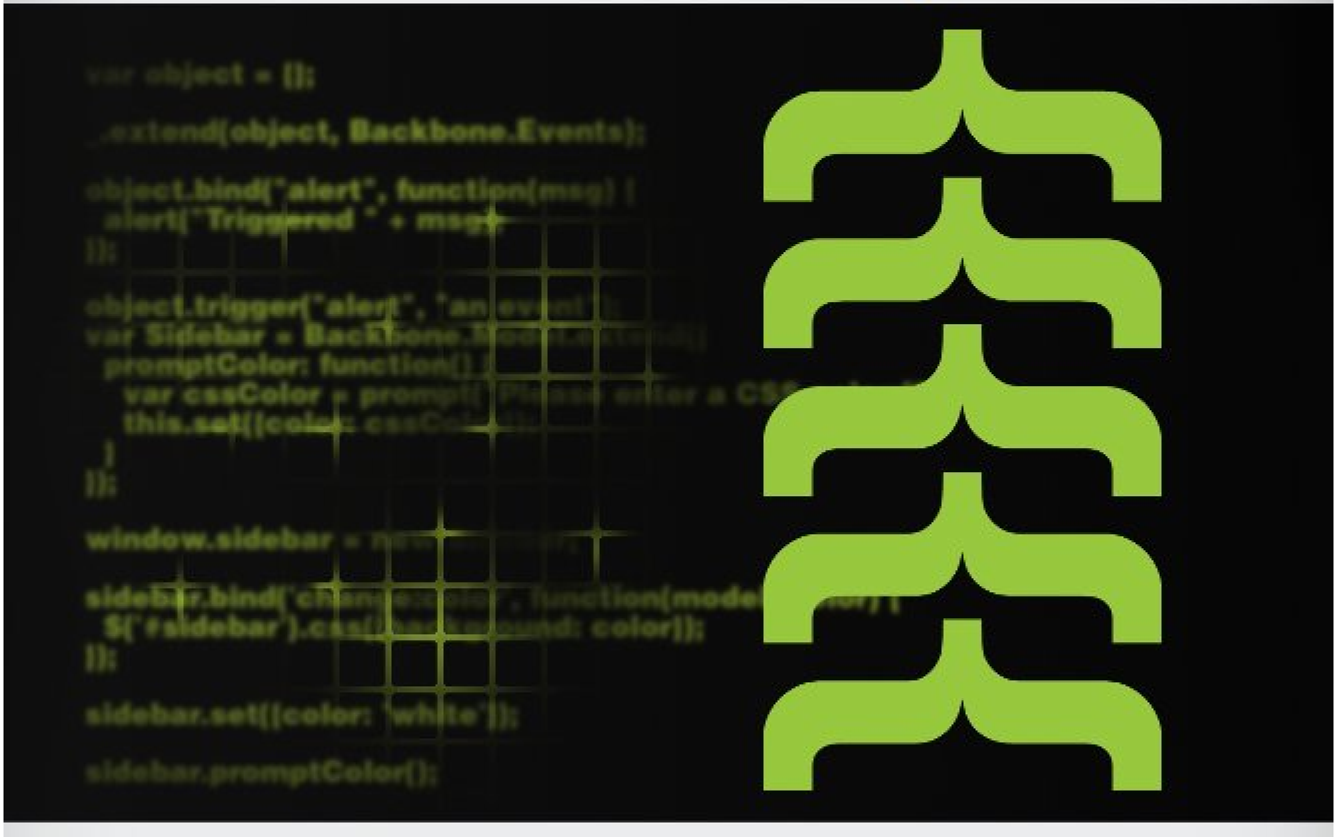
RECIPES WITH BACKBONE

Strategies to accelerate development with Backbone.js



Nick Gauthier & Chris Strom

Recipes with Backbone

Nick Gauthier and Chris Strom

Recipes with Backbone

Nick Gauthier and Chris Strom

Table of Contents

History

- 2011-09-30: Initial alpha release
- 2011-10-06: New recipes: "Non-REST Model" and "Reduced Models and Collections"
- 2011-10-19: New recipes: "Constructor Route" and "Router Redirection". Minor copy edits.
- 2011-10-24: New recipes: "Changes Feed" and "Object References in Backbone".
- 2011-10-31: **Beta.** New recipes: "Underscore Templates", "Pagination and Search", "Evented Routes", and "Custom Events". Converted all CoffeeScript code samples to Javascript. Many corrections (thanks to Ben Morris, Geoffrey Grosenbach).
- 2011-11-30: **1.0** New intro chapters and an appendix on testing with Jasmine. Many, many corrections.
- 2011-12-02: Fix a couple of typos in the introduction chapters (thanks to Luigi Montanez).
- 2011-12-07: More typos / grammar corrections (thanks Martin Harrigan, "jdkealy", and Bob Spryn).
- 2011-12-31: Ensure that code blocks do not split across pages. Tweak to better support Kindle Fire. Additional typo fixes. Much thanks to David Mosher and "simax" for pointing these out.
- 2012-01-02: New Require.js recipe. Sample code fix (de-ruby-ify some Javascript)—thanks to "simax" for identifying the problem.
- 2012-01-18: New Jasmine strategies recipe. Typo fix in the underscore recipe—thanks "Hudon689" for the keen eye.

- 2012-08-24: Updates for Backbone.js 0.9. Many errata fixes—big thanks to Dennis Reimann, Joe Ellis, and "redthor".
- 2012-09-26: Fix typo. Thanks to Wlodek Bzyl for pointing it out.
- 2013-05-01: Updates for Backbone.js 1.0. Updates to the testing chapter. Errata fixes: thanks to Stephen Deese.

Introduction

During its brief existence, Backbone.js has enjoyed tremendous popularity from the web development community. As one of the first "micro-frameworks" to hit the scene, it captured the hearts and minds of developers that had previously been struggling with much heavier frameworks—many of which had a learning curve similar to learning a new programming language.

A significant appeal of Backbone is how small it is. A seasoned developer can pick it up in a day and start cranking out robust code in no time. Its parent organization, DocumentCloud [1], has outstanding documentation and has collected a nice set of tutorial applications. This low barrier to entry coupled with significant power is compelling.

Another appeal is how very agnostic Backbone is about, well… everything. It makes no assumptions about templating libraries that will populate the UI of the application. It defaults to persisting data over a REST layer, but even that is easy to swap out. The power afforded by Backbone, comes from the application structure chosen (models, collections of models, and views) and the convention it describes for different concepts to interact with each other.

The downside of such a simple and agnostic framework is that it can be easy to take approaches that end up being less than ideal as time goes by. We know. We have made these mistakes and have felt the pain of ripping significant chunks of code out so that we could better leverage the browser, the network or the datastore.

In this book, you will find a collection of the strategies that we have found to be most effective. We will discuss the situations in which they apply and how our initial attempts at solving problems common to all Backbone applications failed and why these recipes have been successful.

[1] http://www.documentcloud.org/

1. Who Should Read this Book

This book is not meant as an introduction to Backbone.js. We will provide a quick introduction, but only enough to provide foundation for many of the recipes later in the book. For a solid introduction to Backbone.js, see the online documentation [2]. It is *excellent*. Or better yet, read the source code. It is very approachable Javascript code and is, as you should come to expect of Backbone.js, self-documented very nicely.

This book also assumes a fair level of Javascript knowledge. If you have read "Javascript: The Good Parts" [3], you should be in good shape. If not, do it—it is a small book with excellent discussion of what makes Javascript such a nice language.

2. Some Notes from the Upgrade to 1.x

When we updated this book for Backbone 1.x, we had to make very few changes. This is a huge testament to the Backbone.js maintainers and to the overall stability of the project. Even a major change, like moving from what might be a very unstable 0.5 version all the way up to 0.9, resulted in a remarkably unchanged API. This bodes well for committing applications and development efforts to Backbone.js.

In fact, all of the changes that we made were either optimizations, adopting a new best practice, or preparing for a deprecation. Among the changes:

- We favor `this.listenTo(object)` over the `object.on()` equivalent for better memory management.

[2]http://documentcloud.github.com/backbone/
[3]"Javascript the Good Parts": http://shop.oreilly.com/product/9780596517748.do

- We make use of the new `this.$el` convenience property instead of wrapping each lookup in a `$(this.el)`.
- Since they are stripped anyway, we removed leading slashes from routes.
- In some of our recipes, we change the URL and trigger the route method. In these cases, we now include the new `{trigger: true}` option.
- Instead of the old two-step process to bind event methods, we have switched to the single step that comes with the new `on()` method. That is, instead of: `_.bindAll(this, 'method'); this.bind('event', this.method)`, we are now using: `this.on('event', this.method, this)`.
- When initializing new, empty collections, we use `null` rather than `[]` to avoid a subsequent `reset()`.

That's it!

Aside from those changes, all of the same Backbone goodness that we have come to know and love remains the same.

3. Contact Us

If you have thoughts or suggestions, we would love to hear from you!

If you find any mistakes or have any suggestions, *please* do not hesitate to let us know by adding an item to our TODO list (https://github.com/recipeswithbackbone/recipeswithbackbone.github.com/issues) or by dropping us a line at `errata@recipeswithbackbone.com`.

We will update the mailing list whenever a new version is ready to be downloaded [4], so make sure that you are subscribed.

[4]This book's mailing list: http://eepurl.com/fqMy2

4. How this Book is Organized

We have structured this book so that concepts are introduced from the bottom-up.

We start with a brief introduction to client-side development in the days prior to Backbone (Chapter 1, *Writing Client Side Apps (Without Backbone)*) and then discuss how our poor application might be better served by Backbone (Chapter 2, *Writing Backbone Applications*). These introductory chapters also serve to introduce the sample application with which we will work through many of the recipes. If you are an experienced Backbone.js coder, you can safely skip these introductory chapters.

Next come some "fundamentals" recipes. The first two describe different strategies for Backbone.js organization. Chapter 3, *Namespacing*, is intended for smaller applications. The next, Chapter 4, *Organizing with Require.js*, introduces the very powerful require.js library as an effective means for working with larger Backbone.js codebases. Last up in this section is Chapter 5, *View Templates with Underscore.js*, which introduces the surprisingly powerful built-in templating tool.

With the preliminaries out of the way, we dive into Backbone.js view objects, which is where a surprising amount of action takes place. First up is Chapter 6, *Instantiated View*, which is useful when views only need to be created once. Next is Chapter 7, *Collection View*, which is essential for working with collections. Then we move into a couple of performance optimization recipes: Chapter 8, *View Signature* and Chapter 9, *Fill-In Rendering*. We finish up views with a little eye candy: Chapter 10, *Actions and Animations*.

The next section of the book contains recipes for working with models and collections. First up is an interesting little recipe describing how to work with statistical and aggregating objects: Chapter 11, *Reduced Models and Collections*. Following that is Chapter 12, *Non-REST Models*, which introduces working with legacy server code (sadly it is quite useful). Next comes Chapter 13, *Changes Feed*, which gives some nice tips on how to make your Backbone applications even more dynamic. Lastly is Chapter 14, *Pagination and Search*.

In the routing section of the book, we start off with Chapter 15, *Constructor Route*, which describes an interesting little pattern that can significantly decrease the amount of code required in your Backbone applications. Next comes the Chapter 16, *Router Redirection* which serves up some tricks for implementing redirection-like behaviors in Backbone. Last up is Chapter 17, *Evented Routers* which similarly discusses strategies for keeping your routes DRY.

We finish up the book with two recipes that did not quite fit anywhere else, but are definite must-reads. First is Chapter 18, *Object References in Backbone*. If you read nothing else in this book, read this as it gives a top-down philosophy for building Backbone.js applications that will be applicable almost anywhere. We finish up with a discussion of Chapter 19, *Custom Events*.

If you are still hungry for more, dig into our appendices where we discuss Appendix A, *Getting Started with Jasmine*.

Excited? Let's get started!

Chapter 1. Writing Client Side Apps (Without Backbone)

Before jumping into Backbone.js development, let's take a stroll through life without it. This is not meant to serve as a straw man argument so that in the end, we can jump up and say "look how awesome Backbone.js is!" To be sure, Backbone.js is awesome, but this exercise is meant to give you an idea of where Backbone provides structure. Once we have made it through this exercise, we will be left with a number of questions as to what the next steps should be. Without backbone, these questions would be left to us to answer.

For most of the book, we are going to be discussing Backbone in relationship to a Calendaring application. Here, we will try to get a month view up and running using nothing but server-side code and jQuery. Surely we can do this—our forefathers have been doing this kind of thing for dozens of months.

For our purposes, let's assume that the server is responsible for drawing the HTML of the calendar itself, while the client must make a call to a web service to load appointment for that calendar. Sure, this is a conceit, but it is a conceit born of a thousand implementations in the wild.

1.1. Working with Dates

This is not news, but working with dates in Javascript is not pleasant. We will keep it to a minimum, in part by using the ISO 8601 date format [1]. ISO 8601 date/times take the form of "YYYY-MM-DD HH:MM:SS TZ" [2]. The date that the first edition of this book was published can be represented as "2011-11-30".

[1] http://en.wikipedia.org/wiki/ISO_8601
[2] The official ISO 8601 representation of a datetime includes a `T` in between the date and the time (`2011-11-30T23:59:59`). We prefer omitting the `T` to aid in human readability without degrading machine parsing (`2011-11-30 23:59:59`)

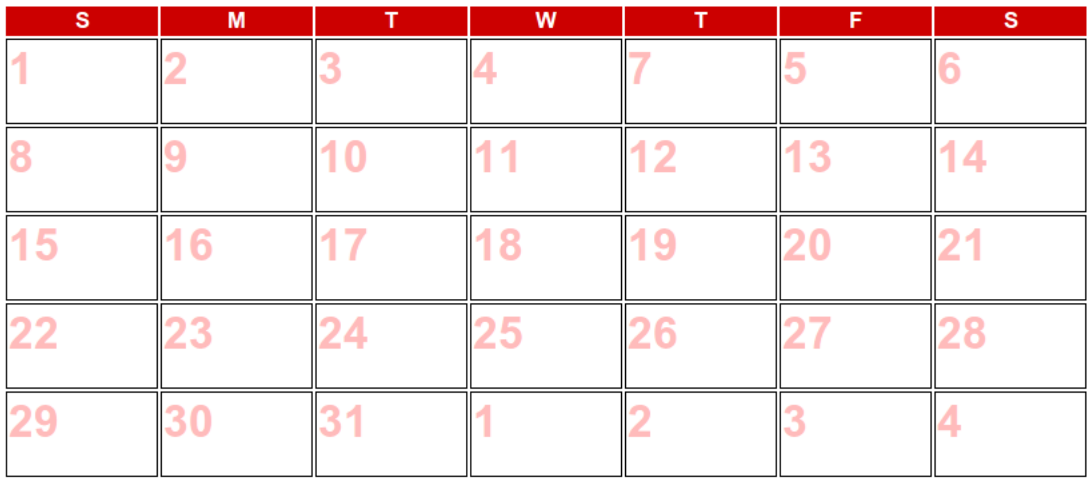
The brilliant simplicity of ISO 8601 is that anyone can read it—even Americans who tend to represent date in nonsensical order. There is no doubt that `2011-11-12` represents the 12th of November, whereas Americans think that `12/11/2011` is the 11th of December, the civilized world know this to be the 12th day of the 11th month of 2011. Reading dates when the units increase or decrease from left-to-right just makes sense.

It even makes sense to a machine since, although `"2011-11-12"` and `"2011-11-30"` are strings, they can still be compared by any programming language. Machines simply compare the two as strings. Since the "2" and "2" are the same, it compares the next two characters in the string (both "0"). Eventually, the "3" and "1" are reached in the days of the month place. Since the character "3" is greater than the character "1", the following would be true regardless of language: `"2011-11-30" > "2011-11-12"`.

Armed with that knowledge, we make the ID element of the table cells ISO 8601 dates, corresponding to the date that the cell represents.

```
<table>
  <tr>
    <th>S</th><th>M</th><th>T</th><th>W</th>
      <th>T</th><th>F</th><th>S</th>
  </tr>
  <tr>
    <td id="2012-01-01"><span class="day-of-month">1</span></td>
    <td id="2012-01-02"><span class="day-of-month">2</span></td>
    <td id="2012-01-03"><span class="day-of-month">3</span></td>
    <td id="2012-01-04"><span class="day-of-month">4</span></td>
    <td id="2012-01-05"><span class="day-of-month">5</span></td>
    <td id="2012-01-06"><span class="day-of-month">6</span></td>
    <td id="2012-01-07"><span class="day-of-month">7</span></td>
  </tr>
  <tr><!-- ... --></tr>
  <tr><!-- ... --></tr>
  <tr><!-- ... --></tr>
  <tr><!-- ... --></tr>
</table>
```

That HTML might generate a calendar that displays something like this in a browser:



S	M	T	W	T	F	S
1	2	3	4	7	5	6
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

So far we have nothing more than a static calendar page. To make things a little more interesting, we add a jQuery AJAX request to the backend asking for all appointments in January:

```
$(function() {
  $.getJSON('/appointments', function(data) {
    $.each(data.rows, function(i, rec) { add_appointment(rec) });
  });
});
```

That request of the `/appointments` resource will return JSON that includes a `rows` attribute. For each record in the list of rows, we want to add a corresponding appointment to the the calendar.

```
{"total_rows":2,"offset":0,"rows":[
  {"id":           "appt-1",
   "startDate":   "2012-01-01",
   "title":       "Recover from Hangover",
   "description": "Hair of the dog that bit you."},
  {"id":           "appt-2",
   "startDate":   "2012-01-02",
   "title":       "Quit drinking",
   "description": "No really, I mean it this year"}
]}
```

The `add_appointment` function need not be anything fancy if we simply want the appointment to display. Something along the lines of the following will suffice:

```
function add_appointment(appointment) {
  var date = appointment.startDate,
      title = appointment.title,
      description = appointment.description;

  $('#' + date).append(
    '<span title="' + description + '">' +
      title +
    '</span>'
  );
}
```

Do you see the ISO 8601 trick in there? The `startDate` attribute is represented as an ISO 8601 date (*e.g.* `"2012-01-01"`). The cells in our calendar `<table>` also have IDs that correspond to the ISO 8601 date:

```
  <tr>
    <td id="2012-01-01"><span class="day-of-month">1</span></td>
    <!-- ... -->
  </tr>
```

Thus, by appending the appointment HTML to `$('#' + date)`, we are really appending to `$('#2012-01-01')` or the date cell for New Year's day. Simple clever [3], eh?

Using this strategy, we could fetch 3 years worth of appointments from the backend and run each through the `add_appointment` function. An appointment from New Year's Day 2010 would not be appended to the calendar because there is no calendar table cell with an ID of `$('#2010-01-10')`. That jQuery selector would produce an empty wrapped set, which results in no change.

The authors have been using a similar technique since the 1900s to great effect. For more than 10 years, it has been possible to do something like this and we did not need any fancy Javascript MVC framework.

So why do we need one now?

The answer to that question is what comes next. As in "What comes next in my calendar application?" Perhaps the user needs to move appointments between dates. Or maybe add new appointments / delete old ones. Regardless of what comes next, we are going to need to answer *how*. And how is the realm of Backbone.js.

Sure we might continue coming up with clever hacks like the ISO 8601 trick for our calendar. But with each clever hack, we risk making the code harder to approach for the next developer.

How do you future proof? How can you be sure that your approach will be understood by the next developer? How can you know that today's simple cleverness will still be easy to read in 3 months?

The answer is to not choose. Rather, let Backbone show you the way.

And that is where we begin in the next chapter…

[3]as opposed to clever clever which is always a bad idea

Chapter 2. Writing Backbone Applications

Having gone through the exercise of loading appointments over AJAX, a picture begins to form of how it will evolve. There will be navigation buttons to move back and forth between months. Controls will need to be added to switch between month, week, and day views. At some point, our calendar will need to create, update, move, and delete appointments on the calendar. To compete in the market, it will even need to support "fancy" features like scheduling recurring appointments and appointments on the second Tuesday of every month.

And throughout the evolution of such features, our calendar application needs to remain nice and snappy. It also needs to be able to store appointments in the backend quickly—again without impacting the performance of the UI.

Changing views from month to week to day does not seem all that hard. We could include hidden `<div>` tags to hold those views, showing them when the user chooses the appropriate control:

```
<div id="calendar">
  <div class="month-view">
    <!-- ... -->
  </div>
  <div class="week-view" style="display:none">
    <!-- ... -->
  </div>
  <div class="day-view" style="display:none">
    <!-- ... -->
  </div>
</div>
```

Now, when updates are made, we need to make sure that they apply to each of the three views. It will not do to create an appointment in the month view, only to have it disappear when switching to the day view.

Instead of updating three different views each time a change occurs, perhaps it would be better to store a copy of all appointments in a global variable. That would allow us to switch quickly between views without needing to make calls to the server each time.

But how will the server get notified when appointments change? How does that global data structure coordinate updates with the server for persistent storage? How do edit / change dialogs coordinate changes with this local store, the server and the current view?

Being developers, we are already starting to envision strategies for handling all this. Coming up with an API to manipulate that local data store. Maybe broadcast some global custom events when changes occur. Ooh! Maybe an API that wraps around the calls to the server…

Yes, there is a lot that we can do here. We could probably solve all of these problems and others that we have not even thought up yet, given enough time. Some of us might even come up with an almost elegant solution.

But the entire time that we are doing all of this, we are not focusing on our application. We are building infrastructure, not value for our customers.

Instead, let's use Backbone.js…

2.1. Converting to Backbone.js

To collect appointments from the server in our vanilla AJAX solution, we are making a jQuery `getJSON()` call:

```
$(function() {
  $.getJSON('/appointments', function(data) {
    $.each(data.rows, function(i, rec) { add_appointment(rec) });
  });
});
```

As is, this fetches the data from the server, displays it, and then promptly forgets about it.

In a Backbone application, the retrieval of a list of objects is always retained locally in a Collection. Once stored, Views can be attached to display the individual objects in a collection. They are not rendered immediately by the Collection as we did in our vanilla AJAX solution. Rather, views spring into existence when the application is started or in response to events.

The reason that Backbone uses a stand-alone Collection like this is so that the Collection can be used as something of a junction for events. If an individual object in the collection changes, it can generate a little event, which will bubble up through the collection and be passed along to any interested observers.

For instance, if an appointment is removed from the Collection, it will generate a "remove" event. The view responsible for displaying this particular appointment can then remove itself from the page. Just as importantly, summary views (e.g. number of events this month) can also listen for the "remove" event, using it as a signal to update themselves.

To actually define one of these Collections, we need to use the built-in `extend` method to extend `Backbone.Collection` into something specific to our appointment:

```
var Appointments = Backbone.Collection.extend({
  model: Appointment,
  url: '/appointments'
});
```

That defines an `Appointments` class, which will retrieve lists of events from the same server on which the Backbone application originated. Instead of retrieving the homepage or a URL specific to the Backbone application, the Collection will retrieve from a REST-like resource [1].

[1] REST is a convention for how to interact with objects stored on the server. If the list of appointments can be retrieved from `/appointments`, then an appointment with ID 42 can be retrieved from `/appointments/42`. To create a new appointment, a client would need to POST the `/appointments` URL. To update an appointment, the client would PUT to `/appointments/42`. To delete an appointment, use the HTTP verb to DELETE `/`

Backbone convention is such that the collection is not responsible for converting the results of fetching the URL. Rather, the collection simply takes the list of attributes [2] and sends each in turn to the model constructor specified by the `model` attribute. As we will see throughout this book, the separation of collections of models and models themselves have some fairly astounding implications. Here, it is enough to see that our Collection class is incredibly small.

This is a class, not the actual object that retrieves and stores collection data. For that, we need an instance of the collection. Instances are generally done inside a Backbone's constructor. For very simple Backbone applications, this might be done in a jQuery onDocumentReady callback:

```
$(function() {
  var appointments = new Appointments();
  appointments.fetch();
});
```

Here we have created an empty collection object. To populate it, we invoke the `fetch()` method. As described, `fetch()` makes an AJAX request of the server, converting the list of attributes returned into individual model objects.

If possible, it is generally considered good practice to create the collection store already populated:

`appointments/42`. This is a gross over simplification of REST. See the appendix for more resources.

[2]If the results of the URL are not a pure list of attributes, they can be "parsed" in the collection. For an example of doing this with CouchDB, see: http://japhr.blogspot.com/2011/08/converting-to-backbonejs.html

```
$(function() {
  var appointments = Appointments.reset(
    [ {startDate: "2011-12-31", /* ... */ },
      {startDate: "2012-01-01", /* ... */ },
      {startDate: "2012-01-02", /* ... */ },
      /* ... */
    ]
  );
});
```

This would initialize the store with 3+ appointments that are immediately available to be consumed and displayed by Backbone views. Doing something like this is only a good idea if the seed data is readily available. If there is any latency, then it is better to present an empty shell of the application to be filled in as quickly as possible by a subsequent `fetch()`.

This is all well and good, but so far we have only succeeded in retrieving data into a collection store. Unless we can display that information to the user, an awesome collection store is of no real benefit. Happily, Backbone views work quite well with this collection store.

But first we need to define the underpinning of that collection: the model.

2.2. Models

Before looking at Views, let's take a quick peek at Backbone models. To completely reproduce our pure AJAX version of the calendar application, almost nothing is required:

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

With that, the collection store is now capable of creating individual objects within the collection. Any attributes defined by the the collection's `/appointments` URL resource will be passed along to the model. To access those attributes—either directly or, more likely, from a View—we can use the `get()` method:

```
var firstAppointment = appointments.at(0)

var firstStartDate = firstAppointment.get('startDate')
```

The above extracts the first appointment model from the collection. From first appointment, we can get the "startDate" attribute.

Tip

In practice, it is quite rare to access individual models in a collection with `at()`. Typically, you should attach views to each member of the collection and have them render as appropriate. The `at()` method can be useful in testing, but in live code it is generally a code smell.

Since the goal of this exercise is to be able to update appointments as well as retrieve them, we need to make one other change to our `Appointment` model. Specifically, we need to tell it where it can access the corresponding server resource:

```
var Appointment = Backbone.Model.extend({
  url: '/appointments'
});
```

Amazingly, that is *all* that Backbone requires. This is because Backbone expects to interact with REST-like server resources. Given that, the above is all that Backbone needs to know so that it can POST new appointments to `/appointments`, PUT updates to `/appointments/42`, and DELETE at `/appointments/42`.

Tip

Of course, Backbone does not limit you to REST-like server code. See Chapter 12, *Non-REST Models* for more information.

At this point, we can retrieve and update appointments. And yet we still have no vehicle for an actual user to do this. Let’s change that next as we describe our first views….

2.3. Views

Just as in our vanilla AJAX application, the server is generating a month view that looks something like this:

```
<table>
  <tr>
    <th>S</th><th>M</th><th>T</th><th>W</th>
      <th>T</th><th>F</th><th>S</th>
  </tr>
  <tr>
    <td id="2012-01-01"><span class="day-of-month">1</span></td>
    <td id="2012-01-02"><span class="day-of-month">2</span></td>
    <td id="2012-01-03"><span class="day-of-month">3</span></td>
    <td id="2012-01-04"><span class="day-of-month">4</span></td>
    <td id="2012-01-05"><span class="day-of-month">5</span></td>
    <td id="2012-01-06"><span class="day-of-month">6</span></td>
    <td id="2012-01-07"><span class="day-of-month">7</span></td>
  </tr>
  <tr><!-- ... --></tr>
  <tr><!-- ... --></tr>
  <tr><!-- ... --></tr>
  <tr><!-- ... --></tr>
</table>
```

Our collection of appointments contains appointments on the 1st and 2nd of the month:

```
$(function() {
  var appointments = new Appointments();
  appointments.reset(
    [ {startDate: "2011-12-31", /* ... */ },
      {startDate: "2012-01-01", /* ... */ },
      {startDate: "2012-01-02", /* ... */ },
      /* ... */
    ]
  );
});
```

These appointments should render a title and possibly some controls on the corresponding date in the calendar. To make this happen, we need Backbone.js views.

An individual view might look something like:

```
var AppointmentView = Backbone.View.extend({
  template: _.template(
    '<span class="appointment" title="{{ description }}">' +
    '  <span class="title">{{title}}</span>' +
    '  <span class="delete">X</span>' +
    '</span>'
  ),
  initialize: function(options) {
    this.container = $('#' + this.model.get('startDate'));
    this.listenTo(this.model, 'change', this.render);
  },
  render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    this.container.append(this.$el);
    return this;
  }
});
```

That is fairly small, but there is also a fair bit going on in there. First is a `template()` method that describes the DOM structure of the appointment as it will be inserted into the calendar:

```
var AppointmentView = Backbone.View.extend({
  template: _.template(
    '<span class="appointment" title="{{ description }}">' +
    '  <span class="title">{{title}}</span>' +
    '  <span class="delete">X</span>' +
    '</span>'
  ),
  // ....
});
```

The description will be a tooltip as the user hovers over the appointment. The title and a "delete" icon are the elements that will actually be displayed [3].

In the `initialize()` method, which Backbone calls automatically if defined, we define where in the calendar the appointment HTML will get attached (*i.e.* the view's container) and we instruct the view to listen to the underlying model for changes:

```
var AppointmentView = Backbone.View.extend({
  // ....
  initialize: function(options) {
    this.container = $('#' + this.model.get('startDate'));
    this.listenTo(this.model, 'change', this.render);
  },
  // ....
});
```

Both the HTML and the model are following our ISO 8601 convention for dates. The HTML page assigns the ISO 8601 date as the ID attribute for the corresponding table cell in the calendar:

```
    <!-- ... -->
    <td id="2012-01-01"><span class="day-of-month">1</span></td>
    <!-- ... -->
```

To access that table cell with jQuery, we would use the ID selector convention of: `$('#2012-01-01')`.

The model has the ISO 8601 date stored in the `startDate` attribute:

```
      // ...
      {startDate: "2012-01-01", /* ... */ },
      // ...
```

To extract attributes from Backbone model objects, we need to use the `get()` method: `model.get('startDate')`.

[3] We are using mustache style templates here to aid in readability. See Chapter 5, *View Templates with Underscore.js* for details.

Thus, in the View's `initialize()` method, we can identify the appointment model's container with: `$('#' + this.model.get('startDate'))`. Astute readers will note that we have not explicitly assigned the model attribute in our view—even though we are making extensive use of it. As we will see in Chapter 18, *Object References in Backbone*, Backbone does this for us.

Last up in the view is the `render()` method, which actually builds the HTML for this view:

```
var AppointmentView = Backbone.View.extend({
  // ...
  render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    this.container.append(this.$el);
    return this;
  }
});
```

Here, we make use of another Model method, `toJSON()`, to get all of the attributes from the model (*e.g.* `startDate`, `title`, and `description`) as an object literal:

```
{
  'startDate': "2012-01-01",
  'title': "Resolve to Learn Backbone.js",
  'description': "Because it is awesome."
}
```

That object literal is then passed to the template that we defined earlier so that the attributes of the object literal can be used to replace the variables of the same name in the template.

Thus, this template:

```
<span class="appointment" title="{{ description }}">
  <span class="title">{{title}}</span>
  <span class="delete">X</span>'
</span>
```

When combined our model's JSON, becomes:

```
<span class="appointment" title="Because it is awesome.">
  <span class="title">Resolve to Learn Backbone.js</span>
  <span class="delete">X</span>'
</span>
```

The rest of the `render()` method inserts this HTML into the View's `el`, and the `el` into the containing table cell:

```
var AppointmentView = Backbone.View.extend({
  // ...
  render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    this.container.append(this.$el);
    return this;
  }
});
```

Important

Backbone views automatically create an anonymous `el` to hold the HTML that it needs. Normally the calling context will append that el into its own element. In the case of our calendar, there is no calling context, so the Appointment view itself assumes this responsibility. In reality this is a Backbone code smell. There should be a Calendar view with this responsibility — that would probably be next on our list of things to do with this application, but we leave it here as an exercise for the reader.

We have our Appointment view defined, but it is *still* not being drawn on the calendar. To actually get it added to the calendar, something needs to create the view objects. Depending on the application, this can either be done in the Backbone application's constructor or in a Collection view.

The collection view is more proper:

```
var AppointmentCollectionView = Backbone.View.extend({
  render: function() {
    this.collection.each(function(appointment) {
      var view = new AppointmentView({model: appointment});
      view.render();
    });
  }
});

// In the application constructor:
var appointments = new Appointments();
appointments.reset(
  [ {startDate: "2011-12-31", /* ... */ },
    {startDate: "2012-01-01", /* ... */ },
    {startDate: "2012-01-02", /* ... */ },
    /* ... */
  ]
);

var collection_view =
  new AppointmentCollectionView({collection: appointments});

collection_view.render();
```

Don't worry if that is a bit much at this point. We discuss the collection view in much more detail in Chapter 7, *Collection View*.

At this point, we have effectively replaced the following non-AJAX code from the previous chapter with the Backbone equivalent:

```
function add_appointment(appointment) {
  var date = appointment.startDate,
      title = appointment.title,
      description = appointment.description;

  $('#' + date).append(
    '<span title="' + description + '">' +
      title +
    '</span>'
  );
}
```

We have a bit more code on our hands, but we have gained much: a persistent store object, separation of concerns, and an idea of how to take the next steps. That last point should not be overlooked. In our non-Backbone version of the application, we have no idea how we might go about adding a delete widget to the appointment on the calendar.

With Backbone, we know that the responsibility for handing the delete click event lies with the individual Appointment views. Backbone would have us do this via an `events` property:

```
var AppointmentView = Backbone.View.extend({
  initialize: function(options) {
    this.container = $('#' + this.model.get('startDate'));
    this.listenTo(this.model, 'destroy', this.remove);
  },
  events: {
    'click .delete': 'handleDelete'
  },
  handleDelete: function() {
    this.model.destroy();
    return false;
  },

  // ....

  template: _.template(
    '<span class="appointment" title="{{ description }}">' +
    '  <span class="title">{{title}}</span>' +
    '  <span class="delete">X</span>' +
    '</span>'
  ),

  // ....
});
```

With that, a click on our delete widget is handed off to the `handleDelete()` method, which signals the model to `destroy()` itself. Once the model has successfully removed itself from the server, it triggers a "destroy" event. In the view's `initialize` method, we bind "destroy" events to the built-in `remove()` method, which removes the view itself.

Tip

Why not just `remove()` the element directly in `handleDelete()`? We wait until the model confirms that it has been removed from the persistence store because something could go wrong. By relying on events in this manner, we can be sure that things only disappear when they are supposed to and that errors can be handled separately. Try doing that in our non-AJAX version of the application!

2.4. Additional Reading

This was by no means an exhaustive introduction to Backbone.js programming. If you are still feeling a little lost, we highly recommend any of the following:

- The main Backbone site [http://documentcloud.github.com/backbone/]. The source code itself is also worth reading—it is quite well written and nicely commented.
- Backbone Resources from Derick Bailey [http://backbonetraining.net/resources]. This is a great resource for all things Backbone. Derick also produces a series of screencasts (both paid and free) that are well worth watching.
- Peepcode screencasts [http://peepcode.com/products/backbone-js] the undisputed kings of the screencast bring their considerable skill to bear on Backbone.js.

2.5. Conclusion

That concludes our whirlwind introduction to Backbone.js as well as the application that will drive our discussion of much deeper topics in the reminder of the book. Even if this was your first exposure to Backbone, you should have the beginnings of understanding as to how Backbone applications are written. Even better, you should have an idea as to why some things are done they way they are.

The rest of the book is dedicated to exploiting this structure so that we might realize some pretty amazing benefits.

Let’s get started…

Chapter 3. Namespacing

3.1. The Problem

This recipe is entirely geared toward long term maintainability of Backbone applications.

As Backbone applications grow, they can quickly pollute the global browser namespace with classes, instances and helper variables. In smaller implementations, this may not be much of a concern. When applications grow (and shouldn't they grow?), a poorly chosen naming scheme can cause all manner of problems.

3.2. The Solution

Two different approaches have worked for us in the past.

3.2.1. Alternative #1: Global Object Namespace

The first involves placing all Backbone class definitions inside of a global "namespace" object:

```
var Calendar = {
  Models: {},
  Collections: {},
  Views: {}
};
```

Then, as you define each model, collection and view class (as well as any helper classes that you might require), you can define them in this structure:

```
Calendar.Models.Appointment = Backbone.Model.extend({
  // ...
});

Calendar.Models.Holiday = Backbone.Model.extend({
  // ...
});

Calendar.Collections.Appointments = Backbone.Collection.extend({
  // ...
});
```

The principal advantage of this approach is that only one variable representing Backbone models, views and collections makes its way into the global namespace: `Calendar`. All other classes are defined inside this global object.

This is especially handy when concepts like "holidays" might have meaning outside of the Backbone application. With this naming scheme, the `Holiday` model is tucked away inside the `Calendar.Models` "namespace" where intent is clear.

Using this naming convention has the side-benefit of making class naming easier. If you attempt to define things in the global namespace, it might make sense to name the appointment model `Appointment`. But what, then, should you name the appointment view?

Tacking on the word `View` to each seems, er... tacky. Models would be given first-class citizen treatment (e.g. `Appointment`). Collections might get this as well (e.g. `Appointments`). But views need the extra `View`? Ugh.

And what if some views do not conflict with models and collection names (e.g. `AddAppointment`)? Do you append "View" to these in order to follow a convention? Do you omit "View" on these special cases to aid clarity?

Following the global object namespace convention means not having to choose. You get a clear convention that aids in maintainability with the added benefit of limiting risk of conflicting class and variable names.

3.2.2. Alternative #2: Javascript Function Constructor

The global object namespace is a simple approach. It also lends itself well to very large applications with numerous classes that you might prefer to keep in separate files. For more self-contained applications, a full blown Javascript object makes more sense.

This approach involves using a function constructor [1]. With a function constructor in Javascript, all manner of objects, functions, data and even classes can be initialized inside the function. Very little need be exposed to the outside world.

In the following, methods and data are defined inside the `public` and `private` object literals. With that out of the way, object initialization can be performed. Lastly, public attributes—and *only* the public attributes—of the resulting object are returned:

```
var Calendar = function(options) {
  var private = { /* private stuff here */ },
      public = { /* stuff for the outside */ };

  // Define methods and properties, adding to public as needed
  // Perform any object initialization

  return public;
}

var calendar = new Calendar();
```

In the global object namespace approach, we would still need to assign collection and view variables in the global namespace:

```
var appointments = new Calendar.Collections.Appointments;
```

[1]This approach relies on Javascript closures. If this is a foreign concept, check out "Inheritance: Functional" in Douglas Crockford's "Javascript: The Good Parts"

With a function constructor, this instance creation can take place inside of the function where there is no danger of conflicting with other variables.

Applying this approach to a Backbone application, we have something along the lines of:

```
var Calendar = function() {
  var Models = { /* model classes */ };

  var Collections = { /* collection classes */ };

  var Views = { /* view classes */ };

  // Initialize the app
  var appointments = new Collections.Appointments;

  new Views.Application({collection: appointments});

  return {
    Models: Models,
    Collections: Collections,
    Views: Views,
    appointments: appointments
  };
};
```

By defining `Models`, `Collections`, and `Views` object literal variables inside of the constructor, we have an easy time referencing things across—or even outside—concerns. For example, when initializing the `Appointments` collection, we can refer to `new Collections.Appointment`. With the global namespace approach, we always have to use the global namespace `new Calendar.Collections.appointment`. Dropping a single word does wonders for code readability and long term maintainability.

By returning each of these classes from the function constructor with a key of the same name, it means that the outside can get access to objects or classes with the exact same naming scheme:

```
var Calendar = function() {
  // ...

  var appointments = new Collections.Appointments;

  return {
    Models: Models,
    Collections: Collections,
    Views: Views,
    appointments: appointments
  };
};

var calendar = new Calendar();

// The appointments collection class
var AppointmentsCollection = calendar.Collections.Appointments;

// Actual appointments from the initialized collection object
var appointments = calendar.appointments;
```

Encapsulating concepts inside functions is so useful, in fact, that it can be used inside the function constructor. Instead of assigning the `Models`, `Collections`, and `Views` variables directly to object literals containing class definitions, we find it best to assign them to the return value of anonymous functions. These anonymous functions, when invoked with the `()` operator, return the same object literals—but with only those attributes that we want exposed to the outside world.

For example, instead of defining the `Models` variable directly:

```
var Calendar = function() {
  var Models = {Appointment: Backbone.Model.extend({...});};

  // ...
};
```

We can define the Models inside an anonymous function:

```
var Calendar = function() {
  var Models = (function() {
    var Appointment = Backbone.Model.extend({...});

    return {Appointment: Appointment};
  })();

  // ...
};
```

In this case, it buys us nothing. The end result of both approaches is a `Models` object variable with an `Appointments` key. This `Appointments` key references the `Appointment` class. This allows us to reference the class as `Models.Appointment` from elsewhere inside the function constructor (and ultimately as `Calendar.Models.Appointment` outside of the function constructor).

Where this approach yields benefit is when you have classes that are only needed by other classes, but not the outside world. For instance, the `Appointment` model may need to create instances of appointment attendees:

```
var Calendar = function() {
  var Models = (function() {
    var Appointment = Backbone.Model.extend({
      // ...
      attendees: function() {
        _.(this.get("emails")).map(function(email) {
          return new Attendee(email);
        });
      }
    });

    var Attendee = Backbone.Model.extend({ /* ... */ });

    // Only return Appointment
    return {Appointment: Appointment};
  })();

  // ...
};
```

This is especially powerful with View classes. Generally, only a handful of View classes need to be seen outside of a Backbone application. The remaining only pop-up on demand from the main view.

In the following, only the `Application` view needs to be accessed from outside. Once it is initialized, instances of the remaining classes are used on demand from the `Application` object or from each other:

```
var Calendar = function() {
  var Models = (function() { /* ... */ })();

  var Collections = (function() { /* ... */ })();

  var Views = (function() {
    var Appointment = Backbone.View.extend({...});
    var AppointmentEdit = Backbone.View.extend({...});
    var AppointmentAdd = new (Backbone.View.extend({...}));
    var Day = Backbone.View.extend({...});
    var Application = Backbone.View.extend({...});

    return {Application: Application};
  })();

  // Initialize the app
  var appointments = new Collections.Appointments;

  new Views.Application({collection: appointments});

  return {
    Models: Models,
    Collections: Collections,
    Views: Views,
    appointments: appointments
  };
};
```

A second advantage of this approach is that, within the constructor, it is possible to reference cross concern classes with less ceremony. When the collection needs to reference the model, it can do so as `Models.Appointment` instead of the full `Calendar.Models.Appointment` that is required in strategy #1:

```
var Collections = (function() {
  var Appointments = Backbone.Collection.extend({
    model: Models.Appointment,
    parse: function(response) {
      return _(response.rows).map(function(row) { return row.doc ;});
    }
  });

  return {Appointments: Appointments};
})();
```

This simple, and seemingly small, change will pay significant dividends over the lifetime of your Backbone applications.

This advantage is even more pronounced when referencing classes within the same concern. For example, if clicking the day view spawns the add-appointment view, this can be done with a simple reference to `AppointmentAdd` instead of needing to type (and read) `Calendar.Views.AppointmentAdd`:

```
var Day = Backbone.View.extend({
  events : {
    'click': 'addClick'
  },
  addClick: function(e) {
    console.log("addClick");

    AppointmentAdd.reset({startDate: this.el.id});
  }
});
```

The last advantage of this approach is the ability to define a very specific API for your Backbone application. Only those properties and methods required by other objects or even other Backbone applications are exposed.

A potential disadvantage of this approach is that individual model, view and collection classes cannot be in separate files and included directly in the page:

```
<script src="/javascript/backbone/calendar/models/appointment.js">
<script src="/javascript/backbone/calendar/collections/appointment.js">
<script src="/javascript/backbone/calendar/views/appointment.js">
<script src="/javascript/backbone/calendar/views/appointment_edit.js">
<script src="/javascript/backbone/calendar/views/appointment_add.js">
<script src="/javascript/backbone/calendar/views/day.js">
<script src="/javascript/backbone/calendar/views/application.js">
```

In the end, the choice is yours. Stick with the simple, global object that allows separate files for each class or go for the self-contained goodness of javascript objects. One is sure to meet your needs.

3.3. Conclusion

Namespacing is one of those concepts that you generally do not think about until it is too late. Even if you are fairly certain that your Backbone application is going to remain small, it is best to initialize and build models, views and controllers inside a common namespace object. This eliminates questions about possible naming conventions and reduces the footprint on the global namespace.

But, if your application is large or has the potential to grow large, it is best to put Javascript's function constructors to good use. These can create a whole application constructor that only exposes those pieces that you definitely want the rest of the page to see. Better still, it gives the individual components of your application more direct access to each other.

Chapter 4. Organizing with Require.js

Unlike most languages, Javascript lacks a built-in mechanism for loading libraries. There are a number of competing solutions, but require.js offers perhaps the most complete.

4.1. The Problem

As we just saw in Chapter 3, *Namespacing*, organizing Backbone code is a significant challenge to the Backbone developer. As difficult as it may be to keep code well organized within a namespace, it may be even more of a challenge to keep code organized on the file system. The require.js library offers just such a solution—doing so by exposing two keywords (`require` and `define`) that give Javascript a very familiar feel.

4.2. The Solution

Consider again our poor Calendar application. To draw the month view of the calendar itself, we might use a series of Backbone views that start at the top-level `CalendarMonth` and work all the way down to `CalendarMonthDay`. In our namespacing solution, this would look something like:

```
window.Cal = function(root_el) {
  var Models = (function() { /* ... */  })();
  var Collections = (function() { /* ... */ })();
  var Appointments = Backbone.Collection.extend({ /* ... */ })();
  var Views = (function() {
    var CalendarMonth = Backbone.View.extend({ /* ... */ });
    var CalendarMonthHeader = Backbone.View.extend({ /* ... */ });
    var CalendarMonthBody = Backbone.View.extend({ /* ... */ });
    var CalendarMonthWeek = Backbone.View.extend({ /* ... */ });
    var CalendarMonthDay = Backbone.View.extend({ /* ... */ });
    // ...
  })();

  // Routers, helpers, initialization...
};
```

In a small Backbone application, that is not *too* bad. There are some definite advantages to having everything in a single editor buffer—especially if everything is fairly small.

But, if the application grows significantly, this can quickly become unwieldy. Searching through a single file for the `Appointment` model can easily become a tedium of by-passing places in which the model is instantiated rather than defined. Or, worse still, where the `Appointment` *view* is defined instead of the model.

In the past, client-side Javascript developers have been reduced to a series of `<script>` tags, each of which populate a global namespace:

```
<script>
var Calendar = {
  Models: {},
  Collections: {},
  Views: {}
};
</script>
<script src="Calendar/Views/CalendarMonth.js" />
<script src="Calendar/Views/CalendarMonthHeader.js" />
<script src="Calendar/Views/CalendarMonthBody.js" />
<script src="Calendar/Views/CalendarMonthWeek.js" />
<script src="Calendar/Views/CalendarMonthDay.js" />
<!-- ... -->
```

Without help, such a solution is very much at the mercy of networking woes. If `CalendarMonth` creates an instance of `CalendarMonthHeader`, but `CalendarMonthHeader` arrives in the browser later than the requiring context, trouble can ensue. Regardless of load order, require.js ensures that no code is evaluated until all requirements have finished loading.

If there is significant network latency, then the round-trip time for the browser to fetch each of these files can significantly degrade application startup. Network issues can be mitigated by packaging all Javascript files into a single bundle. Although that introduces some complexity in the deployment process, it is a fairly well-established practice—with or without require.js.

Another, more subtle problem with this approach is that it encourages inadvertent coupling between the classes. Seemingly innocent references to higher order objects from a lower order object can quickly grow out of hand (see Chapter 18, *Object References in Backbone* for examples). With require.js (and similar mechanisms in server-side languages), dependencies must be explicitly declared. Coupling concerns become that much more readily identified and eliminated.

Let's see how a require.js Backbone application looks in HTML:

```
<script data-main="scripts/main"
          src="scripts/require.js"></script>
```

That’s it! All of the `<script>` tags from our traditional approach have been replaced with a single `<script>` tag. The `src` of that script tag is the require.js library itself.

How, then, does the application code get loaded? The answer is the `data-main` HTML5 attribute, which points to the "main" entry point of the application. The ".js" suffix is optional, so, in our case, we are loading from the `public/scripts/main.js` file.

The entry point for a require.js application is responsible for any configuration that needs to be done as well as initializing objects. For our calendar application, it might look something like:

```
require.config({
  paths: {
    'jquery': 'jquery.min',
    'jquery-ui': 'jquery-ui.min'
  }
});

require(['Calendar'], function(Calendar){
  var calendar = new Calendar($('#calendar'));
});
```

In the configuration section, we are telling require.js where to find libraries that are referenced. For the most part, require.js can guess the library needed. In this case, we tell require.js that, when we `require('jquery')`, that it should use the minified `jquery.min.js` (again the ".js" suffix is not needed). This can be especially handy if libraries include version numbers or other information in the filename (*e.g.* jquery-ui-1.8.16.custom.min.js). There are many `config` options [1], but `paths` suffices 80% of the time.

As for loading and initializing our Backbone application, it requires three lines of Javascript:

[1]http://requirejs.org/docs/api.html#config

```
require(['Calendar'], function(Calendar){
  new Calendar($('#calendar'));
});
```

The first argument to `require()` is a list of dependent libraries. In this case, we only want `public/scripts/Calendar.js`. Surprisingly, we do *not* need to pull in jQuery, Backbone or anything else—those dependencies are resolved lower in the Backbone application. The calendar class is supplied to the anonymous function, to which we bind the `Calendar` variable. At this point, all that is left is to instantiate the application.

For experienced Javascript coders—especially front end developers—this is pretty amazing. It is almost as if our beloved Javascript has become a "real" server-side language like Ruby, Python or Perl—complete with `require` / `import` statements. This, of course, is the entire point of require.js. It allows us to define and require modules, classes, JSON, and even functions.

To see how we might define a require.js module, let's have a look at the `Calendar.js` class that is being required above:

```
// public/scripts/Calendar.js
define(function(require) {
  var $ = require('jquery')
    , _ = require('underscore')
    , Backbone = require('backbone')
    , Router = require('Calendar/Router')
    , Appointments = require('Calendar/Collections.Appointments')
    , Application = require('Calendar/Views.Application')
    , to_iso8601 = require('Calendar/Helpers.to_iso8601');

  return function(root_el) {
    // Instantiate collections, views, routes here
  };
});
```

Require.js modules are built with the `define()` method. The `define()` method is roughly analogous to the `module` keyword in other languages—it encapsulates a code module. By convention, the first thing done inside a require.js module is requiring other libraries. **This** is where jQuery and Backbone dependencies finally start to be seen.

Important

At the time of this writing, this will only work with a minor fork of Backbone maintained by James Burke [2], the require.js maintainer. Jeremy Ashkenas has publicly stated his intention to merge some form of this into Backbone by the next release, so we are not going *too* far out on a limb here.

Also of note, is the naming convention that we use for the individual Backbone classes on the server. Instead of grouping them in `Models`, `Views` and `Collections` sub-directories, we put everything inside the `Calendar` top-level application directory. In there, we embed the type of class into the filename. This makes it easy to tell the difference between `Models.Appointment.js` and `Views.Appointment.js` in our editors (otherwise we would just have two files named `Appointment.js`).

Require.js modules must return a value—this is what gets assigned by the `require()` function. In our `Calendar` class, we use a function constructor to instantiate three things: a Backbone collection, a top-level view and the router. Then, we return an object for the `new Calendar($('#calendar'))` call:

[2]https://github.com/jrburke/backbone/tree/optamd3

```
define(function(require) {
  // require things

  return function(root_el) {
    var appointments = new Appointments()
      , application = new Application({
          collection: appointments,
          el: root_el
        });

    new Router({application: application});
    Backbone.history.start();

    return {
      application: application,
      appointments: appointments
    };
  };
});
```

Taking a quick peek at a how a Backbone view is defined, we again see the `define()` statement at the top, followed by the various `require()` statements. Last up comes the return value, an anonymous view class definition:

```
// scripts/Calendar/Views.Application.js
define(function(require) {
  var Backbone = require('backbone')
    , $ = require('jquery')
    , _ = require('underscore')
    , TitleView = require('Calendar/Views.TitleView')
    , CalendarMonth = require('Calendar/Views.CalendarMonth')
    , Appointment = require('Calendar/Views.Appointment');

  return Backbone.View.extend({
    // ...
  });
});
```

Tip

Things like assigning the jQuery function to the dollar sign are much more explicit in require.js: `$ = require('jquery')`.

At first, returning an anonymous view class might seem a little foreign, but this allows us the flexibility of assigning the class name however we see fit in the requiring context:

```
var Application = require('Calendar/Views.Application');

// or

var Calendar = {
  Views: {
    Application: require('Calendar/Views.Application');
  }
}
```

Note

Require.js is very good about loading modules only once regardless of how many times in the dependency tree a particular module is `require()`'d. Nearly all of your Backbone classes will need to do something along the lines of `Backbone = require('backbone')`. Mercifully, require.js spares the user the overhead of re-requesting that same library repeatedly.

At the risk of being redundant, a model class might be defined as:

```
// scripts/Calendar/Models.Appointment.js
define(function(require) {
  var Backbone = require('backbone')
    , _ = require('underscore');

  return Backbone.Model.extend({
    // Normal model attributes
  });
});
```

The collection that uses this model could then be defined as:

```
// scripts/Calendar/Collections.Appointments.js
define(function(require) {
  var Backbone = require('backbone')
    , Appointment = require('Calendar/Models.Appointment');

  return Backbone.Collection.extend({
    model: Appointment,
    url: '/appointments',
    // Other collection attributes here
  });
});
```

With require.js, the list of individual library files needed to run a Backbone application is no longer the responsibility of the web page that happens to include the application. Now, it is the dependent libraries who are tasked with this job—a much saner, more maintainable solution.

4.2.1. Requiring Other Things

Require.js is a browser hack rather than a language hack. That is, once it analyzes dependencies, it adds new libraries by appending new `<script>` tags to the body of the hosting web page. Since it is already appending things to the page, there is nothing preventing require.js from appending other things—like CSS and HTML templates.

HTML templates, in particular, can further aid in the maintainability of Backbone applications. Consider, for instance, an appointment template (using the mustache-style from Chapter 5, *View Templates with Underscore.js*) that displays the title and a delete widget:

```
// public/javascripts/calendar/Views.Appointment.html
<span class="appointment" title="{{ description }}">
  <span class="title">{{title}}</span>
  <span class="delete">X</span>
</span>
```

There are some advantages to keeping such HTML templates in our views, especially if they are small. Still, there are times when the views themselves get long or the syntax highlighting in our editors would be handy. In such cases, we can install the require.js text plugin [3]. The defined sections of our views can then require the HTML template:

```
define(function(require) {
  var Backbone = require('backbone')
    , _ = require('underscore')
    , html_template = require('text!calendar/views/Appointment.html')
    , template = _.template(html_template)
    // ...

  return Backbone.View.extend({
    template: template,
    // ...
  });
});
```

With that, we are now maintaining templates separately from the views without any significant changes to the overall structure of the code.

4.2.2. Optimization / Asset Packaging

In the end, even a very small Backbone application organized with require.js is going to be comprised of a large number of individual files. By way of example, a limited calendar application might look like:

[3] http://requirejs.org/docs/download.html#text

```
scripts
+-- backbone.js
+-- Calendar
|   +-- Collections.Appointments.js
|   +-- Helpers.template.js
|   +-- Helpers.to_iso8601.js
|   +-- Models.Appointment.js
|   +-- Router.js
|   +-- Views.Application.js
|   +-- Views.AppointmentAdd.js
|   +-- Views.AppointmentEdit.js
|   +-- Views.Appointment.js
|   +-- Views.Appointment.html
|   +-- Views.CalendarMonthBody.js
|   +-- Views.CalendarMonthDay.js
|   +-- Views.CalendarMonthHeader.js
|   +-- Views.CalendarMonth.js
|   +-- Views.CalendarMonthWeek.js
|   +-- Views.CalendarNavigation.js
|   +-- Views.TitleView.js
+-- Calendar.js
+-- jquery.min.js
+-- jquery-ui.min.js
+-- main.js
+-- require.js
+-- underscore.js
```

That is 24 round trips (request / response) that the browser would need to make before it is even capable of booting the application. Even if the client is connected to the server over a fast, low latency connection, there is way too much overhead in that setup [4]. To get around that, of course, modern websites use asset packaging and CDNs.

Most asset packages are ignorant of require.js so we might be given to despair. Happily, require.js includes its own asset packager, `r.js`. There

[4]Unless you are using something like SPDY. By the way, you should totally buy Chris's "The SPDY Book" if you have not already :D

are at least two ways to install `r.js` [5]. Which installation method is best depends on individual development environments and preferences.

To run the optimization tool, it is easiest to create an `app.build.js` file in your application's root directory. This file contains a number of options, some of which will be nearly duplicate of the `require.config` options in the data-main file:

```
({
  // Where HTML and JS is stored:
  appDir: "public",
  // Sub-directory of appDir containing JS:
  baseUrl: "scripts",
  // Where to build the optimized project:
  dir: "public.optimized",
  // Modules to be optimized:
  modules: [
    {
      name: "main"
    }
  ],
  // Resolve any 'jquery' dependencies to the versioned jquery file:
  paths: {
    'jquery': 'jquery-1.7.1'
  }
})
```

The `paths` option has exactly the same meaning in the build configuration that is has in data-main—it tells require.js to map named dependencies to non-inferable filenames. Here, we are telling require.js to require references to `jquery` from the `jquery-1.7.1.js` resource. At some point, the optimization tool may be able to extract this information directly from `data-main`. At the time of this writing, it is separate to allow maximum flexibility when optimizing.

[5]Download and installation instructions are available from the require.js site: http://requirejs.org/docs/optimization.html

Most of the other configuration options are self-explanatory. To slurp in the entire dependency tree, all we need to do is specify the `main.js` module via the `modules` attribute—`r.js` will take care of the rest for us.

With that, it is a simple matter of building the optimized version of our public directory:

That’s it! The optimized version of the site is now available in the directory specified by `dir` (we used `public.optimized`). We can then point our web server at that directory and serve up super fast, packaged code.

4.3. Conclusion

Web developers have lived with the lack of a mechanism to require Javascript files for so long that we are almost numb to the pain. We would belittle a server-side programming language that lacked something so basic—how can we possibly produce quality code without a reliable way to organize it? We could *almost* excuse the lack of this ability in the past with Javascript—after all it is only recently that we have witnessed the explosion of client-side coding.

But in 2012 and beyond, it behooves us to use a module loader like require.js. It makes our code much more readable, and easier to maintain. It’s almost like programming in a real language.

Chapter 5. View Templates with Underscore.js

5.1. The Problem

It will happen.

No matter how hard you try to keep your Backbone.js views free from large amounts of HTML, there really are times when more HTML does solve the problem (unlike XML and guns).

But how do you solve the immediate problem without sacrificing maintainability and readability in the future? Discretionary use of underscore.js templates will go a long way.

5.2. The Solution

Let's consider a calendar navigation view. If the user is currently viewing appointments for January 2012, the view would need to render HTML along the lines of:

```
<div class="previous">
 <a href="/#month/2011-12">previous</a>
</div>
<div class="next">
 <a href="/#month/2012-02">next</a>
</div>
```

One way to accomplish this might be concatenating HTML and date strings:

```
/* This is not a good idea */
var CalendarNavigation = Backbone.View.extend({
  render: function() {
    var date = this.collection.getDate(),
        previous_month = Helpers.previousMonth(date),
        next_month = Helpers.nextMonth(date);

    var html =
      '<div class="previous">' +
        '<a href="/#month/' + previous_month + '">previous</a>' +
      '</div>' +
      '<div class="next">' +
        '<a href="/#month/' +  next_month + '">next</a>' +
      '</div>';

    $(this.el).html(html);

    return this;
  }
});
```

This is a poor long term solution because, as short as it is, the `render()` method is doing way too much. It calculates the next/previous months, it builds HTML, and it inserts that HTML into the DOM.

It also fails the "at a glance" test because it is hard to pick out the variables from the noise of the HTML. Since the variables are the most important thing going on here, it should be immediately obvious how they are used. Besides, it is a pain typing all of those single and double quotes.

Tip

If code fails the "at a glance" test, it lacks long term maintainability. That is, if it is hard to pick out the important elements of a particular segment of code, then it will be that much harder to understand intent when trying to track down a bug or add a feature six months later.

A better solution is to make use of the templating that is built into underscore.js (upon which Backbone is built).

Passing a template string to the `_.template()` method returns a template function:

```
var template_fn =  _.template(
  '<div class="previous">' +
    '<a href="/#month/{{ previous_date }}">previous</a>' +
  '</div>' +
  '<div class="next">' +
    '<a href="/#month/{{ next_date }}">next</a>' +
  '</div>'
);
```

The template function will accept a single argument—an object literal that contains the values to be inserted into the template. In this case, we want to insert values for `previous_date` and `next_date`. If the current month is January 2012, we would want to set these values to December 2011 and February 2012 respectively:

```
template_fn({
  previous_date: "2011-12",
  next_date: "2012-02"
})
```

The result of that template function call would then be:

```
<div class="previous">
 <a href="/#month/2011-12">previous</a>
</div>
<div class="next">
 <a href="/#month/2012-02">next</a>
</div>
```

In Backbone applications, this is typically implemented by assigning the template function to an attribute on the View class. Since this attribute points to a function, we can call it as if it were a method on the object.

Warning

Template methods are not real methods—the special `this` variable will not refer to the current object. Rather it would refer to the template function from underscore. For the most part, this should not matter—you should only pass variables into a template, not assign them from attributes of the object. But, if you really, really need to, the judicious use of an `_.bindAll` in the object's initialize should do the trick.

To illustrate, let's revisit the `CalendarNavigation` view. We assign the template function to the `template` attribute of the view class. The `render` method can then call the template function as `this.template()`:

```
var CalendarNavigation = Backbone.View.extend({
  template: _.template(
    '<div class="previous">' +
      '<a href="#month/{{ previous_date }}">previous</a>' +
    '</div>' +
    '<div class="next">' +
      '<a href="#month/{{ next_date }}">next</a>' +
    '</div>'
  ),
  render: function() {
    var date = this.collection.getDate();
    $(this.el).html(this.template({
      previous_date: Helpers.previousMonth(date),
      next_date: Helpers.nextMonth(date)
    }));

    return this;
  }
});
```

Best of all, the `template` method has a single responsibility: building the HTML structure. Similarly, the `render()` method has a single responsibility as well: inserting the result of the template function call into the DOM.

Both methods are small and focused, which means that there are fewer chances for bugs to occur. If bugs do occur, then it will much easier to identify the culprit. Splitting things up like this keeps things so small, in fact, that including the HTML directly in the template method definition does not feel onerous.

Tip

The underscore `template()` method can take either one or two arguments. The first is the template string, the second is the object literal that is interpolated into the template. If you only pass the template string argument, then `template()` returns a function that accepts one argument—the object literal.

5.2.1. Avoid Script Tag Templates

Many tutorials and documentation give examples of script tag underscore templates such as:

```
<script type="text/template" id="calendar-appointment-template">
  <span class="appointment" title="{{ description }}">
      {{title}}
      <span class="delete">X</span>
  </span>
</script>
```

There are some definite advantages to this kind of approach. By using a `<script>` tag, the intent is quite clear—this will be used to support code. A `type` of `text/template` is not recognized by any browser vendor as a programming language, but it is readily understood by any developer reading the code.

Using such templates is similarly easy. Simply reference the `id` attribute of the script tag and you have access to the template via a jQuery `html()` call:

```
var Appointment = Backbone.View.extend({
  template: _.template($('#calendar-appointment-template').html()),
  render: function() {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
  }
});
```

From there, it is just as easy to apply the script template as it was to apply the inline template.

Easy. Easy. Easy. So what is the problem?

In two words: code organization.

By its very nature, the `<script>` tag is HTML that needs to reside in the HTML document. Most of your Backbone code is better organized in `.js` files loaded elsewhere.

At best, you can include script templates immediately after the script tag defining the Backbone view:

```
<script type="application/javascript">
var Appointment = Backbone.View.extend({
  template: _.template($('#calendar-appointment-template').html()),
  render: function() {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
  }
});
</script>

<script type="text/template" id="calendar-appointment-template">
  <span class="appointment" title="{{ description }}">
      {{title}}
      <span class="delete">X</span>
  </span>
</script>
```

But that is terrible! As you scan through your code, you will be constantly be interrupted by HTML snippets—both in templates and in your code.

It is best to keep the number of script templates to a minimum by inlining underscore templates:

```
var Appointment = Backbone.View.extend({
  template: _.template(
    '<span class="appointment" title="{{ description }}">' +
      '{{title}}' +
      '<span class="delete">X</span>' +
    '</span>'
  ),
  render: function() {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
  }
});
```

Since the template in this case is small, we are not sacrificing much in the way of readability by including it directly in the view class. Not only do we have the template available without jarring `<script>` tags getting in the way, but no searching is required to find the template. The template is right in the Backbone view itself.

Tip

Keep the number of script templates to an absolute minimum. Sometimes they are a necessary evil, but regard them as a code smell. Your code will thank you. Your future self will thank you.

If unsure, look for ways to break large templates into smaller views.

5.2.2. ERB Sucks {{ Use Mustache }}

The default variable / code handling in Backbone.js is intended to make life easy for Ruby programmers. Ruby programmers are a bunch of whiny, elitist hipsters whose opinion should not influence you (much) [1]. Especially since they inflicted ERB on the world. ERB, or embedded ruby allows developers to dynamically insert variables (or even code) into templates.

[1] we are both rubyists by profession

The navigation template from earlier in this recipe can be written with ERB as:

```
<div class="previous">
  <a href="#month/<%= previous_date %>">previous</a>
</div>
<div class="next">
  <a href="#month/<%= next_date %>">next</a>
</div>
```

Do you see it in there? Of course you don't. ERB lacks any semblance of readability when used in HTML since it is delimited with the same less-than and greater-than symbols that HTML uses.

Upon closer inspection of the above, you might notice `<%= previous_date %>`. This is how ERB templates signal that the `previous_date` variable needs to be inserted into the template. "Inserting into" is also known as interpolating.

Happily, Underscore exposes a mechanism for changing from ERB to a saner delimiter. We prefer the mustache [2] format of double curly braces, which is much easier to pick out when scanning the template:

```
<div class="previous">
  <a href="#month/{{ previous_date }}">previous</a>
</div>
<div class="next">
  <a href="#month/{{ next_date }}">next</a>
</div>
```

Less-than, greater-than symbols just blend into the noise of the HTML. There are no common use-cases for double curly braces so they naturally stand out when skimming the template. This readily identifiable aspect of mustache makes it ideal for use in Backbone applications.

To achieve mustache interpolation, change the value of `_.templateSettings`:

[2] http://mustache.github.com/

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

Tip

Ah regular expressions. But as regular expressions go, our mustache interpolation regular expression is not *that* bad:

```
/\{\{(.+?)\}\}/g
```

Obviously, it matches double curly braces before and after `(.+?)`. The parentheses around that indicates that everything inside will be fed back to underscore to be replaced with the corresponding variable name. Of the remaining regular expression, `.+?`, the period matches any character, the plus matches one or more (of any character) and the question mark makes the expression "non-greedy".

Non-greedy regular expressions stop matching sooner rather than later. A greedy version of our regular expression would match this entire string:

```
{{ one }} two three four {{ five }}
```

In other words, it starts matching on the two curly braces at the beginning of the line and does not stop until it reaches the end of the line (after "five"). By making the regular expression non-greedy, the matching stops as soon as the text after the regular express (in this case, `\}\}`) is reached.

Thus we pull out two variables to be interpolated: `one` and `five`.

5.2.3. Avoid Evaluation

By default, underscore templates are not limited to interpolating values into templates. They can also evaluate Javascript. Instead of using the <

`%= ... %>` delimiter, evaluation in underscore templates takes place inside `<% ... %>` (without the equals sign).

By way of example, building a month view for a calendar might look something like:

```
/* Not a good idea */
<% _([0,1,2,3,4,5]).each(function(i) { %>
<tr class="week<%= i %>">
  <td class="sunday"></td>
  <td class="monday"></td>
  <td class="tuesday"></td>
  <td class="wednesday"></td>
  <td class="thursday"></td>
  <td class="friday"></td>
  <td class="saturday"></td>
</tr>
<% }); %>
```

This template features both interpolation (the week number in the class of the `<tr>` tag) and evaluation (the anonymous function called for each week).

It is incredibly awkward to open the block of our function only to immediately close the ERB tag. Even worse, the last line of this template — "`<% }); %>`" — looks as though a cat was loosed on the keyboard of an unsuspecting developer.

Evaluating code in a template is almost always a code smell. Code evaluation is more often than not a separate Backbone view trying to escape. In this `Month` view example, it is more proper to create a `Week` view with the `<tr>` tag. This `Week` view, in turn, would then have 7 `Day` views that will hold the `<td>` elements.

We take this advice so seriously that we normally do not define an `evaluate` template setting:

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

With this setting, it is not possible to evaluate code in our templates. If ever we are tempted by code evaluation in a template, we would be forced to ask ourselves very seriously if we really need it. And we never do [3].

5.3. Conclusion

We use underscore templating almost without thinking in the remainder of the recipes. Yet, it is well worth taking a step back to consider how we really want to use it and why.

To promote code readability, and hence maintainability, we will always use the mustache-style interpolation syntax. Along those same lines, we will never use evaluatable templates in these recipes. Views should be smart, templates dumb.

[3]Well, almost never.

Chapter 6. Instantiated View

6.1. Introduction

The Instantiated View pattern is a simple pattern that changes the way a view is instantiated when it only needs one instantiation throughout the entire application. This pattern is really more of a nifty trick, however we use it so frequently throughout the book that it needed its own chapter.

6.2. The Problem

In web applications there are often objects in the user interface that only exist once. For example, a navigation bar at the top of the page, or a search box, or information about the user that is currently logged in. When we have these objects in a normal program we would use a global variable, or more eloquently a singleton pattern. A singleton is a nice way of auto-instantiating an object on demand and maintaining a single instance under what is more or less a global variable.

A global variable is a much simpler way of accomplishing this task, but is generally frowned upon because it pollutes the global scope of the program. However, in a web application we are making active use of proper namespacing, and can therefore get away with simply making a global variable instead of bothering to make a singleton. Furthermore, jQuery makes it very easy to automatically instantiate an instance on the boot of our application.

In HTML and Javascript, there is already the notion of a global identifier in the DOM, \`id\`s. A div with a global id looks like:

```
<div id="list">I am a list. Because my id is set there should only be one of me
```

In Backbone, we often go about defining our views then instantiating them, like this:

```
MyApplication.Views.List = Backbone.View.extend({/* etc */});
// instantiation
$(function() {
  MyApplication.ListView = new MyApplication.Views.List({
    el: $('#list')
  });
})
```

Notice that we're passing the element in as a parameter to the view. This decouples the view from its actual location from the DOM, and allows us to change that DOM id without having to update the view itself, just the instantiation of the app.

There is nothing functionally wrong with this code, it will behave as intended. The main issue here is the redundancy and ambiguity of the variable names. It is up to the programmer to remember that `MyApplication.Views.List` is the class definition of a list that is only used once in the application and should not be instantiated, and that `MyApplication.ListView` is our reference to the object that we should use in the application.

We can make this code clearer, cleaner, shorter, and less error prone with a small change to our view definition.

6.3. The Solution

The fix is to assign an instantiation of an anonymous class to `MyApplication.ListView`:

```
$(function() {
  MyApplication.ListView = new (Backbone.View.extend({
    render: function() {
      this.$el.html("I am a list");
      return this;
    }
  }))({el: $('#list')}).render()
});
```

In this approach, we are doing the typical extension of a top-level Backbone class with custom attributes and methods. The difference, however, is that we do *not* assign the result to a class name as we did earlier. Instead, we immediately create a `new` instance of this anonymous class. Lastly, `MyApplication.ListView` is assigned to the resulting object.

This lets us make one version of the view without leaving an enticing (or confusing) class around that could be instantiated again. It also saves us a few lines of typing and reduces the number of variables in our namespace. It also lets us avoid confusing nomenclature like `MyApplication.Views.List` and `MyApplication.ListView`. Which one is the instantiated version and which one is the class that we're not supposed to use?

6.4. Conclusion

The end result is that there is no way to instantiate a new instance of the class [1]. A good pattern not only let's you do something great, it also prevents you from doing something bad. The Instantiated View prevents you, or someone else, from accidentally instantiating a view that binds to what should be a unique DOM element. Just as important, the intent of the object is much clearer.

[1]This is Javascript, so it is *possible* to clone the object or add it to a prototype chain. But you, or other developers, would have to work hard to subvert the intent.

Chapter 7. Collection View

7.1. Introduction

The Collection View is a pattern that describes how to render views within views, specifically with a Collection holding many Models which is a common occurrence in a Backbone application. However, this pattern can easily be applied to any situation where you have a view that needs to render a dynamic number of sub-views.

7.2. The Problem

In server-side applications, it is common to see routes that represent many items of the same type. For example, the `/appointments` route might display HTML for all of the appointments in the system.

Typically, this server-side code gathers up all `appointments` via a database query. It then iterates over each record, rendering them as HTML in a template. This is all well and good when the following conditions are true:

1. The server-side template library can handle iteration (or arbitrary code)
2. The generated page is not interactive

Unfortunately, neither of these conditions hold for a modern client-side web application. Additionally, we encounter other obstacles:

1. Maintaining client-side templates quickly grows disorganized and confusing when they are filled with logic and iteration
2. A lot of interactive code is concentrated into a few "master" views instead of spread throughout the models

The first point is immediately apparent for anyone who has worked on a large client-side application (otherwise, take our word for it!). The second point is more subtle and will creep into your application over time.

Consider our `appointments` application, which might consist of:

- `Models.Appointments`
- `Collections.Appointments`
- `Views.Appointments`
- `Templates.Appointments`

Think about what `Templates.Appointments` would look like. One of the first lines will be the beginning of an iteration over individual `appointments`. **The majority of this view will be concerned with rendering an individual `appointment`**. This should immediately be an indicator that `Templates.Appointments` is not doing what it was designed to do. A template for rendering multiple appointments should only be concerned with concepts like lists and ordering, not with the process of rendering individual items.

Additionally, if we have a master `AppointmentsView`, its event bindings will be on the *list of appointments* not on the *individual appointments*. This will be much harder to implement naturally using Backbone's event binding.

Furthermore, if an event is triggered signaling that an individual `appointment` has changed and must be re-rendered, we need to re-render the entire list of `appointments`. This is not only expensive, but can jar the user's view by breaking their scrolling position (if the list is long and they are in the middle). It also means that a single view is listening to events triggered by many models, which is another code smell.

In well designed server-side applications, `Templates.Appointments` will simply loop over the `appointments` and immediately render a `Templates.Appointment` template for each one, thus delegating that work onto another class. This is what we want to do in Backbone. The difference is that, in Backbone, it is much simpler and more natural to have views call subviews, instead of having templates call subtemplates.

7.3. The Solution

First, we need a new application structure:

- `Models.Appointments`
- `Collections.Appointments`
- `Views.Appointments`
- `Templates.Appointments`
- **`Views.Appointment`**
- **`Templates.Appointment`**

We have added a second view and second template to handle individual appointments. Let's take a look at what the top level `Templates.Appointments` and `Views.Appointments` might look like:

```
Templates.Appointments = _.template(
  "<h2>Here is a list of Appointments</h2>"
);
```

```
Views.Appointments = Backbone.View.extend({
  template: Templates.Appointments,

  initialize: function(options) {
    this.listenTo(this.collection, 'add', this.addOne);
  },

  render: function() {
    this.$el.html(this.template());
    this.addAll();
    return this;
  },

  addAll: function() {
    this.collection.each(this.addOne, this);
  },

  addOne: function(model) {
    view = new Views.Appointment({model: model});
    view.render();
    this.$el.append(view.el);
    model.on('remove', view.remove, view);
  }
});
```

```
$(function() {
  // Create a collection
  var appointments = new Collections.Appointments([
    {title: 'Doctor Appointment', date: '2011-01-04'},
    {title: 'Birthday Party', date: '2011-01-07'},
    {title: 'Book Club', date: '2011-01-14'}
  ]);

  // Create our top level view attached to the dom
  new Views.Appointments({
    collection: appointments, el: $('#appointments')
  }).render();
});
```

Here is what `Views.Appointments` is responsible for:

1. Rendering its own template (the template data not relevant to individual appointments)
2. Iterating over `Collections.Appointments`
3. Creating new `Views.Appointment` when a new `appointment` is added to the collection and appending that view's DOM element to its own
4. Asking the view to remove itself when the model is removed from the collection

More importantly, note what `Views.Appointments` is *not* responsible for:

1. Rendering individual `appointments`
2. Listening to events on individual `appointments`
3. Updating the individual `appointment` view
4. Removing the view when a model is destroyed

Now that we have that sorted out, let's look at `Templates.Appointment` and `Views.Appointment`:

```
Templates.Appointment = _.template(
  "<div class='title'>{{ title }}</div>" +
  "<div class='date'>{{ date }}</div>"
);
```

Warning

Try to keep javascript code out of templates. It is a good habit to pass a JSON-style object to a template, not pass a full model to a template. The key point here is to pass key value pairs of JSON primitives like integers and strings, and not expect functions to be available. Avoid iteration by doing the iteration in the view and creating subviews. Conditionals are subjective, if they are short it is OK, but as they grow, consider subtemplates or subviews.

```
Views.Appointment = Backbone.View.extend({
  template: Templates.Appointment,

  initialize: function(options) {
    this.listenTo(this.model, 'change', this.render);
    this.listenTo(this.model, 'destroy', this.remove);
  },

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

Here is what `Views.Appointment` is responsible for:

1. Rendering an individual `appointment`

2. Updating the view when the `appointment` changes

3. Removing the view when the `appointment` is destroyed

An interesting distinction between the `destroy` and `remove` events can be observed here. Both are causing the same effect in the view and in the DOM, but they are very different events!

The `destroy` event occurs when the model is deleted from the persistence system (the server, or client storage, *etc.*). For example, we could have a button on our view called "Delete". Or, **more importantly**, there could be a button on an entirely different part of our application that deletes models.

Consider a side-panel that has a button called "Remove all read appointments" that only removes `appointment` models if they have the `read` attribute set. We could easily say, "when the delete button is clicked, remove this element". If we do that, we would have to do that for every instance that a model is deleted in some way *and* would need to hook it up to every view that displays that model. The power of events in

Backbone is that, by binding to relevant events, we can avoid this duplication.

We also need to be aware of `remove` actions, because we may be maintaining multiple collections with the same set of models in them. Consider if we had all our appointments in a global `MyApplication.Appointments`, but then we created two sub-collections: `MyApplication.ReadAppointments` and `MyApplication.UnreadAppointments`. Any time a `Model.Appointment` was marked as `read` or `unread`, we move it from one collection to the other. In memory, those are the same `Model.Appointment` in the top-level `MyApplication.Appointments` and in the sub-collections.

If we had a `Views.Appointments` for each of the sub-collections, we need to remove the view elements on a `remove` event, but the model is not deleted, just removed from the collection. Since this is a Collection View we are representing the state of the collection, and must modify the view to mirror the collection's state.

Tip

Always try to use the most appropriate event when binding to an action. Don't see the right event? We will cover firing and listening to custom events in a later chapter.

7.4. Conclusion

The Single Responsibility Principle is just as important in client-side Javascript as it is in server-side code. It is an indication of good, object-oriented design that each entity is responsible for a single task. The Collection View divides up the tasks of iteration, interactivity, and output into separate objects, each with their own simple goals.

Chapter 8. View Signature

8.1. Introduction

The View Signature pattern is a form of caching to prevent redraws in the browser. It helps short-circuit unnecessary redraws by defining a cache key method and checking if the key changes.

8.2. The Problem

Whenever a view is rendered, there are costs inflicted on the user:

1. The browser must render the view. This takes time and effort, even if it is only a small amount
2. The view will "blink" while the application empties the element and fills it in with new content
3. Scrolling can be interrupted if the view is long and the user has scrolled down into it

It is disturbing to see content blink on a page, and it is very annoying to have your scrolling position change while using a site. In the worst, case, it may be impossible to scroll the page because the view keeps being re-rendered. This behavior is undesirable in any web application.

8.3. The Solution

There are two tactics for dealing with these problems, each with their own pros and cons. The first is View Signatures, which we will discuss here, and the second is Fill-In Views, which we will discuss in the next chapter.

A View Signature avoids the rendering of a view when the output would be identical to what is currently rendered. Often, events are fired in Backbone that may or may not be relevant to the view. The view does not care to differentiate between different types of events (or different objects on which the event is fired) because that sort of computation could be difficult, costly, and beyond the responsibility of the view.

The goal of a View Signature is to **simply and efficiently** decide if the view is stale and needs to be re-rendered.

8.3.1. What is a Signature?

A View Signature is a condensed representation of the contents of a view. It should conform to the following rules:

1. Two views cannot have the same signature and also have different HTML output
2. Two views cannot have different signatures and also have the same HTML output

In this way it is very much like a cache key for the contents of the view.

8.3.2. Signature Module

First, let's define a simple signature module we can use in our views to keep them cleaner:

```
Modules.SignedView = {
  updateSignature: function() {
    newSignature = this.signature();
    if (newSignature !== this._signature) {
      this._signature = newSignature;
      return true;
    }
    return false;
  }
}
```

The `updateSignature` method updates the signature if it has changed, calling `this.signature()`, which the class being mixed into needs to define.

It will return `true` if the signature changed and `false` if it did not change. Now we can include this module into any view and all we have to do is define `signature` and short-circuit `render` if `updateSignature` returns true. Now that we have that out of the way, let's look at a few signature methods.

8.3.3. A Simple Example: MD5

The simplest example (other than the HTML of the view itself) is an MD5 hash of the rendered HTML. The problem here is that, while it is a *very* simple implementation that prevents the view from being re-rendered, it is not very efficient. The inefficiency stems from the view, which has to be generated in its entirety in order to determine if it is the same. However, since it does accomplish the goal of avoiding a re-render, it is a legitimate solution (especially for an extremely complex view). Let's take a look at what an MD5 signature for `Views.Appointment` from the Collection View chapter looks like:

```
Views.AppointmentMD5 = Backbone.View.extend(_.extend({
  template: Templates.Appointment,
  render: function() {
    if (this.updateSignature()) {
      this.$el.html(this.template(this.model.toJSON()));
    }
    return this;
  },

  signature: function() {
    return hex_md5(this.template(this.model.toJSON()));
  }
}, Modules.SignedView));
```

Note

We are using the MD5 algorithm from http://pajhome.org.uk/crypt/md5/md5.html by Paul "Paj" Johnston

This code generates an MD5 hash of the HTML. We call `this.updateSignature()` to check if the signature has changed, if it has changed, we'll render the view.

The problem with the MD5, as previously stated, is that we need to render the view to HTML to tell if we need to complete the rendering work. The higher up the rendering chain we can push the logic, the more efficient our signature will be. Also, observant readers may have noted that we need to render the template twice, based on how we implemented the signature method and the signature module. We could use an internal variable or some other mechanism to only render once, but the point of this signature was simplicity, not speed, so we won't get into it here.

8.3.4. A Fast Example: Model Data

In this example, we use the data of the model to generate a signature. This will be superior to the MD5 method because we can abort the rendering of HTML by doing a simple string comparison. The only caveat here is that we need to be sure that our view only depends on model data, so anything passed into the template should be included in the signature.

First, let's look at our template:

```
Templates.Appointment = _.template(
  "<div class='title'>{{ title }}</div>" +
  "<div class='date'>{{ date }}</div>"
);
```

As we can see here, our template only depends on the `title` and `date` attributes. So, let's define our signature so that it only depends on those attributes:

```
Views.AppointmentData = Backbone.View.extend(_.extend({
  template: Templates.Appointment,
  render: function() {
    if (this.updateSignature()) {
      this.$el.html(this.template(this.model.toJSON()));
    }
    return this;
  },

  signature: function() {
    return [
      this.model.get('title'),
      this.model.get('date')
    ].join(';');
  }
}, Modules.SignedView));
```

This new method has two advantages:

1. We compute and compare the signature without rendering the template
2. We are not depending on an additional library for our signature computation (since we are already using jQuery)

Now, when the model changes, `change` events would fire for attributes like `read` or `favorite`. Since we only care about `title` and `date`, we will not re-render the view unless those attributes change. The trade-off here is that the programmer needs to keep the signature in sync with the template.

Tip

An even simpler solution in this case would be to remove the binding of `change` to `render` and replace it with a bind of `change:title` and `change:date` to render. That way, we only re-render when those attributes change. However, this solution is limited to situations in which the template only depends on attributes and not computed information performed by the view.

Chapter 9. Fill-In Rendering

9.1. Introduction

Fill-In Rendering is a form of caching to minimize the amount of structural redraws performed when updating the DOM by keeping the structure in the template and then updating elements' contents with attributes from the models directly.

9.2. The Problem

Fill-In Rendering tackles the same problem as View Signatures. Whenever a view is rendered the following costs are inflicted on the user:

1. The browser must render the view. This takes time and effort, even if only a small amount
2. The view will "blink" while the application empties the element and fills it in with new content
3. Scrolling can be interrupted if the view is long and the user has scrolled down into it

View Signatures handle this problem very quickly by short-circuiting rendering when there is no change in the view. However, often data changes and views must be updated to reflect new information. We still want to avoid flickering and scroll interruption, while at the same time minimizing the amount of work the browser has to do. This is where Fill-In Rendering comes into play.

Tip

Fill-In Rendering and View Signatures handle the same problems, but they are not mutually exclusive. In fact, they work very well together!

9.3. The Solution

Fill-In Rendering **only updates the portions of the view that have changed**, while leaving the scaffolding of the view in place. This means that the DOM element should not have its HTML set as a whole. Instead the dynamic attributes will have HTML set individually.

Here is the original template and render method for an `Appointment`:

```
Templates.Appointment = _.template(
  "<div class='title'>{{ title }}</div>" +
  "<div class='date'>{{ date }}</div>"
);
```

```
Views.Appointment = Backbone.View.extend({
  template: Templates.Appointment,

  initialize: function(options) {
    this.listenTo(this.model, 'change', this.render);
  },

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

Note how any time the view is rendered, the `title` and `date` `<div>` tags will be recreated and reset on the DOM element to which the view is bound. This is not *too* bad for `Views.Appointment`, but consider if there was a small static description of what a appointment was and its significance. Every time we have to re-render a bunch of appointments, that text would be reset, which would cause all the divs to bounce up (as the text was removed) then spring back down as they are individually populated.

Let's take a look at how we would implement Fill-In Rendering on `Views.Appointment`:

```
Templates.Appointment = _.template(
  "<div class='title'></div>" +
  "<div class='date'></div>"
);
```

```
Views.Appointment = Backbone.View.extend({
  template: Templates.Appointment,

  initialize: function(options) {
    this.listenTo(this.model, 'change', this.render);
    this.$el.html(this.template());
  },

  render: function() {
    this.$('.title').html(this.model.get('title'));
    this.$('.date').html(this.model.get('date'));
    return this;
  }
});
```

First, we have modified the template such that it no longer renders any dynamic information. It only fills the HTML structure of the view. Second, we have rendered the HTML of the template during initialization. Third, when asked to render, we simply fill in `title` and `date`.

You may be asking yourself, "why is it OK to render during initialization?" On initialization Backbone creates an element, `el`, *not attached to the DOM* for us to manipulate when the view is initialized. It is a good practice to have the caller of a View be responsible for attaching the element to the DOM, and it will do so after it has also called render. Now, subsequent calls to `render` on a `change` event only modify the properties of the DOM that are dynamic.

9.4. A Quick Refactor

Now that we have the Fill-In Rendering pattern defined, let’s look at how we might refactor it:

```
Modules.FillInView = {
  fillInAttributes: function() {
    _(this.fillInBindings).each( function(value, key) {
      this.$(key).html(this.model.get(value))
    }, this);
  }
};
```

```
Views.AppointmentIntrospect = Backbone.View.extend(_.extend({
  template: Templates.Appointment,

  fillInBindings: {
    '.title': 'title',
    '.date': 'date'
  },

  initialize: function(options) {
    this.listenTo(this.model, 'change', this.render);
    this.$el.html(this.template());
  },

  render: function() {
    this.fillInAttributes();
    return this;
  }
}, Modules.FillInView));
```

With that, all we have to do is define a `fillInBindings` object on the view such that the keys are jQuery selectors and the values are model attributes. Then we can call `fillInAttributes` to update the view with the newest information.

Warning

In the examples in this section, we use `model.get`. This is only for data that is already javascript and HTML-safe. If you are rendering user data, use `model.escape`.

9.5. Conclusion

Implementing the Fill-In View pattern results in less page blink, less scrolling interruption, and faster render times. It also lets us use animations to transition changes in elements, which we will cover more in Chapter 10, *Actions and Animations*. Keep in mind that like any caching strategy, this pattern has its trade-offs. Pure templating is a simpler solution and therefore is less prone to developer error.

Chapter 10. Actions and Animations

10.1. Introduction

Due to Backbone's evented nature, Actions and Animations are handled differently during the course of manipulating objects. In this chapter we cover how to bind animations in common scenarios.

10.2. The Problem

Consider a very common feature of many web applications: when something is deleted or removed, its UI element fades away or zips up into nothing. Normally this is implemented by daisy-chaining the UI element removal action to the delete action. Here is an example in jQuery:

```
$.ajax({
  url: "/objects/"+object.id,
  type: 'DELETE',
  success: function() {
    $('object-'+object.id).fadeOut(250);
  }
});
```

We could adapt this exact process pretty easily in Backbone, here is an example of **the wrong way to fade out the view**:

```
/* Hey this code is a bad idea, so don't copy it! */
appointment.destroy({success: function(model, response) {
  $(model.view.el).fadeOut();
})
```

This looks really similar to our jQuery solution, so what's wrong with it?

First, fading out the view is not dependent on the destroy call finishing—it is dependent on the model actually being deleted client-side, which will be conveyed by its state. Second, the model should not have a reference to its view. This is a common shortcut that causes a lot of unidiomatic backbone code. The view should handle any of its actions based on the model's state.

Warning

If you are using callbacks on Backbone actions, **you're doing it wrong!** There are a few rare exceptions to this rule, but you should think twice before using a callback.

10.3. The Solution

From a user's perspective the action of deleting an item and seeing it go away are tightly intertwined. However, are these actions truly directly related? Actually, there is an alternate view of the situation that is much more aligned with how Backbone operates:

1. We ask the server to delete the object, it responds OK
2. The object is updated and is marked as deleted (client-side by Backbone's sync on the model)
3. The UI element is removed because its model was deleted

As server-side programmers, we are always drawn to the fact that the server said OK, confirming the object was deleted. However, from the client-side perspective, we do not care what the server says—we just care about the state of the object. Therefore instead of daisy-chaining the removal of the UI element after the successful response from the server, we will *bind the removal of the UI element to the change in the model.*

Let's see what it would look like to modify our `AppointmentView` so that when we delete an appointment, it fades out the view:

```
Views.Appointment = Backbone.View.extend({
  events: {
    "click .delete": "delete"
  },
  template: Templates.Appointment,
  initialize: function(properties) {
    this.listenTo(this.model, 'change', this.render);
    this.listenTo(this.model, 'destroy', this.remove);
  },
  render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
  },
  remove: function() {
    this.$el.fadeOut(250);
  },
  delete: function() {
    this.model.destroy();
  }
});
```

Pretty simple! Now, any time an object is removed, we fade out the element instead of just removing it.

In a more traditional web application, we might have half a dozen places where an object is deleted. This would require an update to each one with the new behavior. But in Backbone, the deletion of the object is announced with an event to which any object can listen.

Let's take a look at another possible evented change: marking an Appointment as a favorite:

```
Views.FavoriteAppointment = Backbone.View.extend({
  events: {
    "click .set-favorite": "setFavorite"
  },
  template: Templates.Appointment,
  initialize: function(properties) {
    this.listenTo(this.model, 'change', this.render);
    this.listenTo(this.model, 'change:favorite', this.updateFavorite);
  },
  render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
  },
  setFavorite: function() {
    this.model.save({favorite: (!this.model.get('favorite'))});
  },
  updateFavorite: function() {
    this.$el.toggleClass('favorite', this.model.get('favorite'));
  }
});
```

First off, we describe an event. When a user clicks a DOM element with the class `favorite`, we want to run the `setFavorite` method. `setFavorite` will toggle the model's favorite status. Now, look at the initializer: we are binding a change on the favorite attribute to run the `updateFavorite` method. `updateFavorite` is going to toggle a CSS class on the element to mark it as a favorite or not.

The most important and relevant thing to note in this code is that the `save` call to modify the model **does not have a callback**. We have bound all our functionality to events. Thus, we have fully decoupled the operation and its effects.

10.4. Conclusion

Decoupling actions and outcomes is the Backbone Way. Cleaner and less coupled code arises from removing dependencies between objects, and allowing objects to pass messages to each other. In Backbone, message passing is handled through events, and the best way to decouple Backbone objects is to expose a clear API of events to which concerned objects can bind themselves. As we have seen in this chapter, using events to pass messages yields clean and flexible code when dealing with animations.

Chapter 11. Reduced Models and Collections

11.1. Introduction

Reduced Models and Collections is a pattern that describes how to maintain and display aggregated data cleanly by creating meta-models and collections that represent computations performed on the first-class data objects of an application.

11.2. The Problem

Oftentimes you will find yourself wanting to aggregate information for display by Backbone views. The best approach for doing this might not be obvious at first. By leveraging the law of separation of concerns, it is possible to come up with an elegant, maintainable solution.

For this recipe, consider a calendar application backed by a server with a standard RESTful endpoint for `Appointments`. We already have the calendar working—creating appointments, displaying them on the calendar, deleting them, and updating them works just fine. Now we want to make a calendar sidebar that has output that includes:

- January (4)
- February (2)
- March
- April (7)
- etc...

All of the information required to render this sidebar is already available. It is sitting in our collection and models. So let's explore a couple of approaches to get this done.

11.3. The Solution

11.3.1. Simple Solution: A View

The easiest thing to start with is a (somewhat) simple view like this:

```
/* Note: this code may not run. As it is not the ideal solution
   it has not been thoroughly tested. */
Views.MonthCountView = new (Backbone.View.extend({
  initialize: function() {
    this.listenTo(this.collection, 'all', this.render);
  },
  render: function() {
    $(this.el).empty();
    var months = this.collection.reduce(function(memo, appointment) {
      var monthNumber = appointment.get('monthNumber');
      if (!memo[monthNumber]) {
        memo[monthNumber] = 0;
      }
      memo[monthNumber]++;
      return memo;
    }, {});

    _(_(months).keys()).each(function(monthNumber) {
      var monthName = MyCalendar.NumberToMonthName(monthNumber);
      var count = months[monthNumber];
      if (count > 0) {
        $(this.el).append('<li>'+monthName+' ('+count+')</li>');
      } else {
        $(this.el).append('<li>'+monthName+'</li>');
      }
    });
  }
}))({el: $('#month-count'), collection: MyCalendar.Appointments});
```

Let’s walk through this code. In the initializer we are watching the collection. Whenever anything changes we re-render our `MonthCountView`. We are using an Instantiated View pattern here since we bind to a DOM id of `month-count`.

In `render`, we reduce the appointments to an object that will look like this:

```
{
  1: 4,
  2: 2,
  3: 0,
  4: 7
  /* etc */
}
```

The key of this data structure is the month number. The value is the number of appointments in that month.

Next, we iterate over the keys (the `monthNumber`) and exchange the `monthNumber` (e.g. 1) for a `monthName` (e.g. "January"). The `NumberToMonthName` method, which is defined elsewhere in our app, handles this for us. Next, we grab the count from our reduced object.

Lastly, we generate our string and append it to our view. We append `li` elements under the assumption that `#month-count` is a `ul` element.

Some observations about this solution:

1. It works. This does solve our requirement of having a month count list.

2. There is a lot of non-view code in this view. We access a collection that does not directly pertain to the view being rendered. We are accessing the `Appointments` collection but we are not rendering `Appointments`. That is a Backbone code smell.

3. There are only 6 lines of view code. When we empty the element and when we fill it with `li` elements.

4. Our reduced data is only accessible within this view. This are no means for anything else in the application to access the per-month count. If

another view needed this information, it would be forced to calculate it itself, violating the DRY principle [1].

5. The reduced data is not in the form of objects, but as pure JSON with an implicit structure, which makes it brittle.

11.3.2. Better Solution: A Reduced Collection

You might have noticed that this is a Collection View. We are rendering a Collection of aggregated `MonthCount` objects. So what we really want to do is implement the Collection View pattern with a Collection of `MonthCount` models, a `MonthCounts` view, and a `MonthCount` view. Let's take a look at each of these pieces from the bottom up.

The model is just going to be a storage system for attributes and a source of appointments:

```
Models.MonthCount = Backbone.Model.extend();
```

The Collection is where things get interesting. We define a `recompute` method whose job is to reduce `Appointments` down to a collection of `MonthCount` models:

[1]DRY = Don't Repeat Yourself

```
Collections.MonthCounts = Backbone.Collection.extend({
  model: Models.MonthCount,
  initialize: function(models, options) {
    this.collection = options.collection;
    this.listenTo(this.collection, 'all', this.recompute);
    this.recompute();
  },
  recompute: function() {
    this.reset(_.map(_.range(0,12), function(monthNumber) {
      var count = this.collection.select(function(model) {
        return model.getMonth() === monthNumber;
      }).length;

      return {
        monthName: Models.Appointment.numberToMonth(monthNumber),
        count: count
      };
    }, this));
  }
});
```

We iterate over each month number, finding all of the appointments for this month and counting them, then we return an object in the following format:

```
[
  {monthName: "January",  count: 4},
  {monthName: "February", count: 2},
  {monthName: "March",    count: 0},
  {monthName: "April",    count: 7}
  /* etc */
]
```

That array is passed directly to the `reset` method on the `MonthCounts` collection to empty it out and populate it with models with those attributes.

This is a nice encapsulation of the reduction into a Collection. It only deals with the collection it was initialized with, the models in that collection, and `MonthCount` model. There is absolutely no view code here at all.

Let's take a look at what the `MonthCountsView` looks like:

```
Views.MonthCounts = Backbone.View.extend({
  initialize: function() {
    this.listenTo(this.collection, 'all', this.render);
  },
  render: function() {
    this.$el.empty();
    this.collection.each(function(monthCount) {
      this.$el.append(
        (new Views.MonthCount({model: monthCount})).render().el
      );
    }, this);
  }
});
```

This is a straightforward Collection View. We initialize it with a `MonthCounts` collection, to which it binds all the appointments on the collection to render. Rendering empties out our div and fills it with the DOM elements of `MonthCount` views for each of the `MonthCount` models in our collection.

This is a beautiful separation of concerns. All that `MonthCountsView` worries about is aggregating subviews for its collection. There are no reductions being done. There is not even any templating!

Next, let's look at the view for the `MonthCount`:

```
Views.MonthCount = Backbone.View.extend({
  tagName: 'li',
  render: function() {
    var text = this.model.get('monthName');
    if (this.model.get('count') > 0) {
      text += " ("+this.model.get('count')+")";
    }
    this.$el.text(text);
    return this;
  }
})
```

This view is solely responsible for turning a `MonthCount` model into its HTML representation. Again there is no reduction or any reference to any other model or collection other than the one that we explicitly render here.

Finally, here is the code used to bootstrap this application:

```
$(function() {
  var appointments = new Collections.Appointments([
    {title: 'Doctor Appointment', date: '2011-01-04'},
    {title: 'Birthday Party',     date: '2011-01-07'},
    {title: 'Book Club',          date: '2011-02-14'}
  ]);

  var month_counts = new Collections.MonthCounts(
    null, {collection: appointments}
  );

  (new Views.MonthCounts({
    collection: month_counts, el: $('#month-counts')
  })).render();
});
```

11.4. Conclusion

Although our original solution required fewer lines of code, there are several benefits of a Reduced Models and Collection approach:

1. it is re-usable in other areas of the application

2. it is much cleaner and more readable in its individual parts

3. it is made of components that have limited concerns

The most important goal achieved by restructuring our solution is that **every individual component performs only a single task**. The `MonthCountsView` only generates `MonthCountView` views from the `MonthCounts` collection. The `MonthCounts` collection only reduces `Appointments` into `MonthCount` models. The `MonthCount` view only renders a `MonthCount` to HTML. If we need to fix a bug, we know exactly where to look and we know that the extent of the bug will be limited to the object that is failing to perform its duty.

Chapter 12. Non-REST Models

12.1. Introduction

The Non-REST Models pattern covers how to introduce actions into Backbone models that cannot use the default `save` method because the server is not purely REST-ful.

12.2. The Problem

Most web developers would agree that REST is the way to go for clear and accessible APIs. In real life, however, we are often treated to APIs that are anything but easy to use. Sometimes we have to interact with a legacy system, or some special route in our system has to be different in one way or another. Let's look at a couple of situations we might find ourselves in and how to handle them in Backbone.

12.3. The Solution

12.3.1. Special Action

Let's consider our Calendar application again. Our `Appointments` are RESTful, except for one new action: `publish`. Instead of publishing an `Appointment` appointment by saving it with `published: true`, we have a special route in our system `/appointments/<id>/publish` to which we need to `POST`.

In a situation like this, we almost always want to use the normal `save()` method. For the usual `create` and `update`, things should flow through Backbone's `save()`. But, for our "special" API call, we create a separate method:

```
Models.Appointment = Backbone.Model.extend({
  /* the rest of the model is normal */
  publish: function() {
    var model = this;
    var options = {}
    options.url = this.url() + '/publish';
    options.data = {};
    options.success = function(resp, status, xhr) {
      model.set({published: true});
    };
    return (this.sync || Backbone.sync).call(this, 'create', this, options);
  }
});
```

We define a `publish` method on the model that creates the appropriate `Backbone.sync` request. We set the `url` to include a trailing `/publish`. Our `Appointment` model's url is normally `/appointments/<id>` so we turn that into `/appointments/<id>/publish`.

Next, we explicitly set `data` to an empty object so as not to confuse the server with the model parameters, which `sync` would send for us by default. The last option is a `success` callback to set the model's `published` attribute to true if the call succeeds.

The final line is either calling this model's overridden `sync` method or Backbone's `sync` method. We have not overridden `sync`, but it is a good practice to include this check to future-proof code (hint: we do it in this chapter). Calling the sync method with the `create` action will be translated to `POST`. In addition to the action, we also pass the usual model and options arguments to `sync`.

Tip

When we set out to tackle this problem, we could have simply called jQuery directly instead of using Backbone's `sync`. It may have been quicker in the short term, but it is generally a good idea to try to stick to the norm so that future code can continue to follow Backbone's practices. For example, by calling `set`, the event `change:published` will now be fired after this completes. By using `url()` the `publish` method will change with the model if the main `url` changes.

12.3.2. Special Persistence Layer

By default, Backbone's `sync` uses jQuery to make HTTP requests to a remote RESTful resource. But what if we wanted to change this around entirely and just use HTML5 local storage? Luckily, since all persistence is done through `Backbone.sync`, all we have to do is override that method for each case. In fact, the awesome folks at DocumentCloud have already written a Backbone local storage module [1].

Rather than introducing an alternate scenario and solution and explaining how to achieve a special persistence layer in that fashion, let's walk through DocumentCloud's solution and explain how it fits in to the big picture. This discussion won't cover how the Local Storage module is implemented—only how they replace `Backbone.sync` in order to store data via their module:

[1]The Backbone local storage module can be found at: http://documentcloud.github.com/backbone/docs/backbone-localstorage.html

```
/* Source code from:
 * http://documentcloud.github.com/backbone/docs/backbone-localstorage.html
 */
Backbone.sync = function(method, model, options) {

  var resp;
  var store = model.localStorage || model.collection.localStorage;

  switch (method) {
    case "read":    resp = model.id ? store.find(model) : store.findAll(); brea
    case "create":  resp = store.create(model);                            brea
    case "update":  resp = store.update(model);                            brea
    case "delete":  resp = store.destroy(model);                           brea
  }

  if (resp) {
    options.success(resp);
  } else {
    options.error("Record not found");
  }
};
```

There are four possible methods for `sync`:

1. `read`: find a model by id or retrieve all the models
2. `create`: create a model with the given attributes and set the model's `id` on success
3. `update`: update the given model's attributes
4. `delete`: delete a model from the storage layer

In each of these cases, we delegate to the local storage module to perform the corresponding action. To support Backbone, a storage solution only needs to support these four actions and return either a success or failure response.

Notice that there are no mentions of firing events in this code. This is because events are fired *above* the `sync` method in the caller. So, for an `update`, the caller would be the `save` method, which looks like this:

```
/* Set a hash of model attributes, and sync the model to the server.
 * If the server returns an attributes hash that differs, the model's
 * state will be `set` again.
 */
save : function(attrs, options) {
  options || (options = {});
  if (attrs && !this.set(attrs, options)) return false;
  var model = this;
  var success = options.success;
  options.success = function(resp, status, xhr) {
    if (!model.set(model.parse(resp, xhr), options)) return false;
    if (success) success(model, resp, xhr);
  };
  options.error = wrapError(options.error, model, options);
  var method = this.isNew() ? 'create' : 'update';
  return (this.sync || Backbone.sync).call(this, method, this, options);
}
```

Notice how `save` looks a lot like our `publish` method? This is because we modeled our `publish` method after the `save` method! All `save` does is set the attributes to the model, call `sync`, and signal success or failure. On success, it sets the response from the server to the model. On failure, it runs the error callback or triggers the `error` event. By stacking very simple layers on top of each other to form models, Backbone allows us to override any section along the way to suit our needs.

Tip

You can override the `sync` method on just one particular model, or even just one instance of a model, or a collection, or an instance of a collection. This is useful if most models are RESTful and a few are not. Or if most models are server-side, but some models are kept client-side.

12.4. Conclusion

Exceptional cases always occur, and the true strength of any framework is how it handles going off the beaten path. Backbone is an effective tool to be used when it fits the RESTful task at hand, but that is not always the case. Luckily, through methods like `sync`, `set`, and `save`, it is easy to circumvent Backbone’s naturally RESTful style without having to do any monkey-patching. Do not be afraid to step outside the realm of Backbone and write "plain old javascript." When your application’s behavior does not fit with Backbone’s style, go your own way (but try to maintain interoperability).

Chapter 13. Changes Feed

13.1. Introduction

The Changes Feed pattern provides a lightweight way of keeping your data in sync with the server by applying differences as opposed to reloading data.

13.2. The Problem

The user of a web application is not always the only person (or thing) modifying the underlying data. For example, in our calendar application, multiple users could be modifying appointments on the calendar at the same time. An obvious solution to this issue would be periodically running `fetch` on our collections to make sure they are up-to-date. However, this solution has a few problems because a full fetch may:

1. Be a slow operation on the server
2. Re-render a large number of views
3. Incur a large amount of client-side processing
4. Disrupt the browser with a large number of changes

A much better solution would be a changes feed that will only send what has changed since the last time we checked.

13.3. Changes feed on a Collection

Since Collections are responsible for adding, changing, and destroying Models, they are an ideal place for a changes feed. Let's look at how we would implement a changes feed for our calendar:

```
Collections.AppointmentChanges = Backbone.Collection.extend({
  model: Models.Appointment,
  url: "/appointments",
  initialize: function(models, options) {
    this.collection = options.collection;
    this.on('reset', this.processChanges, this);
    setInterval(this.changes, 15*1000);
  },

  since: function() {
    return this.collection.max(function(appointment) {
      return appointment.get('updated_at');
    });
  },

  changes: function() {
    this.fetch({ data: { since: this.since() } });
  },

  processChanges: function() {
    this.each(this.processChange, this);
  },

  processChange: function(appointment) {
    var existing = this.collection.get(appointment.id);
    if (existing) {
      if (appointment.get('deleted')) {
        this.collection.remove(existing);
      } else {
        existing.set(appointment.attributes);
      }
    } else {
      this.collection.add(appointment);
    }
  }
});
```

First of all, we're setting the URL to the same as the url for `Calendar.Appointments`, but when we fetch changes we're passing a since parameter:

```
Collections.AppointmentChanges = Backbone.Collection.extend({
  model: Models.Appointment,
  url: "/appointments",
  changes: function() {
    this.fetch({ data: { since: this.since() } });
  },
  // ...
})
```

The URL in this collection is just a normal `index` call on the `/appointments` route. We're passing the since parameter to fetch so that the server can send change objects instead of the full index. Our implementation here depends on a change object looking exactly like a normal object in the case of an addition or update, and with `deleted: true` in case of a deletion.

The `since` method simply gets the maximum `updated_at` timestamp from all of our appointments and sends that to the server.

```
  since: function() {
    return this.collection.max(function(appointment) {
      return appointment.get('updated_at');
    });
  }
```

If we have no appointments, `since` will be `''`. In that case, the server will treat it like a normal index call.

We also setup a 15 second interval to call the changes method. Thus, we are continuously polling the server for changes:

```
  initialize: function(models, options) {
    this.collection = options.collection;
    this.bind('reset', this.processChanges);
    setInterval(this.changes, 15*1000);
  },
```

If the polling call emits a `reset` event, we run the `processChanges` method.

At this point, our changes collection is populated with a bunch of `Appointment` objects. These appointments represent `Appointments` that have changed since the timestamp. The `processChanges` method is simply an iterator that calls `processChange` on each `Appointment`.

```
processChange: function(appointment) {
  var existing = this.collection.get(appointment.id);
  if (existing) {
    if (appointment.get('deleted')) {
      this.collection.remove(existing);
    } else {
      existing.set(appointment.attributes);
    }
  } else {
    this.collection.add(appointment)
  }
}
```

`processChange` performs the following actions:

1. Search the main collection to see if we already have the `Appointment` in our system

2. If we have the `Appointment`, check to see if it has been deleted (we would implement this server side with a `deleted` boolean), and if it has, remove if from our main collection

3. If it has not been deleted, `set` its attributes to the new ones from the server, because some attribute change has occurred

4. If we do not have an existing `Appointment`, it means it is new and needs to be added to our collection

The great thing about this changes feed is that it is so simple. All we have to do is propagate the change to a corresponding `add`, `update`, or `destroy` call on the model or collection. None of our code needs to know that a changes feed even exists! Also, `AppointmentChanges` only needs to know about the collection it's instantiated with—nothing about views, the router or anything else!

13.4. Conclusion

As in Chapter 12, *Non-REST Models*, sometimes we need to step outside of Backbone's RESTful roots to extend our application. Case in point is the Changes Feed, in which a Backbone collection is not representing data directly. Rather it is a kind of metadata: a collection of change objects. Collections and models do not have to correspond directly with a database table on the server. On the contrary, some of the most powerful and intriguing uses of collections and models are driven by metadata and interact with first-class data objects as a result.

Chapter 14. Pagination and Search

14.1. Introduction

Pagination and Search is a set of recipes that provide an easy way to display and navigate a large amount of information quickly via a user defined-query or by dividing data across pages.

14.2. The Problem

There comes a time in any application's life when there is just too much content to display at one time. Two tactics for displaying more targeted information to a user are pagination and search. Search is a feature where a user provides a query and the server sends back all of the objects that match that particular `query`. Pagination is when the server will only present a certain number of elements at a time (known as the `per_page` number) and the user can then navigate to the next `page`. Often, these two features are found together as a search results page that is paginated.

The simplest solution would be to retrieve all of the data from the server and then perform the pagination or search on the client-side. This can work up to a certain point, but performance becomes an issue on both the client and the server.

Alternatively, we can do the pagination and search on the server, which can be optimized with indexes on the persistence solution. This allows us to only retrieve the exact number of elements needed to the client, right when they want it. In Backbone, both pagination and search are implemented in a Collection in a very similar manner. Let's take a look.

14.3. The Solution

14.3.1. Search

Search is the easier of the two features, so we start there. In search, a user often asks for a standard index route with with an additional `q` parameter, representing a query. For example:

```
http://example.com/widgets?q=brown
```

This URL would return us widgets that matched the query `brown`. We would expect to receive the same sort of object that would be returned from the plain `widgets` url. In Backbone, we need to modify our `url` parameter on a collection to retrieve objects that match a user-given query. Let's look at what our `AppointmentsCollection` would look like if we wanted to show only appointments that matched the query `conference`:

```
MyCalendar.ConferenceAppointmentsCollection = new (Backbone.Collection.extend({
  model: Models.Appointment,
  url: '/events?q=conference'
}))();
```

Now we have a `ConferenceAppointmentsCollection` that only contains appointments that match the query `conference`. Clearly this is only useful if our users only search for conferences. Let's modify this so that the url can be dynamic based on a query:

```
MyCalendar.AppointmentsCollectionSearch = new (Backbone.Collection.extend({
  model: Models.Appointment,
  url: function() {
    if (this._query) return '/appointments?q='+this._query;
    else             return '/appointments';
  },
  query: function(q, options) {
    this._query = escape(q);
    this.fetch(options);
  },
  all: function(options) {
    this._query = false;
    this.fetch(options);
  }
}))();
```

Now we can search for conferences by running `MyCalendar.AppointmentsCollection.query('conference')` and then later reset it by running `MyCalendar.AppointmentsCollection.all()`. After setting or resetting our query we will automatically fetch the collection, which will fire the collection's appointments. In turn, this will trigger the appropriate views to be re-rendered, displaying all of the new data.

Tip

If your `url` is already a function, you can use a practice called Currying to dynamically define the function with the search parameter mixed in. Check out this Wikipedia article on currying [http://en.wikipedia.org/wiki/Currying:].

14.3.2. Pagination

Pagination is really a specific case of search. Instead of asking for a query, however, we ask the user for a `page`. Often, this is done by providing links on our view with a Next and Previous button to change pages. Let's look at how we can tie a pagination view to our collection by first examining our paginator:

```
$(function() {
  MyCalendar.PaginatorSimple = new (Backbone.View.extend({
    template: _.template("<span class='prev'>Previous</span><span class='next'>
    events: {
      'click .prev': 'previous',
      'click .next': 'next'
    },
    render: function() {
      this.$el.html(this.template());
      return this;
    },
    previous: function() {
      this.trigger('previous');
    },
    next: function() {
      this.trigger('next');
    }
  }))({el: '#paginator'}).render();
});
```

The paginator is an Instantiated View. It has a Next and Previous `span` that are linked to methods that simply fire methods of the the same name. This means that the paginator it just a relay for user interaction.

Next, lets look at our collection:

```
MyCalendar.AppointmentsCollectionPaginated = new (Backbone.Collection.extend({
  model: Models.Appointment,
  url: function() {
    return '/appointments?page='+this._page;
  },
  initialize: function() {
    this._page = 1;
  },
  nextPage: function() {
    this.changePage(1);
  },
  prevPage: function() {
    this.changePage(-1);
  },
  changePage: function(delta) {
    this.setPage(this._page + delta);
  },
  setPage: function(page) {
    this._page = page;
    this.fetch();
  }
}))();
```

The collection has a dynamic url that is set via a function. We keep an internal `_page` variable and provide functions to change the page by incrementing or decrementing it. When the page is changed, we fetch the collection.

At this point, all that remains is to wire these two objects together:

```
MyCalendar.Paginator.on(
  'next', MyCalendar.AppointmentsCollectionPaginated.nextPage,
          MyCalendar.AppointmentsCollectionPaginated.nextPage
);
MyCalendar.Paginator.on(
  'previous', MyCalendar.AppointmentsCollectionPaginated.prevPage,
              MyCalendar.AppointmentsCollectionPaginated.nextPage
);
```

When the user clicks Next the paginator fires a `next` event, which we bind to the `nextPage` method on the collection. We may want to add a bit to these objects to make the user experience a little better. For example, we could output the current page in the collection's view so that the user knew relatively where they where. We could also have the paginator listen to a `change:page` event on the collection and display the page number with the Next and previous controls. This is left as an exercise to the reader.

Lastly, a common feature of paginators is to show the total number of pages. This should be done as a separate call to the server, as including this metadata in the collection fetch would be confusing and hard to extract. A simple page metadata model might look like this:

```
MyCalendar.AppointmentPages = new (Backbone.Model.extend({
  url: '/appointments/pages'
}))();
```

We would implement this server route to return a metadata object in a structure such as:

```
{ pages: 42, per_page: 10 }
```

Next, we create a `PageView` to represent a single page:

```
Views.PageView = Backbone.View.extend({
  tagName: 'span',
  className: 'page',
  events: { 'click': 'page' },
  initialize: function(options) {
    this._page = options.page;
  },
  page: function() {
    this.trigger('page', this._page);
  },
  render: function() {
    this.$el.text(this._page);
    return this;
  }
});
```

The `PageView` renders a single page and fires a custom `page` event when it is clicked. We pass the page number in to its initializer and attach it as an event argument when we fire. Next we update our Paginator to follow the Collection View pattern:

```
$(function() {
  MyCalendar.Paginator = new (Backbone.View.extend({
    template: _.template("<span class='prev'>Previous</span><span class='pages'
    events: {
      'click .prev': 'previous',
      'click .next': 'next'
    },
    initialize: function() {
      this.listenTo(this.model, 'change', this.render);
    },
    render: function() {
      this.$el.html(this.template());
      _(this.model.get('pages')).chain().range().each(function(page) {
        var view = new Views.PageView({page: (page+1)});
        this.$('.pages').append(view.render().el);
        this.listenTo(view, 'page', this.page);
      }, this);
      return this;
    },
    previous: function() {
      this.trigger('previous');
    },
    next: function() {
      this.trigger('next');
    },
    page: function(page) {
      this.trigger('page', page);
    }
  }))({el: '#paginator', model: MyCalendar.AppointmentPages}).render();
});
```

The Paginator now takes a model that represents the page metadata. Whenever the metadata model changes we re-render the paginator. The render method here is the same as the Collection View pattern, except we bind the `page` event on the subview to our own `page` method, firing our own `page` event. By coalescing the events of the subviews into this one event on the paginator, we can use our collection's `setPage` method to set the collection to a specific page and simply bind our paginator's page method directly to setPage:

```
MyCalendar.Paginator.bind(
  'page', MyCalendar.AppointmentsCollectionPaginated.setPage
);
```

Because the `page` event fires with a single argument representing the page, and the `setPage` method expects one argument representing the page, we can wire these two up directly. Now, whenever a user clicks the page number, it will update our collection's route and fetch it.

Warning

We did not restrict the page number to the bounds of the page metadata returned to the server. A user could continue to click `nextPage` and go beyond the max page and get empty pages. It would be a good idea to stifle the triggering of the `page`, `next`, and `previous` events on the `Paginator` if the page is out of bounds.

14.4. Conclusion

Similar to Chapter 13, *Changes Feed*, we are creating a set of meta-objects based on meta-data to interact with our first class data. By exposing information from the server in a customizable fashion, we are able to take more control of the way a user interacts with our application. This chapter also illustrates some of the more complicated interactions between a large number of objects, and shows how events can be triggered, captured, manipulated, and re-triggered to create message passing infrastructure in an application.

Chapter 15. Constructor Route

The Constructor Route pattern provides a way to seed a form with dynamic initial data when a user creates an object in a specific context.

15.1. The Problem

Standard Content Management Systems typically contain individual pages for each type of resource. These resources rarely interact with each other without requiring the user to navigate to a new page. However, in a dynamic client-side web application, different resources interact all the time. One pattern that often arises is creating new objects based on existing objects in the application.

For our calendar example, consider the process of creating a new appointment. Clearly, we can have a form in which the user selects the year, month, day, and time, then proceeds to fill in the appointment details. However, we would like to have the user be able to click on a day on the calendar and pre-fill the form with that date. The application already knows the date, so pre-populating the date should be a no-brainer. Little things like add up to respect that our application pays the user. And Backbone makes things like this fairly trivial on developers as well.

Since we are trying to make our user's lives easier, it would be a nice touch if this form was sharable. That is, if the user wants to send the link to someone else to complete or, more simply, bookmark the form—then it should be possible.

15.1.1. A simple specific route

Tip

Since the following example is the simple and non-ideal solution, the code is not guaranteed to run and some sections have been removed for brevity.

A simple solution comprises of a route that has a parameter for the year, month, and day:

```
var Calendar.Router = new (Backbone.Router.extend({
  routes: {
    "create/:year/:month/:day": "create"
  },

  create: function(year, month, day) {
    Calendar.CreateAppointmentView.setDate(year, month, day);
    Calendar.CreateAppointmentView.show();
  }
}))();
```

This route is tied to the `create()` method, which sets the date on the form view and displays it. The form view, and the resulting day view can then be defined as:

```
var Calendar.CreateAppointmentView = new (Backbone.View.extend({
  template: "", //redacted
  initialize: function(options) {
    this.appointment = new Calendar.Appointment();
    this.appointment.on('change', this.render, this);
  },
  setDate: function(year, month, day) {
    this.appointment.set({year: year, month: month, day: day});
  },
  render: function() {
    this.$el.empty();
    this.$el.html(this.template(this.appointment.toJSON()));
  },
  show: function() {
    this.$el.fadeIn();
  }
}))({el: '#create-appointment'});

var Calendar.DayView = Backbone.View.extend({
  events: {
    'click': 'createAppointment'
  },
  createAppointment: function() {
    Backbone.History.navigate([
      'create',
      this.model.get('year'),
      this.model.get('month'),
      this.model.get('day')
    ].join('/'), true);
  }
})
```

The create-appointment view exposes the two methods invoked by the router: `setDate` and `show`. The `setDate` method accepts year, month, and date of the appointment, which the router pulls from the url (*e.g.* `/create/2011/01/01`). The `show` method does just that—shows the form if it is hidden. Thus, if the requested url is `/create/2011/01/01`, the router will send the appropriate date values to the view, which is then pre-populated with the date, and shown to the user.

Internally, the create-appointment view keeps an `Appointment` model to encapsulate the data being entered. For good measure, we listen for changes on this model (*e.g.* from a separate control widget in the application). If the appointment is changed, we automatically re-render..

Last is the `DayView`, in which we bind the `click` event on the `background` div (so that a click on an appointment does not trigger the event) to the creation route for that `DayView`'s `Day`.

Now, how can we improve this? It handles our problem, but it is not very flexible. Consider the case where we want to approximate the time based on where on the `DayView` we clicked. We would have to go through and update the entire chain from the `DayView` to the `Router` to the `CreateAppointmentView` so that the method signature matched the parameters. This is where a Constructor Route shines.

15.2. The Solution

The core idea is to accept object-like syntax through the entire chain so we can have a very flexible route.

In the router, two things change: the route itself and the function invoked. The route changes to accept a "splat" of options. Effectively, this route now matches any URL that begins with `/create` and stuffs the rest into the `options` variable. The `create` route handler then needs to change to handle the new options format:

```
Calendar.Router = new (Backbone.Router.extend({
  routes: {
    "create/*options": "create"
  },

  create: function(options) {
    var params = _.reduce(options.split('/'), function(memo, opt) {
      opt = opt.split(':');
      memo[opt[0]] = opt[1];
      return memo;
    }, {});

    Calendar.CreateAppointmentView.set(params);
    Calendar.CreateAppointmentView.show();
  }
}))();
```

In the create-appointment view, the `set` method becomes much simpler:

```
$(function() {
  Calendar.CreateAppointmentView = new (Backbone.View.extend({
    template: _.template([
      "<input type='text' name='year'  value='{{year}}'/>",
      "<input type='text' name='month' value='{{month}}'/>",
      "<input type='text' name='day'   value='{{day}}'/>"
    ].join('')),

    initialize: function(options) {
      this.appointment = new Models.Appointment();
      this.listenTo(this.appointment, 'change', this.render);
    },

    set: function(options) {
      this.appointment.set(options);
    },

    render: function() {
      this.$el.empty();
      this.$el.html(this.template(this.appointment.toJSON()));
    },

    show: function() {
      this.$el.fadeIn();
    }
  }))({el: '#create-appointment'});
});
```

And finally, the day view that invokes the route needs to be able to send the browser to a route that can be processed by the router and create-appointment form:

```
Views.DayView = Backbone.View.extend({
  className: 'day',
  template: _.template("{{year}}/{{month}}/{{day}}"),
  events: {'click': 'createAppointment'},
  route: _.template("create/year:{{year}}/month:{{month}}/day:{{day}}"),

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

  createAppointment: function() {
    Backbone.history.navigate(this.route(this.model.toJSON()), {trigger: true})
  }
});
```

Let's work our way from the bottom up. First, notice that `DayView` is still specifying which parameters to send, but now it's prefixing each value with a key. So our route would look like:

```
/create/year:2011/month:01/day:01
```

Also, we're using an underscore template for our route, who says views get to have all the fun? It's much more readable than the array join we were doing earlier.

Now, if we look in the `Router`, we can see we're using a splat option `*options` which will match anything after the `/create/`. This means that `options` in the `create` function will be a string like `year:2011/month:01/day:01`. So, we split this string on the `/` and reduce the array of `key:value` tuples by splitting on the `:` and accumulating a params object. Next, we take that params object and send it to `CreateAppointmentView.set`. `CreateAppointmentView.set` merely delegates to `appointment.set` since the params are assumed to translate directly to the appointment's data format.

This means that we can now build virtually any route—as long as the appointment model supports the resulting parameters—and they will be set automatically. We can include a `time` or `allDay` option in the route and it gets passed through to the model backing the view. Nifty!

Warning

We are not encoding the URI components here because we are assuming they are relatively simple. Keep in mind a URI can only contain 2048 characters. Also, if our parameters included the `/` character, this solution would break down. However, this simple strategy works in many cases were your data is short and simple. If your data is more complex or input via the user, you should run `encodeURI` on it.

15.3. Conclusion

Any route in an application can be considered data. In general, many of the routes that an application uses contain only a small amount of data: "I want to see X." However, a URI can be up to 2048 characters long, so there is plenty of room in a route to pass along a substantial message information. This chapter serves as a template for demonstrating ways of encoding dynamic data into routes. It does not just have to be used to render form data, it could filter a collection based on a parameter or be used to categorize or sort elements in a view.

Chapter 16. Router Redirection

16.1. Introduction

Although pure client-side redirection is not possible, this recipe will discuss strategies for achieving the same desired effect.

16.2. The Problem

Redirection—requesting one thing, but being sent another—is not quite as common in Backbone applications as in its server-side brethren. Even so, there are times when it is down right handy—even necessary.

In server-side web applications, redirection serves as a means to delegate an action to the appropriate destination or to send a client to a different destination after an action has been performed. For example, after creating a calendar appointment, a server-side application might redirect the user to the newly created event’s URL.

Browsers are smart enough to recognize a server-side redirect action (*e.g.* an HTTP 302 redirect), skipping them when a user hits the back button. Browsers are not quite so intelligent about client-side redirects. If a client-side route simply navigates to another route, the user cannot navigate beyond this point via the back button. The back button will re-request the client-side redirect, sending the user back to the place from which she was just attempting to navigate away.

This is definitely not ideal.

16.3. The Solution

In our Calendar application, consider the controls at the top of the application that navigate forward and back a month. If we were to store the the current month on the `ControlsView`, then we might be tempted to simply increment or decrement that number based on a user clicking "Next" or "Previous" links.

We could get into trouble if the user manually enters a route or if the application navigates between months in any other way (*e.g.* skips 6 months ahead). The counter would become out of sync with the route itself.

A simple solution to illustrate this problem might look like:

```
/* This code illustrates a BAD IDEA so don't copy it */
var Calendar.ControlsView = new (Backbone.View.extend({
  el: '#calendar .controls',

  events: {
    'click .next': 'nextMonth',
    'click .prev': 'prevMonth'
  },

  nextMonth: function() {
    window.location = 'month/next';
  },

  prevMonth: function() {
    window.location = 'month/prev';
  }
}))();

var Calendar.Router = new (Backbone.Router.extend({
  routes: {
    'month/:id': 'month',
    'month/next': 'nextMonth',
    'month/prev': 'prevMonth'
  }

  initialize: function() {
    this.month = 1;
  }

  month: function(id) {
    this.month = 1;
    /* Display the month */
  },

  nextMonth: function() {
    window.location = 'month/' + ((this.month++) % 12);
  },

  prevMonth: function() {
    window.location = 'month/' + ((this.month--) % 12);
  }
}))();
```

Once the application navigates to `month/next`, it becomes part of the browser history. Consider the following steps:

1. The user clicks the "Next" link, which is linked to the `month/next` client-side route.

2. The application maps `month/next` to the `nextMonth()` router method.

3. This `month/next` URL is stored in the browser's history.

4. The `nextMonth()` method adds *another* entry in the browser's history, as it now sets `window.location` to something like `month/2` (if we had been in November).

5. The browser comes to rest displaying February's appointments.

A user desiring to check on something something back in January might naturally click the browser's back button instead of our UI's "Previous" control. Because `month/next` is the previous resource in `window.history`, the browser would hit the wrong route. Not only is the route wrong, but is also a redirection route. Thus hitting `month/next` would also increment the internally stored `this.month` value past "2", sending the user to March.

In effect the user would get the opposite of the desired / expected effect of clicking the Back button.

From a code quality perspective, there are additional problems with our obvious approach. First, it is an annoyance to keep track of a month variable when the month is right there in the route (`month/2`). Furthermore, performing routing in the view is a brittle solution (the `window.location` call is routing code!). If we change our routes, we have to hunt through the application for these calls. It is much better to move this sort of code into the `Router`.

Thus, a better approach is to create redirection methods on our `Router` that handle modifying the route. While the `ControlsView` is responsible for triggering a next or previous action, it is not responsible for re-routing the application. It is much better to keep all of the routing logic inside our `Router`.

To accomplish this, let's examine the view first. This should look pretty standard, as we are only binding methods to the click event on the `.next` and `.prev` elements in the view:

```
var Calendar.ControlsView = new (Backbone.View.extend({
  className: 'controls',

  events: {
    'click .next': 'nextMonth',
    'click .prev': 'prevMonth'
  },

  nextMonth: function() {
    this.trigger('next');
  },

  prevMonth: function() {
    this.trigger('previous');
  }
}))();
```

Notice that we are no longer setting `window.location` nor are we calling the `Router` from the view. Instead, we fire custom events. The reasoning behind this will be discussed more in Chapter 18, *Object References in Backbone*, but the bottom line is that, since the `Router` is a higher-order object in the application, it would be inappropriate for our lowly view to give it commands. Instead, the view will fire events that the `Router` can listen to, if it deigns to do so.

In the `Router`, we bind routes a little differently than you may be used to. Generally, routes are bound via a property called `routes` that are bound to methods on initialization. Under the covers, Backbone simply passes each of those key-value pairs to the `route` method. Since we are going to re-use the month route, we are binding the route ourselves in the `initialize` method:

```
var Calendar.Router = new (Backbone.Router.extend({
  monthRoute: this._routeToRegExp("month/:id"),

  initialize: function() {
    this.listenTo(Calendar.ControlsView, 'next', this.nextMonth);
    this.listenTo(Calendar.ControlsView, 'previous', this.prevMonth);
    this.route(this.monthRoute, 'month', this.month);
  },
  // ...
}))();
```

The `this.route(...)` call has the same effect of the following entry in the `routes` object:

```
{ 'month/:id', 'month' }
```

But with the additional goodness of establishing a DRY, robust, decoupled binding between the view's next/previous click actions and the next/previous routes.

The rest of our router then becomes:

```
var Calendar.Router = new (Backbone.Router.extend({
  monthRoute: this._routeToRegExp("month/:id"),

  initialize: function() {
    this.listenTo(Calendar.ControlsView, 'next', this.nextMonth);
    this.listenTo(Calendar.ControlsView, 'previous', this.prevMonth);
    this.route(this.monthRoute, 'month', this.month);
  },
  month: function(id) {
    /* Display the month */
  },

  nextMonth: function() {
    this.moveMonth(1);
  },

  prevMonth: function() {
    this.moveMonth(-1);
  },

  moveMonth: function(by) {
    var id = this._extractParameters(
      monthRoute,
      Backbone.history.getFragment()
    )[0];
    Backbone.history.navigate(
      "month/" + ((id+by)%12),
      {trigger: true}
    );
  }
}))();
```

We run the route through an undocumented Backbone method called `_routeToRegExp`. This method converts the Backbone-style route into a RegExp we can use for matching later.

Also in the `initialize` method, we bind our own `nextMonth` and `prevMonth` methods to the `ControlsView`'s custom events. This event-oriented approach to control flow is much safer and more flexible for our application in the long-run. All routing decisions are made in the router, and not spread across various objects throughout the application.

The `nextMonth` and `prevMonth` methods are delegating to a `moveMonth` method because their functionality is extremely similar—differing only in the direction they are moving. `moveMonth` is calling another undocumented method of the Backbone Router: `_extractParameters`. This method takes a route RegExp and a URI fragment, returning us the parameters that we define in the RegExp. `Backbone.history.getFragment` will return a browser-normalized URL fragment representing what comes after the `#` (if using push-state). The end result is that our `:id` will match and we receive `[id]` as a result.

Tip

To be clear, we are using these undocumented Backbone methods to keep our routing DRY.

Now, when a user clicks the next or previous button on the view, our route will be changed to the next or previous month. We did this by firing and listening to custom events and by introspecting and manipulating our current route. This yields a cleaner and decoupled implementation. Our `Router` became smarter (by knowing everything about month routing) and our `ControlsView` became dumber by losing that same knowledge of month routing (and even all routing and the flow of the application).

Tip

Objects in Backbone should only concern themselves with behaviors related to their class name. Views should only have view behaviors, and Routers should only concern themselves with routing. All other actions should be delegated to another object (like a view calling its model) or cause an event to fire to which an interested object can listen (as in the case here where the Router listens to a view).

One final note in our updated solution is the use of `Backbone.history.navigate()` instead of setting `window.location` directly as we had before. Both ultimately do the same thing—adding a new entry to the browser's history and potentially forcing the browser to request that new page. Using `Backbone.history.navigate()` is *always* preferred because it works cross browser with push-state and normal URLs alike.

Important

Manipulating `window.location` directly is a code-smell in Backbone.js applications. Always use `Backbone.history.navigate()`.

We call `Backbone.history.navigate()` with a second argument, `{trigger: true}`. This tells Backbone that, in addition to updating the browser's history and address bar with the new month's URL, the corresponding route method should be invoked. That is, if we `Backbone.history.navigate('month/2')`, then our month route should fire, which tells the `month()` method to fire, rendering the controls view.

Without the `{trigger: true}` second argument, Backbone simply updates the browser history and address bar. The calling context would then be responsible for changing internal state as needed.

16.3.1. Default Routes

Often it makes sense to redirect from an application route or from a resource route to a more specific location. If our calendar's entry URL is http://calendar.example.com, then we might want to redirect the browser to http://calendar.example.com/#month/1 immediately after logging in.

To accomplish something like this, it is again tempting to create a "default" router method that performs a `Backbone.history.navigate()` redirection:

```
/* Don't do this! */
var Calendar.Router = new (Backbone.Router.extend({
  routes: {
    "": "default",
  },

  default: function() {
    Backbone.history.navigate("month/1");
  }
});
```

This constitutes the poor practice of breaking the user's Back button on your site, so do not do it [1]. When the browser hits the default route, two additional browser history entries are created—one for the initial entry point (http://calendar.example.com) and one for the redirected resource (calendar.example.com/#month/1). If the user clicks the Back button, the browser will hit the default route, which will promptly redirect back to the page the user was just trying to leave.

Instead of attempting a true redirect, we do it under the covers:

```
var Calendar.Router = new (Backbone.Router.extend({
  routes: {
    "": "month"
  },

  month: function(id) {
    if (typeof(id) == "undefined") id=1;

    /* Display the month */
  }
});
```

Here, we tie the default route directly to the `month()` method that would have been invoked anyway. That method subsequently needs to be a little smarter about handling undefined IDs, but that is the only real change required.

[1]Well, *maybe* on the application start page, but even there it feels henky.

Ultimately, we achieve our "redirection" to the appropriate resource, without annoying our user by breaking the Back button.

16.4. Conclusion

True redirection will not work in the browser without significant user experience consequences. Even so, we are able to mimic redirection behavior by keeping routing where it belongs: in routing objects.

Chapter 17. Evented Routers

17.1. Introduction

The Evented Router pattern uses route cleanup as an example of how to use Router events to DRY up Router code.

17.2. The Problem

In a modern client-side application, there will be a number of routes, and many ways to transition in between them. When navigating from one route to the next, the previous route needs to be cleaned up and the new route needs to be rendered. Since an application with N routes could potentially have N^2 transitions, adding a new route introduces N new possible transitions. Clearly it is unreasonable to represent every possible transition as an application grows. Let’s consider our calendar application, in which we have three main routes:

1. `month/:id`: Show a month

2. `event/:id`: Show an event

3. `event/new`: Create a new event

A simple router for this scenario might look like:

```
CalendarRouter = Backbone.Router.extend({
  routes: {
    'month/:id': 'month',
    'event/:id': 'event',
    'event/new': 'new_event'
  },

  month: function(id) {
    reset_event();
    reset_new_event();
    Calendar.MonthView.show(id);
  ),
  reset_month: function() { Calendar.MonthView.hide(); },

  event: function(id) {
    reset_month();
    reset_new_event();
    Calendar.EventView.show(id);
  },
  reset_event: function() { Calendar.EventView.hide(); },

  new_event: function() {
    reset_month();
    reset_event();
    Calendar.NewEventView.show();
  },
  reset_new_event: function() { Calendar.NewEventView.hide(); }
})
```

That router is already a little nutty. Clearly adding new routes is only going to make matters worse. What other options are there?

17.3. The Solution

To solve this problem, we will combine Backbone's Events with some metaprogramming. Whenever a Router matches a route, it will fire an event corresponding to the routing method. So visiting `/event/4` matches the route `event` and will fire `route:event`. We will add event binding to our initializer that binds the `route:X` event to the `reset_Y` method whenever X is not the same as Y. In other words, when we route to X, all the other routes will have their reset method triggered, cleaning up everything but X.

```
Calendar.Router = new (Backbone.Router.extend({
  routes: {
    'month/:id': 'month',
    'event/new': 'new_event',
    'event/:id': 'event'
  },

  initialize: function() {
    _(this.routes).each(function(destination) {
      _(this.routes).each(function(other) {
        if (destination === other) return;
        // route:x => reset_y
        this.on('route:'+destination, this['reset_'+other], this);
      }, this);
    }, this);
  },

  month:       function(id) { Calendar.MonthView.show(id); },
  reset_month: function()   { Calendar.MonthView.hide(); },

  event:       function(id) { Calendar.EventView.show(id); },
  reset_event: function()   { Calendar.EventView.hide(); },

  new_event: function() { Calendar.NewEventView.show(); },
  reset_new_event: function() { Calendar.NewEventView.hide(); }
}))();
```

Now, if we add a new route, we need a new route method for it as well as a corresponding reset method. There is no need to modify all the other methods to keep them up to date. Now, adding new routes is O(1) code modifications, instead of O(N^2)!

As in other recipes, we can extract this behavior into a helpful mixin:

```
Mixins.AutoResetRouter = {
  autoResetRoutes: function() {
    _(this.routes).each(function(destination) {
      _(this.routes).each(function(other) {
        if (destination === other) return;
        this.on('route:'+destination, this['reset_'+other], this);
      }, this);
    }, this);
  }
}
```

And now we just need to call this method in our initializer and mix it in:

```
Calendar.Router = new (Backbone.Router.extend(_.extend({
  routes: {
    'month/:id': 'month',
    'event/new': 'new_event',
    'event/:id': 'event'
  },

  initialize: function() {
    this.autoResetRoutes();
  },

  month:       function(id) { Calendar.MonthView.show(id); },
  reset_month: function()   { Calendar.MonthView.hide(); },

  event:       function(id) { Calendar.EventView.show(id); },
  reset_event: function()   { Calendar.EventView.hide(); },

  new_event:       function() { Calendar.NewEventView.show(); },
  reset_new_event: function() { Calendar.NewEventView.hide(); }
}, Mixins.AutoResetRouter)))()
```

Simply by agreeing on this nomenclature of `method` and `reset_method` we can handle all possibly transitions between all states with our mixin and we only have to implement those methods when we add a new route.

Tip

If you are working with an application with multiple routers, bind all of the reset methods to a general `reset` event. Next, define a `reset` method that triggers the reset event. Now, from router X you can call `Y.reset()` whenever X matches a route to reset all of Y's views.

17.4. Conclusion

This is definitely one of the more complicated patterns in the book. We are applying powerful metaprogramming techniques to Backbone's event structure in order to handle a combinatorial growth problem. This serves as a solid base from which to extrapolate other methods of using events in dynamic ways. For example, Backbone models fire `change` events as well as attribute-specific change events, like `change:name` when `name` changes. What interesting problems could you solve by dynamically binding to attribute change events on instantiation of a model?

Chapter 18. Object References in Backbone

Backbone.js is a loose framework. Two different developers can use Backbone with entirely different patterns and Backbone is perfectly fine with it. One of the very first pieces of example code I (Nick) read when I was first learning Backbone looked like this:

```
MyView = Backbone.View.extend({
  initialize: function(options) {
    this.model.view = this;
  }
})

MyModel = Backbone.Model.extend({
  initialize: function(options) {
    this.on('destroy', this.removeView, this);
  },
  removeView: function() {
    this.view.remove();
  }
})

var model = new MyModel();
new MyView({model: model)});
model.destroy();
```

Not knowing any better, this was the pattern I used for quite some time when I needed to remove a view after the model was removed. At this point in the book, it should be clear that this is the wrong way to remove a view. Instead, we should be using events:

```
MyView = Backbone.View.extend({
  initialize: function(options) {
    this.listenTo(this.model, 'destroy', this.remove);
  }
})

MyModel = Backbone.Model.extend({})

var model = new MyModel();
new MyView({model: model)});
model.destroy();
```

This solution is clearly simpler and more succinct. The model has no idea that a view exists at all. The model's implementation is completely independent of any other piece of the application. This is decoupling at its finest.

The view, on the other hand, does still require a reference to the model in order to perform its duties. In this example, we only use it to bind a `destroy` event listener, but Backbone retains the reference under it covers.

So it is OK for a view to hold a reference to a model, but not the other way around. Right?

To answer that question, let's delve into when and where it is appropriate to pass references around in Backbone. In particular, notice the code smell in our "newbie" code: the model knows about its view in order to have the view respond to a change in the model's state:

```
MyModel = Backbone.Model.extend({
  initialize: function(options) {
    this.on('destroy', this.removeView, this);
  },
  // ...
});
```

The model should be indifferent to its listeners. In another case, however, it is reasonable (and 100% necessary) to have a view know about a collection:

```
MyView = Backbone.View.extend({
  /* note that the `collection` attribute on the options
   * objects is automatically assigned to `this.collection`
   */
  render: function() {
    this.$el.empty();
    this.collection.each(this.addOne, this);
  },
  addOne: function() {
    view = new OtherView({model: model});
    view.render();
    this.$el.append(view.el);
  }
})
```

There is no way we could render a view for a collection without the view knowing about the collection. This brings us to the first portion of this recipe: the Precipitation Pattern.

18.1. Precipitation Pattern

The precipitation pattern encapsulates the idea that references in Backbone should flow downstream from higher-order objects. Furthermore, references should never be made back "up" the stream. The order of Backbone objects from highest to lowest is as follows:

1. Router

2. View

3. Collection

4. Model

It is OK to move "sideways", for example from one view to the next (as in the Collection View pattern). It is not OK to move "upstream", for example from a Model to a View or a View to a Router.

Do not think only about the types of objects. Consider their relationships and behavior. It would be inappropriate for a `SidebarView` to reference a `MainPanel` view if one was not the descendant of the other. However having a `TodoView` reference a `TodoControlsView` would be appropriate if the `TodoControlsView` was a child view containing a control panel for a `Todo` model represented by the `TodoView`.

Taking this metaphor another step, the water cycle on Earth sees water evaporate to clouds, which rains down at the tops of the mountains. In Backbone, Events are our rain. A Model may fire events and a View can listen to them and take action. We may even create a special customized event for a Model specifically for one of our Views to use, but we do not reference the View directly. References are the rivers that flow down from the top of the mountain to the ocean. The only way to communicate back up the chain is by raining down some events.

18.2. Dependency Injection

As our references flow downstream, there is a specific code pattern that keeps our code clean and organized: dependency injection. To illustrate how this pattern works, let's look at our Calendar view.

Here, we use a global variable, `window.Appointments`, that holds our collection of Appointment objects. The `MonthView` then binds event listeners to that global object:

```
Calendar.Collections.Appointments = Backbone.Collection.extend({
  model: Calendar.Models.Appointment
});

window.Appointments = new Calendar.Collections.Appointments();

Calendar.Views.MonthView = Backbone.View.extend({
  initialize: function(options) {
    window.Appointments.on('add', this.addOne, this);
  },
  render: function() {
    $(this.el).empty();
    window.Appointments.each(this.addOne, this);
  },
  addOne: function(appointment) {
    var view = new Calendar.Views.AppointmentView({model: appointment});
    view.render();
    this.$el.append(view.el);
  }
});

window.MonthView = new Calendar.Views.MonthView()
```

This is a brittle solution. First, we need to ensure there are no variable collisions on `window`. Secondly, future changes would necessitate that we update all such references. For example, if we find that our application needs `window.ThisMonthAppointments` and `window.LastMonthAppointments`, then we would have to replace references to `window.Appointments` with these new appointment collection slices.

In this case, `MonthView` depends on `window.Appointments`. We cannot change this dependency, but we can inject it to keep our code a little more flexible:

```
Calendar.Collections.Appointments = Backbone.Collection.extend({
  model: Calendar.Models.Appointment
});

Calendar.Views.MonthView = Backbone.View.extend({
  initialize: function(options) {
    this.listenTo(this.collection, 'add', this.addOne);
  },
  render: function() {
    this.$el.empty();
    this.collection.each(this.addOne, this);
  },
  addOne: function(appointment) {
    var view = new Calendar.Views.AppointmentView({model: appointment});
    view.render();
    this.$el.append(view.el);
  }
});

new Calendar.Views.MonthView({
  collection: ( new Calendar.Collections.Appointments() )
});
```

Notice that in this example we still have the same references but keep them local to the objects involved. Because dependency injection is such a helpful pattern, views in Backbone automatically self-assign the keys `model` and `collection`.

If multiple views were to reference this collection, we can still dependency inject the object in our application's start-up using jQuery's `onReady`:

```
$(function() {
  var collection = new Calendar.Collections.Appointments();
  new Calendar.Views.MonthView({collection: collection});
  new Calendar.Views.MonthSidebar({collection: collection});
});
```

Note that the `collection` variable will not survive outside of this method, thus avoiding the dreaded global variable.

Tip

If you are unable to inject a dependency, it may be an indication that your application architecture does not follow the precipitation pattern and should be refactored.

18.3. Conclusion

Keeping your dependencies in check is important for maintaining loosely coupled objects. By limiting the direction in which references flow and limiting the scope to which those references are made, we can reduce the number of dependencies "floating around". We also enforce that the only dependencies of an object are given to it on instantiation. Not only does this help us write better code, it is an integral part of writing code that is easy to unit test [1].

[1]Check out Appendix A, *Getting Started with Jasmine* if you are interested in unit testing Backbone.

Chapter 19. Custom Events

19.1. Introduction

Custom events are a loose mechanism for communicating change between concerns without the risk of collisions with built-in browser and Backbone events.

19.2. The Problem

Events make life worth living. They make coding fun. They solve all of the problems in the world. Except when they don't.

Even after eliminating as many callbacks as possible. Even after dutifully following the Precipitation Pattern. Even after producing the most beautiful Backbone code possible, it is still possible to find yourself in event hell.

Event hell is what happens when you find yourself responding conditionally to events. That is, display this sub-view when the model updates, but only if the collection has this number of models in it. Otherwise, display the non-compact message.

Events are a fantastic mechanism to decouple the various components of a Backbone application. But once there are many events flying, it is almost a necessity to start using custom events.

19.3. The Solution

Backbone invented neither events nor custom events. Events have been part of Javascript for as long as Javascript has been around. Custom events are a natural outgrowth of Javascript's event driven nature. They allow developers to realize the power of DOM events in their own applications.

Since Backbone is not a DOM framework like jQuery, its events are not clicks and mouseOvers. Rather, Backbone's events describe actions that take place in its models and collections. Its events are things like: "add", "remove", "change", "destroy", and the dreaded "error" event.

Backbone also includes a special little event, named `all`, which pretty much does what you might expect. If you are listening for `all` events, your listener will receive *all* events tossed by Backbone parts.

As we have seen throughout this book, there is much power in these events. Naturally, Backbone does not restrict you to just this subset of events.

In fact, Backbone itself fires namespace attribute events when an attribute in the model is updated. For instance, if the title attribute is changed, Backbone models automatically fire a `change:title` event.

There are any number of reasons that we might want to use our own custom events in Backbone applications. As mentioned in the introduction, using custom events helps to further decouple design and to prevent conflicts with built-in Backbone events.

19.3.1. Application Triggered Events

To see this in action, consider the month view of our calendar application once again. When shifting between months, the collection is being updated via a `setDate` method to set the current month (e.g. "2012-02"). When this happens, a number of associated views need to be updated as well. The page title and document title need to be updated with the new date. The navigation elements need to be updated to point to the correct month. The sidebar overview needs to be updated as well.

Listening to the collection's built-in "all", "reset", or "change:date" events is not a solution for these disparate views. Each of them would need to contain identical logic to decide if collection updates are a result of a model being updated or the collection's date changing.

Rather than asking the event subscribers to discern the kind of change, why not simply have the source of the change decide for itself?

For instance, when `setDate()` is invoked, we could immediately trigger the `calendar:change:date` event:

```
/* Good, but we can do better */
var Appointments = Backbone.Collection.extend({
  // ...
  setDate: function(date) {
    this.date = date;
    this.fetch();
    this.trigger('calendar:change:date');
  },
  // ...
});
```

An internal state of the collection is changing, but not necessarily the underlying models or the collection itself will change. As such, we should not risk triggering the built-in `change` Backbone event.

Here, we namespace the event as being specific to our calendar application by prefixing the event with `calendar:`. The remainder of the event describes the thing that is occurring—in this case, our application's date is changing.

Tip

Naming conventions for events are fairly application specific. They should always describe the event that is occurring from the perspective of the source of the event, though it should still read well in the consumer's context. If in doubt, following the Backbone convention, but namespaced for your application, is always a reasonable starting point.

With the collection now generating custom events, the next step it to listen for them. For that, consider the `TitleView`, which updates the page title with the current month:

```
var TitleView = Backbone.View.extend({
  initialize: function(options) {
    this.listenTo(options.collection, 'calendar:change:date', this.render);
  },
  render: function() {
    this.$el.html(' (' + this.collection.getDate() + ') ');
  }
});
```

The `TitleView` needs to bind to the collection so that it can listen for our custom event. With that, the page title will be updated to include the current month, as described by the collection.

Tip

There is no need to explicitly assign the `this.collection` in a Backbone view. Backbone will automatically create the `this.collection` property when it is passed via `new ViewThingy({collection: my_collection})`.

When working with events, always keep them as close to the actual change as possible. When changing months, the event should not fire until the date has actually changed. That is, the calendar's date is not really changed until the collection has been successfully fetched from the backend:

```
var Appointments = Backbone.Collection.extend({
  // ...
  setDate: function(date) {
    this.date = date;

    this.fetch({
      data: {date: this.date},
      success: _.bind(function() {
        this.trigger('calendar:change:date');
      }, this)
    });
  },
  // ...
});
```

Here, we supply the `date` query parameter (via the `data` attribute supplied to `fetch()`) so that we collect only a subset of the backend store's data. By doing so, we have the opportunity to pass a `success` callback to the normal `fetch()` method.

The `success` callback is the ideal place for the "calendar:change:date" event to originate. Only if the server successfully responds with with the updated collection information should the event fire and various views start their corresponding updates.

If anything goes wrong, the same views can remain the same or, possibly, subscribe to an "error" event to indicate the failure condition.

19.3.2. User Triggered Events

Custom events are perhaps even more powerful when the user is triggering them. Consider, for example, that the user is trying to highlight calendar appointments with matching titles.

The `CalendarFilter` view might look something like:

```
var CalendarFilter = Backbone.View.extend({
  template: _.template(
    '<input type="text" name="filter">' +
    '<input type="button" class="filter" value="Filter">'
  ),
  render: function() {
    this.$el.html(this.template());
    return this;
  },
  events: {
    'click .filter':  'filter'
  },
  filter: function() {
    var filter = $('input[type=text]', this.el).val();
    this.trigger('calendar:filter', filter);
  }
});
```

Here, we create a simple view comprised of form elements. Using the `events` attribute of Backbone views, clicks on the "Filter" button will trigger a custom event on the view itself. In this case, when the event is triggered, it makes sense to associate the event with data—the text being used to filter appointment titles.

Events should always bubble up the Precipitation chain. Here, this event should be consumed by the collection view:

```
var CalendarCollection = Backbone.View.extend({
  initialize: function(options) {
    // ...
    this.initialize_filter();
  },
  initialize_filter: function() {
    this.$el.after('<div id="calendar-filter">');

    var filter = new CalendarFilter({
      el: $('#calendar-filter'),
      collection: this.collection
    });
    filter.render();

    /* Bind to the custom event */
    filter.bind('calendar:filter', this.filterCollection, this);
  },
  // ...
});
```

The `filterCollection` method can then tell each appointment view to highlight themselves:

```
  filterCollection: function(string) {
    this.views.each(function(view) {
       view.highlightIfMatch(string);
    });
  }
```

Very little code was required thanks to the use of custom events (this time further customized with event data). Despite the lack of code, we achieve a fairly sophisticated bit of functionality *without* needing to make any requests at all of models, collections or anything else in the the backend.

19.4. Conclusion

Low coupling and high cohesion have long been hallmarks of object oriented coding. Javascript is not known for its object oriented nature, but Backbone.js makes heavy use of objects in its Models, Views and Collections. To prevent views from being coupled to one another or, worse yet, coupled to collections and models, we can leverage what Javascript *is* well known for: its functional and event-driven nature.

Chapter 20. Testing with Jasmine

This recipe gives a few pointers to maximize effectiveness when testing Backbone applications with Jasmine [1].

20.1. The Problem

A strong test suite is a must for maintaining robust, accurate code. How else can we be sure that changes do not break existing functionality? In addition to preventing breakages, testing our Backbone applications can also significantly enhance the cleanliness, and hence maintainability, of the code.

Even though we may accept that testing is valuable, browser testing is notoriously difficult. Let's take a look at some strategies for long term success with a Jasmine [2] test suite.

20.2. The Solution

There are two kinds of tests that we are going to consider in this recipe: high-level, integration tests and individual unit tests. Both have their uses and it is inappropriate to use one or the other exclusively.

Integration tests are great for interacting with Backbone applications across concerns. When we want to verify that a text field change effects a change in a model, which ultimately results in a change in a separate view, we are performing an integration test.

Unit tests are what we use to test individual views, collections and models in isolation. These are generally smaller and less likely to catch regressions. As such, many developers tend to view them as less important than their higher level brethren. But unit tests are great at

[1]http://pivotal.github.com/jasmine/
[2]More information on getting started with Jasmine can be found in Appendix A, *Getting Started with Jasmine*

forcing us to view our classes an individual objects with their own API. In practice, if it is hard writing a test for a class, it is almost certainly because there is too much coupling between it and other classes.

To see both types of tests in action, we will walk through developing a simple list of upcoming appointments in our calendar application.

20.2.1. Ingredients

We are going to make use of the following ingredients in this particular recipe:

- jasmine [3], a relatively small, browser-based Javascript library that facilitates well-written tests.
- sinon.js [4], which we will use to mimic backend server calls.
- jasmine-jquery [5], which dramatically improves the clarity of Jasmine tests.

20.2.2. Integration Testing with Jasmine

In the list view, if there are three items in the data store, then it stands to reason that three list items should be shown on the page. Or, in Jasmine-ese:

```
describe("list view, with three items in the data store", function() {
  it("lists all appointments", function() {
    expect($("li", "#appointment-list").length).toEqual(3);
  });
});
```

In this case, we expect a tag with an ID of `appointment-list` to include a bunch of list items. The jQuery selector of `$("li", "#appointment-list")`

[3]http://pivotal.github.com/jasmine/
[4]http://sinonjs.org/
[5]https://github.com/velesin/jasmine-jquery

will yield a wrapped set of all such list items. We can then examine the `length` property of the wrapped set to compare it to our expectation that there are three elements in our list view.

What makes this test an integration test rather than a unit test has nothing to do with our spec so far. Rather, it is the context in which we are running and checking expectations. If we had said that the view, given a model with three items, should add three items to the DOM, we might have been able to use this as a unit test. Instead, we couched this test in terms of what was saved in the database ("three items in the data store").

To describe that backend store, we must either point our Backbone application at a test server or stub out calls to the server. To minimize the hassle of building and tearing down a test database, we will do the latter. We stub out HTTP calls with sinon.js.

Any sinon.js interaction requires three steps: (1) start the sinon.js fake server, (2) perform some client-side action, and (3) tell sinon.js how to respond. The code "outline" for such a test might look like:

```
describe("list view, with three items in the data store", function() {
  // Stub web requests
  beforeEach(function() { /* sinon fakeServer */ });

  // Connect the Backbone app to the DOM
  beforeEach(function() { /* Backbone initialization */ });

  // Respond to any queued AJAX calls
  beforeEach(function() { /* sinon respondWith */ });

  it("lists all appointments", function() {
    expect($("li", "#appointment-list").length).toEqual(3);
  });
});
```

Let's look at each step in turn.

First, we need to intercept all web requests. This is accomplish by creating a sinon fake server:

```
// Stub web requests
var server;
beforeEach(function() {
  server = sinon.fakeServer.create();
});
```

As the name implies, the `beforeEach()` function will be executed before each test. This `beforeEach()` instantiates a fake server, which will intercept *all* outgoing HTTP requests from our Backbone.js application (or from any other widget on the current page). We can use this fake server to respond to requests as we desire.

Important

Unless otherwise specified, all requests intercepted by sinon.js will come back as 404 Not Found (more specifically, `[404, {}, ""]`).

With the fake server ready, we next need to exercise code that will generate web requests. That is, we need our Backbone application to do something that will send requests to the backend. For that, we need another `beforeEach()` setup block. This `beforeEach()` will be responsible for initializing the Backbone application:

```
// Connect the Backbone app to the DOM
beforeEach(function() {
  window.calendar = new Cal($('#calendar'));
  Backbone.history.loadUrl();
});
```

Alternatively, if the test page does not already have a `#calendar` element, it is easy enough to add:

```
// Connect the Backbone app to the DOM
beforeEach(function() {
  if ($('#calendar').length == 0)
    $('body').append('<div id="calendar"/>');
  window.calendar = new Cal($('#calendar'));
  Backbone.history.loadUrl();
});
```

The call to `Backbone.history.loadUrl()` is the second of two hacks to prod Backbone routes to work under test. The first hack occurs in the application code itself. `Backbone.history` is normally started in the application. Once started, it cannot be started twice without throwing an exception.

Unfortunately, creating multiple instances of a Backbone application is a necessity for unit testing. Creating multiple instances typically means multiple history starts. The only way to allow the application code to be tested is with a try-catch in the application code:

```
new Routes({application: application});
try { Backbone.history.start(); }
catch (x) { console.log(x); }
```

In other words, we always ignore errors from `Backbone.history.start()`. If history is already monitoring for `hashchange` events, then leave it be. It will work just as well for new instances of the application as it did for the first instance.

This is the first part of the Backbone history hack that ends with `loadUrl()`. The call to `loadUrl()` does just enough history work to trigger the proper routes for testing.

At this point, we have a fake server intercepting all web requests and we have a Backbone application generating web requests. The next step is to respond to different web requests. This is accomplished with sinon's `respondWith()` method:

```
// Respond to any queued AJAX calls
var list = [{}, {}, {}];
beforeEach(function() {
  server.respondWith(
    'GET',
    /\/appointments/,
    [
      200,
      { "Content-Type": "application/json" },
      JSON.stringify(list)
    ]
  );
  server.respond();
});
```

Here, we have most definitely veered into the realm of integration testing. We began with a test's expectation describing elements on a web page. Now, we are describing a JSON response from a backend server.

Sinon.js supports a very flexible syntax. In this case, we use the form of `respondWith()` that accepts three arguments: the HTTP method used (`GET`), the resource being requested (any URL containing `/appointments`) and the response. The response, in turn, consists of three parts: the HTTP status code (`200 OK`), any HTTP headers that we wish to include (`Content-Type: application/json`) and the body of the response (the JSON describing a list of three documents).

Tip

Sinon.js supports terser syntax, but we prefer the more verbose version shown here. We are being very explicit about expectations, leaving less room to be surprised.

After describing `respondWith()`, we tell the server to `respond()` immediately to any pending queries with `server.respond()`. Up to this point, our earlier `sinon.fakeServer.create()` had been merrily intercepting all HTTP activity and placing it into a queue. The `server.respond()` call immediately works through the queue, replying with any `respondWith()` responses that we have set (or 404s otherwise).

With all of the setup out of the way, we are finally ready to write the View code to make the spec pass. Despite all of the setup, our core behavior remains simple. We want three appointments to show up on the list (`expect($("li", "#appointment-list").length).toEqual(3)`). A smallish view like the following will do the trick:

```
var Cal = Backbone.View.extend({
  initialize: function(options) {
    this.listenTo(options.collection, 'reset', this.render);
  },
  render: function() {
    this.$el.html(
      '<h2>Appointment List</h2>' +
      '<ol id="appointment-list">' +
        this.collection.reduce(function(memo, appointment) {
          return memo + '<li></li>';
        }, "") +
      '</ol>'
    );
    return this;
  }
});
```

Important

Using a tool like sinon.js to stub out the backend can be dangerous if the backend API is undergoing rapid change. It is quite easy to get into a situation where the backend undergoes a breaking change, but all of the tests continue to pass. The test code would still serve up the old JSON API, which passes when run against old code.

In practice, this is not as fearsome as it might seem at first. First, most API changes are non-breaking (adding an attribute or two to JSON). Second, breaking changes usually go hand-in-hand with major front-end changes, which will necessitate test changes anyway. Thirdly, smoke tests of features are normally going to catch these kinds of errors.

> Still not convinced? Good, paranoia is a good character trait to have when testing. If you want to test full stack, then running your tests against a Jasmine server (from the Jasmine Ruby gem) can allow you to run tests against Backbone code hitting a real test server. This, however, comes at the added expense of maintaining code to build and tear down the test server.

In addition to sinon.js, the other invaluable tool for effective testing with Jasmine is jasmine-jquery. This tool adds several methods to jasmine that are based on the jQuery library. These methods are largely dedicated to making our lives easier by making Jasmine specs more readable—both in the spec code itself and the resultant output.

Recall that we made our spec pass by adding empty list items. To verify that the list items have the text that we expect, we will make use of the jasmine-jquery `toHaveText()` matcher. To match text in the list view, we provide details for the document list:

```
describe("list view", function() {
  var server = sinon.fakeServer.create(),
      list = [
        {"title": "Appt 001", "date": "2011-10-01"},
        {"title": "Appt 002", "date": "2011-11-25"},
        {"title": "Appt 003", "date": "2011-12-31"}
      ];
  // Setup app and sinon respondWith, then...
  it("displays appointment titles", function() {
    expect($('#appointment-list')).toHaveText(/Appt 001/);
  });
});
```

If we cared about the order in which our Backbone application displays the appointments, we might write another test ensuring that the first appointment in the DOM is from October. Here, it is sufficient to test that the `#appointment-list` element contains the expected text: "Appt 001".

It may seem overkill to pull in an entire testing library for something as simple as `toHaveText()`. To be sure, we could have asked `#appointment-`

`list` for its `text()` and then used a regular expression to match for the appointment title. The trouble with that approach is twofold. First, the intent of the test is not as clear, hidden behind all of that jQuery code. Second, we are just as likely to end up debugging the test as we are our application.

Jasmine-jquery is a huge win. Do not try to test without it.

20.2.3. Unit Testing

Unit tests, those tests that exercise behavior of individual components of an application, are very similar in structure to integration tests. In fact, it is easy to get the two confused. We generally like to follow the convention of putting the various tests into separate sub-directories of a top-level `specs' directory. Typically, these sub-directories are named: `integration`, `models`, `collections`, and `views` (the latter three all holding unit tests).

One way of thinking about unit tests is that we are asking how they will behave in the presence of a thing that acts like a model, view or a collection. For instance, if we wanted to test our appointment view in isolation, we might use a model-like thing on which the appointment view can operate:

```
describe("Calendar.Views.Appointment", function() {
  var el = $('<div></div>'),
      appointment = {
        title: 'Title Foo',
        description: 'Description bar.'
      },
      model = {
        toJSON: function() {return appointment;}
      };

  it("shows the title", function() {
    var ViewClass = Calendar.Views.Appointment,
        view = new ViewClass({model: model, el: el});

    view.render();
    expect($(el)).toHaveText(/Title Foo/);
  });
});
```

Here, our "model" defines an object literal with a single attribute: a function that answers to `toJSON()` calls.

This particular test is not all that exciting. We have succeeded only in verifying that the template method works:

```
var Calendar = {
  Views: {}
};

Calendar.Views.Appointment = Backbone.View.extend({
    template: _.template(
      '<span class="appointment" title="{{ description }}">' +
      '  <span class="title">{{title}}</span>' +
      '  <span class="delete">X</span>' +
      '</span>'
    ),
    render: function() {
      this.$el.html(this.template(this.model.toJSON()));
      return this;
    }
});
```

Unit tests are more helpful in Backbone applications when verifying callbacks. For example, if we click on the title of an appointment, we might expect that the appointment is "activated" on the page (*e.g.* it now has a CSS class applied for highlighting the selection). The simplest way to accomplish this is to check that the element has the "active" class enabled after a click. A more precise way to check the behavior is to verify that the `makeActive()` callback is invoked in response to the click event. For doing something like that, we use spies:

```
describe("Views.Appointment", function() {
  var el = $('<div></div>'),
      appointment = { /*... */ },
      model = { /*... */ };

  it("shows the title", function() { /*... */ });

  it("makes the view active when clicked", function() {
    var ViewClass = Calendar.Views.Appointment,
        view = new ViewClass({model: model, el: el});

    view.render();

    // Spy on the makeActive method
    sinon.spy(view, 'makeActive');

    // Simulate a click on the appointment
    $('.title', el).click();

    expect(view.makeActive.calledOnce).toBeTruthy();
  });
});
```

That is quite small and easy to read, but significantly helps to ease some of the worry of "callback hell" in Backbone applications. We are now 100% certain that, whenever the appointment is clicked, it will become active in the UI.

Trying to do something like this in an integration test would have been a morass of setup and jQuery selectors that find the right appointment. Worst of all, we would have no way to be certain that the appointment becomes active in response to the click event or from a side-effect of some other event (*e.g.* a model event).

Tip

Spies in sinon.js are a little nicer than the built-in Jasmine spies, which is why we use them here.

20.3. Conclusion

Simple Backbone applications do not need tests. After writing a tutorial application or our first, simple application, it is tempting to think that we can get away without tests for bigger Backbone applications. As with any codebase of significant proportions, this is inevitably a mistake as obvious use-cases are broken when new functionality is added. A solid test suite goes a long way toward mitigating such a situation. With the two types of tests described here, integration and unit, we should be well prepared to weather the growth of even the most complex Backbone applications.

Appendix A. Getting Started with Jasmine

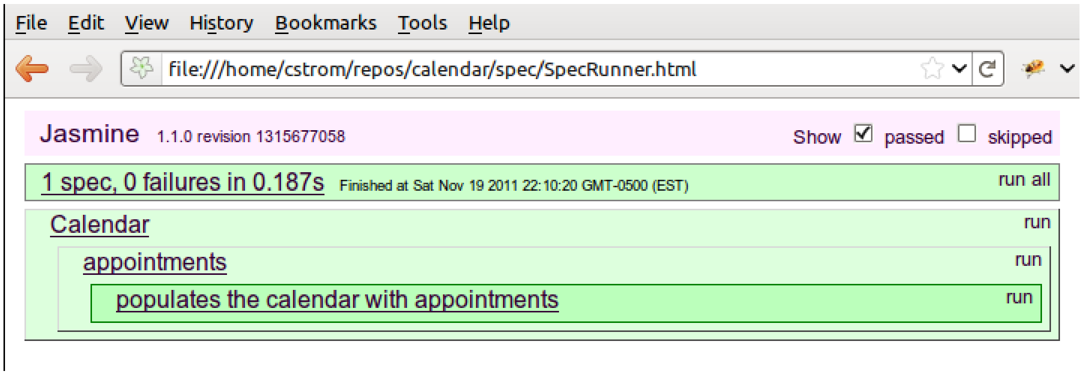
In Chapter 20, *Testing with Jasmine*, we discussed successful strategies for testing Backbone applications with Jasmine [1]. Here we present a brief introduction to Jasmine itself, with an eye toward testing Backbone code.

A.1. Your First Jasmine Test

Jasmine tests are written in Javascript and look something like:

```
describe("appointments", function() {
  it("populates the calendar with appointments", function() {
    expect($('#' + fifteenth)).toHaveText(/Get Funky/);
  });
});
```

In this test, we have a bunch of appointments that have been attached to our calendar application. One of them indicates an appointment to get on the funk on the fifteenth of the month [2]. If this test were passing, is might look something like:


File Edit View History Bookmarks Tools Help
file:///home/cstrom/repos/calendar/spec/SpecRunner.html

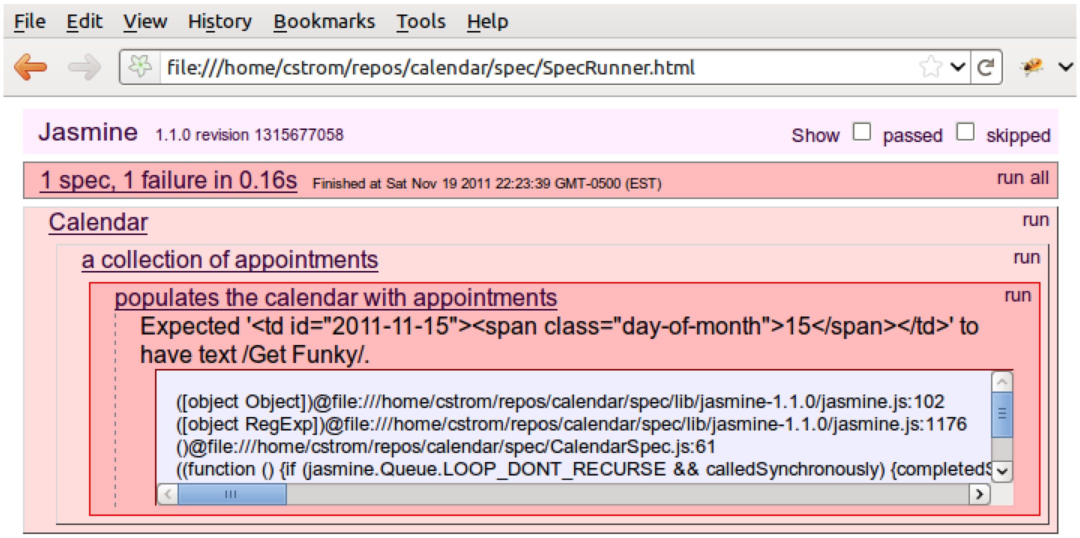
```
Jasmine 1.1.0 revision 1315677058                    Show ☑ passed ☐ skipped
1 spec, 0 failures in 0.187s  Finished at Sat Nov 19 2011 22:10:20 GMT-0500 (EST)   run all
Calendar                                                                  run
  appointments                                                            run
    populates the calendar with appointments                              run
```

[1]http://pivotal.github.com/jasmine/
[2]The `toHaveText()` matcher comes from the jquery-jasmine plugin [https://github.com/velesin/jasmine-jquery]

As can been seen, Jasmine tests are evaluated in a browser [3]. Looking at those Jasmine results, we can get a decent idea of what Jasmine means when it claims to facilitate BDD. Specifically, the output of the tests almost reads like a specification.

From top to bottom, we are talking about a Calendar Backbone application. In that calendar application, we expect that a collection of appointments will populate the calendar with UI representations of themselves.

We are getting ahead of ourselves of course. Since we want to be counted among the cool kids and/or hipsters of the programming world, we want to drive the implementation of this feature via our test. Without an appointment view class, our test fails:



```
Jasmine 1.1.0 revision 1315677058
1 spec, 1 failure in 0.16s  Finished at Sat Nov 19 2011 22:23:39 GMT-0500 (EST)
Calendar
  a collection of appointments
    populates the calendar with appointments
      Expected '<td id="2011-11-15"><span class="day-of-month">15</span></td>' to
      have text /Get Funky/.
        ([object Object])@file:///home/cstrom/repos/calendar/spec/lib/jasmine-1.1.0/jasmine.js:102
        ([object RegExp])@file:///home/cstrom/repos/calendar/spec/lib/jasmine-1.1.0/jasmine.js:1176
        ()@file:///home/cstrom/repos/calendar/spec/CalendarSpec.js:61
        ((function () {if (jasmine.Queue.LOOP_DONT_RECURSE && calledSynchronously) {completedS
```

It fails because our Appointment view does not actually do anything:

```javascript
var Appointment = Backbone.View.extend({});
```

[3]At the time of this writing Firefox is the browser that works best with Jasmine

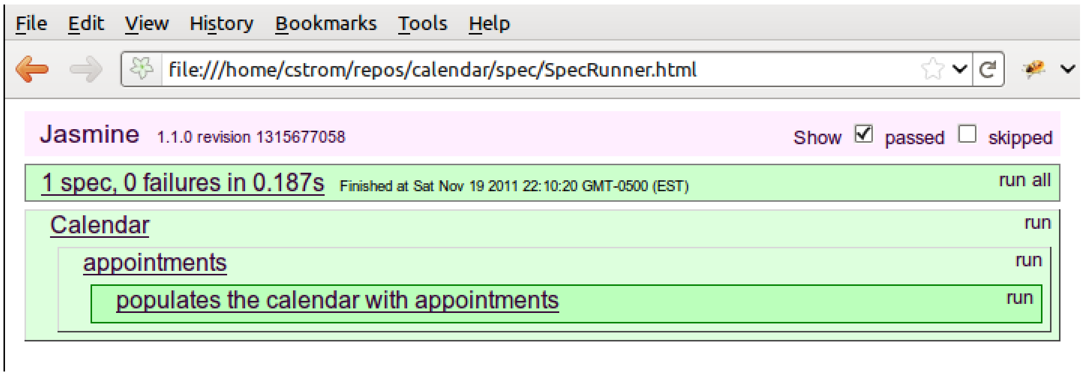
It is a Backbone view, so it will respond to `render()` calls—but it will not actually render anything. It has an `el` property that can be inserted into the DOM, but it is an empty `<div>`. So, of course, our test fails.

If this were a book on BDD, we might take you through the steps of demonstrating simple tests that get something displayed. Then, we could write a second test that verifies that information is coming from our collection rather than being hard-coded to allow the first test to pass. But, since this is a Backbone book, let's skip ahead to what is necessary to make this test pass with data from the collection:

```
var Appointment = Backbone.View.extend({
  template: _.template(
    '<span class="appointment" title="{{ description }}">' +
    '  <span class="title">{{title}}</span>' +
    '  <span class="delete">X</span>' +
    '</span>'
  ),
  render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
  }
});
```

Our collection view has ultimate responsibility for inserting this individual appointment View into the DOM at the correct location. All we have to do is ensure that the view will include the model's title.

Done and done thanks to our view class. Now we have legitimately achieved our passing test:


File Edit View History Bookmarks Tools Help
file:///home/cstrom/repos/calendar/spec/SpecRunner.html
Show ☑ passed ☐ skipped
run all
run
run
run

Jasmine 1.1.0 revision 1315677058

1 spec, 0 failures in 0.187s Finished at Sat Nov 19 2011 22:10:20 GMT-0500 (EST)

Calendar

appointments

populates the calendar with appointments

Important

Even if you are not adhering to the principals of BDD, you should always ensure that removing code makes tests fail. If you remove code and the test still passes, chances are you are not testing what you think you are testing. Actually, strike that, if you remove code and your tests still pass, then your test is worthless.

A.2. Jasmine Standalone

When first getting started with a Backbone / Jasmine test suite, the quick and dirty thing to do is "Jasmine Standalone". This involves pointing our browser at a single web page *on the local filesystem*. For instance, if we are building our application in `$HOME/repos/calendar`, then we would want to load our tests at `file:///home/dafunk/repos/calendar/spec/SpecRunner.html`.

The contents of the spec runner looks something like:

```
<head>
  <!-- Jasmine files -->
  ...

  <!-- Library files -->
  ...

  <!-- Test helpers -->
  ...

  <!-- Include spec files ... -->
  ...

  <!-- Include source files ... -->
  ...

  <!-- Initialize Jasmine env -->
  <script type="text/javascript">
    (function() {
      // ...
    })();
  </script>
</head>
```

Since this test is file-based, all of those sections need to include references to other files on the file system. For example, the jasmine files would be linked in as:

```
<!-- Jasmine files -->
<script type="text/javascript"
  src="lib/jasmine-1.1.0/jasmine.js"></script>
<script type="text/javascript"
  src="lib/jasmine-1.1.0/jasmine-html.js"></script>
```

The specs themselves would be linked in as:

```
<!-- include spec files here... -->
<script type="text/javascript"
  src="CalendarSpec.js"></script>

<!-- include source files here... -->
<script type="text/javascript"
  src="../public/javascripts/Calendar.js"></script>
```

The specs and testing libraries should generally be kept in a separate directory from the actual code. Here, we have the Jasmine test libraries and the actual test file in the same directory while the application code being tested is in a separate `public` top-level directory (see below for a more detailed breakdown of how these files might be organized).

As for the spec itself, it should look something like:

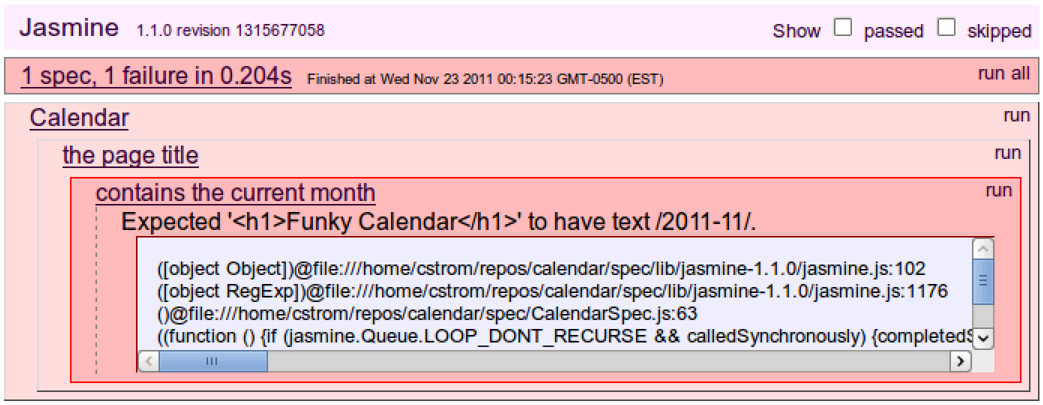
```
describe("Calendar", function() {
  describe("the page title", function() {
    it("contains the current month", function() {
      /* Code verifying the title */
    });
  });
});
```

The first, outermost spec describes the highest level concept being tested —here the calendar Backbone application. Inside that, we describe the specific aspect of the application being tested—in this case, the title of the page. Finally, we have one or more blocks that enumerate the expected behavior of the code.

An `it()` block uses Jasmine "matchers" to describe the expected behavior. To describe the expectation that the title should contain the ISO 8601 date, we can write:

```
  describe("the page title", function() {
    it("contains the current month", function() {
      expect($('h1')).toHaveText(/2011-11/);
    });
  });
```

At first, this will fail with an error along the lines of:


Jasmine 1.1.0 revision 1315677058 Show ☐ passed ☐ skipped

1 spec, 1 failure in 0.204s Finished at Wed Nov 23 2011 00:15:23 GMT-0500 (EST) run all

Calendar run

the page title run

contains the current month run

Expected '<h1>Funky Calendar</h1>' to have text /2011-11/.

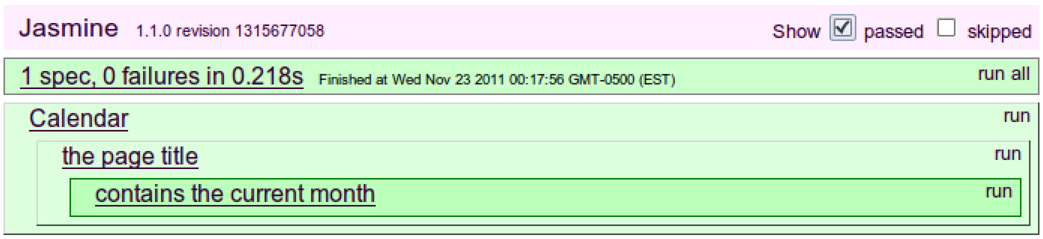
```
([object Object])@file:///home/cstrom/repos/calendar/spec/lib/jasmine-1.1.0/jasmine.js:102
([object RegExp])@file:///home/cstrom/repos/calendar/spec/lib/jasmine-1.1.0/jasmine.js:1176
()@file:///home/cstrom/repos/calendar/spec/CalendarSpec.js:63
((function () {if (jasmine.Queue.LOOP_DONT_RECURSE && calledSynchronously) {completed
```

But we can make our spec pass by implementing a Backbone title view:

```
var TitleView = Backbone.View.extend({
  tagName: 'span',
  initialize: function(options) {
    this.listenTo(options.collection, 'calendar:change:date', this.render);

    $('span.year-and-month', 'h1').
      replaceWith(this.el);
  },
  render: function() {
    this.$el.html(' (' + this.collection.getDate() + ') ');
  }
});
```

With that, we have our passing test:


Jasmine 1.1.0 revision 1315677058 Show ☑ passed ☐ skipped

1 spec, 0 failures in 0.218s Finished at Wed Nov 23 2011 00:17:56 GMT-0500 (EST) run all

Calendar run

the page title run

contains the current month run

There are definitely times that standalone Jasmine tests are not sufficient, which is what the next section discusses.

A.3. Jasmine (Ruby) Server

If your application grows, it will soon become too large for standalone Jasmine. The standalone approach lacks the capability to run under a continuous integration server. Standalone also make network requests very difficult. This is where the jasmine ruby gem steps into the picture.

Tip

The jasmine server is implemented in Ruby, so you will need that installed on your system. The best resource for this is the "Downloads" link on http://ruby-lang.org.

You will also need the rubygems library. Despite being universal in the Ruby community, the rubygems library is not bundled with Ruby. You can find instructions for installing rubygems at: http://docs.rubygems.org/read/chapter/3

With ruby and rubygems installed, you are ready to install the jasmine server. Per the jasmine server instructions [https://github.com/pivotal/jasmine/wiki/A-ruby-project:], installation is accomplished via two commands:

```
$ gem install jasmine
$ jasmine init
```

At this point the server can be run as:

```
$ rake jasmine
```

There is a fair bit of configuration required in the Jasmine server. After installation of the server (and assuming that we already have the Backbone application started in `Calendar.js`), our directory structure might include the following:

```
...
├── public
│   └── javascripts
│       ├── backbone.js
│       ├── Calendar.js
│       ├── jquery.min.js
│       ├── jquery-ui.min.js
│       └── underscore.js
├── spec
│   └── javascripts
│       ├── helpers
│       │   ├── jasmine-jquery.js
│       │   ├── sinon.js
│       │   └── SpecHelper.js
│       ├── CalendarSpec.js
│       └── support
│           ├── jasmine_config.rb
│           ├── jasmine_runner.rb
│           └── jasmine.yml
...
```

Libraries used by the actual application (our Backbone app and various supporting libraries) are stored under `public/javascripts`. Libraries only used for testing are stored along with the rest of the testing material under the `spec/javascripts/` directory.

Configuration is done almost exclusively in the `jasmine.yml` configuration file. For the most part, the defaults are sound. The exception for Backbone applications is the `src_files` directive. By default, this loads javascript source files (the things under `public/javascripts` for us) in alphabetical order. This is disastrous for a Backbone application because it means that `underscore.js` would be loaded after `backbone.js` (ditto `jquery.js`).

To work around this, we need to explicitly layout our library files in the same order as they would be in the web page. Something along the lines of:

```
src_files:
    - public/javascripts/jquery.min.js
    - public/javascripts/underscore.js
    - public/javascripts/backbone.js
    - public/javascripts/**/*.js
```

The last line instructs Jasmine to slurp up everything (Jasmine is smart enough to ignore anything it has already loaded).

With that, we can run our test suite—either locally or under a continuous integration server—by issuing the `rake jasmine:ci` command:

```
#  calendar git:(jasmine) rake jasmine:ci
 Waiting for jasmine server on 58538...
 Waiting for jasmine server on 58538...
 Waiting for jasmine server on 58538...
 [2011-11-19 23:38:14] INFO  WEBrick 1.3.1
 [2011-11-19 23:38:14] INFO  ruby 1.9.2 (2011-07-09) [x86_64-linux]
 [2011-11-19 23:38:14] WARN  TCPServer Error: Address already in use - bind(2)
 [2011-11-19 23:38:14] INFO  WEBrick::HTTPServer#start: pid=6921 port=58538
 Waiting for jasmine server on 58538...
 jasmine server started.
 Waiting for suite to finish in browser ...
 .........

 Finished in 1.17 seconds
 9 examples, 0 failures
```

It is wonderful to have a reproducible test environment for all team members as well as for our continuous integration server. This will save hours upon hours of tracking down idiosyncrasies between different environments.

The other benefit of running the jasmine server is that it is a server. To see this in action, we can start the server with the `rake jasmine` (omitting the `:ci` from the continuous integration version of the command):

```
#  calendar git:(jasmine) rake jasmine
your tests are here:
  http://localhost:8888/

 [2011-11-19 23:58:02] INFO  WEBrick 1.3.1
 [2011-11-19 23:58:02] INFO  ruby 1.9.2 (2011-07-09) [x86_64-linux]
 [2011-11-19 23:58:02] WARN  TCPServer Error: Address already in use - bind(2)
 [2011-11-19 23:58:02] INFO  WEBrick::HTTPServer#start: pid=7077 port=8888
```

And, as the output instructs us, we can find our specs at http://localhost:8888/. At this point, nothing prevents us from making real HTTP request of an application server also listening on `localhost`. We are no longer restricted to simulating browser interaction. We can test the real thing.

A.3.1. Continuous Integration

Tests are only useful when run on each commit. Unless you have the kind of team that is intensely committed to manually running the Jasmine suite before each commit, you will need a continuous integration server.

For ruby shops, the jasmine gem includes rake [4] commands that can be used in continuous integration. Instead of running the server manually with `rake jasmine`, a continuous integration server would invoke the `rake jasmine:ci` command. The exit status and output from this command work nicely with most continuous integration environments.

The one caveat with using the jasmine gem for continuous integration is that the jasmine gem needs to start up an actual web browser to execute the tests. This can be difficult to configure. There are headless alternatives to the jasmine gem. The jasmine-headless-webkit [5] is a good starting place. The PhantomJS [6] javascript environment is another. The latter even includes a `run-jasmine.js` script which is easily adaptable for use in a continuous integration environment. Of the two, PhantomJS is currently

[4]Rake is the ruby equivalent of the venerable Unix command, `make`
[5]http://johnbintz.github.com/jasmine-headless-webkit/
[6]http://phantomjs.org

better suited for Backbone development, but both are undergoing active development.