide a solution for modular JavaScript that developers can use today. The two key concepts you need to be aware of when using it with a script-loader are the **define()** method for defining modules and the **require()** method for loading dependencies. **define()** is used to define named or unnamed modules using the following signature:

```
define(  
  module_id /*optional*/,  
  [dependencies] /*optional*/,  
  definition function /*function for instantiating the module or object*/  
);
```

As you can tell by the inline comments, the `module_id` is an optional argument which is typically only required when non-AMD concatenation tools are being used (there may be some other edge cases where it's useful too). When this argument is left out, we call the module 'anonymous'. When working with anonymous modules, RequireJS will use a module's file path as its module id, so the adage Don't Repeat Yourself (DRY) should be applied by omitting the module id in the `define()` invocation.

The dependencies argument is an array representing all of the other modules that this module depends on and the third argument is a factory that can either be a function that should be executed to instantiate the module or an object.

A barebones module (compatible with RequireJS) could be defined using `define()` as follows:

```
// A module ID has been omitted here to make the module anonymous

define(['foo', 'bar'],
  // module definition function
  // dependencies (foo and bar) are mapped to function parameters
  function ( foo, bar ) {
    // return a value that defines the module export
    // (i.e the functionality we want to expose for consumption)

    // create your module here
    var myModule = {
      doStuff:function(){
        console.log('Yay! Stuff');
      }
    }

    return myModule;
  });
```

Note: RequireJS is intelligent enough to automatically infer the '.js' extension to your script file names. As such, this extension is generally omitted when specifying dependencies.

Alternate syntax There is also a [sugared version](#) of `define()` available that allows you to declare your dependencies as local variables using `require()`. This will feel familiar to anyone who's used node, and can be easier to add or remove dependencies. Here is the previous snippet using the alternate syntax:

```
// A module ID has been omitted here to make the module anonymous
```

```

define(function(require){
    // module definition function
    // dependencies (foo and bar) are defined as local vars
    var foo = require('foo'),
        bar = require('bar');

    // return a value that defines the module export
    // (i.e the functionality we want to expose for consumption)

    // create your module here
    var myModule = {
        doStuff:function(){
            console.log('Yay! Stuff');
        }
    }

    return myModule;
});

```

The `require()` method is typically used to load code in a top-level JavaScript file or within a module should you wish to dynamically fetch dependencies. An example of its usage is:

```

// Consider 'foo' and 'bar' are two external modules
// In this example, the 'exports' from the two modules loaded are passed as
// function arguments to the callback (foo and bar)
// so that they can similarly be accessed

require( ['foo', 'bar'], function ( foo, bar ) {
    // rest of your code here
    foo.doSomething();
});

```

Addy's post on [Writing Modular JS](#) covers the AMD specification in much more detail. Defining and using modules will be covered in this book shortly when we look at more structured examples of using RequireJS.

Getting Started with RequireJS

Before using RequireJS and Backbone we will first set up a very basic RequireJS project to demonstrate how it works. The first thing to do is to [Download RequireJS](#). When you load in the RequireJS script in your HTML file, you need to also tell it where your main JavaScript file is located. Typically this will be called something like "app.js", and is the main entry point for your application. You do this by adding in a `data-main` attribute to the `script` tag:

```
<script data-main="app.js" src="lib/require.js"></script>
```

Now, RequireJS will automatically load `app.js` for you.

RequireJS Configuration In the main JavaScript file that you load with the `data-main` attribute you can configure how RequireJS loads the rest of your application. This is done by calling `require.config`, and passing in an object:

```
require.config({  
  // your configuration key/values here  
  baseUrl: "app", // generally the same directory as the script used in a data-main attribute  
  paths: {}, // set up custom paths to libraries, or paths to RequireJS plugins  
  shim: {}, // used for setting up all Shims (see below for more detail)  
});
```

The main reason you'd want to configure RequireJS is to add shims, which we'll cover next. To see other configuration options available to you, I recommend checking out the [RequireJS documentation](#).

RequireJS Shims Ideally, each library that we use with RequireJS will come with AMD support. That is, it uses the `define` method to define the library as a module. However, some libraries - including Backbone and one of its dependencies, Underscore - don't do this. Fortunately RequireJS comes with a way to work around this.

To demonstrate this, first let's shim Underscore, and then we'll shim Backbone too. Shims are very simple to implement:

```
require.config({  
  shim: {  
    'lib/underscore': {  
      exports: '_'  
    }  
  }  
});
```

Note that when specifying paths for RequireJS you should omit the `.js` from the end of script names.

The important line here is `exports: '_'`. This line tells RequireJS that the script in `'lib/underscore.js'` creates a global variable called `_` instead of defining a module. Now when we list Underscore as a dependency RequireJS will know to give us the `_` global variable as though it was the module defined by that script. We can set up a shim for Backbone too:

```
require.config({
  shim: {
    'lib/underscore': {
      exports: '_'
    },
    'lib/backbone': {
      deps: ['lib/underscore', 'jquery'],
      exports: 'Backbone'
    }
  }
});
```

Again, that configuration tells RequireJS to return the global **Backbone** variable that Backbone exports, but this time you'll notice that Backbone's dependencies are defined. This means whenever this:

```
require( 'lib/backbone', function( Backbone ) {...} );
```

Is run, it will first make sure the dependencies are met, and then pass the global **Backbone** object into the callback function. You don't need to do this with every library, only the ones that don't support AMD. For example, jQuery does support it, as of jQuery 1.7.

If you'd like to read more about general RequireJS usage, the [RequireJS API docs](#) are incredibly thorough and easy to read.

Custom Paths Typing long paths to file names like `lib/backbone` can get tedious. RequireJS lets us set up custom paths in our configuration object. Here, whenever I refer to “underscore”, RequireJS will look for the file `lib/underscore.js`:

```
require.config({
  paths: {
    'underscore': 'lib/underscore'
  }
});
```

Of course, this can be combined with a shim:

```
require.config({
  paths: {
    'underscore': 'lib/underscore'
  },
  shim: {
```

```

        'underscore': {
            exports: '_'
        }
    }
});

```

Just make sure that you refer to the custom path in your shim settings, too. Now you can do

```

require( ['underscore'], function(_) {
    // code here
});

```

to shim Underscore but still use a custom path.

Require.js and Backbone Examples

Now that we've taken a look at how to define AMD modules, let's review how to go about wrapping components like views and collections so that they can also be easily loaded as dependencies for any parts of your application that require them. At its simplest, a Backbone model may just require Backbone and Underscore.js. These are dependencies, so we can define those when defining the new modules. Note that the following examples presume you have configured RequireJS to shim Backbone and Underscore, as discussed previously.

Wrapping models, views, and other components with AMD For example, here is how a model is defined.

```

define(['underscore', 'backbone'], function(_, Backbone) {
    var myModel = Backbone.Model.extend({

        // Default attributes
        defaults: {
            content: 'hello world',
        },

        // A dummy initialization method
        initialize: function() {
        },

        clear: function() {
            this.destroy();
            this.view.remove();
        }
    });
});

```



```

    }

  });
  return myModel;
});

```

Note how we alias Underscore.js's instance to `_` and Backbone to just `Backbone`, making it very trivial to convert non-AMD code over to using this module format. For a view which might require other dependencies such as jQuery, this can similarly be done as follows:

```

define([
  'jquery',
  'underscore',
  'backbone',
  'collections/mycollection',
  'views/myview'
], function($, _, Backbone, myCollection, myView){

  var AppView = Backbone.View.extend({
    ...

```

Aliasing to the dollar-sign (\$) once again makes it very easy to encapsulate any part of an application you wish using AMD.

Doing it this way makes it easy to organize your Backbone application as you like. It's recommended to separate modules into folders. For example, individual folders for models, collections, views and so on. RequireJS doesn't care about what folder structure you use; as long as you use the correct path when using `require`, it will happily pull in the file.

As part of this chapter I've made a very simple [Backbone application with RequireJS that you can find on Github](#). It is a stock application for a manager of a shop. They can add new items and filter down the items based on price, but nothing more. Because it's so simple it's easier to focus purely on the RequireJS part of the implementation, rather than deal with complex JavaScript and Backbone logic too.

At the base of this application is the `Item` model, which describes a single item in the stock. Its implementation is very straight forward:

```

define( ["lib/backbone"], function ( Backbone ) {
  var Item = Backbone.Model.extend({
    defaults: {
      price: 35,
      photo: "http://www.placedog.com/100/100"

```

```

    }
  });
  return Item;
});

```

Converting an individual model, collection, view or similar into an AMD, RequireJS compliant one is typically very straight forward. Usually all that's needed is the first line, calling `define`, and to make sure that once you've defined your object - in this case, the `Item` model, to return it.

Let's now set up a view for that individual item:

```

define( ["lib/backbone"], function ( Backbone ) {
  var ItemView = Backbone.View.extend({
    tagName: "div",
    className: "item-wrap",
    template: _.template($("#itemTemplate").html()),

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

```

This view doesn't actually depend on the model it will be used with, so again the only dependency is Backbone. Other than that it's just a regular Backbone view. There's nothing special going on here, other than returning the object and using `define` so RequireJS can pick it up. Now let's make a collection to view a list of items. This time we will need to reference the `Item` model, so we add it as a dependency:

```

define(["lib/backbone", "models/item"], function(Backbone, Item) {
  var Cart = Backbone.Collection.extend({
    model: Item,
    initialize: function() {
      this.on("add", this.updateSet, this);
    },
    updateSet: function() {
      items = this.models;
    }
  });
  return Cart;
});

```

I've called this collection `Cart`, as it's a group of items. As the `Item` model is the second dependency, I can bind the variable `Item` to it by declaring it as the second argument to the callback function. I can then refer to this within my collection implementation.

Finally, let's have a look at the view for this collection. (This file is much bigger in the application, but I've taken some bits out so it's easier to examine).

```
define(["lib/backbone", "models/item", "views/itemview"], function(Backbone, Item, ItemView) {
  var ItemCollectionView = Backbone.View.extend({
    el: '#yourcart',
    initialize: function(collection) {
      this.collection = collection;
      this.render();
      this.collection.on("reset", this.render, this);
    },
    render: function() {
      this.$el.html("");
      this.collection.each(function(item) {
        this.renderItem(item);
      }, this);
    },
    renderItem: function(item) {
      var itemView = new ItemView({model: item});
      this.$el.append(itemView.render().el);
    },
    // more methods here removed
  });
  return ItemCollectionView;
});
```

There really is nothing to it once you've got the general pattern. Define each "object" (a model, view, collection, router or otherwise) through RequireJS, and then specify them as dependencies to other objects that need them. Again, you can find this entire application [on Github](#).

If you'd like to take a look at how others do it, [Pete Hawkins' Backbone Stack repository](#) is a good example of structuring a Backbone application using RequireJS. Greg Franko has also written [an overview of how he uses Backbone and Require](#), and [Jeremy Kahn's post](#) neatly describes his approach. For a look at a full sample application, the [Backbone and Require version](#) of the TodoMVC application is a good starting point.

Keeping Your Templates External Using RequireJS And The Text Plugin

Moving your templates to external files is actually quite straight-forward, whether they are Underscore, Mustache, Handlebars or any other text-based template format. Let's look at how we do that with RequireJS.

RequireJS has a special plugin called text.js which is used to load in text file dependencies. To use the text plugin, follow these steps:

1. Download the plugin from <http://requirejs.org/docs/download.html#text> and place it in either the same directory as your application's main JS file or a suitable sub-directory.
2. Next, include the text.js plugin in your initial RequireJS configuration options. In the code snippet below, we assume that RequireJS is being included in our page prior to this code snippet being executed.

```
require.config( {  
  paths: {  
    'text': 'libs/require/text',  
  },  
  baseUrl: 'app'  
} );
```

3. When the `text!` prefix is used for a dependency, RequireJS will automatically load the text plugin and treat the dependency as a text resource. A typical example of this in action may look like:

```
require(['js/app', 'text!templates/mainView.html'],  
  function( app, mainView ) {  
    // the contents of the mainView file will be  
    // loaded into mainView for usage.  
  }  
);
```

4. Finally we can use the text resource that's been loaded for templating purposes. You're probably used to storing your HTML templates inline using a script with a specific identifier.

With Underscore.js's micro-templating (and jQuery) this would typically be:

HTML:

```

<script type="text/template" id="mainViewTemplate">
  <% _.each( person, function( person_item ){ %>
    <li><%= person_item.get('name') %></li>
  <% }>; %>
</script>

```

JS:

```

var compiled_template = _.template( $('#mainViewTemplate').html() );

```

With RequireJS and the text plugin however, it's as simple as saving the same template into an external text file (say, `mainView.html`) and doing the following:

```

require(['js/app', 'text!templates/mainView.html'],
  function(app, mainView){
    var compiled_template = _.template( mainView );
  }
);

```

That's it! Now you can apply your template to a view in Backbone with something like:

```

collection.someview.$el.html( compiled_template( { results: collection.models } ) );

```

All templating solutions will have their own custom methods for handling template compilation, but if you understand the above, substituting Underscore's micro-templating for any other solution should be fairly trivial.

Optimizing Backbone apps for production with the RequireJS Optimizer

Once you're written your application, the next important step is to prepare it for deployment to production. The majority of non-trivial apps are likely to consist of several scripts and so optimizing, minimizing, and concatenating your scripts prior to pushing can reduce the number of scripts your users need to download.

A command-line optimization tool for RequireJS projects called `r.js` is available to help with this workflow. It offers a number of capabilities, including:

- Concatenating specific scripts and minifying them using external tools such as UglifyJS (which is used by default) or Google's Closure Compiler for optimal browser delivery, whilst preserving the ability to dynamically load modules

- Optimizing CSS and stylesheets by inlining CSS files imported using `@import`, stripping out comments, etc.
- The ability to run AMD projects in both Node and Rhino (more on this later)

If you find yourself wanting to ship a single file with all dependencies included, `r.js` can help with this too. Whilst `RequireJS` does support lazy-loading, your application may be small enough that reducing HTTP requests to a single script file is feasible.

You'll notice that I mentioned the word 'specific' in the first bullet point. The `RequireJS` optimizer only concatenates module scripts that have been specified as string literals in `require` and `define` calls (which you've probably used). As clarified by the [optimizer docs](#) this means that Backbone modules defined like this:

```
define(['jquery', 'backbone', 'underscore', 'collections/sample', 'views/test'],
  function($, Backbone, _, Sample, Test){
    //...
  });
```

will combine fine, however dynamic dependencies such as:

```
var models = someCondition ? ['models/ab', 'models/ac'] : ['models/ba', 'models/bc'];
define(['jquery', 'backbone', 'underscore'].concat(models),
  function($, Backbone, _, firstModel, secondModel){
    //...
  });
```

will be ignored. This is by design as it ensures that dynamic dependency/module loading can still take place even after optimization.

Although the `RequireJS` optimizer works fine in both Node and Java environments, it's strongly recommended to run it under Node as it executes significantly faster there.

To get started with `r.js`, grab it from the [RequireJS download page](#) or [through NPM](#). To begin getting our project to build with `r.js`, we will need to create a new build profile.

Assuming the code for our application and external dependencies are in `app/libs`, our `build.js` build profile could simply be:

```
({
  baseUrl: 'app',
  out: 'dist/main.js',
```

The paths above are relative to the `baseUrl` for our project and in our case it would make sense to make this the `app` folder. The `out` parameter informs `r.js` that we want to concatenate everything into a single file.

Alternatively, we can specify `dir`, which will ensure the contents of our `app` directory are copied into this directory. e.g:

```
({
  baseUrl: 'app',
  dir: 'release',
  out: 'dist/main.js'
```

Additional options that can be specified such as `modules` and `appDir` are not compatible with `out`, however let's briefly discuss them in case you do wish to use them.

`modules` is an array where we can explicitly specify the module names we would like to have optimized.

```
modules: [
  {
    name: 'app',
    exclude: [
      // If you prefer not to include certain
      // libs exclude them here
    ]
  }
]
```

`appDir` - when specified, `baseUrl` is relative to this parameter. If `appDir` is not defined, `baseUrl` is simply relative to the `build.js` file.

```
appDir: './',
```

Back to our build profile, the `main` parameter is used to specify our main module - we are making use of `include` here as we're going to take advantage of [Almond](#) - a stripped down loader for RequireJS modules which is useful should you not need to load modules in dynamically.

```
include: ['libs/almond', 'main'],
wrap: true,
```

`include` is another array which specifies the modules we want to include in the build. By specifying "main", `r.js` will trace over all modules main depends on and will include them. `wrap` wraps modules which RequireJS needs into a closure so that only what we export is included in the global environment.

```

    paths: {
      backbone: 'libs/backbone',
      underscore: 'libs/underscore',
      jquery: 'libs/jquery',
      text: 'libs/text'
    }
  })

```

The remainder of the build.js file would be a regular paths configuration object. We can compile our project into a target file by running:

```
node r.js -o build.js
```

which should place our compiled project into dist/main.js.

The build profile is usually placed inside the 'scripts' or 'js' directory of your project. As per the docs, this file can however exist anywhere you wish, but you'll need to edit the contents of your build profile accordingly.

That's it. As long as you have UglifyJS/Closure tools setup correctly, r.js should be able to easily optimize your entire Backbone project in just a few key-strokes.

If you would like to learn more about build profiles, James Burke has a [heavily commented sample file](#) with all the possible options available.

Exercise: Your First Modular Backbone + RequireJS App

In this chapter, we'll look at our first practical Backbone & RequireJS project - how to build a modular Todo application. Similar to exercise 1, the application will allow us to add new todos, edit new todos and clear todo items that have been marked as completed. For a more advanced practical, see the section on mobile Backbone development.

The complete code for the application can be found in the `practicals/modular-todo-app` folder of this repo (thanks to Thomas Davis and Jérôme Gravel-Niquet). Alternatively grab a copy of my side-project [TodoMVC](#) which contains the sources to both AMD and non-AMD versions.

Overview

Writing a modular Backbone application can be a straight-forward process. There are however, some key conceptual differences to be aware of if opting to use AMD as your module format of choice:

- As AMD isn't a standard native to JavaScript or the browser, it's necessary to use a script loader (such as RequireJS or curl.js) in order to support defining components and modules using this module format. As we've already reviewed, there are a number of advantages to using the AMD as well as RequireJS to assist here.
- Models, views, controllers and routers need to be encapsulated *using* the AMD-format. This allows each component of our Backbone application to cleanly manage dependencies (e.g collections required by a view) in the same way that AMD allows non-Backbone modules to.
- Non-Backbone components/modules (such as utilities or application helpers) can also be encapsulated using AMD. I encourage you to try developing these modules in such a way that they can both be used and tested independent of your Backbone code as this will increase their ability to be re-used elsewhere.

Now that we've reviewed the basics, let's take a look at developing our application. For reference, the structure of our app is as follows:

```
index.html
...js/
  main.js
  .../models
    todo.js
  .../views
    app.js
    todos.js
  .../collections
    todos.js
  .../templates
    stats.html
    todos.html
  ../libs
    .../backbone
    .../jquery
    .../underscore
    .../require
      require.js
      text.js
...css/
```

Markup

The markup for the application is relatively simple and consists of three primary parts: an input section for entering new todo items (**create-todo**), a list

section to display existing items (which can also be edited in-place) (`todo-list`) and finally a section summarizing how many items are left to be completed (`todo-stats`).

```
<div id="todoapp">

  <div class="content">

    <div id="create-todo">
      <input id="new-todo" placeholder="What needs to be done?" type="text" />
      <span class="ui-tooltip-top">Press Enter to save this task</span>
    </div>

    <div id="todos">
      <ul id="todo-list"></ul>
    </div>

    <div id="todo-stats"></div>

  </div>

</div>
```

The rest of the tutorial will now focus on the JavaScript side of the practical.

Configuration options

If you've read the earlier chapter on AMD, you may have noticed that explicitly needing to define each dependency a Backbone module (view, collection or other module) may require with it can get a little tedious. This can however be improved.

In order to simplify referencing common paths the modules in our application may use, we use a RequireJS [configuration object](#), which is typically defined as a top-level script file. Configuration objects have a number of useful capabilities, the most useful being module name-mapping. Name-maps are basically a key:value pair, where the key defines the alias you wish to use for a path and the value represents the true location of the path.

In the code-sample below, you can see some typical examples of common name-maps which include: `backbone`, `underscore`, `jquery` and depending on your choice, the RequireJS `text` plugin, which assists with loading text assets like templates.

main.js

```

require.config({
  baseUrl: '../',
  paths: {
    jquery: 'libs/jquery/jquery-min',
    underscore: 'libs/underscore/underscore-min',
    backbone: 'libs/backbone/backbone-optamd3-min',
    text: 'libs/require/text'
  }
});

require(['views/app'], function(AppView){
  var app_view = new AppView;
});

```

The `require()` at the end of our `main.js` file is simply there so we can load and instantiate the primary view for our application (`views/app.js`). You'll commonly see both this and the configuration object included in most top-level script files for a project.

In addition to offering name-mapping, the configuration object can be used to define additional properties such as `waitSeconds` - the number of seconds to wait before script loading times out and `locale`, should you wish to load up i18n bundles for custom languages. The `baseUrl` is simply the path to use for module lookups.

For more information on configuration objects, please feel free to check out the excellent guide to them in the [RequireJS docs](#).

Modularizing our models, views and collections

Before we dive into AMD-wrapped versions of our Backbone components, let's review a sample of a non-AMD view. The following view listens for changes to its model (a `Todo` item) and re-renders if a user edits the value of the item.

```

var TodoView = Backbone.View.extend({

  //... is a list tag.
  tagName: 'li',

  // Cache the template function for a single item.
  template: _.template($('#item-template').html()),

  // The DOM events specific to an item.
  events: {
    'click .check' : 'toggleDone',
    'dblclick div.todo-content' : 'edit',

```

```

        'click span.todo-destroy'    : 'clear',
        'keypress .todo-input'      : 'updateOnEnter'
    },

    // The TodoView listens for changes to its model, re-rendering. Since there's
    // a one-to-one correspondence between a Todo and a TodoView in this
    // app, we set a direct reference on the model for convenience.
    initialize: function() {
        this.model.on('change', this.render, this);
        this.model.view = this;
    },
    ...

```

Note how for templating the common practice of referencing a script by an ID (or other selector) and obtaining its value is used. This of course requires that the template being accessed is implicitly defined in our markup. The following is the ‘embedded’ version of our template being referenced above:

```

<script type="text/template" id="item-template">
    <div class="todo <%= done ? 'done' : '' %>">
        <div class="display">
            <input class="check" type="checkbox" <%= done ? 'checked="checked"' : '' %> />
            <div class="todo-content"></div>
            <span class="todo-destroy"></span>
        </div>
        <div class="edit">
            <input class="todo-input" type="text" value="" />
        </div>
    </div>
</script>

```

Whilst there is nothing wrong with the template itself, once we begin to develop larger applications requiring multiple templates, including them all in our markup on page-load can quickly become both unmanageable and come with performance costs. We’ll look at solving this problem in a minute.

Let’s now take a look at the AMD-version of our view. As discussed earlier, the ‘module’ is wrapped using AMD’s **define()** which allows us to specify the dependencies our view requires. Using the mapped paths to ‘jquery’ etc. simplifies referencing common dependencies and instances of dependencies are themselves mapped to local variables that we can access (e.g ‘jquery’ is mapped to \$).

views/todo.js

```

define([
    'jquery',

```

```

    'underscore',
    'backbone',
    'text!templates/todos.html'
  ], function($, _, Backbone, todosTemplate){
    var TodoView = Backbone.View.extend({

      //... is a list tag.
      tagName: 'li',

      // Cache the template function for a single item.
      template: _.template(todosTemplate),

      // The DOM events specific to an item.
      events: {
        'click .check'           : 'toggleDone',
        'dblclick div.todo-content' : 'edit',
        'click span.todo-destroy'  : 'clear',
        'keypress .todo-input'     : 'updateOnEnter'
      },

      // The TodoView listens for changes to its model, re-rendering. Since there's
      // a one-to-one correspondence between a **Todo** and a **TodoView** in this
      // app, we set a direct reference on the model for convenience.
      initialize: function() {
        this.model.on('change', this.render, this);
        this.model.view = this;
      },

      // Re-render the contents of the todo item.
      render: function() {
        this.$el.html(this.template(this.model.toJSON()));
        this.setContent();
        return this;
      },

      // Use 'jQuery.text' to set the contents of the todo item.
      setContent: function() {
        var content = this.model.get('content');
        this.$('.todo-content').text(content);
        this.input = this.$('.todo-input');
        this.input.on('blur', this.close);
        this.input.val(content);
      },
      ...
    });
  });

```

From a maintenance perspective, there's nothing logically different in this version of our view, except for how we approach templating.

Using the RequireJS text plugin (the dependency marked `text`), we can actually store all of the contents for the template we looked at earlier in an external file (`todos.html`).

`templates/todos.html`

```
<div class="todo <%= done ? 'done' : '' %>">
  <div class="display">
    <input class="check" type="checkbox" <%= done ? 'checked="checked"' : '' %> />
    <div class="todo-content"></div>
    <span class="todo-destroy"></span>
  </div>
  <div class="edit">
    <input class="todo-input" type="text" value="" />
  </div>
</div>
```

There's no longer a need to be concerned with IDs for the template as we can map its contents to a local variable (in this case `todosTemplate`). We then simply pass this to the Underscore.js templating function `_.template()` the same way we normally would have the value of our template script.

Next, let's look at how to define models as dependencies which can be pulled into collections. Here's an AMD-compatible model module, which has two default values: a `content` attribute for the content of a Todo item and a boolean `done` state, allowing us to trigger whether the item has been completed or not.

`models/todo.js`

```
define(['underscore', 'backbone'], function(_, Backbone) {
  var TodoModel = Backbone.Model.extend({

    // Default attributes for the todo.
    defaults: {
      // Ensure that each todo created has 'content'.
      content: 'empty todo...',
      done: false
    },

    initialize: function() {
    },

    // Toggle the 'done' state of this todo item.
    toggle: function() {
```

```

        this.save({done: !this.get('done')});
    },

    // Remove this Todo from *localStorage* and delete its view.
    clear: function() {
        this.destroy();
        this.view.remove();
    }

});
return TodoModel;
});

```

As per other types of dependencies, we can easily map our model module to a local variable (in this case `Todo`) so it can be referenced as the model to use for our `TodosCollection`. This collection also supports a simple `done()` filter for narrowing down `Todo` items that have been completed and a `remaining()` filter for those that are still outstanding.

collections/todos.js

```

define([
    'underscore',
    'backbone',
    'libs/backbone/localstorage',
    'models/todo'
], function(_, Backbone, Store, Todo){

    var TodosCollection = Backbone.Collection.extend({

        // Reference to this collection's model.
        model: Todo,

        // Save all of the todo items under the 'todos' namespace.
        localStorage: new Store('todos'),

        // Filter down the list of all todo items that are finished.
        done: function() {
            return this.filter(function(todo){ return todo.get('done'); });
        },

        // Filter down the list to only todo items that are still not finished.
        remaining: function() {
            return this.without.apply(this, this.done());
        },
        ...
    });

```

In addition to allowing users to add new Todo items from views (which we then insert as models in a collection), we ideally also want to be able to display how many items have been completed and how many are remaining. We've already defined filters that can provide us this information in the above collection, so let's use them in our main application view.

views/app.js

```
define([
  'jquery',
  'underscore',
  'backbone',
  'collections/todos',
  'views/todo',
  'text!templates/stats.html'
], function($, _, Backbone, Todos, TodoView, statsTemplate){

  var AppView = Backbone.View.extend({

    // Instead of generating a new element, bind to the existing skeleton of
    // the App already present in the HTML.
    el: $('#todoapp'),

    // Our template for the line of statistics at the bottom of the app.
    statsTemplate: _.template(statsTemplate),

    // ...events, initialize() etc. can be seen in the complete file

    // Re-rendering the App just means refreshing the statistics -- the rest
    // of the app doesn't change.
    render: function() {
      var done = Todos.done().length;
      this.$('#todo-stats').html(this.statsTemplate({
        total:      Todos.length,
        done:       Todos.done().length,
        remaining:  Todos.remaining().length
      }));
    },
    ...
  });
});
```

Above, we map the second template for this project, `templates/stats.html` to `statsTemplate` which is used for rendering the overall `done` and `remaining` states. This works by simply passing our template the length of our overall `Todos` collection (`Todos.length` - the number of `Todo` items created so far) and similarly the length (counts) for items that have been completed (`Todos.done().length`) or are remaining (`Todos.remaining().length`).

The contents of our `statsTemplate` can be seen below. It's nothing too complicated, but does use ternary conditions to evaluate whether we should state there's "1 item" or "2 items" in a particular state.

```
<% if (total) { %>
  <span class="todo-count">
    <span class="number"><%= remaining %></span>
    <span class="word"><%= remaining == 1 ? 'item' : 'items' %></span> left.
  </span>
<% } %>
<% if (done) { %>
  <span class="todo-clear">
    <a href="#">
      Clear <span class="number-done"><%= done %></span>
      completed <span class="word-done"><%= done == 1 ? 'item' : 'items' %></span>
    </a>
  </span>
<% } %>
```

The rest of the source for the Todo app mainly consists of code for handling user and application events, but that rounds up most of the core concepts for this practical.

To see how everything ties together, feel free to grab the source by cloning this repo or browse it [online](#) to learn more. I hope you find it helpful!

Note: While this first practical doesn't use a build profile as outlined in the chapter on using the RequireJS optimizer, we will be using one in the section on building mobile Backbone applications.

Route-based module loading

This section will discuss a route based approach to module loading as implemented in [Lumbar](#) by Kevin Decker. Like RequireJS, Lumbar is also a modular build system, but the pattern it implements for loading routes may be used with any build system.

The specifics of the Lumbar build tool are not discussed in this book. To see a complete Lumbar based project with the loader and build system see [Thorax](#) which provides boilerplate projects for various environments including Lumbar.

JSON-based module configuration

RequireJS defines dependencies per file, while Lumbar defines a list of files for each module in a central JSON configuration file, outputting a single JavaScript

file for each defined module. Lumbar requires that each module (except the base module) define a single router and a list of routes. An example file might look like:

```
{
  "modules": {
    "base": {
      "scripts": [
        "js/lib/underscore.js",
        "js/lib/backbone.js",
        "etc"
      ]
    },
    "pages": {
      "scripts": [
        "js/routers/pages.js",
        "js/views/pages/index.js",
        "etc"
      ],
      "routes": {
        "": "index",
        "contact": "contact"
      }
    }
  }
}
```

Every JavaScript file defined in a module will have a `module` object in scope which contains the `name` and `routes` for the module. In `js/routers/pages.js` we could define a Backbone router for our `pages` module like so:

```
new (Backbone.Router.extend({
  routes: module.routes,
  index: function() {},
  contact: function() {}
}));
```

Module loader Router

A little used feature of `Backbone.Router` is its ability to create multiple routers that listen to the same set of routes. Lumbar uses this feature to create a router that listens to all routes in the application. When a route is matched, this master router checks to see if the needed module is loaded. If the module is already loaded, then the master router takes no action and the router defined by the

module will handle the route. If the needed module has not yet been loaded, it will be loaded, then `Backbone.history.loadUrl` will be called. This reloads the route, causes the master router to take no further action and the router defined in the freshly loaded module to respond.

A sample implementation is provided below. The `config` object would need to contain the data from our sample configuration JSON file above, and the `loader` object would need to implement `isLoaded` and `loadModule` methods. Note that Lumbar provides all of these implementations, the examples are provided to create your own implementation.

```
// Create an object that will be used as the prototype
// for our master router
var handlers = {
  routes: {}
};

_.each(config.modules, function(module, moduleName) {
  if (module.routes) {
    // Generate a loading callback for the module
    var callbackName = "loader_" + moduleName;
    handlers[callbackName] = function() {
      if (loader.isLoaded(moduleName)) {
        // Do nothing if the module is loaded
        return;
      } else {
        //the module needs to be loaded
        loader.loadModule(moduleName, function() {
          // Module is loaded, reloading the route
          // will trigger callback in the module's
          // router
          Backbone.history.loadUrl();
        });
      }
    };
    // Each route in the module should trigger the
    // loading callback
    _.each(module.routes, function(methodName, route) {
      handlers.routes[route] = callbackName;
    });
  }
});

// Create the master router
new (Backbone.Router.extend(handlers));
```

Using NodeJS to handle pushState

`window.history.pushState` support (serving Backbone routes without a hash mark) requires that the server be aware of what URLs your Backbone application will handle, since the user should be able to enter the app at any of those routes (or hit reload after navigating to a pushState URL).

Another advantage to defining all routes in a single location is that the same JSON configuration file provided above could be loaded by the server, listening to each route. A sample implementation in NodeJS and Express:

```
var fs = require('fs'),
    _ = require('underscore'),
    express = require('express'),
    server = express(),
    config = JSON.parse(fs.readFileSync('path/to/config.json'));

_.each(config.modules, function(module, moduleName) {
  if (module.routes) {
    _.each(module.routes, function(methodName, route) {
      server.get(route, function(req, res) {
        res.sendFile('public/index.html');
      });
    });
  }
});
```

This assumes that `index.html` will be serving out your Backbone application. The `Backbone.History` object can handle the rest of the routing logic as long as a `root` option is specified. A sample configuration for a simple application that lives at the root might look like:

```
Backbone.history || (Backbone.history = new Backbone.History());
Backbone.history.start({
  pushState: true,
  root: '/'
});
```

An asset package alternative for dependency management

For more than trivial views, DocumentCloud have a home-built asset packager called [Jammit](#), which has easy integration with Underscore.js templates and can also be used for dependency management.

Jammit expects your JavaScript templates (JST) to live alongside any ERB templates you're using in the form of `.jst` files. It packages the templates into a

global JST object which can be used to render templates into strings. Making Jammit aware of your templates is straight-forward - just add an entry for something like `views/**/*.jst` to your app package in `assets.yml`.

To provide Jammit dependencies you simply write out an `assets.yml` file that either listed the dependencies in order or used a combination of free capture directories (for example: `//.js`, `templates/.js`, and specific files).

A template using Jammit can derive it's data from the collection object that is passed to it:

```
this.$el.html(JST.myTemplate({ collection: this.collection }));
```

Paginating Backbone.js Requests & Collections

Pagination is a ubiquitous problem we often find ourselves needing to solve on the web. Perhaps most predominantly when working with back-end APIs and JavaScript-heavy clients which consume them.

On this topic, we're going to go through a set of **pagination components** I wrote for Backbone.js, which should hopefully come in useful if you're working on applications which need to tackle this problem. They're part of an extension called [Backbone.Paginator](#).

When working with a structural framework like Backbone.js, the three types of pagination we are most likely to run into are:

Requests to a service layer (API)- e.g query for results containing the term 'Brendan' - if 5,000 results are available only display 20 results per page (leaving us with 250 possible result pages that can be navigated to).

This problem actually has quite a great deal more to it, such as maintaining persistence of other URL parameters (e.g sort, query, order) which can change based on a user's search configuration in a UI. One also has to think of a clean way of hooking views up to this pagination so you can easily navigate between pages (e.g., First, Last, Next, Previous, 1,2,3), manage the number of results displayed per page and so on.

Further client-side pagination of data returned - e.g we've been returned a JSON response containing 100 results. Rather than displaying all 100 to the user, we only display 20 of these results within a navigable UI in the browser.

Similar to the request problem, client-pagination has its own challenges like navigation once again (Next, Previous, 1,2,3), sorting, order, switching the number of results to display per page and so on.

Infinite results - with services such as Facebook, the concept of numeric pagination is instead replaced with a 'Load More' or 'View More' button. Triggering

this normally fetches the next ‘page’ of N results but rather than replacing the previous set of results loaded entirely, we simply append to them instead.

A request pager which simply appends results in a view rather than replacing on each new fetch is effectively an ‘infinite’ pager.

Let’s now take a look at exactly what we’re getting out of the box:

Paginator is a set of opinionated components for paginating collections of data using Backbone.js. It aims to provide both solutions for assisting with pagination of requests to a server (e.g an API) as well as pagination of single-loads of data, where we may wish to further paginate a collection of N results into M pages within a view.

Paginator’s pieces

Backbone.Paginator supports two main pagination components:

- **Backbone.Paginator.requestPager:** For pagination of requests between a client and a server-side API
- **Backbone.Paginator.clientPager:** For pagination of data returned from a server which you would like to further paginate within the UI (e.g 60 results are returned, paginate into 3 pages of 20)

Live Examples

Live previews of both pagination components using the Netflix API can be found below. Fork the repository to experiment with these examples further.

- [Backbone.Paginator.requestPager\(\)](#)
- [Backbone.Paginator.clientPager\(\)](#)
- [Infinite Pagination \(Backbone.Paginator.requestPager\(\)\)](#)
- [Diacritic Plugin](#)

Paginator.requestPager

In this section we’re going to walkthrough actually using the requestPager.

1. Create a new Paginated collection First, we define a new Paginated collection using `Backbone.Paginator.requestPager()` as follows:

```
var PaginatedCollection = Backbone.Paginator.requestPager.extend({
```

2: Set the model for the collection as normal Within our collection, we then (as normal) specify the model to be used with this collection followed by the URL (or base URL) for the service providing our data (e.g the Netflix API).

```
model: model,
```

3. Configure the base URL and the type of the request We need to set a base URL. The type of the request is GET by default, and the `dataType` is `jsonp` in order to enable cross-domain requests.

```
paginator_core: {  
  // the type of the request (GET by default)  
  type: 'GET',  
  
  // the type of reply (jsonp by default)  
  dataType: 'jsonp',  
  
  // the URL (or base URL) for the service  
  url: 'http://odata.netflix.com/Catalog/People(49446)/TitlesActedIn?'  
},
```

4. Configure how the library will show the results We need to tell the library how many items per page would we like to see, etc...

```
paginator_ui: {  
  // the lowest page index your API allows to be accessed  
  firstPage: 0,  
  
  // which page should the paginator start from  
  // (also, the actual page the paginator is on)  
  currentPage: 0,  
  
  // how many items per page should be shown  
  perPage: 3,  
  
  // a default number of total pages to query in case the API or  
  // service you are using does not support providing the total  
  // number of pages for us.  
  // 10 as a default in case your service doesn't return the total  
  totalPages: 10  
},
```

5. Configure the parameters we want to send to the server The base URL on its own won't be enough for most cases, so you can pass more parameters to the server.

Note how you can use functions instead of hardcoded values, and you can also refer to the values you specified in `paginator_ui`.

```
server_api: {
  // the query field in the request
  '$filter': '',

  // number of items to return per request/page
  '$top': function() { return this.perPage },

  // how many results the request should skip ahead to
  // customize as needed. For the Netflix API, skipping ahead based on
  // page * number of results per page was necessary.
  '$skip': function() { return this.currentPage * this.perPage },

  // field to sort by
  '$orderby': 'ReleaseYear',

  // what format would you like to request results in?
  '$format': 'json',

  // custom parameters
  '$inlinecount': 'allpages',
  '$callback': 'callback'
},
```

6. Finally, configure `Collection.parse()` and we're done The last thing we need to do is configure our collection's `parse()` method. We want to ensure we're returning the correct part of our JSON response containing the data our collection will be populated with, which below is `response.d.results` (for the Netflix API).

You might also notice that we're setting `this.totalPages` to the total page count returned by the API. This allows us to define the maximum number of (result) pages available for the current/last request so that we can clearly display this in the UI. It also allows us to influence whether clicking say, a 'next' button should proceed with a request or not.

```
parse: function (response) {
  // Be sure to change this based on how your results
  // are structured (e.g d.results is Netflix specific)
  var tags = response.d.results;
```



```

        //Normally this.totalPages would equal response.d.__count
        //but as this particular Netflix request only returns a
        //total count of items for the search, we divide.
        this.totalPages = Math.floor(response.d.__count / this.perPage);
        return tags;
    }
  });
});

```

Convenience methods: For your convenience, the following methods are made available for use in your views to interact with the `requestPager`:

- `Collection.goTo(n, options)` - go to a specific page
- `Collection.requestNextPage(options)` - go to the next page
- `Collection.requestPreviousPage(options)` - go to the previous page
- `Collection.howManyPer(n)` - set the number of items to display per page

`requestPager` collection's methods `.goTo()`, `.requestNextPage()` and `.requestPreviousPage()` are all extensions of the original [Backbone Collection.fetch\(\) method](#). As such, they can all take the same option object as parameter.

This option object can use `success` and `error` parameters to pass a function to be executed after server answer.

```

Collection.goTo(n, {
  success: function( collection, response ) {
    // called if server request succeeds
  },
  error: function( collection, response ) {
    // called if server request fails
  }
});

```

You could also use the [jqXHR](#) returned by these methods to manage callbacks.

```

Collection
  .requestNextPage()
  .done(function( data, textStatus, jqXHR ) {
    // called is server request success
  })
  .fail(function( data, textStatus, jqXHR ) {

```

```

        // called if server request fail
    })
    .always(function( data, textStatus, jqXHR ) {
        // do something after server request is complete
    });
});

```

If you'd like to add the incoming models to the current collection, instead of replacing the collection's contents, pass `{add: true}` as an option to these methods.

```
Collection.requestPreviousPage({ add: true });
```

Paginator.clientPager

The `clientPager` works similar to the `requestPager`, except that our configuration values influence the pagination of data already returned at a UI-level. Whilst not shown (yet) there is also a lot more UI logic that ties in with the `clientPager`. An example of this can be seen in 'views/clientPagination.js'.

1. Create a new paginated collection with a model and URL As with `requestPager`, let's first create a new `Paginated Backbone.Paginator.clientPager` collection, with a model:

```

var PaginatedCollection = Backbone.Paginator.clientPager.extend({
    model: model,

```

2. Configure the base URL and the type of the request We need to set a base URL. The `type` of the request is `GET` by default, and the `dataType` is `jsonp` in order to enable cross-domain requests.

```

paginator_core: {
    // the type of the request (GET by default)
    type: 'GET',

    // the type of reply (jsonp by default)
    dataType: 'jsonp',

    // the URL (or base URL) for the service
    url: 'http://odata.netflix.com/v2/Catalog/Titles?&'
},

```

3. Configure how the library will show the results We need to tell the library how many items per page would we like to see, etc. . .

```
paginator_ui: {  
  // the lowest page index your API allows to be accessed  
  firstPage: 1,  
  
  // which page should the paginator start from  
  // (also, the actual page the paginator is on)  
  currentPage: 1,  
  
  // how many items per page should be shown  
  perPage: 3,  
  
  // a default number of total pages to query in case the API or  
  // service you are using does not support providing the total  
  // number of pages for us.  
  // 10 as a default in case your service doesn't return the total  
  totalPages: 10  
},
```

4. Configure the parameters we want to send to the server The base URL on its own won't be enough for most cases, so you can pass more parameters to the server.

Note how you can use functions instead of hardcoded values, and you can also refer to the values you specified in `paginator_ui`.

```
server_api: {  
  // the query field in the request  
  '$filter': 'substringof(\'america\',Name)',  
  
  // number of items to return per request/page  
  '$top': function() { return this.perPage },  
  
  // how many results the request should skip ahead to  
  // customize as needed. For the Netflix API, skipping ahead based on  
  // page * number of results per page was necessary.  
  '$skip': function() { return this.currentPage * this.perPage },  
  
  // field to sort by  
  '$orderby': 'ReleaseYear',  
  
  // what format would you like to request results in?  
  '$format': 'json',
```

```

    // custom parameters
    '$inlinecount': 'allpages',
    '$callback': 'callback'
  },

```

5. Finally, configure `Collection.parse()` and we're done And finally we have our `parse()` method, which in this case isn't concerned with the total number of result pages available on the server as we have our own total count of pages for the paginated data in the UI.

```

  parse: function (response) {
    var tags = response.d.results;
    return tags;
  }

});

```

Convenience methods: As mentioned, your views can hook into a number of convenience methods to navigate around UI-paginated data. For `clientPager` these include:

- **`Collection.goTo(n)`** - go to a specific page
- **`Collection.previousPage()`** - go to the previous page
- **`Collection.nextPage()`** - go to the next page
- **`Collection.howManyPer(n)`** - set how many items to display per page
- **`Collection.setSort(sortBy, sortDirection)`** - update sort on the current view. Sorting will automatically detect if you're trying to sort numbers (even if they're stored as strings) and will do the right thing.
- **`Collection.setFilter(filterFields, filterWords)`** - filter the current view. Filtering supports multiple words without any specific order, so you'll basically get a full-text search ability. Also, you can pass it only one field from the model, or you can pass an array with fields and all of them will get filtered. Last option is to pass it an object containing a comparison method and rules. Currently, only the `levenshtein` method is available.

```

this.collection.setFilter(
  {'Name': {cmp_method: 'levenshtein', max_distance: 7}}
  , 'Amreican P' // Note the switched 'r' and 'e', and the 'P' from 'Pie'
);

```

Also note that the `levenshtein` plug-in should be loaded and enabled using the `useLevenshteinPlugin` variable.

Last but not less important: Performing Levenshtein comparison returns the `distance` between two strings. It won't let you *search* lengthy text.

The Levenshtein distance between two strings is the number of characters that should be added, removed or moved to the left or to the right to transform one string into the other.

That means that comparing "Something" in "This is a test that could show something" will return 32, which is bigger than comparing "Something" and "ABCDEFGH" (9).

Use levenshtein only for short texts (titles, names, etc).

- **Collection.doFakeFilter(filterFields, filterWords)** - returns the models count after fake-applying a call to **Collection.setFilter**.
- **Collection.setFieldFilter(rules)** - filter each value of each model according to **rules** that you pass as argument. Example: You have a collection of books with 'release year' and 'author'. You can filter only the books that were released between 1999 and 2003. And then you can add another **rule** that will filter those books only to authors whose names start with 'A'. Possible rules: function, required, min, max, range, minLength, maxLength, rangeLength, oneOf, equalTo, pattern.

```
my_collection.setFieldFilter([
  {field: 'release_year', type: 'range', value: {min: '1999', max: '2003'}},
  {field: 'author', type: 'pattern', value: new RegExp('A*', 'igm')}
]);
```

```
//Rules:
//
//var my_var = 'green';
//
//{field: 'color', type: 'equalTo', value: my_var}
//{field: 'color', type: 'function', value: function(field_value){ return field_value == 'green'}}
//{field: 'color', type: 'required'}
//{field: 'number_of_colors', type: 'min', value: '2'}
//{field: 'number_of_colors', type: 'max', value: '4'}
//{field: 'number_of_colors', type: 'range', value: {min: '2', max: '4'}}
//{field: 'color_name', type: 'minLength', value: '4'}
//{field: 'color_name', type: 'maxLength', value: '6'}
//{field: 'color_name', type: 'rangeLength', value: {min: '4', max: '6'}}
//{field: 'color_name', type: 'oneOf', value: ['green', 'yellow']}
//{field: 'color_name', type: 'pattern', value: new RegExp('gre*', 'ig')}
```

- **Collection.doFakeFieldFilter(rules)** - returns the models count after fake-applying a call to **Collection.setFieldFilter**.

Implementation notes: You can use some variables in your **View** to represent the actual state of the paginator.

totalUnfilteredRecords - Contains the number of records, including all records filtered in any way. (Only available in **clientPager**)

totalRecords - Contains the number of records

currentPage - The actual page were the paginator is at.

perPage - The number of records the paginator will show per page.

totalPages - The number of total pages.

startRecord - The position of the first record shown in the current page (eg 41 to 50 from 2000 records) (Only available in **clientPager**)

endRecord - The position of the last record shown in the current page (eg 41 to 50 from 2000 records) (Only available in **clientPager**)

Plugins

Diacritic.js

A plugin for Backbone.Paginator that replaces diacritic characters (ă,ș,ț etc) with characters that match them most closely. This is particularly useful for filtering.

To enable the plugin, set **this.useDiacriticsPlugin** to true, as can be seen in the example below:

```
Paginator.clientPager = Backbone.Collection.extend({  
  
    // Default values used when sorting and/or filtering.  
    initialize: function(){  
        this.useDiacriticsPlugin = true; // use diacritics plugin if available  
        ...  
    }  
});
```

Backbone Boilerplate And Grunt-BBB

Boilerplates provide us a starting point for working on projects. They're a base for building upon using the minimum required code to get something functional put together. When you're working on a new Backbone application, a new Model typically only takes a few lines of code to get working.

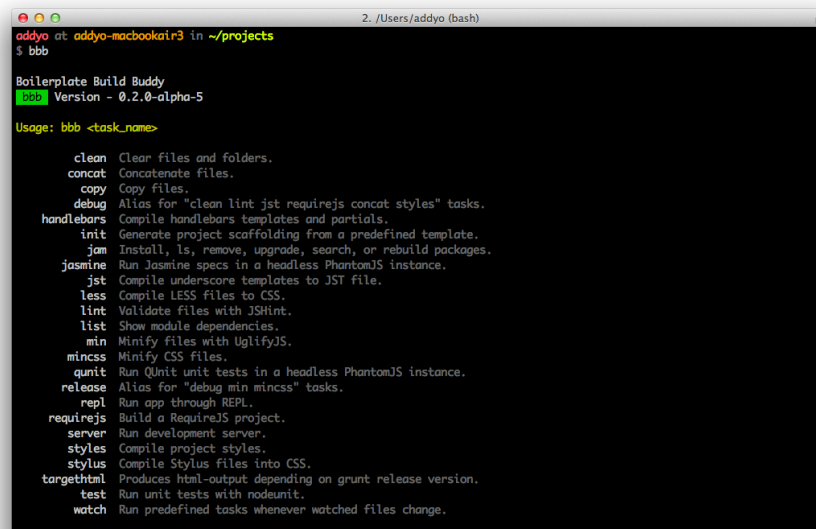
That alone probably isn't enough however, as you'll need a **Collection** to group those models, a **View** to render them and perhaps a **router** if you're looking to making specific views of your **Collection** data bookmarkable. If you're starting

on a completely fresh project, you may also need a build process in place to produce an optimized version of your app that can be pushed to production.

This is where boilerplate solutions are useful. Rather than having to manually write out the initial code for each piece of your Backbone app, a boilerplate could do this for you, also ideally taking care of the build process.

[Backbone Boilerplate](#) (or just BB) provides just this. It is an excellent set of best practices and utilities for building Backbone.js applications, created by Backbone contributor [Tim Branyen](#). He took the the gotchas, pitfalls and common tasks he ran into while heavily using Backbone to build apps and crafted BB as a result of this experience.

[Grunt-BBB or Boilerplate Build Buddy](#) is the companion tool to BB, which offers scaffolding, file watcher and build capabilities. Used together with BB it provides an excellent base for quickly starting new Backbone applications.



```
addyo at addyo-macbookair3 in ~/projects
$ bbb

Boilerplate Build Buddy
bbb Version - 0.2.0-alpha-5

Usage: bbb <task_name>

clean  Clear files and folders.
concat Concatenate files.
copy   Copy files.
debug  Alias for "clean lint jst requirejs concat styles" tasks.
handlebars Compile handlebars templates and partials.
init   Generate project scaffolding from a predefined template.
jam    Install, ls, remove, upgrade, search, or rebuild packages.
jasmine Run Jasmine specs in a headless PhantomJS instance.
jst     Compile underscore templates to JST file.
less    Compile LESS files to CSS.
lint    Validate files with JSHint.
list    Show module dependencies.
min     Minify files with UglifyJS.
mincss  Minify CSS files.
qunit   Run QUnit unit tests in a headless PhantomJS instance.
release Alias for "debug min mincss" tasks.
repl    Run app through REPL.
requirejs Build a RequireJS project.
server  Run development server.
styles  Compile project styles.
stylus  Compile Stylus files into CSS.
targethtml Produces html-output depending on grunt release version.
test    Run unit tests with nodeunit.
watch   Run predefined tasks whenever watched files change.
```

Out of the box, BB and Grunt-BBB provide provide us with:

- Backbone, [Lodash](#) (an [Underscore.js](#) alternative) and [jQuery](#) with an [HTML5 Boilerplate](#) foundation
- Boilerplate and scaffolding support, allowing us to spend minimal time writing boilerplate for modules, collections and so on.
- A build tool for template pre-compilation and, concatenation & minification of all our libraries, application code and stylesheets
- A Lightweight node.js webserver

Notes on build tool steps:

- Template pre-compilation: using a template library such as Underscore micro-templating or Handlebars.js generally involves three steps: (1) reading a raw template, (2) compiling it into a JavaScript function and (3) running the compiled template with your desired data. Precompiling eliminates the second step from runtime, by moving this process into a build step.
- Concatenation is the process of combining a number of assets (in our case, script files) into a single (or fewer number) of files to reduce the number of HTTP requests required to obtain them.
- Minification is the process of removing unnecessary characters (e.g white space, new lines, comments) from code and compressing it to reduce the overall size of the scripts being served.

Getting Started

Backbone Boilerplate and Grunt-BBB

To get started we're going to install Grunt-BBB, which will include Backbone Boilerplate and any third-party dependencies it might need such as the Grunt build tool.

We can install Grunt-bbb via NPM by running:

```
npm install -g bbb
```

That's it. We should now be good to go.

A typical workflow for using grunt-bbb, which we will use later on is:

- Initialize a new project (**bbb init**)
- Add new modules and templates (**bbb init:module**)
- Preview changes using the built in server (**bbb server**)
- Run the build tool (**bbb build**)
- Lint JavaScript, compile templates, build your application using r.js, minify CSS and JavaScript (using **bbb release**)

Creating a new project

Let's create a new directory for our project and run **bbb init** to kick things off. A number of project sub-directories and files will be stubbed out for us, as shown below:

```
$ bbb init
```


Running "init" task

This task will create one or more files in the current directory, based on the environment and the answers to a few questions. Note that answering "?" to any question will show question-specific help and answering "none" to most questions will leave its value blank.

"bbb" template notes:

This tool will help you install, configure, build, and maintain your Backbone Boilerplate project.

Writing app/app.js...OK

Writing app/config.js...OK

Writing app/main.js...OK

Writing app/router.js...OK

Writing app/styles/index.css...OK

Writing favicon.ico...OK

Writing grunt.js...OK

Writing index.html...OK

Writing package.json...OK

Writing readme.md...OK

Writing test/jasmine/index.html...OK

Writing test/jasmine/spec/example.js...OK

Writing test/jasmine/vendor/jasmine-html.js...OK

Writing test/jasmine/vendor/jasmine.css...OK

Writing test/jasmine/vendor/jasmine.js...OK

Writing test/jasmine/vendor/jasmine_favicon.png...OK

Writing test/jasmine/vendor/MIT.LICENSE...OK

Writing test/qunit/index.html...OK

Writing test/qunit/tests/example.js...OK

Writing test/qunit/vendor/qunit.css...OK

Writing test/qunit/vendor/qunit.js...OK

Writing vendor/h5bp/css/main.css...OK

Writing vendor/h5bp/css/normalize.css...OK

Writing vendor/jam/backbone/backbone.js...OK

Writing vendor/jam/backbone/package.json...OK

Writing vendor/jam/backbone.layoutmanager/backbone.layoutmanager.js...OK

Writing vendor/jam/backbone.layoutmanager/package.json...OK

Writing vendor/jam/jquery/jquery.js...OK

Writing vendor/jam/jquery/package.json...OK

Writing vendor/jam/lodash/lodash.js...OK

Writing vendor/jam/lodash/lodash.min.js...OK

Writing vendor/jam/lodash/lodash.underscore.min.js...OK

Writing vendor/jam/lodash/package.json...OK

Writing vendor/jam/require.config.js...OK

Writing vendor/jam/require.js...OK

Writing vendor/js/libs/almond.js...OK

Writing vendor/js/libs/require.js...OK

Initialized from template "bbb".

Done, without errors.

Let's review what has been generated.

index.html

This is a fairly standard stripped-down HTML5 Boilerplate foundation with the notable exception of including [RequireJS](#) at the bottom of the page.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <meta name="viewport" content="width=device-width,initial-scale=1">

  <title>Backbone Boilerplate</title>

  <!-- Application styles. -->
  <!--(if target dummy)><!-->
  <link rel="stylesheet" href="/app/styles/index.css">
  <!--!(endif)-->
</head>
<body>
  <!-- Application container. -->
  <main role="main" id="main"></main>

  <!-- Application source. -->
  <!--(if target dummy)><!-->
  <script data-main="/app/config" src="/vendor/js/libs/require.js"></script>
  <!--!(endif)-->

</body>
</html>
```

RequireJS - the [AMD](#) (Asynchronous Module Definition) module and script loader - will assist us with managing the modules in our application. We've already covered it in the last chapter, but let's recap what this particular block does in terms of the Boilerplate:

```
<script data-main="/app/config" src="/vendor/js/libs/require.js"></script>
```

The `data-main` attribute is used to inform RequireJS to load `app/config.js` (a configuration object) after it has finished loading itself. You'll notice that we've omitted the `.js` extension here as RequireJS can automatically add this for us, however it will respect your paths if we do choose to include it regardless. Let's now look at the config file being referenced.

config.js

A RequireJS configuration object allows us to specify aliases and paths for dependencies we're likely to reference often (e.g., jQuery), bootstrap properties like our base application URL, and `shim` libraries that don't support AMD natively.

This is what the config file in Backbone Boilerplate looks like:

```
// Set the require.js configuration for your application.
require.config({

    // Initialize the application with the main application file and the JamJS
    // generated configuration file.
    deps: ["../vendor/jam/require.config", "main"],

    paths: {
        // Put paths here.
    },

    shim: {
        // Put shims here.
    }

});
```

The first option defined in the above config is `deps: ["../vendor/jam/require.config", "main"]`. This informs RequireJS to load up additional RequireJS configuration as well as a `main.js` file, which is considered the entry point for our application.

You may notice that we haven't specified any other path information for `main`. Require will infer the default `baseUrl` using the path from our `data-main` attribute in `index.html`. In other words, our `baseUrl` is `app/` and any scripts we require will be loaded relative to this location. We could use the `baseUrl` option to override this default if we wanted to use a different location.

The next block is `paths`, which we can use to specify paths relative to the `baseUrl` as well as the paths/aliases to dependencies we're likely to regularly reference.

After this comes **shim**, an important part of our RequireJS configuration which allows us to load libraries which are not AMD compliant. The basic idea here is that rather than requiring all libraries to implement support for AMD, the **shim** takes care of the hard work for us.

Going back to **deps**, the contents of our **require.config** file can be seen below.

```
var jam = {
  "packages": [
    {
      "name": "backbone",
      "location": "../vendor/jam/backbone",
      "main": "backbone.js"
    },
    {
      "name": "backbone.layoutmanager",
      "location": "../vendor/jam/backbone.layoutmanager",
      "main": "backbone.layoutmanager.js"
    },
    {
      "name": "jquery",
      "location": "../vendor/jam/jquery",
      "main": "jquery.js"
    },
    {
      "name": "lodash",
      "location": "../vendor/jam/lodash",
      "main": "../lodash.js"
    }
  ],
  "version": "0.2.11",
  "shim": {
    "backbone": {
      "deps": [
        "jquery",
        "lodash"
      ],
      "exports": "Backbone"
    },
    "backbone.layoutmanager": {
      "deps": [
        "jquery",
        "backbone",
        "lodash"
      ],
      "exports": "Backbone.LayoutManager"
    }
  }
}
```

```

    }
  }
};

```

The `jam` object is to support configuration of [Jam](#) - a package manager for the front-end which helps instal, upgrade and configurate the dependencies used by your project. It is currently the package manager of choice for Backbone Boilerplate.

Under the `packages` array, a number of dependencies are specified for inclusion, such as Backbone, the Backbone.LayoutManager plugin, jQuery and Lo-dash.

For those curious about [Backbone.LayoutManager](#), it's a Backbone plugin that provides a foundation for assembling layouts and views within Backbone.

Additional packages you install using Jam will have a corresponding entry added to `packages`.

main.js

Next, we have `main.js`, which defines the entry point for our application. We use a global `require()` method to load an array containing any other scripts needed, such as our application `app.js` and our main router `router.js`. Note that most of the time, we will only use `require()` for bootstrapping an application and a similar method called `define()` for all other purposes.

The function defined after our array of dependencies is a callback which doesn't fire until these scripts have loaded. Notice how we're able to locally alias references to "app" and "router" as `app` and `Router` for convenience.

```

require([
  // Application.
  "app",

  // Main Router.
  "router"
],

function(app, Router) {

  // Define your master router on the application namespace and trigger all
  // navigation from this instance.
  app.router = new Router();

  // Trigger the initial route and enable HTML5 History API support, set the
  // root folder to '/' by default. Change in app.js.
  Backbone.history.start({ pushState: true, root: app.root });

```

```

// All navigation that is relative should be passed through the navigate
// method, to be processed by the router. If the link has a 'data-bypass'
// attribute, bypass the delegation completely.
$(document).on("click", "a[href]:not([data-bypass])", function(evt) {
  // Get the absolute anchor href.
  var href = { prop: $(this).prop("href"), attr: $(this).attr("href") };
  // Get the absolute root.
  var root = location.protocol + "://" + location.host + app.root;

  // Ensure the root is part of the anchor href, meaning it's relative.
  if (href.prop.slice(0, root.length) === root) {
    // Stop the default event to ensure the link will not cause a page
    // refresh.
    evt.preventDefault();

    // 'Backbone.history.navigate' is sufficient for all Routers and will
    // trigger the correct events. The Router's internal 'navigate' method
    // calls this anyways. The fragment is sliced from the root.
    Backbone.history.navigate(href.attr, true);
  }
});

});

```

Inline, Backbone Boilerplate includes boilerplate code for initializing our router with HTML5 History API support and handling other navigation scenarios, so we don't have to.

app.js

Let us now look at our `app.js` module. Typically, in non-Backbone Boilerplate applications, an `app.js` file may contain the core logic or module references needed to kick start an app.

In this case however, this file is used to define templating and layout configuration options as well as utilities for consuming layouts. To a beginner, this might look like a lot of code to comprehend, but the good news is that for basic apps, you're unlikely to need to heavily modify this. Instead, you'll be more concerned with modules for your app, which we'll look at next.

```

define([
  "backbone.layoutmanager"
], function() {

```

```

// Provide a global location to place configuration settings and module
// creation.
var app = {
  // The root path to run the application.
  root: "/"
};

// Localize or create a new JavaScript Template object.
var JST = window.JST = window.JST || {};

// Configure LayoutManager with Backbone Boilerplate defaults.
Backbone.LayoutManager.configure({
  // Allow LayoutManager to augment Backbone.View.prototype.
  manage: true,

  prefix: "app/templates/",

  fetch: function(path) {
    // Concatenate the file extension.
    path = path + ".html";

    // If cached, use the compiled template.
    if (JST[path]) {
      return JST[path];
    }

    // Put fetch into 'async-mode'.
    var done = this.async();

    // Seek out the template asynchronously.
    $.get(app.root + path, function(contents) {
      done(JST[path] = _.template(contents));
    });
  }
});

// Mix Backbone.Events, modules, and layout management into the app object.
return _.extend(app, {
  // Create a custom object with a nested Views object.
  module: function(additionalProps) {
    return _.extend({ Views: {} }, additionalProps);
  },

  // Helper for using layouts.
  useLayout: function(name, options) {
    // Enable variable arity by allowing the first argument to be the options

```

```

    // object and omitting the name argument.
    if (_.isObject(name)) {
        options = name;
    }

    // Ensure options is an object.
    options = options || {};

    // If a name property was specified use that as the template.
    if (_.isString(name)) {
        options.template = name;
    }

    // Create a new Layout with options.
    var layout = new Backbone.Layout(_.extend({
        el: "#main"
    }, options));

    // Cache the reference.
    return this.layout = layout;
}
}, Backbone.Events);
});

```

Note: JST stands for JavaScript templates and generally refers to templates which have been (or will be) precompiled as part of a build step. When running `bbb release` or `bbb debug`, Underscore/Lo-dash templates will be precompiled to avoid the need to compile them at runtime within the browser.

Creating Backbone Boilerplate Modules

Not to be confused with simply being just an AMD module, a Backbone Boilerplate module is a script composed of a:

- Model
- Collection
- Views (optional)

We can easily create a new Boilerplate module using `grunt-bbb` once again using `init`:

```

# Create a new module
$ bbb init:module

```



```
# Grunt prompt
Please answer the following:
[?] Module Name foo
[?] Do you need to make any changes to the above before continuing? (y/N)

Writing app/modules/foo.js...OK
Writing app/styles/foo.styl...OK
Writing app/templates/foo.html...OK

Initialized from template "module".

Done, without errors.
```

This will generate a module foo.js as follows:

```
// Foo module
define([
  // Application.
  "app"
],

// Map dependencies from above array.
function(app) {

  // Create a new module.
  var Foo = app.module();

  // Default Model.
  Foo.Model = Backbone.Model.extend({

  });

  // Default Collection.
  Foo.Collection = Backbone.Collection.extend({
    model: Foo.Model
  });

  // Default View.
  Foo.Views.Layout = Backbone.Layout.extend({
    template: "foo"
  });

  // Return the module for AMD compliance.
  return Foo;
```

```
});
```

Notice how boilerplate code for a model, collection and view have been scaffolded out for us.

Optionally, we may also wish to include references to plugins such as the Backbone LocalStorage or Offline adapters. One clean way of including a plugin in the above boilerplate could be:

```
// Foo module
define([
  // Application.
  "app",
  // Plugins
  'plugins/backbone-localstorage'
],

// Map dependencies from above array.
function(app) {

  // Create a new module.
  var Foo = app.module();

  // Default Model.
  Foo.Model = Backbone.Model.extend({
    // Save all of the items under the "foo" namespace.
    localStorage: new Store('foo-backbone'),
  });

  // Default Collection.
  Foo.Collection = Backbone.Collection.extend({
    model: Foo.Model
  });

  // Default View.
  Foo.Views.Layout = Backbone.Layout.extend({
    template: "foo"
  });

  // Return the module for AMD compliance.
  return Foo;

});
```

router.js

Finally, let's look at our application router which is used for handling navigation. The default router Backbone Boilerplate generates for us includes sane defaults out of the box and can be easily extended.

```
define([
  // Application.
  "app"
],

function(app) {

  // Defining the application router, you can attach sub routers here.
  var Router = Backbone.Router.extend({
    routes: {
      "": "index"
    },

    index: function() {

    }
  });

  return Router;
});
```

If however we would like to execute some module-specific logic, when the page loads (i.e when a user hits the default route), we can pull in a module as a dependency and optionally use the Backbone LayoutManager to attach Views to our layout as follows:

```
define([
  // Application.
  'app',

  // Modules
  'modules/foo'
],

function(app, Foo) {

  // Defining the application router, you can attach sub routers here.
  var Router = Backbone.Router.extend({
```

```

    routes: {
      '': 'index'
    },

    index: function() {
      // Create a new Collection
      var collection = new Foo.Collection();

      // Use and configure a 'main' layout
      app.useLayout('main').setViews({
        // Attach the bar View into the content View
        '.bar': new Foo.Views.Bar({
          collection: collection
        })
      }).render();
    }
  });

  // Fetch data (e.g., from localStorage)
  collection.fetch();

  return Router;
});

```

Related Tools & Projects

As we've seen, scaffolding tools can assist in expediting how quickly you can begin a new application by creating the basic files required for a project automatically.

If you appreciated Grunt-BBB and would like to explore similar tools for the broader app development workflow, I'm happy to also recommend checking out the [Yeoman](#) and workflow [Brunch](#).

Both projects offer application scaffolding, a file-watcher and build system however Yeoman achieves the latter through [Grunt](#) and also helps with client-side package management via [Bower](#).

Conclusions

In this section we reviewed Backbone Boilerplate and learned how to use the `bbb` tool to help us scaffold out our application.

If you would like to learn more about how this project helps structure your app, BBB includes some built-in boilerplate sample apps that can be easily generated for review.

These include a boilerplate tutorial project (`bbb init:tutorial`) and an implementation of my [TodoMVC](#) project (`bbb init:todomvc`). I recommend checking these out as they'll provide you with a more complete picture of how Backbone Boilerplate, its templates, and so on fit into the overall setup for a web app.

For more about Grunt-BBB, remember to take a look at the official project [repository](#). There is also a related [slide-deck](#) available for those interested in reading more.

Mobile Applications

Backbone & jQuery Mobile

Resolving the routing conflicts

The first major hurdle developers typically run into when building Backbone applications with jQuery Mobile is that both frameworks have their own opinions about how to handle application navigation.

Backbone's routers offer an explicit way to define custom navigation routes through `Backbone.Router`, whilst jQuery Mobile encourages the use of URL hash fragments to reference separate 'pages' or views in the same document. jQuery Mobile also supports automatically pulling in external content for links through XHR calls meaning that there can be quite a lot of inter-framework confusion about what a link pointing at `#photo/id` should actually be doing.

Some of the solutions that have been previously proposed to work-around this problem included manually patching Backbone or jQuery Mobile. I discourage opting for these techniques as it becomes necessary to manually patch your framework builds when new releases get made upstream.

There's also [jQueryMobile router](#), which tries to solve this problem differently, however I think my proposed solution is both simpler and allows both frameworks to cohabit quite peacefully without the need to extend either. What we're after is a way to prevent one framework from listening to hash changes so that we can fully rely on the other (e.g. `Backbone.Router`) to handle this for us exclusively.

Using jQuery Mobile this can be done by setting:

```
$.mobile.hashListeningEnabled = false;
```

prior to initializing any of your other code.

I discovered this method looking through some jQuery Mobile commits that didn't make their way into the official docs, but am happy to see that they are now [covered in more detail](#).

The next question that arises is, if we're preventing jQuery Mobile from listening to URL hash changes, how can we still get the benefit of being able to navigate to other sections in a document using the built-in transitions and effects supported? Good question. This can be solved by simply calling `$.mobile.changePage()` as follows:

```
var url = '#about',
    effect = 'slideup',
    reverse = false,
    changeHash = false;

$.mobile.changePage( url , { transition: effect}, reverse, changeHash );
```

In the above sample, `url` can refer to a URL or a hash identifier to navigate to, `effect` is simply the transition effect to animate the page in with and the final two parameters decide the direction for the transition (`reverse`) and whether or not the hash in the address bar should be updated (`changeHash`). With respect to the latter, I typically set this to false to avoid managing two sources for hash updates, but feel free to set this to true if you're comfortable doing so.

Note: For some parallel work being done to explore how well the jQuery Mobile Router plugin works with Backbone, you may be interested in checking out <https://github.com/Filirom1/jquery-mobile-backbone-requirejs>.

Exercise: A Backbone, Require.js/AMD app with jQuery Mobile

Note: The code for this exercise can be found in `practicals/modular-mobile-app`.

Getting started

Once you feel comfortable with the [Backbone fundamentals](#) and you've put together a rough wireframe of the app you may wish to build, start to think about your application architecture. Ideally, you'll want to logically separate concerns so that it's as easy as possible to maintain the app in the future.

Namespacing

For this application, I opted for the nested namespacing pattern. Implemented correctly, this enables you to clearly identify if items being referenced in your app are views, other modules and so on. This initial structure is a sane place to also include application defaults (unless you prefer maintaining those in a separate file).

```
window.mobileSearch = window.mobileSearch || {
  views: {
```

```

        appview: new AppView()
    },
    routers:{
        workspace:new Workspace()
    },
    utils: utils,
    defaults: {
        resultsPerPage: 16,
        safeSearch: 2,
        maxDate:'',
        minDate:'01/01/1970'
    }
}

```

Models

In the Flickr application, there are at least two unique types of data that need to be modeled - search results and individual photos, both of which contain additional meta-data like photo titles. If you simplify this down, search results are actually groups of photos in their own right, so the application only requires:

- A single model (a photo or ‘result’ entry)
- A result collection (containing a group of result entries) for search results
- A photo collection (containing one or more result entries) for individual photos or photos with more than one image

Views

The views we’ll need include an application view, a search results view and a photo view. Static views or pages of the single-page application which do not require a dynamic element for them (e.g an ‘about’ page) can be easily coded up in your document’s markup, independent of Backbone.

Routers

A number of possible routes need to be taken into consideration:

- Basic search queries `#search/kiwis`
- Search queries with additional parameters (e.g sort, pagination) `#search/kiwis/srelevance/p7`
- Queries for specific photos `#photo/93839`
- A default route (no parameters passed)

This tutorial will be expanded shortly to fully cover the demo application. In the mean time, please see the practicals folder for the completed application that demonstrates the router resolution discussed earlier between Backbone and jQuery Mobile.

jQuery Mobile: Going beyond mobile application development

The majority of jQM apps I've seen in production have been developed for the purpose of providing an optimal experience to users on mobile devices. Given that the framework was developed for this purpose, there's nothing fundamentally wrong with this, but many developers forget that jQM is a UI framework not dissimilar to jQuery UI. It's using the widget factory and is capable of being used for a lot more than we give it credit for.

If you open up Flickr in a desktop browser, you'll get an image search UI that's modeled on Google.com, however, review the components (buttons, text inputs, tabs) on the page for a moment. The desktop UI doesn't look anything like a mobile application yet I'm still using jQM for theming mobile components; the tabs, date-picker, sliders - everything in the desktop UI is re-using what jQM would be providing users on mobile devices. Thanks to some media queries, the desktop UI can make optimal use of whitespace, expanding component blocks out and providing alternative layouts whilst still making use of jQM as a component framework.

The benefit of this is that I don't need to go pulling in jQuery UI separately to be able to take advantage of these features. Thanks to the recent ThemeRoller my components can look pretty much exactly how I would like them to and users of the app can get a jQM UI for lower-resolutions and a jQM-ish UI for everything else.

The takeaway here is just to remember that if you're not (already) going through the hassle of conditional script/style loading based on screen-resolution (using `matchMedia.js` etc), there are simpler approaches that can be taken to cross-device component theming.

Unit Testing

One definition of unit testing is the process of taking the smallest piece of testable code in an application, isolating it from the remainder of your codebase, and determining if it behaves exactly as expected.

For an application to be considered 'well-tested', each function should ideally have its own separate unit tests where it's tested against the different conditions you expect it to handle. All tests must pass before functionality is considered 'complete'. This allows developers to both modify a unit of code and its dependencies with a level of confidence about whether these changes have caused any breakage.

A basic example of unit testing is where a developer asserts that passing specific values to a sum function results in the correct value being returned. For an example more relevant to this book, we may wish to assert that adding a new Todo item to a list correctly adds a Model of a specific type to a Todos Collection.

When building modern web-applications, it's typically considered best-practice to include automated unit testing as a part of your development process. In the following chapters we are going to look at three different solutions for unit testing your Backbone.js apps - Jasmine, QUnit and SinonJS.

Jasmine

Behavior-Driven Development

In this section, we'll be taking a look at how to unit test Backbone applications using a popular JavaScript testing framework called [Jasmine](#) from Pivotal Labs.

Jasmine describes itself as a behavior-driven development (BDD) framework for testing JavaScript code. Before we jump into how the framework works, it's useful to understand exactly what [BDD](#) is.

BDD is a second-generation testing approach first described by [Dan North](#) (the authority on BDD) which attempts to test the behavior of software. It's considered second-generation as it came out of merging ideas from Domain driven design (DDD) and lean software development. BDD helps teams deliver high-quality software by answering many of the more confusing questions early on in the agile process. Such questions commonly include those concerning documentation and testing.

If you were to read a book on BDD, it's likely that it would be described as being 'outside-in and pull-based'. The reason for this is that it borrows the idea of 'pulling features' from Lean manufacturing which effectively ensures that the right software solutions are being written by a) focusing on the expected outputs of the system and b) ensuring these outputs are achieved.

BDD recognizes that there are usually multiple stakeholders in a project and not a single amorphous user of the system. These different groups will be affected by the software being written in differing ways and will have varying opinions of what quality in the system means to them. It's for this reason that it's important to understand who the software will be bringing value to and exactly what in it will be valuable to them.

Finally, BDD relies on automation. Once you've defined the quality expected, your team will want to check on the functionality of the solution being built regularly and compare it to the results they expect. In order to facilitate this efficiently, the process has to be automated. BDD relies heavily on the automation of specification-testing and Jasmine is a tool which can assist with this.

BDD helps both developers and non-technical stakeholders:

- Better understand and represent the models of the problems being solved

- Explain supported test cases in a language that non-developers can read
- Focus on minimizing translation of the technical code being written and the domain language spoken by the business

What this means is that developers should be able to show Jasmine unit tests to a project stakeholder and (at a high level, thanks to a common vocabulary being used) they'll ideally be able to understand what the code supports.

Developers often implement BDD in unison with another testing paradigm known as [TDD](#) (test-driven development). The main idea behind TDD is using the following development process:

1. Write unit tests which describe the functionality you would like your code to support
2. Watch these tests fail (as the code to support them hasn't yet been written)
3. Write code to make the tests pass
4. Rinse, repeat, and refactor

In this chapter we're going to use BDD (with TDD) to write unit tests for a Backbone application.

Note: I've seen a lot of developers also opt for writing tests to validate behavior of their code after having written it. While this is fine, note that it can come with pitfalls such as only testing for behavior your code currently supports, rather than the behavior needed to fully solve the problem.

Suites, Specs, & Spies

When using Jasmine, you'll be writing suites and specifications (specs). Suites basically describe scenarios while specs describe what can be done in these scenarios.

Each spec is a JavaScript function, described with a call to `it()` using a description string and a function. The description should describe the behaviour the particular unit of code should exhibit and, keeping in mind BDD, it should ideally be meaningful. Here's an example of a basic spec:

```
it('should be incrementing in value', function(){
  var counter = 0;
  counter++;
});
```

On its own, a spec isn't particularly useful until expectations are set about the behavior of the code. Expectations in specs are defined using the `expect()` function and an [expectation matcher](#) (e.g., `toEqual()`, `toBeTruthy()`, `toContain()`). A revised example using an expectation matcher would look like:

```
it('should be incrementing in value', function(){
    var counter = 0;
    counter++;
    expect(counter).toEqual(1);
});
```

The above code passes our behavioral expectation as `counter` equals 1. Notice how easy it was to read the expectation on the last line (you probably grokked it without any explanation).

Specs are grouped into suites which we describe using Jasmine's `describe()` function, again passing a string as a description and a function as we did for `it()`. The name/description for your suite is typically that of the component or module you're testing.

Jasmine will use the description as the group name when it reports the results of the specs you've asked it to run. A simple suite containing our sample spec could look like:

```
describe('Stats', function(){
    it('can increment a number', function(){
        ...
    });

    it('can subtract a number', function(){
        ...
    });
});
```

Suites also share a functional scope, so it's possible to declare variables and functions inside a describe block which are accessible within specs:

```
describe('Stats', function(){
    var counter = 1;

    it('can increment a number', function(){
        // the counter was = 1
        counter = counter + 1;
        expect(counter).toEqual(2);
    });

    it('can subtract a number', function(){
        // the counter was = 2
        counter = counter - 1;
        expect(counter).toEqual(1);
    });
});
```

Note: Suites are executed in the order in which they are described, which can be useful to know if you would prefer to see test results for specific parts of your application reported first.

Jasmine also supports **spies** - a way to mock, spy, and fake behavior in our unit tests. Spies replace the function they're spying on, allowing us to simulate behavior we would like to mock (i.e., test without using the actual implementation).

In the example below, we're spying on the `setComplete` method of a dummy `Todo` function to test that arguments can be passed to it as expected.

```
var Todo = function(){
};

Todo.prototype.setComplete = function (arg){
  return arg;
}

describe('a simple spy', function(){
  it('should spy on an instance method of a Todo', function(){
    var myTodo = new Todo();
    spyOn(myTodo, 'setComplete');
    myTodo.setComplete('foo bar');

    expect(myTodo.setComplete).toHaveBeenCalledWith('foo bar');

    var myTodo2 = new Todo();
    spyOn(myTodo2, 'setComplete');

    expect(myTodo2.setComplete).not.toHaveBeenCalled();

  });
});
```

You are more likely to use spies for testing **asynchronous** behavior in your application such as AJAX requests. Jasmine supports:

- Writing tests which can mock AJAX requests using spies. This allows us to test both the code that initiates the AJAX request and the code executed upon its completion. It's also possible to mock/fake the server responses. The benefit of this type of testing is that it's faster as no real calls are being made to a server. The ability to simulate any response from the server is also of great benefit.
- Asynchronous tests which don't rely on spies

This example of the first kind of test shows how to fake an AJAX request and verify that the request was both calling the correct URL and executed a callback where one was provided.

```
it('the callback should be executed on success', function () {

    // 'andCallFake()' calls a passed function when a spy
    // has been called
    spyOn($, 'ajax').andCallFake(function(options) {
        options.success();
    });

    // Create a new spy
    var callback = jasmine.createSpy();

    // Execute the spy callback if the
    // request for Todo 15 is successful
    getTodo(15, callback);

    // Verify that the URL of the most recent call
    // matches our expected Todo item.
    expect($.ajax.mostRecentCall.args[0]['url']).toEqual('/todos/15');

    // 'expect(x).toHaveBeenCalled()' will pass if 'x' is a
    // spy and was called.
    expect(callback).toHaveBeenCalled();
});

function getTodo(id, callback) {
    $.ajax({
        type: 'GET',
        url: '/todos/' + id,
        dataType: 'json',
        success: callback
    });
}
```

All of these are Spy-specific matchers and are documented on the Jasmine [wiki](#).

For the second type of test (asynchronous tests), we can take the above further by taking advantage of three other methods Jasmine supports:

- [waits\(timeout\)](#) - a native timeout before the next block is run
- [waitsFor\(function, optional message, optional timeout\)](#) - a way to pause specs until some other work has completed. Jasmine waits until the supplied function returns true here before it moves on to the next block.

- `runs(function)` - a block which runs as if it was directly called. They exist so that we can test asynchronous processes.

```
it('should make an actual AJAX request to a server', function () {

    // Create a new spy
    var callback = jasmine.createSpy();

    // Execute the spy callback if the
    // request for Todo 16 is successful
    getTodo(16, callback);

    // Pause the spec until the callback count is
    // greater than 0
    waitsFor(function() {
        return callback.callCount > 0;
    });

    // Once the wait is complete, our runs() block
    // will check to ensure our spy callback has been
    // called
    runs(function() {
        expect(callback).toHaveBeenCalled();
    });
});

function getTodo(id, callback) {
    $.ajax({
        type: 'GET',
        url: 'todos.json',
        dataType: 'json',
        success: callback
    });
}
```

Note: It's useful to remember that when making real requests to a web server in your unit tests, this has the potential to massively slow down the speed at which tests run (due to many factors including server latency). As this also introduces an external dependency that can (and should) be minimized in your unit testing, it is strongly recommended that you opt for spies to remove the dependency on a web server.

beforeEach() and afterEach()

Jasmine also supports specifying code that can be run before each (`beforeEach()`) and after each (`afterEach()`) test. This is useful for enforcing consistent conditions (such as resetting variables that may be required by specs). In the following example, `beforeEach()` is used to create a new sample Todo model which specs can use for testing attributes.

```
beforeEach(function(){
  this.todo = new Backbone.Model({
    text: 'Buy some more groceries',
    done: false
  });
});

it('should contain a text value if not the default value', function(){
  expect(this.todo.get('text')).toEqual('Buy some more groceries');
});
```

Each nested `describe()` in your tests can have their own `beforeEach()` and `afterEach()` methods which support including setup and teardown methods relevant to a particular suite.

`beforeEach()` and `afterEach()` can be used together to write tests verifying that our Backbone routes are being correctly triggered when we navigate to the URL. We can start with the `index` action:

```
describe('Todo routes', function(){

  beforeEach(function(){

    // Create a new router
    this.router = new App.TODORouter();

    // Create a new spy
    this.routerSpy = sinon.spy();

    // Begin monitoring hashchange events
    try{
      Backbone.history.start({
        silent:true,
        pushState: true
      });
    }catch(e){
      // ...
    }
  });
});
```

```

    }

    // Navigate to a URL
    this.router.navigate('/js/spec/SpecRunner.html');
  });

  afterEach(function(){

    // Navigate back to the URL
    this.router.navigate('/js/spec/SpecRunner.html');

    // Disable Backbone.history temporarily.
    // Note that this is not really useful in real apps but is
    // good for testing routers
    Backbone.history.stop();
  });

  it('should call the index route correctly', function(){
    this.router.bind('route:index', this.routerSpy, this);
    this.router.navigate('', {trigger: true});

    // If everything in our beforeEach() and afterEach()
    // calls have been correctly executed, the following
    // should now pass.
    expect(this.routerSpy).toHaveBeenCalledOnce();
    expect(this.routerSpy).toHaveBeenCalled();
  });
});

```

The actual TodoRouter for that would make the above test pass looks like:

```

var App = App || {};
App.TodoRouter = Backbone.Router.extend({
  routes:{
    '': 'index'
  },
  index: function(){
    //...
  }
});

```


Shared scope

Let's imagine we have a Suite where we wish to check for the existence of a new Todo item instance. This could be done by duplicating the spec as follows:

```
describe("Todo tests", function(){

    // Spec
    it("Should be defined when we create it", function(){
        // A Todo item we are testing
        var todo = new Todo("Get the milk", "Tuesday");
        expect(todo).toBeDefined();
    });

    it("Should have the correct title", function(){
        // Where we introduce code duplication
        var todo = new Todo("Get the milk", "Tuesday");
        expect(todo.title).toBe("Get the milk");
    });

});
```

As you can see, we've introduced duplication that should ideally be refactored into something cleaner. We can do this using Jasmine's Suite (Shared) Functional Scope.

All of the specs within the same Suite share the same functional scope, meaning that variables declared within the Suite itself are available to all of the Specs in that suite. This gives us a way to work around our duplication problem by moving the creation of our Todo objects into the common functional scope:

```
describe("Todo tests", function(){

    // The instance of Todo, the object we wish to test
    // is now in the shared functional scope
    var todo = new Todo("Get the milk", "Tuesday");

    // Spec
    it("should be correctly defined", function(){
        expect(todo).toBeDefined();
    });

    it("should have the correct title", function(){
        expect(todo.title).toBe("Get the milk");
    });

});
```

```
});
```

In the previous section you may have noticed that we initially declared `this.todo` within the scope of our `beforeEach()` call and were then able to continue using this reference in `afterEach()`.

This is again down to shared function scope, which allows such declarations to be common to all blocks (including `runs()`).

Variables declared outside of the shared scope (i.e within the local scope `var todo=...`) will however not be shared.

Getting set up

Now that we've reviewed some fundamentals, let's go through downloading Jasmine and getting everything set up to write tests.

A standalone release of Jasmine can be [downloaded](#) from the official release page.

You'll need a file called `SpecRunner.html` in addition to the release. It can be downloaded from <https://github.com/pivotal/jasmine/tree/master/lib/jasmine-core/example> or as part of a download of the complete Jasmine [repo](#). Alternatively, you can [git clone](#) the main Jasmine repository from <https://github.com/pivotal/jasmine.git>.

Let's review [SpecRunner.html.jst](#):

It first includes both Jasmine and the necessary CSS required for reporting:

```
<link rel="stylesheet" type="text/css" href="lib/jasmine-<%= jasmineVersion %>/jasmine.css">
<script type="text/javascript" src="lib/jasmine-<%= jasmineVersion %>/jasmine.js"></script>
<script type="text/javascript" src="lib/jasmine-<%= jasmineVersion %>/jasmine-html.js"></script>
<script type="text/javascript" src="lib/jasmine-<%= jasmineVersion %>/boot.js"></script>
```

Next come the sources being tested:

```
<!-- include source files here... -->
<script type="text/javascript" src="src/Player.js"></script>
<script type="text/javascript" src="src/Song.js"></script>
```

Finally, some sample tests are included:

```
<!-- include spec files here... -->
<script type="text/javascript" src="spec/SpecHelper.js"></script>
<script type="text/javascript" src="spec/PlayerSpec.js"></script>
```

Note: Below this section of SpecRunner is code responsible for running the actual tests. Given that we won't be covering modifying this code, I'm going to skip reviewing it. I do however encourage you to take a look through [PlayerSpec.js](#) and [SpecHelper.js](#). They're a useful basic example to go through how a minimal set of tests might work.

Also note that for the purposes of introduction, some of the examples in this section will be testing aspects of Backbone.js itself, just to give you a feel for how Jasmine works. You generally will not need to write testing ensuring a framework is working as expected.

TDD With Backbone

When developing applications with Backbone, it can be necessary to test both individual modules of code as well as models, views, collections, and routers. Taking a TDD approach to testing, let's review some specs for testing these Backbone components using the popular Backbone [Todo](#) application.

Models

The complexity of Backbone models can vary greatly depending on what your application is trying to achieve. In the following example, we're going to test default values, attributes, state changes, and validation rules.

First, we begin our suite for model testing using `describe()`:

```
describe('Tests for Todo', function() {
```

Models should ideally have default values for attributes. This helps ensure that when creating instances without a value set for any specific attribute, a default one (e.g., an empty string) is used instead. The idea here is to allow your application to interact with models without any unexpected behavior.

In the following spec, we create a new `Todo` without any attributes passed then check to find out what the value of the `text` attribute is. As no value has been set, we expect a default value of `"` to be returned.

```
it('Can be created with default values for its attributes.', function() {  
    var todo = new Todo();  
    expect(todo.get('text')).toBe('');  
});
```

If testing this spec before your models have been written, you'll incur a failing test, as expected. What's required for the spec to pass is a default value for the attribute `text`. We can set this and some other useful defaults (which we'll be using shortly) in our `Todo` model as follows:

```

window.Todo = Backbone.Model.extend({

  defaults: {
    text: '',
    done: false,
    order: 0
  }
});

```

Next, it is common to include validation logic in your models to ensure that input passed from users or other modules in the application are valid.

A Todo app may wish to validate the text input supplied in case it contains rude words. Similarly if we're storing the `done` state of a Todo item using booleans, we need to validate that truthy/falsy values are passed and not just any arbitrary string.

In the following spec, we take advantage of the fact that validations which fail `model.validate()` trigger an “invalid” event. This allows us to test if validations are correctly failing when invalid input is supplied.

We create an errorCallback spy using Jasmine's built in `createSpy()` method which allows us to spy on the invalid event as follows:

```

it('Can contain custom validation rules, and will trigger an invalid event on failed validation', function() {

  var errorCallback = jasmine.createSpy('-invalid event callback-');

  var todo = new Todo();

  todo.on('invalid', errorCallback);

  // What would you need to set on the todo properties to
  // cause validation to fail?

  todo.set({done: 'a non-boolean value'});

  var errorArgs = errorCallback.mostRecentCall.args;

  expect(errorArgs).toBeDefined();
  expect(errorArgs[0]).toBe(todo);
  expect(errorArgs[1]).toBe('Todo.done must be a boolean value.');
```

The code to make the above failing test support validation is relatively simple. In our model, we override the `validate()` method (as recommended in the Backbone docs), checking to make sure a model both has a 'done' property and that its value is a valid boolean before allowing it to pass.

```

validate: function(attrs) {
  if (attrs.hasOwnProperty('done') && !_.isBoolean(attrs.done)) {
    return 'Todo.done must be a boolean value.';
  }
}

```

If you would like to review the final code for our Todo model, you can find it below:

```

window.TODO = Backbone.Model.extend({

  defaults: {
    text: '',
    done: false,
    order: 0
  },

  initialize: function() {
    this.set({text: this.get('text')}, {silent: true});
  },

  validate: function(attrs) {
    if (attrs.hasOwnProperty('done') && !_.isBoolean(attrs.done)) {
      return 'Todo.done must be a boolean value.';
    }
  },

  toggle: function() {
    this.save({done: !this.get('done')});
  }

});

```

Collections

We now need to define specs to test a Backbone collection of Todo models (a `TodoList`). Collections are responsible for a number of list tasks including managing order and filtering.

A few specific specs that come to mind when working with collections are:

- Making sure we can add new Todo models as both objects and arrays
- Attribute testing to make sure attributes such as the base URL of the collection are values we expect

- Purposefully adding items with a status of `done:true` and checking against how many items the collection thinks have been completed vs. those that are remaining

In this section we're going to cover the first two of these with the third left as an extended exercise you can try on your own.

Testing that Todo models can be added to a collection as objects or arrays is relatively trivial. First, we initialize a new `TodoList` collection and check to make sure its length (i.e., the number of Todo models it contains) is 0. Next, we add new Todos, both as objects and arrays, checking the length property of the collection at each stage to ensure the overall count is what we expect:

```
describe('Tests for TodoList', function() {

  it('Can add Model instances as objects and arrays.', function() {
    var todos = new TodoList();

    expect(todos.length).toBe(0);

    todos.add({ text: 'Clean the kitchen' });

    // how many todos have been added so far?
    expect(todos.length).toBe(1);

    todos.add([
      { text: 'Do the laundry', done: true },
      { text: 'Go to the gym' }
    ]);

    // how many are there in total now?
    expect(todos.length).toBe(3);
  });
  ...
});
```

Similar to model attributes, it's also quite straight-forward to test attributes in collections. Here we have a spec that ensures the collection url (i.e., the url reference to the collection's location on the server) is what we expect it to be:

```
it('Can have a url property to define the basic url structure for all contained models.', function() {
  var todos = new TodoList();

  // what has been specified as the url base in our model?
  expect(todos.url).toBe('/todos/');
});
```

For the third spec (which you will write as an exercise), note that the implementation for our collection will have methods for filtering how many Todo items are done and how many are remaining - we'll call these `done()` and `remaining()`. Consider writing a spec which creates a new collection and adds one new model that has a preset `done` state of `true` and two others that have the default `done` state of `false`. Testing the length of what's returned using `done()` and `remaining()` will tell us whether the state management in our application is working or needs a little tweaking.

The final implementation for our `TodoList` collection can be found below:

```
window.TodoList = Backbone.Collection.extend({

  model: Todo,

  url: '/todos/',

  done: function() {
    return this.filter(function(todo) { return todo.get('done'); });
  },

  remaining: function() {
    return this.without.apply(this, this.done());
  },

  nextOrder: function() {
    if (!this.length) {
      return 1;
    }

    return this.last().get('order') + 1;
  },

  comparator: function(todo) {
    return todo.get('order');
  }

});
```

Views

Before we take a look at testing Backbone views, let's briefly review a jQuery plugin that can assist with writing Jasmine specs for them.

The Jasmine jQuery Plugin

As we know our Todo application will be using jQuery for DOM manipulation, there's a useful jQuery plugin called [jasmine-jquery](#) we can use to help simplify BDD testing of the rendering performed by our views.

The plugin provides a number of additional Jasmine [matchers](#) to help test jQuery-wrapped sets such as:

- `toBe(jQuerySelector)` e.g., `expect($('<div id="some-id"></div>')).toBe('div#some-id')`
- `toBeChecked()` e.g., `expect($('<input type="checkbox" checked="checked"/>')).toBeChecked()`
- `toBeSelected()` e.g., `expect($('<option selected="selected"></option>')).toBeSelected()`

and [many others](#). The complete list of matchers supported can be found on the project homepage. It's useful to know that similar to the standard Jasmine matchers, the custom matchers above can be inverted using the `.not` prefix (i.e. `expect(x).not.toBe(y)`):

```
expect($('<div>I am an example</div>')).not.toHaveText(/other/)
```

`jasmine-jquery` also includes a fixtures module that can be used to load arbitrary HTML content we wish to use in our tests. Fixtures can be used as follows:

Include some HTML in an external fixtures file:

```
some.fixture.html: <div id="sample-fixture">some HTML content</div>
```

Then inside our actual test we would load it as follows:

```
loadFixtures('some.fixture.html')
$('#some-fixture').myTestedPlugin();
expect($('#some-fixture')).to<the rest of your matcher would go here>
```

The `jasmine-jquery` plugin loads fixtures from a directory named `spec/javascripts/fixtures` by default. If you wish to configure this path you can do so by initially setting `jasmine.getFixtures().fixturesPath = 'your custom path'`.

Finally, `jasmine-jquery` includes support for spying on jQuery events without the need for any extra plumbing work. This can be done using the `spyOnEvent()` and `assert(eventName).toHaveBeenTriggered(selector)` functions. For example:

```
spyOnEvent($('#el'), 'click');
$('#el').click();
expect('click').toHaveBeenTriggeredOn($('#el'));
```


View testing

In this section we will review the three dimensions of specs writing for Backbone Views: initial setup, view rendering, and templating. The latter two of these are the most commonly tested, however we'll see shortly why writing specs for the initialization of your views can also be of benefit.

Initial setup At their most basic, specs for Backbone views should validate that they are being correctly tied to specific DOM elements and are backed by valid data models. The reason to consider doing this is that these specs can identify issues which will trip up more complex tests later on. Also, they're fairly simple to write given the overall value offered.

To help ensure a consistent testing setup for our specs, we use `beforeEach()` to append both an empty `` (`#todoList`) to the DOM and initialize a new instance of a `TodoView` using an empty `Todo` model. `afterEach()` is used to remove the previous `#todoList ` as well as the previous instance of the view.

```
describe('Tests for TodoView', function() {

  beforeEach(function() {
    $('body').append('<ul id="todoList"></ul>');
    this.todoView = new TodoView({ model: new Todo() });
  });

  afterEach(function() {
    this.todoView.remove();
    $('#todoList').remove();
  });

  ...
});
```

The first spec useful to write is a check that the `TodoView` we've created is using the correct `tagName` (element or className). The purpose of this test is to make sure it's been correctly tied to a DOM element when it was created.

Backbone views typically create empty DOM elements once initialized, however these elements are not attached to the visible DOM in order to allow them to be constructed without an impact on the performance of rendering.

```
it('Should be tied to a DOM element when created, based off the property provided.', function() {
  //what html element tag name represents this view?
  expect(todoView.el.tagName.toLowerCase()).toBe('li');
});
```

Once again, if the `TodoView` has not already been written, we will experience failing specs. Thankfully, solving this is as simple as creating a new `Backbone.View` with a specific `tagName`.

```
var todoView = Backbone.View.extend({
  tagName: 'li'
});
```

If instead of testing against the `tagName` you would prefer to use a `className` instead, we can take advantage of `jasmine-jquery`'s `toHaveClass()` matcher:

```
it('Should have a class of "todos"', function(){
  expect(this.view.$el).toHaveClass('todos');
});
```

The `toHaveClass()` matcher operates on jQuery objects and if the plugin hadn't been used, an exception would have been thrown. It is of course also possible to test for the `className` by accessing `el.className` if you don't use `jasmine-jquery`.

You may have noticed that in `beforeEach()`, we passed our view an initial (albeit unfilled) `Todo` model. Views should be backed by a model instance which provides data. As this is quite important to our view's ability to function, we can write a spec to ensure a model is defined (using the `toBeDefined()` matcher) and then test attributes of the model to ensure defaults both exist and are the values we expect them to be.

```
it('Is backed by a model instance, which provides the data.', function() {

  expect(todoView.model).toBeDefined();

  // what's the value for Todo.get('done') here?
  expect(todoView.model.get('done')).toBe(false); //or toBeFalsy()
});
```

View rendering Next we're going to take a look at writing specs for view rendering. Specifically, we want to test that our `TodoView` elements are actually rendering as expected.

In smaller applications, those new to BDD might argue that visual confirmation of view rendering could replace unit testing of views. The reality is that when dealing with applications that might grow to a large number of views, it makes sense to automate this process as much as possible from the get-go. There are also aspects of rendering that require verification beyond what is visually presented on-screen (which we'll see very shortly).

We're going to begin testing views by writing two specs. The first spec will check that the view's `render()` method is correctly returning the view instance, which is necessary for chaining. Our second spec will check that the HTML produced is exactly what we expect based on the properties of the model instance that's been associated with our `TodoView`.

Unlike some of the previous specs we've covered, this section will make greater use of `beforeEach()` to both demonstrate how to use nested suites and also ensure a consistent set of conditions for our specs. In our first example we're simply going to create a sample model (based on `Todo`) and instantiate a `TodoView` with it.

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

  beforeEach(function() {
    this.model = new Backbone.Model({
      text: 'My Todo',
      order: 1,
      done: false
    });
    this.view = new TodoView({model:this.model});
  });

  describe('Rendering', function() {

    it('returns the view object', function() {
      expect(this.view.render()).toEqual(this.view);
    });

    it('produces the correct HTML', function() {
      this.view.render();

      // let's use jasmine-jquery's toContain() to avoid
      // testing for the complete content of a todo's markup
      expect(this.view.el.innerHTML)
        .toContain('<label class="todo-content">My Todo</label>');
    });

  });

});
```

When these specs are run, only the second one ('produces the correct HTML') fails. Our first spec ('returns the view object'), which is testing that the `TodoView` instance is returned from `render()`, passes since this is Backbone's

default behavior and we haven't overwritten the `render()` method with our own version yet.

Note: For the purposes of maintaining readability, all template examples in this section will use a minimal version of the following Todo view template. As it's relatively trivial to expand this, please feel free to refer to this sample if needed:

```
<div class="todo <%= done ? 'done' : '' %>">
  <div class="display">
    <input class="check" type="checkbox" <%= done ? 'checked="checked"' : '' %> />
    <label class="todo-content"><%= text %></label>
    <span class="todo-destroy"></span>
  </div>
  <div class="edit">
    <input class="todo-input" type="text" value="<%= content %>" />
  </div>
</div>
```

The second spec fails with the following message:

Expected " to contain '<label class="todo-content">My Todo</label>'.

The reason for this is the default behavior for `render()` doesn't create any markup.

Let's write a replacement for `render()` which fixes this:

```
render: function() {
  var template = '<label class="todo-content">+++PLACEHOLDER+++</label>';
  var output = template
    .replace('+++PLACEHOLDER+++', this.model.get('text'));
  this.$el.html(output);
  return this;
}
```

The above specifies an inline string template and replaces fields found in the template within the “+++PLACEHOLDER+++” blocks with their corresponding values from the associated model. As we're also returning the `TodoView` instance from the method, the first spec will still pass.

It would be impossible to discuss unit testing without mentioning fixtures. Fixtures typically contain test data (e.g., HTML) that is loaded in when needed (either locally or from an external file) for unit testing. So far we've been establishing jQuery expectations based on the view's `el` property. This works for a number of cases, however, there are instances where it may be necessary to render markup into the document. The most optimal way to handle this within specs is through using fixtures (another feature brought to us by the `jasmine-jquery` plugin).

Re-writing the last spec to use fixtures would look as follows:

```

describe('TodoView', function() {

  beforeEach(function() {
    ...
    setFixtures('<ul class="todos"></ul>');
  });

  ...

  describe('Template', function() {

    beforeEach(function() {
      $('<div> .todos').append(this.view.render().el);
    });

    it('has the correct text content', function() {
      expect($('<div> .todos').find('<div> .todo-content'))
        .toHaveText('My Todo');
    });

  });

});

```

What we're now doing in the above spec is appending the rendered todo item into the fixture. We then set expectations against the fixture, which may be something desirable when a view is setup against an element which already exists in the DOM. It would be necessary to provide both the fixture and test the `el` property correctly picking up the element expected when the view is instantiated.

Rendering with a templating system When a user marks a Todo item as complete (done), we may wish to provide them with visual feedback (such as a striked line through the text) to differentiate the item from those that are remaining. This can be done by attaching a new class to the item. Let's begin by writing a test:

```

describe('When a todo is done', function() {

  beforeEach(function() {
    this.model.set({done: true}, {silent: true});
    $('<div> .todos').append(this.view.render().el);
  });

  it('has a done class', function() {

```

```

        expect($('.todos .todo-content:first-child'))
            .toHaveClass('done');
    });

});

```

This will fail with the following message:

Expected '<label class="todo-content">My Todo</label>' to have class 'done'.

which can be fixed in the existing render() method as follows:

```

render: function() {
    var template = '<label class="todo-content">' +
        '<%= text %></label>';
    var output = template
        .replace('<%= text %>', this.model.get('text'));
    this.$el.html(output);
    if (this.model.get('done')) {
        this.$('.todo-content').addClass('done');
    }
    return this;
}

```

However, this can get unwieldily fairly quickly. As the level of complexity and logic in our templates increase, so do the challenges associated with testing them. We can ease this process by taking advantage of modern templating libraries, many of which have already been demonstrated to work well with testing solutions such as Jasmine.

JavaScript templating systems (such as [Handlebars](#), [Mustache](#), and Underscore's own [micro-templating](#)) support conditional logic in template strings. What this effectively means is that we can add if/else/ternery expressions inline which can then be evaluated as needed, allowing us to build even more powerful templates.

In our case, we are going to use the micro-templating found in Underscore.js as no additional files are required to use it and we can easily modify our existing specs to use it without a great deal of effort.

Assuming our template is defined using a script tag of ID myTemplate:

```

<script type="text/template" id="myTemplate">
    <div class="todo <%= done ? 'done' : '' %>">
        <div class="display">
            <input class="check" type="checkbox" <%= done ? 'checked="checked"' : '' %> />
            <label class="todo-content"><%= text %></label>
        </div>
    </div>
</script>

```

```

        <span class="todo-destroy"></span>
      </div>
      <div class="edit">
        <input class="todo-input" type="text" value="<%= content %>" />
      </div>
    </div>
  </script>

```

Our `TodoView` can be modified to use Underscore templating as follows:

```

var TodoView = Backbone.View.extend({

  tagName: 'li',
  template: _.template($('#myTemplate').html()),

  initialize: function(options) {
    // ...
  },

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

  ...

});

```

So, what's going on here? We're first defining our template in a script tag with a custom script type (e.g., `type="text/template"`). As this isn't a script type any browser understands, it's simply ignored, however referencing the script by an `id` attribute allows the template to be kept separate to other parts of the page.

In our view, we're the using the Underscore `_.template()` method to compile our template into a function that we can easily pass model data to later on. In the line `this.model.toJSON()` we are simply returning a copy of the model's attributes for JSON stringification to the `template` method, creating a block of HTML that can now be appended to the DOM.

Note: Ideally all of your template logic should exist outside of your specs, either in individual template files or embedded using script tags within your `SpecRunner`. This is generally more maintainable.

If you are working with much smaller templates and are not doing this, there is however a useful trick that can be applied to automatically create or extend templates in the Jasmine shared functional scope for each test.

By creating a new directory (say, ‘templates’) in the ‘spec’ folder and including a new script file with the following contents into SpecRunner.html, we can manually add custom attributes representing smaller templates we wish to use:

```
beforeEach(function() {
  this.templates = _.extend(this.templates || {}, {
    todo: '<label class="todo-content">' +
      '<%= text %>' +
      '</label>'
  });
});
```

To finish this off, we simply update our existing spec to reference the template when instantiating the `TodoView`:

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

  beforeEach(function() {
    ...
    this.view = new TodoView({
      model: this.model,
      template: this.templates.todo
    });
  });

  ...

});
```

The existing specs we’ve looked at would continue to pass using this approach, leaving us free to adjust the template with some additional conditional logic for Todos with a status of ‘done’:

```
beforeEach(function() {
  this.templates = _.extend(this.templates || {}, {
    todo: '<label class="todo-content <%= done ? \'done\' : \' \' %>">' +
      '<%= text %>' +
      '</label>'
  });
});
```

This will now also pass without any issues, however as mentioned, this last approach probably only makes sense if you’re working with smaller, highly dynamic templates.

Conclusions

We have now covered how to write Jasmine tests for Backbone.js models, collections, and views. While testing routing can at times be desirable, some developers feel it can be more optimal to leave this to third-party tools such as Selenium, so do keep this in mind.

Exercise

As an exercise, I recommend now trying the Jasmine Koans in `practicals\jasmine-koans` and trying to fix some of the purposefully failing tests it has to offer. This is an excellent way of not just learning how Jasmine specs and suites work, but working through the examples (without peeking back) will also put your Backbone skills to the test too.

Further reading

- [Testing Backbone Apps With SinonJS](#) by James Newbry
- [Jasmine + Backbone Revisited](#)
- [Backbone, PhantomJS and Jasmine](#)

QUnit

Introduction

QUnit is a powerful JavaScript test suite written by jQuery team member [Jörn Zaefferer](#) and used by many large open-source projects (such as jQuery and Backbone.js) to test their code. It's both capable of testing standard JavaScript code in the browser as well as code on the server-side (where environments supported include Rhino, V8 and SpiderMonkey). This makes it a robust solution for a large number of use-cases.

Quite a few Backbone.js contributors feel that QUnit is a better introductory framework for testing if you don't wish to start off with Jasmine and BDD right away. As we'll see later on in this chapter, QUnit can also be combined with third-party solutions such as SinonJS to produce an even more powerful testing solution supporting spies and mocks, which some say is preferable over Jasmine.

My personal recommendation is that it's worth comparing both frameworks and opting for the solution that you feel the most comfortable with.

Getting Setup

Luckily, getting QUnit setup is a fairly straight-forward process that will take less than 5 minutes.

We first setup a testing environment composed of three files:

- An HTML **structure** for displaying test results
- The **qunit.js** file composing the testing framework
- The **qunit.css** file for styling test results

The latter two of these can be downloaded from the [QUnit website](#).

If you would prefer, you can use a hosted version of the QUnit source files for testing purposes. The hosted URLs can be found at <http://github.com/jquery/qunit/raw/master/qunit/>.

Sample HTML with QUnit-compatible markup:

```
<!DOCTYPE html>
<html>
<head>
  <title>QUnit Test Suite</title>

  <link rel="stylesheet" href="qunit.css">
  <script src="qunit.js"></script>

  <!-- Your application -->
  <script src="app.js"></script>

  <!-- Your tests -->
  <script src="tests.js"></script>
</head>
<body>
  <h1 id="qunit-header">QUnit Test Suite</h1>
  <h2 id="qunit-banner"></h2>
  <div id="qunit-testrunner-toolbar"></div>
  <h2 id="qunit-userAgent"></h2>
  <ol id="qunit-tests">test markup, hidden.</ol>
</body>
</html>
```

Let's go through the elements above with `qunit` mentioned in their ID. When QUnit is running:

- **qunit-header** shows the name of the test suite

- **qunit-banner** shows up as red if a test fails and green if all tests pass
- **qunit-testrunner-toolbar** contains additional options for configuring the display of tests
- **qunit-userAgent** displays the navigator.userAgent property
- **qunit-tests** is a container for our test results

When running correctly, the above test runner looks as follows:



Figure 1: screenshot 1

The numbers of the form (a, b, c) after each test name correspond to a) failed asserts, b) passed asserts and c) total asserts. Clicking on a test name expands it to display all of the assertions for that test case. Assertions in green have successfully passed.

If however any tests fail, the test gets highlighted (and the qunit-banner at the top switches to red):

Assertions

QUnit supports a number of basic **assertions**, which are used in tests to verify that the result being returned by our code is what we expect. If an assertion fails, we know that a bug exists. Similar to Jasmine, QUnit can be used to easily test for regressions. Specifically, when a bug is found one can write an assertion to test the existence of the bug, write a patch, and then commit both. If subsequent changes to the code break the test you'll know what was responsible and be able to address it more easily.

Some of the supported QUnit assertions we're going to look at first are:

- `ok (state, message)` - passes if the first argument is truthy



Figure 2: screenshot 2



Figure 3: screenshot 3

- `equal (actual, expected, message)` - a simple comparison assertion with type coercion
- `notEqual (actual, expected, message)` - the opposite of the above
- `expect(amount)` - the number of assertions expected to run within each test
- `strictEqual(actual, expected, message)` - offers a much stricter comparison than `equal()` and is considered the preferred method of checking equality as it avoids stumbling on subtle coercion bugs
- `deepEqual(actual, expected, message)` - similar to `strictEqual`, comparing the contents (with `===`) of the given objects, arrays and primitives.

Basic test case using `test(name, callback)` Creating new test cases with QUnit is relatively straight-forward and can be done using `test()`, which constructs a test where the first argument is the **name** of the test to be displayed in our results and the second is a **callback** function containing all of our assertions. This is called as soon as QUnit is running.

```
var myString = 'Hello Backbone.js';

test( 'Our first QUnit test - asserting results', function(){

    // ok( boolean, message )
    ok( true, 'the test succeeds');
    ok( false, 'the test fails');

    // equal( actualValue, expectedValue, message )
    equal( myString, 'Hello Backbone.js', 'The value expected is Hello Backbone.js!');
});
```

What we're doing in the above is defining a variable with a specific value and then testing to ensure the value was what we expected it to be. This was done using the comparison assertion, `equal()`, which expects its first argument to be a value being tested and the second argument to be the expected value. We also used `ok()`, which allows us to easily test against functions or variables that evaluate to booleans.

Note: Optionally in our test case, we could have passed an 'expected' value to `test()` defining the number of assertions we expect to run. This takes the form: `test(name, [expected], test);` or by manually settings the expectation at the top of the test function, like so: `expect(1)`. I recommend you make a habit of always defining how many assertions you expect. More on this later.

Comparing the actual output of a function against the expected output As testing a simple static variable is fairly trivial, we can take this further

to test actual functions. In the following example we test the output of a function that reverses a string to ensure that the output is correct using `equal()` and `notEqual()`:

```
function reverseString( str ){
    return str.split('').reverse().join('');
}

test( 'reverseString()', function() {
    expect( 5 );
    equal( reverseString('hello'), 'olleh', 'The value expected was olleh' );
    equal( reverseString('foobar'), 'raboof', 'The value expected was raboof' );
    equal( reverseString('world'), 'dlrow', 'The value expected was dlrow' );
    notEqual( reverseString('world'), 'dlroo', 'The value was expected to not be dlroo' );
    equal( reverseString('bubble'), 'double', 'The value expected was elbbub' );
})
```

Running these tests in the QUnit test runner (which you would see when your HTML test page was loaded) we would find that four of the assertions pass while the last one does not. The reason the test against `'double'` fails is because it was purposefully written incorrectly. In your own projects if a test fails to pass and your assertions are correct, you've probably just found a bug!

Adding structure to assertions

Housing all of our assertions in one test case can quickly become difficult to maintain, but luckily QUnit supports structuring blocks of assertions more cleanly. This can be done using `module()` - a method that allows us to easily group tests together. A typical approach to grouping might be keeping multiple tests for a specific method as part of the same group (module).

Basic QUnit Modules:

```
module( 'Module One' );
test( 'first test', function() {} );
test( 'another test', function() {} );

module( 'Module Two' );
test( 'second test', function() {} );
test( 'another test', function() {} );

module( 'Module Three' );
test( 'third test', function() {} );
test( 'another test', function() {} );
```

We can take this further by introducing `setup()` and `teardown()` callbacks to our modules, where `setup()` is run before each test and `teardown()` is run after each test.

Using `setup()` and `teardown()` :

```
module( 'Module One', {
  setup: function() {
    // run before
  },
  teardown: function() {
    // run after
  }
});

test('first test', function() {
  // run the first test
});
```

These callbacks can be used to define (or clear) any components we wish to instantiate for use in one or more of our tests. As we'll see shortly, this is ideal for defining new instances of views, collections, models, or routers from a project that we can then reference across multiple tests.

Using `setup()` and `teardown()` for instantiation and clean-up

```
// Define a simple model and collection modeling a store and
// list of stores

var Store = Backbone.Model.extend({});

var StoreList = Backbone.Collection.extend({
  model: store,
  comparator: function( store ) { return store.get('name') }
});

// Define a group for our tests
module( 'StoreList sanity check', {
  setup: function() {
    this.list = new StoreList;
    this.list.add(new Store({ name: 'Costcutter' }));
    this.list.add(new Store({ name: 'Target' }));
    this.list.add(new Store({ name: 'Walmart' }));
    this.list.add(new Store({ name: 'Barnes & Noble' }));
  }
});
```

```

    },
    teardown: function() {
        window.errors = null;
    }
});

// Test the order of items added
test( 'test ordering', function() {
    expect( 1 );
    var expected = ['Barnes & Noble', 'Costcutter', 'Target', 'Walmart'];
    var actual = this.list.pluck('name');
    deepEqual( actual, expected, 'is maintained by comparator' );
});

```

Here, a list of stores is created and stored on `setup()`. A `teardown()` callback is used to simply clear a list of errors we might be storing within the window scope, but is otherwise not needed.

Assertion examples

Before we continue any further, let's review some more examples of how QUnit's various assertions can be correctly used when writing tests:

equal - a comparison assertion. It passes if `actual == expected`

```

test( 'equal', 2, function() {
    var actual = 6 - 5;
    equal( actual, true, 'passes as 1 == true' );
    equal( actual, 1, 'passes as 1 == 1' );
});

```

notEqual - a comparison assertion. It passes if `actual != expected`

```

test( 'notEqual', 2, function() {
    var actual = 6 - 5;
    notEqual( actual, false, 'passes as 1 != false' );
    notEqual( actual, 0, 'passes as 1 != 0' );
});

```

strictEqual - a comparison assertion. It passes if `actual === expected`.

```

test( 'strictEqual', 2, function() {
    var actual = 6 - 5;

```



```

    strictEqual( actual, true, 'fails as 1 !== true' );
    strictEqual( actual, 1, 'passes as 1 === 1' );
  });

```

notStrictEqual - a comparison assertion. It passes if actual !== expected.

```

test('notStrictEqual', 2, function() {
  var actual = 6 - 5;
  notStrictEqual( actual, true, 'passes as 1 !== true' );
  notStrictEqual( actual, 1, 'fails as 1 === 1' );
});

```

deepEqual - a recursive comparison assertion. Unlike strictEqual(), it works on objects, arrays and primitives.

```

test('deepEqual', 4, function() {
  var actual = {q: 'foo', t: 'bar'};
  var el = $('div');
  var children = $('div').children();

  equal( actual, {q: 'foo', t: 'bar'}, 'fails - objects are not equal using equal()' );
  deepEqual( actual, {q: 'foo', t: 'bar'}, 'passes - objects are equal' );
  equal( el, children, 'fails - jQuery objects are not the same' );
  deepEqual(el, children, 'fails - objects not equivalent' );

});

```

notDeepEqual - a comparison assertion. This returns the opposite of deepEqual

```

test('notDeepEqual', 2, function() {
  var actual = {q: 'foo', t: 'bar'};
  notEqual( actual, {q: 'foo', t: 'bar'}, 'passes - objects are not equal' );
  notDeepEqual( actual, {q: 'foo', t: 'bar'}, 'fails - objects are equivalent' );
});

```

raises - an assertion which tests if a callback throws any exceptions

```

test('raises', 1, function() {
  raises(function() {
    throw new Error( 'Oh no! It's an error!' );
  }, 'passes - an error was thrown inside our callback');
});

```

Fixtures

From time to time we may need to write tests that modify the DOM. Managing the clean-up of such operations between tests can be a genuine pain, but thankfully QUnit has a solution to this problem in the form of the `#qunit-fixture` element, seen below.

Fixture markup:

```
<!DOCTYPE html>
<html>
<head>
  <title>QUnit Test</title>
  <link rel="stylesheet" href="qunit.css">
  <script src="qunit.js"></script>
  <script src="app.js"></script>
  <script src="tests.js"></script>
</head>
<body>
  <h1 id="qunit-header">QUnit Test</h1>
  <h2 id="qunit-banner"></h2>
  <div id="qunit-testrunner-toolbar"></div>
  <h2 id="qunit-userAgent"></h2>
  <ol id="qunit-tests"></ol>
  <div id="qunit-fixture"></div>
</body>
</html>
```

We can either opt to place static markup in the fixture or just insert/append any DOM elements we may need to it. QUnit will automatically reset the `innerHTML` of the fixture after each test to its original value. In case you're using jQuery, it's useful to know that QUnit checks for its availability and will opt to use `$(el).html()` instead, which will cleanup any jQuery event handlers too.

Fixtures example:

Let us now go through a more complete example of using fixtures. One thing that most of us are used to doing in jQuery is working with lists - they're often used to define the markup for menus, grids, and a number of other components. You may have used jQuery plugins before that manipulated a given list in a particular way and it can be useful to test that the final (manipulated) output of the plugin is what was expected.

For the purposes of our next example, we're going to use Ben Alman's `$.enumerate()` plugin, which can prepend each item in a list by its index,

optionally allowing us to set what the first number in the list is. The code snippet for the plugin can be found below, followed by an example of the output it generates:

```
$.fn.enumerate = function( start ) {
    if ( typeof start !== 'undefined' ) {
        // Since 'start' value was provided, enumerate and return
        // the initial jQuery object to allow chaining.

        return this.each(function(i){
            $(this).prepend( '<b>' + ( i + start ) + '</b> ' );
        });

    } else {
        // Since no 'start' value was provided, function as a
        // getter, returning the appropriate value from the first
        // selected element.

        var val = this.eq( 0 ).children( 'b' ).eq( 0 ).text();
        return Number( val );
    }
};

/*
<ul>
  <li>1. hello</li>
  <li>2. world</li>
  <li>3. i</li>
  <li>4. am</li>
  <li>5. foo</li>
</ul>
*/
```

Let's now write some tests for the plugin. First, we define the markup for a list containing some sample items inside our `qunit-fixture` element:

```
<div id="qunit-fixture">
  <ul>
    <li>hello</li>
    <li>world</li>
    <li>i</li>
    <li>am</li>
    <li>foo</li>
  </ul>
</div>
```

Next, we need to think about what should be tested. `$.enumerate()` supports a few different use cases, including:

- **No arguments passed** - i.e., `$(el).enumerate()`
- **0 passed as an argument** - i.e., `$(el).enumerate(0)`
- **1 passed as an argument** - i.e., `$(el).enumerate(1)`

As the text value for each list item is of the form “n. item-text” and we only require this to test against the expected output, we can simply access the content using `$(el).eq(index).text()` (for more information on `.eq()` see [here](#)).

and finally, here are our test cases:

```
module('jQuery#enumerate');

test( 'No arguments passed', 5, function() {
    var items = $('#qunit-fixture li').enumerate(); // 0
    equal( items.eq(0).text(), '0. hello', 'first item should have index 0' );
    equal( items.eq(1).text(), '1. world', 'second item should have index 1' );
    equal( items.eq(2).text(), '2. i', 'third item should have index 2' );
    equal( items.eq(3).text(), '3. am', 'fourth item should have index 3' );
    equal( items.eq(4).text(), '4. foo', 'fifth item should have index 4' );
});

test( '0 passed as an argument', 5, function() {
    var items = $('#qunit-fixture li').enumerate( 0 );
    equal( items.eq(0).text(), '0. hello', 'first item should have index 0' );
    equal( items.eq(1).text(), '1. world', 'second item should have index 1' );
    equal( items.eq(2).text(), '2. i', 'third item should have index 2' );
    equal( items.eq(3).text(), '3. am', 'fourth item should have index 3' );
    equal( items.eq(4).text(), '4. foo', 'fifth item should have index 4' );
});

test( '1 passed as an argument', 3, function() {
    var items = $('#qunit-fixture li').enumerate( 1 );
    equal( items.eq(0).text(), '1. hello', 'first item should have index 1' );
    equal( items.eq(1).text(), '2. world', 'second item should have index 2' );
    equal( items.eq(2).text(), '3. i', 'third item should have index 3' );
    equal( items.eq(3).text(), '4. am', 'fourth item should have index 4' );
    equal( items.eq(4).text(), '5. foo', 'fifth item should have index 5' );
});
```

Asynchronous code

As with Jasmine, the effort required to run synchronous tests with QUnit is fairly minimal. That said, what about tests that require asynchronous callbacks

(such as expensive processes, Ajax requests, and so on)? When we're dealing with asynchronous code, rather than letting QUnit control when the next test runs, we can tell it that we need it to stop running and wait until it's okay to continue once again.

Remember: running asynchronous code without any special considerations can cause incorrect assertions to appear in other tests, so we want to make sure we get it right.

Writing QUnit tests for asynchronous code is made possible using the `start()` and `stop()` methods, which programmatically set the start and stop points during such tests. Here's a simple example:

```
test('An async test', function(){
    stop();
    expect( 1 );
    $.ajax({
        url: '/test',
        dataType: 'json',
        success: function( data ){
            deepEqual(data, {
                topic: 'hello',
                message: 'hi there!'
            });
            ok(true, 'Asynchronous test passed!');
            start();
        }
    });
});
```

A jQuery `$.ajax()` request is used to connect to a test resource and assert that the data returned is correct. `deepEqual()` is used here as it allows us to compare different data types (e.g., objects, arrays) and ensures that what is returned is exactly what we're expecting. We know that our Ajax request is asynchronous and so we first call `stop()`, then run the code making the request, and finally, at the very end of our callback, inform QUnit that it is okay to continue running other tests.

Note: rather than including `stop()`, we can simply exclude it and substitute `test()` with `asyncTest()` if we prefer. This improves readability when dealing with a mixture of asynchronous and synchronous tests in your suite. While this setup should work fine for many use-cases, there is no guarantee that the callback in our `$.ajax()` request will actually get called. To factor this into our tests, we can use `expect()` once again to define how many assertions we expect to see within our test. This is a healthy safety blanket as it ensures that if a test completes with an insufficient number of assertions, we know something went wrong and can fix it.

SinonJS

Similar to the section on testing Backbone.js apps using the Jasmine BDD framework, we're nearly ready to take what we've learned and write a number of QUnit tests for our Todo application.

Before we start though, you may have noticed that QUnit doesn't support test spies. Test spies are functions which record arguments, exceptions, and return values for any of their calls. They're typically used to test callbacks and how functions may be used in the application being tested. In testing frameworks, spies usually are anonymous functions or wrappers around functions which already exist.

What is SinonJS?

In order for us to substitute support for spies in QUnit, we will be taking advantage of a mocking framework called [SinonJS](#) by Christian Johansen. We will also be using the [SinonJS-QUnit adapter](#) which provides seamless integration with QUnit (meaning setup is minimal). SinonJS is completely test-framework agnostic and should be easy to use with any testing framework, so it's ideal for our needs.

The framework supports three features we'll be taking advantage of for unit testing our application:

- **Anonymous spies**
- **Spying on existing methods**
- **A rich inspection interface**

Basic Spies Using `this.spy()` without any arguments creates an anonymous spy. This is comparable to `jasmine.createSpy()`. We can observe basic usage of a SinonJS spy in the following example:

```
test('should call all subscribers for a message exactly once', function () {
    var message = getUniqueString();
    var spy = this.spy();

    PubSub.subscribe( message, spy );
    PubSub.publishSync( message, 'Hello World' );

    ok( spy.calledOnce, 'the subscriber was called once' );
});
```

Spying On Existing Functions We can also use `this.spy()` to spy on existing functions (like jQuery's `$.ajax`) in the example below. When spying on a function which already exists, the function behaves normally but we get access to data about its calls which can be very useful for testing purposes.

```
test( 'should inspect the jQuery.getJSON usage of jQuery.ajax', function () {
    this.spy( jQuery, 'ajax' );

    jQuery.getJSON( '/todos/completed' );

    ok( jQuery.ajax.calledOnce );
    equals( jQuery.ajax.getCall(0).args[0].url, '/todos/completed' );
    equals( jQuery.ajax.getCall(0).args[0].dataType, 'json' );
});
```

Inspection Interface SinonJS comes with a rich spy interface which allows us to test whether a spy was called with a specific argument, if it was called a specific number of times, and test against the values of arguments. A complete list of features supported in the interface can be found on [SinonJS.org](http://sinonjs.org), but let's take a look at some examples demonstrating some of the most commonly used ones:

Matching arguments: test a spy was called with a specific set of arguments:

```
test( 'Should call a subscriber with standard matching': function () {
    var spy = sinon.spy();

    PubSub.subscribe( 'message', spy );
    PubSub.publishSync( 'message', { id: 45 } );

    assertTrue( spy.calledWith( { id: 45 } ) );
});
```

Stricter argument matching: test a spy was called at least once with specific arguments and no others:

```
test( 'Should call a subscriber with strict matching': function () {
    var spy = sinon.spy();

    PubSub.subscribe( 'message', spy );
    PubSub.publishSync( 'message', 'many', 'arguments' );
    PubSub.publishSync( 'message', 12, 34 );

    // This passes
```

```

    assertTrue( spy.calledWith('many') );

    // This however, fails
    assertTrue( spy.calledWithExactly( 'many' ) );
  });

```

Testing call order: testing if a spy was called before or after another spy:

```

test( 'Should call a subscriber and maintain call order': function () {
  var a = sinon.spy();
  var b = sinon.spy();

  PubSub.subscribe( 'message', a );
  PubSub.subscribe( 'event', b );

  PubSub.publishSync( 'message', { id: 45 } );
  PubSub.publishSync( 'event', [1, 2, 3] );

  assertTrue( a.calledBefore(b) );
  assertTrue( b.calledAfter(a) );
});

```

Match execution counts: test a spy was called a specific number of times:

```

test( 'Should call a subscriber and check call counts', function () {
  var message = getUniqueString();
  var spy = this.spy();

  PubSub.subscribe( message, spy );
  PubSub.publishSync( message, 'some payload' );

  // Passes if spy was called once and only once.
  ok( spy.calledOnce ); // calledTwice and calledThrice are also supported

  // The number of recorded calls.
  equal( spy.callCount, 1 );

  // Directly checking the arguments of the call
  equals( spy.getCall(0).args[0], message );
});

```


Stubs and mocks

SinonJS also supports two other powerful features: stubs and mocks. Both stubs and mocks implement all of the features of the spy API, but have some added functionality.

Stubs

A stub allows us to replace any existing behaviour for a specific method with something else. They can be very useful for simulating exceptions and are most often used to write test cases when certain dependencies of your code-base may not yet be written.

Let us briefly re-explore our Backbone Todo application, which contained a Todo model and a TodoList collection. For the purpose of this walkthrough, we want to isolate our TodoList collection and fake the Todo model to test how adding new models might behave.

We can pretend that the models have yet to be written just to demonstrate how stubbing might be carried out. A shell collection just containing a reference to the model to be used might look like this:

```
var TodoList = Backbone.Collection.extend({
  model: Todo
});

// Let's assume our instance of this collection is
this.todoList;
```

Assuming our collection is instantiating new models itself, it's necessary for us to stub the model's constructor function for the test. This can be done by creating a simple stub as follows:

```
this.todoStub = sinon.stub( window, 'Todo' );
```

The above creates a stub of the Todo method on the window object. When stubbing a persistent object, it's necessary to restore it to its original state. This can be done in a `teardown()` as follows:

```
this.todoStub.restore();
```

After this, we need to alter what the constructor returns, which can be efficiently done using a plain `Backbone.Model` constructor. While this isn't a Todo model, it does still provide us an actual Backbone model.

```

setup: function() {
  this.model = new Backbone.Model({
    id: 2,
    title: 'Hello world'
  });
  this.todoStub.returns( this.model );
});

```

The expectation here might be that this snippet would ensure our `TodoList` collection always instantiates a stubbed `Todo` model, but because a reference to the model in the collection is already present, we need to reset the model property of our collection as follows:

```

this.todoList.model = Todo;

```

The result of this is that when our `TodoList` collection instantiates new `Todo` models, it will return our plain Backbone model instance as desired. This allows us to write a test for the addition of new model literals as follows:

```

module( 'Should function when instantiated with model literals', {

  setup: function() {

    this.todoStub = sinon.stub(window, 'Todo');
    this.model = new Backbone.Model({
      id: 2,
      title: 'Hello world'
    });

    this.todoStub.returns(this.model);
    this.todos = new TodoList();

    // Let's reset the relationship to use a stub
    this.todos.model = Todo;

    // add a model
    this.todos.add({
      id: 2,
      title: 'Hello world'
    });
  },

  teardown: function() {
    this.todoStub.restore();
  }
}

```

```
});

test('should add a model', function() {
    equal( this.todos.length, 1 );
});

test('should find a model by id', function() {
    equal( this.todos.get(5).get('id'), 5 );
});
});
```

Mocks

Mocks are effectively the same as stubs, however they mock a complete API and have some built-in expectations for how they should be used. The difference between a mock and a spy is that as the expectations for their use are pre-defined and the test will fail if any of these are not met.

Here's a snippet with sample usage of a mock based on PubSubJS. Here, we have a `clearTodo()` method as a callback and use mocks to verify its behavior.

```
test('should call all subscribers when exceptions', function () {
    var myAPI = { clearTodo: function () {} };

    var spy = this.spy();
    var mock = this.mock( myAPI );
    mock.expects( 'clearTodo' ).once().throws();

    PubSub.subscribe( 'message', myAPI.clearTodo );
    PubSub.subscribe( 'message', spy );
    PubSub.publishSync( 'message', undefined );

    mock.verify();
    ok( spy.calledOnce );
});
```

Exercise

We can now begin writing tests for our Todo application, which are listed and separated by component (e.g., Models, Collections, etc.). It's useful to pay attention to the name of the test, the logic being tested, and most importantly the assertions being made as this will give you some insight into how what we've learned can be applied to a complete application.

To get the most out of this section, I recommend looking at the QUnit Koans included in the `practicals/qunit-koans` folder - this is a port of the Backbone.js Jasmine Koans over to QUnit.

In case you haven't had a chance to try out one of the Koans kits as yet, they are a set of unit tests using a specific testing framework that both demonstrate how a set of tests for an application may be written, but also leave some tests unfilled so that you can complete them as an exercise.

Models

For our models we want to at minimum test that:

- New instances can be created with the expected default values
- Attributes can be set and retrieved correctly
- Changes to state correctly fire off custom events where needed
- Validation rules are correctly enforced

```
module( 'About Backbone.Model' );

test('Can be created with default values for its attributes.', function() {
    expect( 1 );

    var todo = new Todo();
    equal( todo.get('text'), '' );
    equal( todo.get('done'), false );
    equal( todo.get('order'), 0 );
});

test('Will set attributes on the model instance when created.', function() {
    expect( 3 );

    var todo = new Todo( { text: 'Get oil change for car.' } );
    equal( todo.get('text'), 'Get oil change for car.' );
});

test('Will call a custom initialize function on the model instance when created.', function() {
    expect( 1 );

    var toot = new Todo({ text: 'Stop monkeys from throwing their own crap!' });
    equal( toot.get('text'), 'Stop monkeys from throwing their own rainbows!' );
});

test('Fires a custom event when the state changes.', function() {
```

```

    expect( 1 );

    var spy = this.spy();
    var todo = new Todo();

    todo.on( 'change', spy );
    // How would you update a property on the todo here?
    todo.set( { text: 'new text' } );

    ok( spy.calledOnce, 'A change event callback was correctly triggered' );
  });

test('Can contain custom validation rules, and will trigger an invalid event on failed validation', function() {
  expect( 3 );

  var errorCallback = this.spy();
  var todo = new Todo();

  todo.on('invalid', errorCallback);
  // What would you need to set on the todo properties to cause validation to fail?
  todo.set( { done: 'not a boolean' } );

  ok( errorCallback.called, 'A failed validation correctly triggered an error' );
  notEqual( errorCallback.getCall(0), undefined );
  equal( errorCallback.getCall(0).args[1], 'Todo.done must be a boolean value.' );
});

```

Collections

For our collection we'll want to test that:

- The Collection has a Todo Model
- Uses localStorage for syncing
- That done(), remaining() and clear() work as expected
- The order for Todos is numerically correct

```

describe('Test Collection', function() {

  beforeEach(function() {

    // Define new todos
    this.todoOne = new Todo;
  });

```

```

    this.todoTwo = new Todo({
      title: "Buy some milk"
    });

    // Create a new collection of todos for testing
    return this.todos = new TodoList([this.todoOne, this.todoTwo]);
  });

  it('Has the Todo model', function() {
    return expect(this.todos.model).toBe(Todo);
  });

  it('Uses local storage', function() {
    return expect(this.todos.localStorage).toEqual(new Store('todos-backbone'));
  });

  describe('done', function() {
    return it('returns an array of the todos that are done', function() {
      this.todoTwo.done = true;
      return expect(this.todos.done()).toEqual([this.todoTwo]);
    });
  });

  describe('remaining', function() {
    return it('returns an array of the todos that are not done', function() {
      this.todoTwo.done = true;
      return expect(this.todos.remaining()).toEqual([this.todoOne]);
    });
  });

  describe('clear', function() {
    return it('destroys the current todo from local storage', function() {
      expect(this.todos.models).toEqual([this.todoOne, this.todoTwo]);
      this.todos.clear(this.todoOne);
      return expect(this.todos.models).toEqual([this.todoTwo]);
    });
  });

  return describe('Order sets the order on todos ascending numerically', function() {
    it('defaults to one when there arent any items in the collection', function() {
      this.emptyTodos = new TodoApp.Collections.TodoList;
      return expect(this.emptyTodos.order()).toEqual(0);
    });

    return it('Increments the order by one each time', function() {
      expect(this.todos.order(this.todoOne)).toEqual(1);
    });
  });

```

```

        return expect(this.todos.order(this.todoTwo)).toEqual(2);
    });
});
});

```

Views

For our views we want to ensure:

- They are being correctly tied to a DOM element when created
- They can render, after which the DOM representation of the view should be visible
- They support wiring up view methods to DOM elements

One could also take this further and test that user interactions with the view correctly result in any models that need to be changed being updated correctly.

```

module( 'About Backbone.View', {
  setup: function() {
    $('body').append('<ul id="todoList"></ul>');
    this.todoView = new TodoView({ model: new Todo() });
  },
  teardown: function() {
    this.todoView.remove();
    $('#todoList').remove();
  }
});

test('Should be tied to a DOM element when created, based off the property provided.', function() {
  expect( 1 );
  equal( this.todoView.el.tagName.toLowerCase(), 'li' );
});

test('Is backed by a model instance, which provides the data.', function() {
  expect( 2 );
  notEqual( this.todoView.model, undefined );
  equal( this.todoView.model.get('done'), false );
});

test('Can render, after which the DOM representation of the view will be visible.', function() {
  this.todoView.render();

  // Hint: render() just builds the DOM representation of the view, but doesn't insert it

```

```

    // How would you append it to the ul#todoList?
    // How do you access the view's DOM representation?

    $('ul#todoList').append(this.todoView.el);
    equal($('#todoList').find('li').length, 1);
  });

  asyncTest('Can wire up view methods to DOM elements.', function() {
    expect( 2 );
    var viewElt;

    $('#todoList').append( this.todoView.render().el );

    setTimeout(function() {
      viewElt = $('#todoList li input.check').filter(':first');

      equal(viewElt.length > 0, true);

      // Make sure that QUnit knows we can continue
      start();
    }, 1000, 'Expected DOM Elt to exist');

    // Hint: How would you trigger the view, via a DOM Event, to toggle the 'done' status.
    // (See todos.js line 70, where the events hash is defined.)

    $('#todoList li input.check').click();
    equal( this.todoView.model.get('done'), true );
  });

```

App

It can also be useful to write tests for any application bootstrap you may have in place. For the following module, our setup instantiates and appends to a `TodoApp` view and we can test anything from local instances of views being correctly defined to application interactions correctly resulting in changes to instances of local collections.

```

module( 'About Backbone Applications' , {
  setup: function() {
    Backbone.localStorageDB = new Store('testTodos');
    $('#qunit-fixture').append('<div id="app"></div>');
    this.App = new TodoApp({ appendTo: $('#app') });
  },

```



```

teardown: function() {
  this.App.todos.reset();
  $('#app').remove();
}
});

test('Should bootstrap the application by initializing the Collection.', function() {
  expect( 2 );

  notEqual( this.App.todos, undefined );
  equal( this.App.todos.length, 0 );
});

test( 'Should bind Collection events to View creation.' , function() {
  $('#new-todo').val( 'Foo' );
  $('#new-todo').trigger(new $.Event( 'keypress', { keyCode: 13 } ));

  equal( this.App.todos.length, 1 );
});

```

Further Reading & Resources

That's it for this section on testing applications with QUnit and SinonJS. I encourage you to try out the [QUnit Backbone.js Koans](#) and see if you can extend some of the examples. For further reading consider looking at some of the additional resources below:

- [Test-driven JavaScript Development \(book\)](#)
- [SinonJS/QUnit Adapter](#)
- [SinonJS and QUnit](#)
- [Automating JavaScript Testing With QUnit](#)
- [Ben Alman's Unit Testing With QUnit](#)
- [Another QUnit/Backbone.js demo project](#)
- [SinonJS helpers for Backbone](#)

Resources

Books & Courses

- [PeepCode: Backbone.js Basics](#)
- [CodeSchool: Anatomy Of Backbone](#)
- [Recipes With Backbone](#)
- [Backbone Patterns](#)

- [Backbone On Rails](#)
- [MVC In JavaScript With Backbone](#)
- [Backbone Tutorials](#)
- [Derick Bailey's Resources For Learning Backbone](#)

Extensions/Libraries

- [MarionetteJS](#)
- [AuraJS](#)
- [Thorax](#)
- [Lumbar](#)
- [Backbone Layout Manager](#)
- [Backbone Boilerplate](#)
- [Backbone.ModelBinder](#)
- [Backbone Relational - for model relationships](#)
- [Backbone CouchDB](#)
- [Backbone Validations - HTML5 inspired validations](#)

Conclusions

I hope that you've found this introduction to Backbone.js of value. What you've hopefully learned is that whilst building a JavaScript-heavy application using nothing more than a DOM manipulation library (such as jQuery) is certainly a possible feat, it is difficult to build anything non-trivial without any formal structure in place. Your nested pile of jQuery callbacks and DOM elements are unlikely to scale and they can be very difficult to maintain as your application grows.

The beauty of Backbone.js is it's simplicity. It's very small given the functionality and flexibility it provides, which is evident if you begin to study the Backbone.js source. In the words of Jeremy Ashkenas, "The essential premise at the heart of Backbone has always been to try and discover the minimal set of data-structuring (Models and Collections) and user interface (Views and URLs) primitives that are useful when building web applications with JavaScript." It just helps you improve the structure of your applications, helping you better separate concerns. There isn't anything more to it than that.

Backbone offers Models with key-value bindings and events, Collections with an API of rich enumerable methods, declarative Views with event handling and a simple way to connect an existing API to your client-side application over a RESTful JSON interface. Use it and you can abstract away data into sane models and your DOM manipulation into views, binding together using nothing more than events.

Almost any developer working on JavaScript applications for a while will ultimately come to creating a similar solution to it on their own if they value architecture and maintainability. The alternative to using it or something similar is rolling your own - often a process that involves glueing together a diverse set of libraries that weren't built to work together. You might use jQuery BBQ for history management and Handlebars for templating, whilst writing abstracts for organizing and testing code by yourself.

Contrast this with Backbone, which has [literate documentation](#) of the source code, a thriving community of both users and hackers and a large number of questions about it asked and answered daily on sites like [Stack Overflow](#). Rather than re-inventing the wheel there are many advantages to structuring your application using a solution based on the collective knowledge and experience of an entire community.

In addition to helping provide sane structure to your applications, Backbone is highly extensible supporting more custom architecture should you require more than what is prescribed out of the box. This is evident by the number of extensions and plugins which have been released for it over the past year, including those which we have touched upon such as MarionetteJS and Thorax.

These days Backbone.js powers many complex web applications, ranging from the LinkedIn [mobile app](#) to popular RSS readers such as [NewsBlur](#) through to social commentary widgets such as [Disqus](#). This small library of simple, but sane abstractions has helped to create a new generation of rich web applications, and I and my collaborators hope that in time it can help you too.

If you're wondering whether it is worth using Backbone on a project, ask yourself whether what you are building is complex enough to merit using it. Are you hitting the limits of your ability to organize your code? Will your application have regular changes to what is displayed in the UI without a trip back to the server for new pages? Would you benefit from a separation of concerns? If so, a solution like Backbone may be able to help.

Google's GMail is often cited as an example of a well built single-page app. If you've used it, you might have noticed that it requests a large initial chunk, representing most of the JavaScript, CSS and HTML most users will need and everything extra needed after that occurs in the background. GMail can easily switch between your inbox to your spam folder without needing the whole page to be re-rendered. Libraries like Backbone make it easier for web developers to create experiences like this.

That said, Backbone won't be able to help if you're planning on building something which isn't worth the learning curve associated with a library. If your application or site is still will still heavily be using the server to do the heavy lifting of constructing and serving complete pages to the browser, you may find just using plain JavaScript or jQuery for simple effects or interactions to be more appropriate. Spend time assessing how suitable Backbone might be for you and make the right choice on a per-project basis.

Backbone is neither difficult to learn nor use, however the time and effort you spend learning how to structure applications using it will be well worth it. Whilst reading this book will equip you with the fundamentals needed to understand the library, the best way to learn is to try building your own real-world applications. You will hopefully find that the end product is cleaner, better organized and more maintainable code.

With that, I wish you the very best with your onward journey into the world of Backbone and will leave you with a quote from American writer [Henry Miller](#) - “One’s destination is never a place, but a new way of seeing things.”

Appendix

A Simple JavaScript MVC Implementation

A comprehensive discussion of Backbone’s implementation is beyond the scope of this book. We can, however, present a simple MVC library - which we will call `Cranium.js` - that illustrates how frameworks such as Backbone implement the MVC pattern.

Like Backbone, we will rely on [Underscore](#) for inheritance and templating.

Event System

At the heart of our JavaScript MVC implementation is an **Event** system (object) based on the Publisher-Subscriber Pattern which makes it possible for MVC components to communicate in an elegant, decoupled manner. Subscribers ‘listen’ for specific events of interest and react when Publishers broadcast these events.

Event is mixed into both the View and Model components so that instances of either of these components can publish events of interest.

```
// cranium.js - Cranium.Events

var Cranium = Cranium || {};

// Set DOM selection utility
var $ = document.querySelector.bind(document) || this.jQuery || this.Zepto;

// Mix in to any object in order to provide it with custom events.
var Events = Cranium.Events = {
  // Keeps list of events and associated listeners
  channels: {},

  // Counter
```

```

    eventNumber: 0,

    // Announce events and passes data to the listeners;
    trigger: function (events, data) {
        for (var topic in Cranium.Events.channels){
            if (Cranium.Events.channels.hasOwnProperty(topic)) {
                if (topic.split("-")[0] == events){
                    Cranium.Events.channels[topic](data) !== false || delete Cranium.Events.channels[topic];
                }
            }
        }
    },
    // Registers an event type and its listener
    on: function (events, callback) {
        Cranium.Events.channels[events + --Cranium.Events.eventNumber] = callback;
    },
    // Unregisters an event type and its listener
    off: function(topic) {
        delete Cranium.Events.channels[topic];
    }
};

```

The Event system makes it possible for:

- a View to notify its subscribers of user interaction (e.g., clicks or input in a form), to update/re-render its presentation, etc.
- a Model whose data has changed to notify its Subscribers to update themselves (e.g., view to re-render to show accurate/updated data), etc.

Models

Models manage the (domain-specific) data for an application. They are concerned with neither the user-interface nor presentation layers, but instead represent structured data that an application may require. When a model changes (e.g. when it is updated), it will typically notify its observers (Subscribers) that a change has occurred so that they may react accordingly.

Let's see a simple implementation of the Model:

```

// cranium.js - Cranium.Model

// Attributes represents data, model's properties.
// These are to be passed at Model instantiation.
// Also we are creating id for each Model instance
// so that it can identify itself (e.g. on change

```

```

// announcements)
var Model = Cranium.Model = function (attributes) {
  this.id = _.uniqueId('model');
  this.attributes = attributes || {};
};

// Getter (accessor) method;
// returns named data item
Cranium.Model.prototype.get = function(attrName) {
  return this.attributes[attrName];
};

// Setter (mutator) method;
// Set/mix in into model mapped data (e.g.{name: "John"})
// and publishes the change event
Cranium.Model.prototype.set = function(attrs){
  if (_.isObject(attrs)) {
    _.extend(this.attributes, attrs);
    this.change(this.attributes);
  }
  return this;
};

// Returns clone of the Models data object
// (used for view template rendering)
Cranium.Model.prototype.toJSON = function(options) {
  return _.clone(this.attributes);
};

// Helper function that announces changes to the Model
// and passes the new data
Cranium.Model.prototype.change = function(attrs){
  this.trigger(this.id + 'update', attrs);
};

// Mix in Event system
_.extend(Cranium.Model.prototype, Cranium.Events);

```

Views

Views are a visual representation of models that present a filtered view of their current state. A view typically observes a model and is notified when the model changes, allowing the view to update itself accordingly. Design pattern literature commonly refers to views as ‘dumb’, given that their knowledge of models and controllers in an application is limited.

Let's explore Views a little further using a simple JavaScript example:

```
// DOM View
var View = Cranium.View = function (options) {
  // Mix in options object (e.g extending functionality)
  _.extend(this, options);
  this.id = _.uniqueId('view');
};

// Mix in Event system
_.extend(Cranium.View.prototype, Cranium.Events);
```

Controllers

Controllers are an intermediary between models and views which are classically responsible for two tasks:

- they update the view when the model changes
- they update the model when the user manipulates the view

```
// cranium.js - Cranium.Controller

// Controller tying together a model and view
var Controller = Cranium.Controller = function(options){
  // Mix in options object (e.g extending functionality)
  _.extend(this, options);
  this.id = _.uniqueId('controller');
  var parts, selector, eventType;

  // Parses Events object passed during the definition of the
  // controller and maps it to the defined method to handle it;
  if(this.events){
    _.each(this.events, function(method, eventName){
      parts = eventName.split('.');
      selector = parts[0];
      eventType = parts[1];
      $(selector)['on' + eventType] = this[method];
    }.bind(this));
  }
};
```

Practical Usage

HTML template for the primer that follows:

```

<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
  <meta name="description" content="">
</head>
<body>
<div id="todo">
</div>
  <script type="text/template" class="todo-template">
    <div>
      <input id="todo_complete" type="checkbox" <%= completed %>>
      <%= title %>
    </div>
  </script>
  <script src="underscore-min.js"></script>
  <script src="cranium.js"></script>
  <script src="example.js"></script>
</body>
</html>

```

Cranium.js usage:

```

// example.js - usage of Cranium MVC

// And todo instance
var todo1 = new Cranium.Model({
  title: "",
  completed: ""
});

console.log("First todo title - nothing set: " + todo1.get('title'));
todo1.set({title: "Do something"});
console.log("Its changed now: " + todo1.get('title'));
,

// View instance
var todoView = new Cranium.View({
  // DOM element selector
  el: '#todo',

  // Todo template; Underscore templating used
  template: _.template($(' .todo-template').innerHTML),

```



```

    init: function (model) {
        this.render( model.toJSON() );

        this.on(model.id + 'update', this.render.bind(this));
    },
    render: function (data) {
        console.log("View about to render.");
        $(this.el).innerHTML = this.template( data );
    }
});

var todoController = new Cranium.Controller({
    // Specify the model to update
    model: todo1,

    // and the view to observe this model
    view: todoView,

    events: {
        "#todo.click" : "toggleComplete"
    },

    // Initialize everything
    initialize: function () {
        this.view.init(this.model);
        return this;
    },
    // Toggles the value of the todo in the Model
    toggleComplete: function () {
        var completed = todoController.model.get('completed');
        console.log("Todo old 'completed' value?", completed);
        todoController.model.set({ completed: (!completed) ? 'checked': '' });
        console.log("Todo new 'completed' value?", todoController.model.get('completed'));
        return this;
    }
});

// Let's kick start things off
todoController.initialize();

todo1.set({ title: "Due to this change Model will notify View and it will re-render"});

```

Samuel Clay, one of the authors of the first version of Backbone.js says of cranium.js: “Unsurprisingly, it looks a whole lot like the beginnings of Backbone. Views are dumb, so they get very little boilerplate and setup. Models are

responsible for their attributes and announcing changes to those models.”

I hope you’ve found this implementation helpful in understanding how one would go about writing their own library like Backbone from scratch, but moreso that it encourages you to take advantage of mature existing solutions where possible but never be afraid to explore deeper down into what makes them tick.

MVP

Model-View-Presenter (MVP) is a derivative of the MVC design pattern which focuses on improving presentation logic. It originated at a company named [Taligent](#) in the early 1990s while they were working on a model for a C++ CommonPoint environment. Whilst both MVC and MVP target the separation of concerns across multiple components, there are some fundamental differences between them.

For the purposes of this summary we will focus on the version of MVP most suitable for web-based architectures.

Models, Views & Presenters

The P in MVP stands for presenter. It’s a component which contains the user-interface business logic for the view. Unlike MVC, invocations from the view are delegated to the presenter, which are decoupled from the view and instead talk to it through an interface. This allows for all kinds of useful things such as being able to mock views in unit tests.

The most common implementation of MVP is one which uses a Passive View (a view which is for all intents and purposes “dumb”), containing little to no logic. MVP models are almost identical to MVC models and handle application data. The presenter acts as a mediator which talks to both the view and model, however both of these are isolated from each other. They effectively bind models to views, a responsibility held by Controllers in MVC. Presenters are at the heart of the MVP pattern and as you can guess, incorporate the presentation logic behind views.

Solicited by a view, presenters perform any work to do with user requests and pass data back to them. In this respect, they retrieve data, manipulate it and determine how the data should be displayed in the view. In some implementations, the presenter also interacts with a service layer to persist data (models). Models may trigger events but it’s the presenter’s role to subscribe to them so that it can update the view. In this passive architecture, we have no concept of direct data binding. Views expose setters which presenters can use to set data.

The benefit of this change from MVC is that it increases the testability of your application and provides a more clean separation between the view and the

model. This isn't however without its costs as the lack of data binding support in the pattern can often mean having to take care of this task separately.

Although a common implementation of a [Passive View](#) is for the view to implement an interface, there are variations on it, including the use of events which can decouple the View from the Presenter a little more. As we don't have the interface construct in JavaScript, we're using it more and more as a protocol than an explicit interface here. It's technically still an API and it's probably fair for us to refer to it as an interface from that perspective.

There is also a [Supervising Controller](#) variation of MVP, which is closer to the MVC and [MVVM - Model-View-ViewModel](#) patterns as it provides data-binding from the Model directly from the View. Key-value observing (KVO) plugins (such as Derick Bailey's Backbone.ModelBinding plugin) introduce this idea of a Supervising Controller to Backbone.

MVP or MVC?

MVP is generally used most often in enterprise-level applications where it's necessary to reuse as much presentation logic as possible. Applications with very complex views and a great deal of user interaction may find that MVC doesn't quite fit the bill here as solving this problem may mean heavily relying on multiple controllers. In MVP, all of this complex logic can be encapsulated in a presenter, which can simplify maintenance greatly.

As MVP views are defined through an interface and the interface is technically the only point of contact between the system and the view (other than a presenter), this pattern also allows developers to write presentation logic without needing to wait for designers to produce layouts and graphics for the application.

Depending on the implementation, MVP may be more easy to automatically unit test than MVC. The reason often cited for this is that the presenter can be used as a complete mock of the user-interface and so it can be unit tested independent of other components. In my experience this really depends on the languages you are implementing MVP in (there's quite a difference between opting for MVP for a JavaScript project over one for say, ASP.NET).

At the end of the day, the underlying concerns you may have with MVC will likely hold true for MVP given that the differences between them are mainly semantic. As long as you are cleanly separating concerns into models, views and controllers (or presenters) you should be achieving most of the same benefits regardless of the pattern you opt for.

MVC, MVP and Backbone.js

There are very few, if any architectural JavaScript frameworks that claim to implement the MVC or MVP patterns in their classical form as many JavaScript

developers don't view MVC and MVP as being mutually exclusive (we are actually more likely to see MVP strictly implemented when looking at web frameworks such as ASP.NET or GWT). This is because it's possible to have additional presenter/view logic in your application and yet still consider it a flavor of MVC.

Backbone contributor [Irene Ros](#) subscribes to this way of thinking as when she separates Backbone views out into their own distinct components, she needs something to actually assemble them for her. This could either be a controller route (such as a `Backbone.Router`) or a callback in response to data being fetched.

That said, some developers do however feel that Backbone.js better fits the description of MVP than it does MVC. Their view is that:

- The presenter in MVP better describes the `Backbone.View` (the layer between View templates and the data bound to it) than a controller does
- The model fits `Backbone.Model` (it isn't that different from the classical MVC "Model")
- The views best represent templates (e.g Handlebars/Mustache markup templates)

A response to this could be that the view can also just be a View (as per MVC) because Backbone is flexible enough to let it be used for multiple purposes. The V in MVC and the P in MVP can both be accomplished by `Backbone.View` because they're able to achieve two purposes: both rendering atomic components and assembling those components rendered by other views.

We've also seen that in Backbone the responsibility of a controller is shared with both the `Backbone.View` and `Backbone.Router` and in the following example we can actually see that aspects of that are certainly true.

Here, our Backbone `TodoView` uses the Observer pattern to 'subscribe' to changes to a View's model in the line `this.model.on('change', ...)`. It also handles templating in the `render()` method, but unlike some other implementations, user interaction is also handled in the View (see `events`).

```
// The DOM element for a todo item...
app.TodoView = Backbone.View.extend({

  //... is a list tag.
  tagName: 'li',

  // Pass the contents of the todo template through a templating
  // function, cache it for a single todo
  template: _.template( $('#item-template').html() ),
```

```

// The DOM events specific to an item.
events: {
  'click .toggle': 'togglecompleted'
},

// The TodoView listens for changes to its model, re-rendering. Since there's
// a one-to-one correspondence between a Todo and a TodoView in this
// app, we set a direct reference on the model for convenience.
initialize: function() {
  this.model.on( 'change', this.render, this );
  this.model.on( 'destroy', this.remove, this );
},

// Re-render the titles of the todo item.
render: function() {
  this.$el.html( this.template( this.model.toJSON() ) );
  return this;
},

// Toggle the "completed" state of the model.
togglecompleted: function() {
  this.model.toggle();
},
});

```

Another (quite different) opinion is that Backbone more closely resembles [Smalltalk-80 MVC](#), which we went through earlier.

As MarionetteJS author Derick Bailey has [written](#), it's ultimately best not to force Backbone to fit any specific design patterns. Design patterns should be considered flexible guides to how applications may be structured and in this respect, Backbone doesn't fit either MVC nor MVP perfectly. Instead, it borrows some of the best concepts from multiple architectural patterns and creates a flexible framework that just works well. Call it **the Backbone way**, MV* or whatever helps reference its flavor of application architecture.

It *is* however worth understanding where and why these concepts originated, so I hope that my explanations of MVC and MVP have been of help. Most structural JavaScript frameworks will adopt their own take on classical patterns, either intentionally or by accident, but the important thing is that they help us develop applications which are organized, clean and can be easily maintained.

Namespacing

When learning how to use Backbone, an important and commonly overlooked area by tutorials is namespacing. If you already have experience with names-

spacing in JavaScript, the following section will provide some advice on how to specifically apply concepts you know to Backbone, however I will also be covering explanations for beginners to ensure everyone is on the same page.

What is namespacing? Namespacing is a way to avoid collisions with other objects or variables in the global namespace. Using namespacing reduces the potential of your code breaking because another script on the page is using the same variable names that you are. As a good ‘citizen’ of the global namespace, it’s also imperative that you do your best to minimize the possibility of your code breaking other developer’s scripts.

JavaScript doesn’t really have built-in support for namespaces like other languages, however it does have closures which can be used to achieve a similar effect.

In this section we’ll be taking a look shortly at some examples of how you can namespace your models, views, routers and other components. The patterns we’ll be examining are:

- Single global variables
- Object Literals
- Nested namespacing

Single global variables

One popular pattern for namespacing in JavaScript is opting for a single global variable as your primary object of reference. A skeleton implementation of this where we return an object with functions and properties can be found below:

```
var myApplication = (function(){
  function(){
    // ...
  },
  return {
    // ...
  }
})();
```

You’ve probably seen this technique before. A Backbone-specific example might look like this:

```
var myViews = (function(){
  return {
    TodoView: Backbone.View.extend({ .. }),
    TodosView: Backbone.View.extend({ .. }),
```

```

        AboutView: Backbone.View.extend({ .. });
        //etc.
    };
  })();

```

Here we can return a set of views, but the same technique could return an entire collection of models, views and routers depending on how you decide to structure your application. Although this works for certain situations, the biggest challenge with the single global variable pattern is ensuring that no one else has used the same global variable name as you have in the page.

One solution to this problem, as mentioned by Peter Michaux, is to use prefix namespacing. It's a simple concept at heart, but the idea is you select a common prefix name (in this example, `myApplication_`) and then define any methods, variables or other objects after the prefix.

```

var myApplication_todoView = Backbone.View.extend({}),
    myApplication_todosView = Backbone.View.extend({});

```

This is effective from the perspective of trying to lower the chances of a particular variable existing in the global scope, but remember that a uniquely named object can have the same effect. This aside, the biggest issue with the pattern is that it can result in a large number of global objects once your application starts to grow.

For more on Peter's views about the single global variable pattern, read his [excellent post on them](#).

Note: There are several other variations on the single global variable pattern out in the wild, however having reviewed quite a few, I felt the prefixing approach applied best to Backbone.

Object Literals

Object Literals have the advantage of not polluting the global namespace but assist in organizing code and parameters logically. They're beneficial if you wish to create easily readable structures that can be expanded to support deep nesting. Unlike simple global variables, Object Literals often also take into account tests for the existence of a variable by the same name, which helps reduce the chances of collision.

This example demonstrates two ways you can check to see if a namespace already exists before defining it. I commonly use Option 2.

```

/* Doesn't check for existence of myApplication */
var myApplication = {};

/*

```

Does check for existence. If already defined, we use that instance.

Option 1: if(!myApplication) myApplication = {};

Option 2: var myApplication = myApplication || {};

We can then populate our object literal to support models, views and collections (or any data structure).

```
var myApplication = {
  models : {},
  views : {
    pages : {}
  },
  collections : {}
};
```

One can also opt for adding properties directly to the namespace (such as your views, in the following example):

```
var myTodosViews = myTodosViews || {};
myTodosViews.todoView = Backbone.View.extend({});
myTodosViews.todosView = Backbone.View.extend({});
```

The benefit of this pattern is that you're able to easily encapsulate all of your models, views, routers etc. in a way that clearly separates them and provides a solid foundation for extending your code.

This pattern has a number of benefits. It's often a good idea to decouple the default configuration for your application into a single area that can be easily modified without the need to search through your entire codebase just to alter it. Here's an example of a hypothetical object literal that stores application configuration settings:

```
var myConfig = {
  language: 'english',
  defaults: {
    enableDelegation: true,
    maxTodos: 40
  },
  theme: {
    skin: 'a',
    toolbars: {
      index: 'ui-navigation-toolbar',
      pages: 'ui-custom-toolbar'
    }
  }
}
```


Note that there are really only minor syntactical differences between the Object Literal pattern and a standard JSON data set. If for any reason you wish to use JSON for storing your configurations instead (e.g. for simpler storage when sending to the back-end), feel free to.

For more on the Object Literal pattern, I recommend reading Rebecca Murphey's [excellent article on the topic](#).

Nested namespacing

An extension of the Object Literal pattern is nested namespacing. It's another common pattern used that offers a lower risk of collision due to the fact that even if a top-level namespace already exists, it's unlikely the same nested children do. For example, Yahoo's YUI uses the nested object namespacing pattern extensively:

```
YAHOO.util.Dom.getElementsByClassName('test');
```

Yahoo's YUI uses the nested object namespacing pattern regularly and even DocumentCloud (the creators of Backbone) use the nested namespacing pattern in their main applications. A sample implementation of nested namespacing with Backbone may look like this:

```
var todoApp = todoApp || {};  
  
// perform similar check for nested children  
todoApp.routers = todoApp.routers || {};  
todoApp.model = todoApp.model || {};  
todoApp.model.special = todoApp.model.special || {};  
  
// routers  
todoApp.routers.Workspace = Backbone.Router.extend({});  
todoApp.routers.TODOSearch = Backbone.Router.extend({});  
  
// models  
todoApp.model.TODO = Backbone.Model.extend({});  
todoApp.model.Notes = Backbone.Model.extend({});  
  
// special models  
todoApp.model.special.Admin = Backbone.Model.extend({});
```

This is readable, clearly organized, and is a relatively safe way of namespacing your Backbone application. The only real caveat however is that it requires your browser's JavaScript engine to first locate the todoApp object, then dig down until it gets to the function you're calling. However, developers such as Juriy Zaytsev (kangax) have tested and found the performance differences between single object namespacing vs the 'nested' approach to be quite negligible.

What does DocumentCloud use?

In case you were wondering, here is the original DocumentCloud (remember those guys that created Backbone?) workspace that uses namespacing in a necessary way. This approach makes sense as their documents (and annotations and document lists) are embedded on third-party news sites.

```
// Provide top-level namespaces for our javascript.
(function() {
  window.dc = {};
  dc.controllers = {};
  dc.model = {};
  dc.app = {};
  dc.ui = {};
})();
```

As you can see, they opt for declaring a top-level namespace on the `window` called `dc`, a short-form name of their app, followed by nested namespaces for the controllers, models, UI and other pieces of their application.

Recommendation

Reviewing the namespace patterns above, the option that I prefer when writing Backbone applications is nested object namespacing with the object literal pattern.

Single global variables may work fine for applications that are relatively trivial. However, larger codebases requiring both namespaces and deep sub-namespaces require a succinct solution that's both readable and scalable. I feel this pattern achieves both of these objectives and is a good choice for most Backbone development.

Backbone Dependency Details

The following sections provide insight into how Backbone uses jQuery/Zepto and Underscore.js.

DOM Manipulation

Although most developers won't need it, Backbone does support setting a custom DOM library to be used instead of these options. From the source:

```
// Set the JavaScript library that will be used for DOM manipulation and
// Ajax calls (a.k.a. the '$' variable). By default Backbone will use: jQuery,
```

```
// Zepto, or Ender; but the 'setDomLibrary()' method lets you inject an
// alternate JavaScript library (or a mock library for testing your views
// outside of a browser).
```

```
Backbone.setDomLibrary = function(lib) {
  $ = lib;
};
```

Calling this method will allow you to use any custom DOM-manipulation library.
e.g:

```
Backbone.setDomLibrary(aCustomLibrary);
```

Utilities

Underscore.js is heavily used in Backbone behind the scenes for everything from object extension to event binding. As the entire library is generally included, we get free access to a number of useful utilities we can use on Collections such as filtering `_.filter()`, sorting `_.sortBy()`, mapping `_.map()` and so on.

From the source:

```
// Underscore methods that we want to implement on the Collection.
var methods = ['forEach', 'each', 'map', 'reduce', 'reduceRight', 'find',
  'detect', 'filter', 'select', 'reject', 'every', 'all', 'some', 'any',
  'include', 'contains', 'invoke', 'max', 'min', 'sortBy', 'sortedIndex',
  'toArray', 'size', 'first', 'initial', 'rest', 'last', 'without', 'indexOf',
  'shuffle', 'lastIndexOf', 'isEmpty', 'groupBy'];

// Mix in each Underscore method as a proxy to Collection#models.
_.each(methods, function(method) {
  Collection.prototype[method] = function() {
    return _[method].apply(_, [this.models].concat(_.toArray(arguments)));
  };
});
```

However, for a complete linked list of methods supported, see the [official documentation](#).

RESTful persistence

Models and collections in Backbone can be “sync”ed with the server using the `fetch`, `save` and `destroy` methods. All of these methods delegate back to the `Backbone.sync` function, which actually wraps jQuery/Zepto’s `$.ajax` function,

calling GET, POST and DELETE for the respective persistence methods on Backbone models.

From the the source for `Backbone.sync`:

```
var methodMap = {
  'create': 'POST',
  'update': 'PUT',
  'patch': 'PATCH',
  'delete': 'DELETE',
  'read': 'GET'
};

Backbone.sync = function(method, model, options) {
  var type = methodMap[method];

  // ... Followed by lots of Backbone.js configuration, then..

  // Make the request, allowing the user to override any Ajax options.
  var xhr = options.xhr = Backbone.ajax(_.extend(params, options));
  model.trigger('request', model, xhr, options);
  return xhr;
};
```

Routing

Calls to `Backbone.History.start` rely on jQuery/Zepto binding `popState` or `hashchange` event listeners back to the window object.

From the source for `Backbone.history.start`:

```
// Depending on whether we're using pushState or hashes, and whether
// 'onhashchange' is supported, determine how we check the URL state.
if (this._hasPushState) {
  Backbone.$(window).on('popstate', this.checkUrl);
} else if (this._wantsHashChange && ('onhashchange' in window) && !oldIE) {
  Backbone.$(window).on('hashchange', this.checkUrl);
}
...

```

`Backbone.History.stop` similarly uses your DOM manipulation library to unbind these event listeners.

Upgrading to Backbone 0.9.10

Developing Backbone.js Applications is currently based on Backbone 0.9.10. If you are transitioning from 0.9.2 to 0.9.10 or above, the following is a guide of [changes](#) grouped by classes, where applicable.

Note: We aim to update the entirety of this book to Backbone 1.0 once it has been tagged.

Model

- Model validation is now only enforced by default in `Model#save` and is no longer enforced by default upon construction or in `Model#set`, unless the `{validate:true}` option is passed:

```
var model = new Backbone.Model({name: "One"});
model.validate = function(attrs) {
  if (!attrs.name) {
    return "No thanks.";
  }
};
model.set({name: "Two"});
console.log(model.get('name'));
// 'Two'
model.unset('name', {validate: true});
// false
```

- Passing `{silent:true}` on change will no longer delay individual "change:attr" events, instead they are silenced entirely.

```
var model = new Backbone.Model();
model.set({x: true}, {silent: true});

console.log(!model.hasChanged(0));
// true
console.log(!model.hasChanged(''));
// true
```

- The `Model#change` method has been removed, as delayed attribute changes are no longer available.
- Calling `destroy` on a Model will now return `false` if the model is new.

```
var model = new Backbone.Model();
console.log(model.destroy());
// false
```

- After fetching a model or a collection, all defined parse functions will now be run. So fetching a collection and getting back new models could cause both the collection to parse the list, and then each model to be parsed in turn, if you have both functions defined.

- HTTP PATCH support allows us to send only changed attributes (i.e partial updates) to the server by passing `{patch: true}` i.e `model.save(attrs, {patch: true})`.

```
// Save partial using PATCH
model.clear().set({id: 1, a: 1, b: 2, c: 3, d: 4});
model.save();
model.save({b: 2, d: 4}, {patch: true});
console.log(this.syncArgs.method);
// 'patch'
```

- When using `add` on a collection, passing `{merge: true}` will now cause duplicate models to have their attributes merged in to the existing models, instead of being ignored.

```
var items = new Backbone.Collection;
items.add([{ id : 1, name: "Dog" , age: 3}, { id : 2, name: "cat" , age: 2}]);
items.add([{ id : 1, name: "Bear" }], {merge: true });
items.add([{ id : 2, name: "lion" }]); // merge: false

console.log(JSON.stringify(items.toJSON()));
// [{"id":1,"name":"Bear","age":3},{id":2,"name":"cat","age":2}]
```

Collection

- While listening to a `reset` event, the list of previous models is now available in `options.previousModels`, for convenience.

```
var model = new Backbone.Model();
var collection = new Backbone.Collection([model])
.on('reset', function(collection, options) {
  console.log(options.previousModels);
  console.log([model]);
  console.log(options.previousModels[0] === model); // true
});
collection.reset([]);
```

- `Collection#sort` now triggers a `sort` event, instead of a `reset` event.
- Removed `getByCid` from Collections. `collection.get` now supports lookup by both `id` and `cid`.
- Collections now also proxy Underscore method name aliases (`collect`, `inject`, `foldl`, `foldr`, `head`, `tail`, `take`, and so on...)

- Added `update` (which is also available as an option to `fetch`) for “smart” updating of sets of models.

The `update` method attempts to perform smart updating of a collection using a specified list of models. When a model in this list isn’t present in the collection, it is added. If it is, its attributes will be merged. Models which are present in the collection but not in the list are removed.

```
var theBeatles = new Collection(['john', 'paul', 'george', 'ringo']);

theBeatles.update(['john', 'paul', 'george', 'pete']);

// Fires a 'remove' event for 'ringo', and an 'add' event for 'pete'.
// Updates any of john, paul and george's attributes that may have
// changed over the years.
```

- `collection.indexOf(model)` can be used to retrieve the index of a model as necessary.

```
var col = new Backbone.Collection;

col.comparator = function(a, b) {
  return a.get('name') < b.get('name') ? -1 : 1;
};

var tom = new Backbone.Model({name: 'Tom'});
var rob = new Backbone.Model({name: 'Rob'});
var tim = new Backbone.Model({name: 'Tim'});

col.add(tom);
col.add(rob);
col.add(tim);

console.log(col.indexOf(rob) === 0); // true
console.log(col.indexOf(tim) === 1); // true
console.log(col.indexOf(tom) === 2); // true
```

View

- `View#make` has been removed. You’ll need to use `$` directly to construct DOM elements now.
- When declaring a View, `options`, `el`, `tagName`, `id` and `className` may now be defined as functions, if you want their values to be determined at runtime.

Events

- Backbone events now support jQuery-style event maps `obj.on({click: action})`. This is clearer than needing three separate calls to `.on` and should align better with the events hash used in Views:

```
model.on({
  'change:name' : this.nameChanged,
  'change:age' : this.ageChanged,
  'change:height' : this.heightChanges
});
```

- The Backbone object now extends Events so that you can use it as a global event bus, if you like.
- Backbone events now supports `once`, similar to Node's `once`, or jQuery's `one`. A call to `once()` ensures that the callback only fires once when a notification arrives.

```
// Use once rather than having to explicitly unbind
var obj = { counterA: 0, counterB: 0 };
_.extend(obj, Backbone.Events);

var incrA = function(){ obj.counterA += 1; obj.trigger('event'); };
var incrB = function(){ obj.counterB += 1; };

obj.once('event', incrA);
obj.once('event', incrB);
obj.trigger('event');

console.log(obj.counterA === 1); // true
console.log(obj.counterB === 1); // true
```

`counterA` and `counterB` should only have been incremented once.

- Added `listenTo` and `stopListening` to Events. They can be used as inversion-of-control flavors of `on` and `off`, for convenient unbinding of all events an object is currently listening to. `view.remove()` automatically calls `view.stopListening()`.

If you've had a chance to work on a few Backbone projects by this point, you may know that every `on` called on an object also requires an `off` to be called in order for the garbage collector to do its job.

This can sometimes be overlooked when Views are binding to Models. In 0.9.10, this can now be done the other way around - Views can bind to Model

notifications and unbind from all of them with just one call. We achieve this using `view.listenTo(model, 'eventName', func)` and `view.stopListening()`.

The default `remove()` of Views will call `stopListening()` for you, just in case you don't remember to.

```
var a = _.extend({}, Backbone.Events);
var b = _.extend({}, Backbone.Events);
a.listenTo(b, 'all', function(){ console.log(true); });
b.trigger('anything');
a.listenTo(b, 'all', function(){ console.log(false); });
a.stopListening();
b.trigger('anything');
```

A more complex example (from [Just JSON](#)) might require our Views to respond to “no connection” and “connection resume” events to re-fetch data on demand in an application.

In 0.9.2, we have to do this to achieve what we need:

```
// In BaseView definition
var BaseView = Backbone.View.extend({
  destroy: function() {
    // Allow child views to hook to this event to unsubscribe
    // anything they may have subscribed to to other objects.
    this.trigger('beforedestroy');
    if (this.model) {
      this.model.off(null, null, this);
    }

    if (this.collection) {
      this.collection.off(null, null, this);
    }

    this.remove();
    this.unbind();
  }
});

// In MyView definition.
// We have a global EventBus that allows elements on the app to subscribe to global events.
// connection/disconnected, connection/resume is two of them.

var MyView = BaseView.extend({
  initialize: function() {
    this.on('beforedestroy', this.onBeforeDestroy, this);
```

```

        this.model.on('reset', this.onModelLoaded, this);
        EventBus.on('connection/disconnected', this.onDisconnect, this);
        EventBus.on('connection/resume', this.onConnectionResumed, this);
    },
    onModelLoaded: function() {
        // We only need this to be done once! (Kinda weird...)
        this.model.off('load', this.onModelLoaded, this);
    },
    onDisconnect: function() {
        // Figure out what state we are currently on, display View-specific messaging, etc.
    },
    onConnectionResumed: function() {
        // Re-do previous network request that failed.
    },
    onBeforeDestroy: function() {
        EventBus.off('connection/resume', this.onConnectionResumed, this);
        EventBus.off('connection/disconnected', this.onDisconnect, this);
    }
});

```

However, in 0.9.10, what we need to do is quite simple:

```

// In BaseView definition
var BaseView = Backbone.View.extend({
    destroy: function() {
        this.trigger('beforedestroy');
        this.remove();
    }
});

// In MyView definition.

var MyView = BaseView.extend({
    initialize: function() {
        this.listenTo(EventBus, 'connection/disconnected', this.onDisconnect);
        this.listenTo(EventBus, 'connection/resume', this.onConnectionResumed);
        this.once(this.model, 'load', this.onModelLoaded);
    },
    onModelLoaded: function() {
        // Don't need to unsubscribe anymore!
    },
    onDisconnect: function() {
        // Figure out the state, display messaging, etc.
    },

```

```

onConnectionResumed: function() {
  // Re-do previous network request that failed.
}
// Most importantly, we no longer need onBeforeDestroy() anymore!
});

```

Routers

- A “route” event is triggered on the router in addition to being fired on Backbone.history.

```

Backbone.history.on('route', onRoute);

// Trigger 'route' event on router instance."
router.on('route', function(name, args) {
  console.log(name === 'routeEvent');
});

location.replace('http://example.com#route-event/x');
Backbone.history.checkUrl();

```

- For semantic and cross browser reasons, routes will now ignore search parameters. Routes like `search?query=...&page=3` should become `search/.../3`.
- Bugfix for normalizing leading and trailing slashes in the Router definitions. Their presence (or absence) should not affect behavior.
- Router URLs now support optional parts via parentheses, without having to use a regex.

```

var Router = Backbone.Router.extend({
  routes: {
    "optional(/:item)": "optionalItem",
    "named/optional/(y:z)": "namedOptionalItem",
  },
  ...
});

```

Sync

- For mixed-mode APIs, Backbone.sync now accepts `emulateHTTP` and `emulateJSON` as inline options.

```

var Library = Backbone.Collection.extend({
  url : function() { return '/library'; }
});

```

```

});

var attrs = {
  title : "The Tempest",
  author : "Bill Shakespeare",
  length : 123
};

library = new Library;
library.create(attrs, {wait: false});

// update with just emulateHTTP
library.first().save({id: '2-the-tempest', author: 'Tim Shakespeare'}, {
  emulateHTTP: true
});

console.log(this.ajaxSettings.url === '/library/2-the-tempest'); // true
console.log(this.ajaxSettings.type === 'POST'); // true
console.log(this.ajaxSettings.contentType === 'application/json'); // true

var data = JSON.parse(this.ajaxSettings.data);
console.log(data.id === '2-the-tempest');
console.log(data.author === 'Tim Shakespeare');
console.log(data.length === 123);

// or update with just emulateJSON

library.first().save({id: '2-the-tempest', author: 'Tim Shakespeare'}, {
  emulateJSON: true
});

console.log(this.ajaxSettings.url === '/library/2-the-tempest'); // true
console.log(this.ajaxSettings.type === 'PUT'); // true
console.log(this.ajaxSettings.contentType === 'application/x-www-form-urlencoded'); // true

var data = JSON.parse(this.ajaxSettings.data.model);
console.log(data.id === '2-the-tempest');
console.log(data.author === 'Tim Shakespeare');
console.log(data.length === 123);

```

- Consolidated "sync" and "error" events within Backbone.sync. They are now triggered regardless of the existence of success or error callbacks.
- Added a "request" event to Backbone.sync, which triggers whenever a request begins to be made to the server. The natural complement to the "sync" event.

Other

- Bug fix on change where attribute comparison uses `!==` instead of `_.isEqual`.
- Bug fix where an empty response from the server on save would not call the success function.
- To improve the performance of add, `options.index` will no longer be set in the `add` event callback.
- Removed the `Backbone.wrapError` helper method. Overriding `sync` should work better for those particular use cases.
- To set what library Backbone uses for DOM manipulation and Ajax calls, use `Backbone.$ = ...` instead of `setDomLibrary`.
- Added a `Backbone.ajax` hook for more convenient overriding of the default use of `$.ajax`. If AJAX is too passé, set it to your preferred method for server communication.
- Validation now occurs even during `"silent"` changes. This change means that the `isValid` method has been removed. Failed validations also trigger an error, even if an error callback is specified in the options.

Where relevant, copyright Addy Osmani, 2012-2013.