MASTER SPACE AND TIME WITH JAVASCRIPT

BOOK 3: BACKBONE

Noel Rappin

a noelrappin.com publication

Master Space and Time With JavaScript

Book 3: Backbone

By Noel Rappin
http://www.noelrappin.com

Release 007 January, 2013

The original image used as the basis of the cover is described at http://commons.wikimedia.org/wiki/File:Challenger1983.jpg as a scene of the Space Shuttle Challenger taken during STS-6 in 1983 with a 70mm camera onboard the shuttle pallet satellite. According to NASA policy, the photo is in the public domain.

CONTENTS

Chapter 1

Welcome to *Master Space and Time With JavaScript*

Thanks for purchasing (hopefully) or otherwise acquiring (I won't tell, but it'd be nice if you paid...) *Master Space and Time With JavaScript*.

Here are a few things I'd like for you to know:

Please note that Book 3 is a work in progress and is incomplete

Section 1.1

What have I purchased?

Master Space and Time With JavaScript is a book in four parts. All four parts are available at http://www.noelrappin.com/mstwjs.

The first part was *Part 1: The Basics*, which is available for free. It contains an introduction to Jasmine testing and jQuery, plus a look at JavaScript's object model.

Book 2: Objects in JavaScript, is currently available for $7. It contains more complete examples of using and testing objects in JavaScript, including communication with a remote server and JSON.

You are currently reading *Book 3: Backbone*, The beta currently consists of about half of the finished book. It continues building the website using Backbone.js to create single-page interfaces for some more complex user interaction.

Book 4: Ember, will probably be available by the end of 2012, also for $7. It will build out completely different parts of the website using Ember.js to create complex client-side interactions.

You may pre-purchase all four parts of the book for $15, a $6 savings over buying all three parts separately. This deal may not be available indefinitely.

This book has no digital rights management or copy protection. As far as I am concerned, you have much the same rights with this file as you would with an physical book. I would appreciate if you would support this book by keeping the files away from public file-sharing servers. If you do wish to distribute this book throughout your organization, it would be great if you would purchase additional copies or contact me at `noel@noelrappin.com` to work out a way for you to purchase a site license.

Section 1.2

Who Are You? Who? Who?

Inevitably, when writing a book like this, I need to make some assumptions about you. In addition to being smart, and obviously possessing great taste in self-published technical books, you already know some things, and you probably are hoping to learn some other things from this book.

In some ways, intermediate level books are the hardest to write. In a beginner book, you can assume the reader knows nothing, in an advanced hyper-specific book, you don't really care what the reader knows, they've probably self-selected just by needing the book. In an intermediate book, though, you are potentially dealing with a wider range of reader knowledge.

Here's a rundown of what I think you know:

JavaScript: I'm assuming that you have a basic familiarity with what JavaScript looks like. In other words, we're not going to explain what an `if` statement is or what a string is. Ideally, you're like I was a several months before I started this project – you've dealt with JavaScript when you had to, then had your mind blown by what somebody who really knew what they were doing could do. You specifically do not need any knowledge of JavaScript's object or prototype model, we'll talk about that.

Server Stuff: Since this is a JavaScript book, the overwhelming majority of the topics are on the client side and have nothing to do with any specific server-side tool. That said, there is a sample application that we'll be working on, and that application happens to use Ruby on Rails. You don't actually need to know anything about Rails to run the JavaScript examples, though if you are a Rails programmer, there will be one or two extras. It will be helpful if you are good enough at a command prompt to install the sample application per the instructions later in this preface.

CoffeeScript: I'm not assuming any knowledge of CoffeeScript. If you happen to have some, and want to follow along with the examples using CoffeeScript, have at it, I'll provide a separate CoffeeScript version of the code repository. NOTE: This isn't ready yet, don't go looking for it.

Testing Tools: I'm not assuming any knowledge of testing tools or of any test-first testing process. We'll cover all of that.

jQuery: I'm not assuming any prior jQuery knowledge.

Backbone.js: I'm not assuming any prior Backbone.js knowledge.

Ember.js: I'm not assuming any prior Ember.js knowlege.

Section 1.3

What to Expect When You Are Reading

On the flip side, it's fair of you to have certain expectations and assumptions about me and about this book. Here are a couple of things to keep in mind:

- I firmly and passionately believe in the effectiveness of Behavior-Driven Development as a way of writing great code, especially in a dynamic language like JavaScript. As a result, we're going to write tests for as much of the functionality in this book as is possible, and we're going to write the tests first, before the we write the code. If you are completely unfamiliar with testing, this may be a challenge in the early going, as we're juggling Jasmine and jQuery. Don't worry, you can do it, and the rewards will be high.

- This book is focused on the current versions of the languages and libraries available. As I write this, that means ECMA Script 5 for JavaScript, jQuery 1.7.2, Jasmine 1.2.0, Backbone.js 0.9.2, Ember.js 0.9.8.1 and Rails 3.2.x. Keeping up with current versions is hard enough, without worrying about the interactions among multiple versions.

Section 1.4

But is it finished?

This book is incomplete. You will be notified from time to time that a new version of the book is available.

Here's a partial list of things that still need to be done in Books 1, 2, and 3:

- CoffeeScript versions of all the sample code in the book will be made available.
- Formatting for Kindle and ePub versions is still a little wonky in spots. I'm working on it.
- The directions for setting up the sample application probably need to be improved.
- Book 3 is only half complete – there's another extended example that's coming.
- Something only you know – if you think there's something missing in the book let me know via any of the mechanisms listed below.

Section 1.5

What if I want to talk about this book?

Please do! The only way this book will be distributed widely is if people who find something useful in it tell their friends and colleagues.

You can reach me with email comments about the book at `noel@noelrappin.com`. Or you can reach me on twitter as `@noelrap`. If you want to talk about the book on Twitter, it'd be great if you use the hashtag `#mstwjs`, which gives me a good chance of seeing your comment.

This book has mistakes, I just don't know what they are yet. If you happen to find an error in the book that needs correcting, please use the email address `errata@noelrappin.com` to let me know.

There's also a discussion forum for the book at `http://www.noelrappin.com/mstwjs-forum/`. You do need to sign up in order to post, which you can do at `http://www.noelrappin.com/register-for-book-forum/`.

Section 1.6

Following Along

The source code for this book is at https://github.com/noelrappin/mstwjs_code. The server side part of the code of this is a Rails application. You won't need to understand any of the Rails code to work through the examples in the book, but you will need to make the application run. Also, a basic knowledge of the Git source control application will help you view the source code.

I've tried to make this simple. The external prerequisites to run the code are Ruby 1.9 and MySQL. RailsInstaller http://www.railsinstaller.org is an easy way to get the Ruby prerequisites for this application installed if you do not already have them. MySQL can be installed via your system's package manager or from http://www.mysql.com.

Once you have the prerequisites installed and the repository copied, you can set everything up for the system by going to the new directory and entering the command `rake mstwjs:setup`. This command will install bundler, load all the Ruby Gems needed for the application, and set up databases. Then you can run the server with the command `rails server`, and the application should be visible at `http://localhost:3000`. Please contact noel@noelrappin.com if you have configuration issues with the setup, and we'll try to work through them.

The git repository for this application has separate branches for each section of the book with source code. Code samples that come from the repository are captioned with the filename and branch they were retrieved from. In order to view the branch, you need to run `git checkout -b <BRANCH> origin/<BRANCH>` from the command line.

Okay, let's get on with it.

Chapter 2

Grow A Spine With Backbone.js

Once again, you hear from Dr. What. The Doctor is excited. Maybe too excited:

> It's time to stop messing around. You've convinced me. JavaScript is the way to go. I want more, more more JavaScript. JavaScript everywhere. Be bold. Show a little backbone.
>
> Dr, What

You decide to take the Doctor literally.

Backbone.js is a simple JavaScript library that brings an MVC structure to your JavaScript client code. With Backbone, you can create single-page web applications that manage the browser display, and even manage user history. It's an exceptionally useful framework for structuring client side applications. However, using Backbone effectively requires a change in how you think about the relationship between your data, your application code, and the view layer.

In the next few chapters, we'll extend our Time Travel Adventures site. First we'll rebuild our home page in Backbone, which isn't necessarily the greatest Backbone use case, but is a page we're all familiar with, and gives us yet another way to rewrite the famous show/hide toggler.

Then we'll build a combination trip display and order page that will take better advantage of Backbone's ability to manage a single page application. We'll also have our Backbone application communicate back to our server.

Section 2.1

Setting Up Backbone.js

As is our usual mode, we're going to do stuff, then explain it, then do some more stuff – I apologize for the technical terms there. The first thing we need to do is install Backbone.

Backbone itself

Backbone has only one dependency, which is the Underscore.js library. In the general case, you need to make sure that Backbone.js and Underscore.js are both in your JavaScript load path. As I write this, the current version of Backbone is 0.9.2, and the current version of Underscore is 1.3.1. Backbone can be downloaded from http://backbonejs.org and Underscore is available from http://documentcloud.github.com/underscore/.

In our Rails sites, we're going to use the `backbone-rails` gem, http://github.com/codebrew/backbone-rails, to allow Rails to track Backbone and Underscore as dependencies and align Backbone with the Rails asset pipeline. All we need to do for that to work is add the following line to our `Gemfile`:

```
gem "rails-backbone"
```

Sample 2-1-1:

That's not a typo, the URL is `backbone-rails`, but the gem name is `rails-backbone`. No, I don't understand it either.

If you then run `rails g backbone:install`, the gem will create the directory `app/assets/javascripts/backbone`, along with a bunch of subdirectories. It will also add Backbone to the `application.js` manifest file, and it will create a setup file, about which more in a bit. The `rails-backbone` gem also has a bunch of generators for setting up Backbone pieces that we are going to ignore.

The Server-Side part

In order for Backbone to work inside our Time Travel application, it needs to get some communication from the server. In general, a Backbone application treats the server as a data store, expecting only JSON data from the server and sending REST requests with JSON data back to the server. In addition, the initial request for the page needs to set up the Backbone page, at least to the limited amount that the Backbone framework needs to be invoked. It's also considered good practice to seed the client-side page with the initial data used to start up the page, to save Backbone from making an extra request back to the server to request data that you already know that you need.

In our case, we're going to add a separate controller and route to the Time Travel application to create an index page. Obviously, in a real application you would just re-purpose the index

page that already exists, but for instructional purposes, we're going to create a new page. That will require a controller, route, and a view setup. If you are using Backbone with something other than Rails as your server-side framework, you'll still need the same basic steps, but the exact details will differ.

In our case, we need a home page controller. All the home page needs to do is gather all the trips from the database and make them available to the view as JSON.

Filename: app/controllers/home_controller.rb (Branch: section_9_1)

```
class HomeController < ApplicationController

  respond_to :html, :json

  def index
    @trips = Trip.all
    respond_with(@trips)
  end

end
```

Sample 2-1-2: Home controller for our backbone page

All this is doing is responding to the `index` action by returning a list of all trips. Because of the way Rails handles requests, we can either respond to an HTML request with a simple view, or if we make a JavaScript request, we'll get a list of trips formatted as JSON. In our case, this is a little confusing, since, we'll be making an HTML request to set up the page, but we need to deal with the trip data as JSON. As you'll see in a second, the HTML view will convert the trip data.

We'll need a route that points to that controller. Which involves changing the `root` line in the `config/routes.rb` file to the following:

```
root :to => "home#index"
```

Sample 2-1-3:

We also want a basic view. All the view needs to do is include the JSON trips and start up Backbone. We'll include a nearly-boilerplate layout:

Filename: app/views/layouts/home.html.erb (Branch: section_9_1)

```
<!DOCTYPE html>
<html>
<head>
  <title>Time Travel Adventures</title>
  <%= stylesheet_link_tag    "application" %>
  <%= javascript_include_tag "application" %>
  <%= yield :javascript %>
  <%= csrf_meta_tags %>
</head>
<body>
  <div id="container"><%= yield %></div>
</body>
</html>
```

Sample 2-1-4: Simple layout for just our assets and page code

And an actual index page with just the smallest bit of JavaScript:

Filename: app/views/home/index.html.erb (Branch: section_9_1)

```
<%= content_for :javascript do -%>
  <%= javascript_tag do %>
    $(function() {
      TimeTravel.init(<%== @trips.to_json %>);
    });
  <% end %>
<% end -%>
```

Sample 2-1-5: Home view that just includes our trips as JSON

All that code is doing is calling a JavaScript function called `TimeTravel.init` with an argument of all the JSON data. The `TimeTravel.init` function will start the Backbone ball rolling. And what does `TimeTravel.init` do? Pretty much nothing yet.

Filename: app/assets/javascripts/backbone/time_travel.js (Branch: section_9_1)

```
//= require_self
//= require_tree ./templates
//= require_tree ./models
//= require_tree ./views
//= require_tree ./routers
```

```
var TimeTravel = {
  Models: {},
  Collections: {},
  Routers: {},
  Views: {},

  init: function(tripData) {
    this.trips = new TimeTravel.Collections.Trips(tripData);
    this.app = new TimeTravel.Routers.TripRouter();
    Backbone.history.start({pushState: true});
    return this;
  }
}
```

Sample 2-1-6: Our application object

First off, we're setting up some namespaces for the various pieces of Backbone that we're going to write. The only thing the `init` function is doing right now is starting the Backbone history object, which Backbone uses to manage internal history, including the browser's Back button.

Section 2.2

What has to happen for somebody to say "We need Backbone.js"?

When you are trying to understand how a programming tool works, it's a good idea to start with the kinds of problems that the tool was created to solve.

Like many successful frameworks, Backbone was abstracted from the common practices of successful projects. For Backbone, the successful projects are rich client-side web applications, particularly those that appear to be a single page from the users point of view. The common abstraction is a relatively simple structure that separates model data from view and layout concerns, and also allows you to "route" user actions to particular parts of the application and manage user history, while still allowing you ample space to structure your application as you see fit. A... backbone, if you will.

So far, we haven’t had a problem with code structure on our project because, honestly, we haven’t added a ton of logic to the client-side – most of our work still takes place on the server, including merging our data with a template to create HTML. So our structural needs are small.

As we move logic from the server to the client, that lack of structure can become a problem. In particular, encapsulation and the ability to work on one part of our application without dealing with the rest will become critical.

Backbone helps our structure problem in two ways. First, like Rails does for server-side applications, Backbone gives you a Model/View structure that gives you a place to slot in your code – though, as we’ll see, Backbone is significantly less opinionated than Rails. Secondly, and I think more importantly, Backbone gives you a structure that makes it easy to think of your web page as the interaction of multiple small pieces that work together to build the page.

Another way to look at Backbone is as a way to bring the best parts of desktop interface frameworks to the best parts of web development. Both modern web development and traditional GUI development have Model/View/Controller (MVC) patterns as a major component of their structure, but the implementations are very different. In web development (particularly Rails web development), the interface is specified as HTML text, the controller often has a very small role to play, and the bulk of the output is essentially the result of one view template. While the view template may be broken up into smaller parts, that’s more a convenience than a significant part of the application architecture.

In a classic desktop MVC, there are multiple view objects, each of which is responsible for a particular portion of the screen (and in some frameworks, each view potentially having its own controller). There’s a much higher emphasis on internal events to cause parts of the screen to update, where as in non-Ajax web development, updating part of the screen is not a major design motivation.

Backbone gives you a similar structure; although the Backbone creators go out of their way to avoid claiming that Backbone is an MVC design. The design structure of Backbone is sometimes described as MVVM for Model/View/View Model – and the lack of a slash between the second View and Model is deliberate. What Backbone provides is a “View Model”, also sometimes called a Presenter, that encapsulates the display logic for a particular model and ties it to a particular part of the screen.

Section 2.3

Enough Yammering, Let's Code

Good point, person who writes the section headings. Let's code.

Our first Backbone task is going to be to replicate the functionality of our home page using Backbone. This involves displaying trip data, and recreating our old friend, the show/hide toggler. (In my defense, Backbone allows us to handle the show/hide toggler completely differently, and the difference is worth discussing.)

I freely admit up front that this task is not quite the Backbone sweet spot – not really enough logic on the page. Next chapter will be a little closer. The advantage of the task in this chapter is that it's a task we already understand, so the Backbone specific features come out into more relief.

We have another problem, which is that I want to start off by writing a test, but we haven't discussed any of the features of the Backbone framework. So, in keeping with Book 1's "test first, ask questions later" approach, I'm going to write out the tests that I want, and we'll discuss the Backbone structures as they come up.

Section 2.4

First, An Integration Test

I want to start our task here with a kind of test we haven't seen yet, something that is closer to an integration test than a unit test. What this test will do is outline the end-to-end conditions, starting with the entrance into the Backbone stack and ending with the DOM structures that Backbone will create.

Here's an attempt at an integration test. The goal here is to get something in the test suite to drive us to start adding the Backbone structures:

Filename: spec/javascripts/homePageSpec.js (Branch: section_9_1)

```
describe("rendering the home page with Backbone", function() {

  var tripData = [{"description":"A cruise","end_date":"1620-11-21",
```

```
    "id":13, "image_name":"mayflower.jpg", "name":"Mayflower Luxury Cruise",
     "price":1204.0, "start_date":"1620-05-17",
    "tag_line":"Enjoy The Cruise That Started It All"},
    {"description":"See plays", "end_date":"1605-10-31", "id":14,
    "image_name":"shakespeare.jpg", "name":"See the Plays of Shakespeare",
    "price": 1313.0, "start_date":"1604-11-01",
    "tag_line":"See The Master As Intended"}];

  describe("end to end", function() {

    it("renders trips on the page", function() {
      affix("#container");
      TimeTravel.init(tripData);
      expect(TimeTravel.trips.size()).toEqual(2);
      var $container = TimeTravel.app.index();
      console.log($container);
      expect($(".trip").size()).toEqual(2);
    });

  });

});
```

Sample 2-4-1: An initial integration test

What are we doing here? First off, we're using the `tripData` variable to store some JSON descriptions of trips (not all the fields of trips are in this data, just a couple of interesting ones.) In the actual spec, we're calling our `TimeTravel.init()` method, which we'll expect to initialize some Backbone objects.

After the application initializes itself, we look for two objects. The first is `TimeTravel.trips`. The `TimeTravel.trips` object is a Backbone *`Collection`* object, which is a useful abstraction that Backbone provides to act as a collection of Backbone *`Model`* objects. The other object is called `TimeTravel.app`, which will be a Backbone *`Router`* object.

Once we have the router object, we'll call the `index()` method on it. That's a method we'll define ourself, and which will also be called to start the action when the page gets hit in the actual browser. The `index()` method will trigger the actual creation of the page, and in the

final line of the code, we test to make sure that, whatever else Backbone has done, it's created `trip-container` elements for each of the trips that for which it has data.

If we go to run this spec, given the version of `TimeTravel.init` listed above, the spec fails on the creation of the class `TimeTravel.Collections.Trips`, because we haven't defined that class yet. Leading to the obvious question, "What the heck is `TimeTravel.Collections.Trips`, and why do we need one?"

Section 2.5

What's Going On Here?

Let's step back a second to talk about the basic structure of a Backbone.js application. There are four main classes in your typical Backbone.js workflow, plus a couple of other support objects. You've got the `Router`, the `View`, the `Model`, and the `Collection`. Here's a diagram that shows the Backbone workflow: (Note to readers: I am bad at diagrams. I will try to get a better one here in the future.)

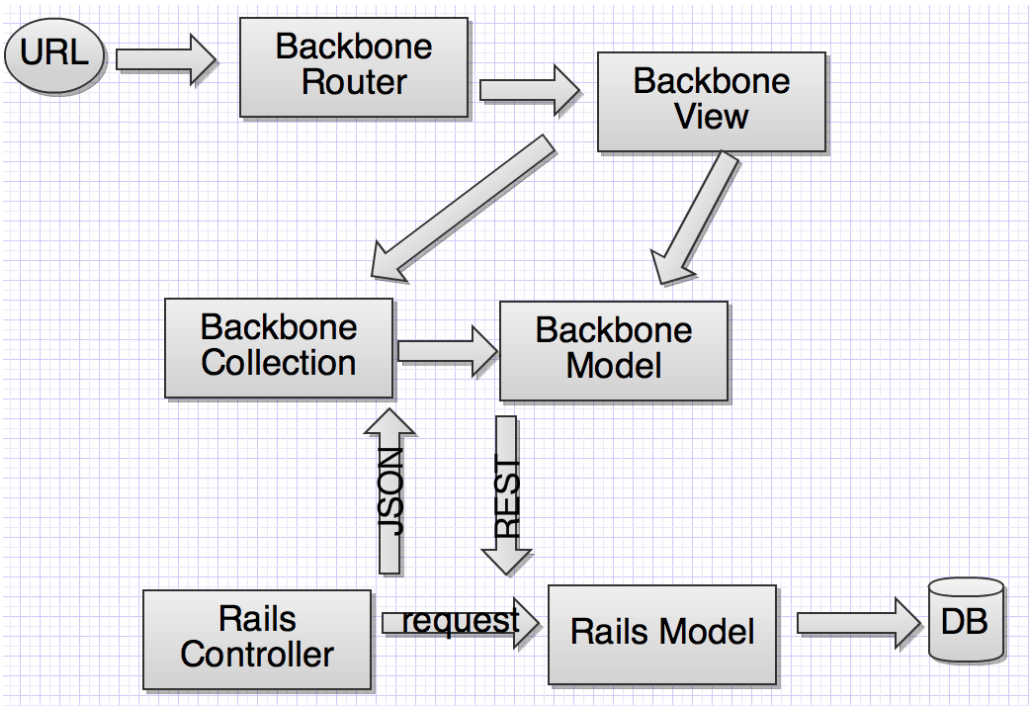

Figure 1: Backbone Workflow

What's happening in that collection of chicken scratches purporting to be a diagram? Well, it goes like this:

1. When the user changes the URL in a web page under Backbone's control, it goes to an instance of `Backbone.Router`. The Backbone router object maintains a mapping between a URL pattern and a method of the router instance that is called when a request matches the particular pattern.

2. So the router matches the URL and calls a method. What happens in that method is completely up to us. Since we are goal-oriented web developers and not, say, nuclear physicists, we typically will use that method to gather data from Backbone models and collections, and associate those items with the Backbone views. We than ask those views to render themselves to the page. It is the router's responsibility to insert the view generated DOM elements into the page.

3. If you think this sounds like what a Rails controller does, that's because it is a lot like what a Rails controller does. In fact, older versions of Backbone did call this class `Controller`.

4. Typically, the router creates instances of particular model, collection, and view classes, associating the views with the model or collection they are responsible to draw.

5. Number five is alive.

6. A Backbone `Model` is where data and business logic live. When you update a Backbone model, it triggers an event which can be caught to cause a view to redraw all or part of itself.

7. In fact, that's a critical idea in Backbone. You don't change the view directly, instead you update data in the model and cause the view to change as a result.

8. A Backbone `Collection` is a wrapper around an array of Model objects of a particular type. Collections trigger events when the contents of the collection changes. Collections also have a rich set of enumerable methods to act on their data, including methods from the Underscore.js library.

9. Both Models and Collections can fetch themselves from a server that adheres to Rails REST convention and returns JSON.

10. A Backbone `View` essentially converts a model or a collection into a DOM tree. The view defines the outermost element of the DOM tree and also a `render` method that fills in that element.

11. What happens in the `render` method of the View is completely up to us. However, since we are still goal-oriented web developers, we typically merge a template with data from the view's model to create more markup. Normally, Backbone templates have very little logic and Backbone view `render` methods apply logic to the templates.

12. Backbone views can be nested, typically meaning that a view that is tied to a collection will, as part of its render action, create subviews tied to each model and trigger the `render` method on each of the subviews. Again, Backbone does not provide specific support for this plan.

13. Number thirteen is unlucky.

14. Views can register events, both by registering themselves as observers of model change events and by registering events that would be triggered by user actions, such as mouse clicks, that happen inside the view and trigger view methods.

That seems like a lot of things – 14 (well, 12 if you skip the dumb jokes), but nearly everything that we're going to do in Backbone is going to be covered by those points.

Section 2.6

Back to the Code

Okay, normally, I like to fix the immediate error first, as simply as possible. In this case, the first error is the fact that our `Trips` collection doesn't exist. In our case, it makes no sense to have a collection without defining the model, I'll throw in the definition of the model as well.

Filename: app/assets/javascripts/backbone/models/trip.js (Branch: section_9_1)

```
TimeTravel.Models.Trip = Backbone.Model.extend({

});

TimeTravel.Collections.Trips = Backbone.Collection.extend({
  model: TimeTravel.Models.Trip,
```

```
  url: "/trips"
});
```

Sample 2-6-1: Our skeleton trip and trip collection

Our trip model here does exactly nothing except extend the relevant Backbone parent class. The use of `extend` here essentially uses the `_.extend` functionality from the Underscore.js library. Without getting into the details too much, we're effectively setting up `Backbone.Model` as a superclass of our `Trip` model, meaning that new instances of `Trip` have all the generic model behavior that Backbone defines. Technically, we're copying all the data from `Backbone.Model`, including method definitions, into our new class.

Our `Trips` collection does do a couple of minor things. When you call any of Backbone's `extend` methods, the argument is a JavaScript literal object that is the description of your specific class's behavior. In other words, inside that literal, you'd define any methods or attributes of your specific class. In addition to our class-specific logic, there are a handful of attributes that Backbone looks for to define your class's behavior.

In the case of collections, two of the most important literals are already defined in our `Trips` collection. First, we have the `model` attribute, whose value is the model class whose instances make up the collection.[1]

Backbone does not insist on any inherent relationship between a collection and a model – unlike Rails, it's not going to try to infer default behavior, so we need to explicitly say that the `TimeTravel.Collections.Trips` collection is made up of `TimeTravel.Model.Trip` objects. This means that Backbone collections are not heterogeneous – every object in the collection is an instance of the same model.[2]

We've also set the `url` attribute of the collection. This represents the server URL that the collection can use to create itself from server data. The server is expected to return a JSON list of the collection's model objects. Models can also fetch themselves, and Backbone is smart enough to use the URL from the collection related to the model when the model does not specify its own URL. We will talk more about the data fetching process later.

1. Yes, yes, I know, JavaScript doesn't have classes. If you are feeling pedantic, insert the phrase: "the object that extends `Backbone.Model`" for "model class".

2. I don't think there's any naming convention for collections that's consistently adhered to. I've seen both the plural (`Trips`) and the suffix (`TripCollection`) used. I prefer the plural because it's shorter, but it's potentially easier to confuse with the model class name.

With the model and the collection in place, we can make it a whole one line before we hit the next problem, namely that `TimeTravel.Routers.TripRouter` doesn't exist. That's fixable.

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_9_1)

```
TimeTravel.Routers.TripRouter = Backbone.Router.extend({
  routes: {
    "": "index"
  },

  index: function() {
  }
});
```

Sample 2-6-2: Our skeleton router

We've got two attributes in our router class. The `routes` attribute is used by Backbone to associate request URL's with outcomes. The keys of that object are route patterns. In our first case, we're using the empty string, which matches the route URL. The values of the object are the names of methods of the router. We'll also talk more about routing as we go.

At this point our code failure is actually a logic failure, namely that we expect the trip data to show up on our page and it does not. At this point, we've laid all the support foundation we need to start and its time to actually figure out how the heck we actually put stuff on the screen.

Section 2.7

How The Heck We Put Stuff On The Screen

The great thing about Backbone is that you have total freedom to do anything you want in response to a URL request.

The frustrating thing about Backbone is that the framework provides no guidance as to what you might do in response to a URL request.

Luckily for us, we kind of know what we want to do. Well, I know what we want to do, and you'll know by the end of this paragraph. We want to create a bunch of views and ask them to render themselves.

Each view we create will be responsible for a particular part of the screen. The overall strategy looks like this:

Figure 2: View Structure

We're creating three separate view structures, a top navigation view, a sidebar view, and a main view containing the trip list. The trip list view itself contains a separate individual trip view for each trip.

I've made the top view and the side view deliberately simple so we can focus on the dynamic behavior we want. Since they are basically static and function as a layout, meaning that we will keep them as we navigate the page, we can just initialize them once by adding them to the `initialize` method of the router object. Backbone automatically calls `initialize` when a new instance is created (across Routers, Views, Models, and Collections). Our initialize function can be pretty simple:

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_9_2)

```
  initialize: function() {
    this.topNavigationView = new TimeTravel.Views.TopNavigationView();
    this.sidebarView = new TimeTravel.Views.SidebarView();
  },
```

Sample 2-7-1: The initialize function creates two views

As promised, this just creates two new Backbone view objects. These Backbone view objects are very boilerplate with basically no dynamic behavior at this point. (Although you'd imagine that a really fully featured time travel agency site would have dynamic behavior here.) Let's look at the top one first:

Filename: app/assets/javascripts/backbone/views/topNavigationView.js (Branch: section_9_2)

```
TimeTravel.Views.TopNavigationView = Backbone.View.extend({
  tagName: 'nav',
  className: 'topNav',

  initialize: function() {
    _.bindAll(this, 'render');
  },

  render: function() {
    this.$el.html(
        TimeTravel.template('topNavigationViewTemplate').render());
    return this;
  }
});
```

Sample 2-7-2: The basic view for the top navigation

The structure here starts from the same place as the other Backbone objects we've created, it's our class, namespaced into the `TimeTravel` object and extending from the core Backbone library, this time `Backbone.View`. The argument to extend, again, is a JavaScript literal object.

Each of the three bits of code inside the extend argument is a little bit of how Backbone views work.

First up, we're setting two literal attributes, `tagName` and `className`. A Backbone view controls a specific DOM element, which is referred to within the object as `el`, or `$el` for the jQuerified version of the element. You can specify the basic parts of the DOM element by setting the `tagName`, `className`, and `id` attributes, each of which goes to creating the eventual `el` – you can also specify `el` directly. If you don't specify the `tagName`, the default is `div`. Our `TopNavigationView` has an HTML5 buzzword-complient `nav` tag, and the class name of `topNav`. For no particular reason, I've declined to specify a DOM id here.

The `initialize` method is automatically called when a new view is created. We don't have any logic to instantiate, but we do use the Undescore.js `bindAll` method. This is a very common piece of Backbone boilerplate. The `bindAll` method takes a first argument, which is a context, and an arbitrary number of future arguments, which are the string names of the context's methods. Using `bindAll`, you guarantee that the methods are always executed in the context of the first argument – meaning that `this` within the method will *always* refer to the object that is the first argument to `bindAll`. (If you remember back to Book One, we're behaving similar to the jQuery `$.proxy`.) This would be true even in cases, such as event handlers, where jQuery or Backbone might try to reset `this` for the method. It's quite typical

Should I be testing these simple views?

Hey there, mister "more Jasmine than a Disney Princess convention". Here we just created two view objects and we haven't written any tests for them at all. Hypocritical much?

Okay, you got me... I can explain, though.

My goal with testing is not to perfectly verify the behavior of my code, though that's a nice side effect. And it's not to reflexively coat the entire application in tests. My goal is to improve the quality of the code by building it incrementally, and to have confidence in my ability to change the code without breaking things.

Often, in a case where I'm adding little to no actual logic, in a place that might be costly to set up a test, and where a failure would be transparently obvious during use, I'll skip BDD cycle. For these views, the code is boilerplate and has no logic. The most common failure mode – a typo – would be caught by the acceptance test, and the second most common – failure to connect the view – would be obvious. So I don't have tests.

Later, if I add actual user login logic to the top navigation, then I would, of course, use tests to drive that change.

to use `bindAll` in a Backbone view to ensure that `render` or any related methods always have the view itself as context.[3]

In the `render` method, we draw the view. Again, Backbone provides no particular structure for rendering, but generally you're going to wind up setting the `html` attribute of the `$el` element. By convention you return `this`, which is to say, return the view itself, so that the render call can be chained with other view methods. In our `render` method, all we are doing is filling the HTML with the result of rendering a Mustache template. We're storing our Mustache templates in `app/assets/javascripts/backbone/templates` and using the `HoganTemplates` gem to manage them. The `template` method of `TimeTravel` that's being called looks like this:

Filename: app/assets/javascripts/backbone/time_travel.js (Branch: section_9_2)

```
  template: function(filename) {
    return HoganTemplates["backbone/templates/" + filename];
  }
```

Sample 2-7-3: Our helper function to render templates

It's just using `HoganTemplates` to find the template. The template itself is just plain HTML, not dynamic at all:

Filename: app/assets/javascripts/backbone/templates/topNavigationViewTemplate.mustache (Branch: section_9_2)

```
<h1 class="span-10">
  <a href="/">Time Travel Adventures</a>
</h1>
<div class="span-14 last">
  <div>
    <a href="#">Log In</a>
    <a href="#">Sign Up</a>
  </div>
</div>
<br clear="all" />
```

Sample 2-7-4: The top nav

3. Yes, CoffeeScript fans, in CoffeeScript you would usually just define `render` with the fat arrow (`=>`) to get the same behavior.

Our sidebar view is identical, except for a few different literals:

Filename: app/assets/javascripts/backbone/views/sidebarView.js (Branch: section_9_2)

```
TimeTravel.Views.SidebarView = Backbone.View.extend({
  tagName: 'section',
  className: 'span-4',

  initialize: function(){
    _.bindAll(this, 'render');
  },

  render: function(){
    this.$el.html(TimeTravel.template('sidebarViewTemplate').render());
    return this;
  }
});
```

Sample 2-7-5: The basic view for the top navigation

I think we can all figure out what the sidebar HTML looks like – again, there's nothing dynamic there.

Okay, that takes us through the boring views. Next we need to invoke them when the router draws the page. What we need to do here is ask the views to render themselves. Since the router's `index` method is invoked for page draw, we make a couple of calls there:

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_9_2)

```
  index: function() {
    $container = $("#container");
    $container.append(this.topNavigationView.render().el);
    $container.append(this.sidebarView.render().el);
  }
```

Sample 2-7-6: The router index function attaches two views

Again, we have total freedom to do whatever we want in this method. We've started by grabbing the `$('#container')` element, using jQuery – this element was defined in the

original Rails layout as being inside the body and is the place to put page content. We then take each view instance in turn, call its `render` method, and then use the `el` attribute, which, if you remember, was the element that the view was managing. (You see here why the `render` method returns `this` – it's a shortcut to allow us to chain the request for the `el` element.) We take that element and use jQuery's `append` method to insert the element on the page.

At this point, our Backbone app actually does something – if you hit `localhost:3000`, then you'll see the top navigation and sidebar. Yay?

It's time to be a little more dynamic and get our trip list displayed.

Section 2.8

Drawing our Trip Collection

Our Backbone page does draw stuff to the screen, but our end-to-end test still doesn't pass because we don't have trips on the screen yet. (And don't worry, we will have more logic that requires writing tests, especially in the next chapter). Let's fix that problem.

What we need to do in the router action is create a new trip collection view, associate it with our list of trips, and append the result into our container element. The trip collection view needs to draw itself, then create a new trip view for each trip, then draw those.

We have a lot of possible angles of attack. My inclination is to start with the boilerplate stuff, then write tests at the point where more involved coding is needed.

Our boilerplate for the collection view starts off as very similar to the top and sidebar views. We create one and render it:

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_9_3)

```
  index: function() {
    $container = $("#container");
    $container.append(this.topNavigationView.render().el);
    $container.append(this.sidebarView.render().el);
    var $content = $("<div/>").attr("id", "content");
    $container.append($content);
    var tripsView = new TimeTravel.Views.TripsView({
```

```
        collection: TimeTravel.trips});
    $content.append(tripsView.render().el);
  }
```

Sample 2-8-1: The router index now with a trip collection view

Two quick points here. We're creating our `TripsView` instance inside the `index` method, because it belongs to this specific page hit, and not to the application as a whole. We've also wrapped at inside a `#content` tag, which is there for eventual styling purposes. Spoiler Alert: eventually, we'll factor out the top and sidebar into their own layout method callable from any routing action.

TripsCollection Boilerplate

Our basic `TripsView` is pretty much identical to the views we've seen to date. It won't stay that way, but let's start from here. If you are getting the sense that there's some repetitive boilerplate in Backbone, you are kind of right, though I think the Backbone designers would say that the tradeoff of being able to be flexible in your structure is worth it.

Filename: app/assets/javascripts/backbone/views/tripsView.js (Branch: section_9_3)

```
TimeTravel.Views.TripsView = Backbone.View.extend({
  tagName:'section',
  className:'all-trips span-20',

  initialize: function() {
    _.bindAll(this, 'render');
    this.collection.bind('reset', this.render, this);
  },

  render: function() {
    this.$el.html(TimeTravel.template('tripsViewTemplate').render());
    return this;
  }
});
```

Sample 2-8-2: The boilerplate Trips view

And the template is super, super simple.

Filename: app/assets/javascripts/backbone/templates/tripsViewTemplate.mustache (Branch: section_9_3)

```
<h1 class="trip_index_header" id="headline">
  Choose Your Time Adventure!
</h1>
<div class="trips"></div>
```

Sample 2-8-3: A boring trip template

At this point, our code will run and draw the header for Time Travel Adventures.

If you are familiar with Rails, you might be surprised that we're not putting any kind of loop logic in the template to draw individual views. Instead, we're going to do that from inside the view's `render` method.

Finally, Drawing Some Trips.

Which brings us to some actual testable logic, which is good, because my trusty testing fingers were getting itchy. What we want to do is create an individual view for each trip, and render it. This is actually logic beyond Backbone boiler plate, and so we should actually drive this with Jasmine tests.

I'm going to break this up into two separate, testable pieces. Piece 1: can the `TestsView` cause an individual `Test` to be rendered? Piece 2: does the normal render process cause all the tests to be rendered? Breaking these into two tests probably leads to better code because the we're building in smaller steps.[4]

Here's that first test:

Filename: spec/javascripts/tripsViewSpec.js (Branch: section_9_4)

```
describe("rendering all the trips with Backbone", function() {

  var tripData = [{"description":"A cruise","end_date":"1620-11-21",
    "id":13, "image_name":"mayflower.jpg", "name":"Mayflower Luxury Cruise",
     "price": 1204.0, "start_date":"1620-05-17",
```

4. Admittedly, this wasn't my first choice for how to test this – I originally wanted to spy on the creation of `Test`, but there was a weird incompatibility between Jasmine spies and Backbone objects that I didn't want to track down. Turns out, I like the way I finally did it better.

```
    "tag_line":"Enjoy The Cruise That Started It All"},
    {"description":"See plays", "end_date":"1605-10-31", "id":14,
    "image_name":"shakespeare.jpg", "name":"See the Plays of Shakespeare",
    "price": 1313.0, "start_date":"1604-11-01",
    "tag_line":"See The Master As Intended"}];

  describe("renders individual views", function() {

    beforeEach(function() {
      this.trips = new TimeTravel.Collections.Trips(tripData);
      this.tripsView = new TimeTravel.Views.TripsView({
              collection: this.trips})
    });

    it("renders a single trip", function() {
      tripView = this.tripsView.renderTrip(this.trips.at(0));
      expect(tripView.$el).toHaveClass("trip");
    });

...

  });

});
```

Sample 2-8-4: We're testing that a `TripView` can draw a `Trip`

I've copied over the `tripData` object, which probably I should just keep in a common file. In my `beforeEach` function, which I have because there will eventually be two specs with common setup, I'm creating a new `Trips` collection from the raw data and directly associating that collection with a `TripsView` – notice that we explictly pass the collection to the view as part of the argument to the constructor. In the spec itself, I'm calling a method called `renderTrip`, which will take a `Trip` as an argument, and should return a Backbone view. And I'm verifying that the result of rendering a trip is, well, that a trip gets rendered.

To make this work, we need the `renderTrip` method to be defined in `TripsView`, and we need a boilerplate view for the singular `TripView`.

In the interests of completeness, and since digital pages are cheap, I'll post the boilerplate part first, which is the `TripView` class. By this point, though, you should get most of the basic drill.

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_9_4)

```
TimeTravel.Views.TripView = Backbone.View.extend({
  className: 'trip trip_entry span-6',

  initialize: function () {
    _.bindAll(this, 'render');
  },

  render: function() {
    this.$el.html(TimeTravel.template(
        'tripViewTemplate').render(this.model.attributes));
    return this;
  }
});
```

Sample 2-8-5: The boilerplate single trip view

This view points to a template that is, if not actually complex, at least has some dynamic mustache-ness in it. You can tell that the template is dynamic, because now we're passing the model to the `render` method. (Technically, we're passing the attributes of the model, rather than the model iteself, since the template expects a plain old JavaScript object). I'm only putting the whole template here as part of the boilerplate because it's easily adapted from the existing server-side Rails app. Normally, I'd just build it minimally at this point:

Filename: app/assets/javascripts/backbone/templates/tripViewTemplate.mustache (Branch: section_9_4)

```
<div class="trip_header">
  <a href="/trips/detail/{{id}}" class="detail_page_link">{{name}}</a>
</div>
<div class="trip_tag">{{tag_line}}</div>
<div class="trip_dates">{{start_date}} - {{end_date}}</div>
<div style="text-align: center">
  <img src="assets/{{image_name}}" height="150" width="150" />
</div>
<div class="trip_price">{{price}}</div>
```

```
<div class="trip_links">
  <a class="trip_detail_link" href="#">Show Details</a>
  <div class="hidden trip_details">{{description}}</div>
</div>
```

Sample 2-8-6: The template for a single trip.

Which brings us, at last, to code that will make one of our tests pass.

Filename: app/assets/javascripts/backbone/views/tripsView.js (Branch: section_9_4)

```
  renderTrip: function(trip) {
    var view = new TimeTravel.Views.TripView(
        {model: trip, collection: this.collection});
    return view.render();
  }
```

Sample 2-8-7: The boilerplate Trips view

And this isn't even all that much yet. All we're doing is creating the `TripView`, associating it with the given trip and trip collection, and rendering it. Note that we're returning the same `TripView` that is returned by the `TripView`'s actual `render` method, on the grounds that Backbone style suggests that `render` methods should return view instances.

Now, all we need to do is ensure that the `renderTrip` method is actually used. Here's the spec – it goes at the ⋯ point in the `tripsViewSpec.js` listing above.

Filename: spec/javascripts/tripsViewSpec.js (Branch: section_9_4)

```
    it("renders all the trips in context", function() {
      spyOn(this.tripsView, 'renderTrip').andCallThrough();
      this.tripsView.render();
      expect(this.tripsView.renderTrip.calls.length).toEqual(2);
    });
```

Sample 2-8-8: We're testing that a TripView can draw all the Trips

This spec is a little bit more interesting. Having established in the previous spec that `renderTrip` is actually used, all we need to establish in this spec is that the method is invoked. Since we aren't testing the state after the method, just the behavior of the `render` along the

way, a Jasmine spy is the perfect thing. Here we spy on the `renderTrip` method, call the main `tripsView.render`, then check to see that `renderTrip` is called twice, or once for each trip.[5]

At long last, we can change the `render` method to call the trip.

Filename: app/assets/javascripts/backbone/views/tripsView.js (Branch: section_9_4)

```
render: function() {
  this.$el.html(TimeTravel.template('tripsViewTemplate').render());
  var $trips = this.$('.trips');
  this.collection.each(function(trip, index) {
    $trips.append(this.renderTrip(trip).el)
  }, this);
  return this;
},
```

Sample 2-8-9: The boilerplate Trips view

There are a couple of items in this method that we haven't seen before. After rendering the collection template, we grab the `.trips` element via jQuery – that's where we will put all the trips. We then use `each` to iterate over each element in the collection. The `each` method is defined by the Underscore.js library – Backbone allows it's collections to define all of Underscore's iterator methods. We've got kind of a weird method signature for `each`, though, the first argument is a function, and the second argument is `this`.

The anonymous function is invoked once for each element in the collection, with the element itself being the first argument and a counter index being the second. The second argument, `this`, placed awkwardly after the anonymous function, specifies the context that the anonymous function is evaluated within. In other words, we are ensuring that inside the anonymous function, `this` will be the same as that second argument, here representing the `TripsView` instance itself. If the context argument is not included, then `this` inside the anonymous function will be set to the global `window` object, and the call to `this.renderTrip` will fail.[6].

5. Some people would advocate replacing `toEqual(2)` with `toEqual(tripData.length)`, on the grounds that it's not a "magic literal". I have no strong feelings about this – I can make a case either way, so do which ever one you are comfortable with.

6. CoffeeScript fans are invited to be smug about the fact that the fat arrow would avoid the problem entirely.

And at this point, not only does our immediate spec pass, but the original acceptance spec also passes – we've wired up all the pieces. Even better, if you hit `localhost:3000` in the browser, you get a Backbone version of the trip listing that is nearly identical to our original Rails and jQuery one.

Section 2.9

Time Out

Let's pause for a second. We've thrown a lot of code around, used the word "boilerplate" too many times, and pretty much just created the same page that we started with. And not even that, because we haven't done the show/hide toggle yet. What have we actually accomplished?

We've created the foundation. We've explored many of the basic concepts underpinning Backbone's flavor of Model/View/Controller, and we've wired them together to build a page.

In Book 1, the Rails server sent us HTML, and we used jQuery to modify the resulting DOM tree in a relatively unstructured way – even when we added structure to turn our toggler into an object, the interaction with the DOM was still haphazard, and the DOM itself had no semantic structure beyond that given to us in the initial HTML.

In the Backbone version of the page, we've been sent the raw data, and built the page up ourselves – it's the difference between being handed a house and being handed the raw materials. We've had to put together the house ourselves, but we have much more flexibility from this point on. We have the raw data to redraw all or part of a page in response to a user action, and we've structured the page to match the structure of our data. This will wind up being useful when we start talking about more complex user actions than just drawing the page.

At this point, we can already see that Backbone's relative simplicity is both a strength and a weakness. The strength is that it gives us extraordinary flexibility to do whatever we want. However, that flexibility comes at the cost of having to tell Backbone what we want to do, there's relatively little in the way of Backbone trying to guess or assume what we want to do. Hence, the fact that we are continually telling our Backbone views that "render" means "find a template and draw it". If you are familiar with Rails, this style will seem verbose, at least for a while, and there's something of an ecosystem for add-ons to Backbone that offer some common abstractions.

Okay, time out is almost over. There are a few more things we need to clean up on this page.

Section 2.10

Time In

There are two things about the page that I'd like to fix. The second – building the show/hide toggle – we'll get to momentarily. The first is a relatively minor cosmetic issue – some of our data is being displayed poorly. Look:

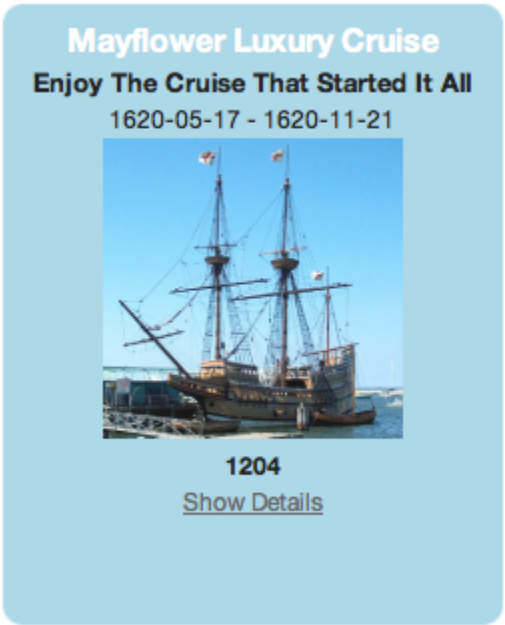

Figure 3: A Badly Formatted Trip

The dates are in SQL format, which is not particularly user-friendly, and the prices are listed as a bare integer, rather than the actual formatted price. While these are minor issues in this case, the generic case where we want to display information from a model that might be go beyond the raw data is pretty common.

How you handle this in Backbone depends on which templating engine you are using and also where you feel like landing on the purist vs. pragmatist spectrum. Let me put that another way. You might imagine that the way to handle, say, the date formatting would be to create a method called `formattedDate` in the `Trip` model class, then have something like `{{trip.formattedDate}}` in the template.

That's straightforward, but that won't quite work as-is in Backbone. On a pragmatic level, we don't actually pass a `Trip` instance to the template. If you look at the `TripView` `render` method, it contains this line:

```
this.$el.html(TimeTravel.template(
        'tripViewTemplate').render(this.model.attributes));
```

Sample 2-10-1: We actually don't pass the model directly to the view...

Our argument to the template's `render` function[7] is `this.model.attributes`, which is not the entire Backbone model, but rather just a JavaScript object containing the attribute values. (That's one reason why you are expected to deal with Backbone model attributes using `get` and `set` – it allows the Backbone model to deal with the internal `attributes` data behind the scenes.)

Okay, so our practical issue is that we can't just add a `formatted_date` method to the model, because the model is not available in the template for the method to be called. We'll probably still need some kind of `formatted_date` method, but we'll have to add the display-ready data from that method to the object that we send to the template.

We need a `formatted_date` method. Great. Where do we define it? Noodling over this for a moment, it seems to me like there are four answers that you can make an argument for.[8]

1. The model. After all, that's where the raw data is.
2. The view. Date formatting is a display concern, not a model concern.
3. A presenter object between the model and the view.
4. The server should pass the formatted version to the client.

The server option might be appropriate if there's a metric boatload of processing to be done on the data, and if it's the kind of thing you might cache in the server, it's probably not necessary here. Similarly, adding an intermediate presenter object seems like overkill at this

7. Yes, it's confusing that both Backbone `views` and Mustache templates both use the word "render" for their drawing method.

8. And yes, it is absolutely worth taking a couple of minutes to think about where your code is going and make sure that it's in the right place. Even for a method this trivial. The best way to keep a code base clean is to keep it from getting messy in the first place.

point, though its something I might consider if a model had radically different presentations in different views.

For the simple display changes we're looking to add, then, the question is whether they belong to the model or the view. I confess that I go back and forth on this. On the one hand, display specific information seems almost by definition to be the responsibility of the view. On the other hand, Backbone models are in some sense "view models" and have a responsibility to present themselves to the view.

In the end, I think the question comes down to dependencies. Whatever we name these formatted attriubtes, if we want the formatted version to appear on the screen, that name has to appear in the template. If we also have that name come from the model, that adds a dependency where the model needs to know about the details of the template. Put that way, it seems like display-only attributes should be created by the view.

For better or for worse, this does appear to be accepted style in Backbone, where the majority of logic winds up in the various View classes – which is an incentive to keep individual views focused on small parts of the page so that they don't fly out of control.

Section 2.11

Testing View Logic

Having decided where to put our display code, it's time to start. First, we write a test:

Filename: spec/javascripts/tripViewSpec.js (Branch: section_9_5)

```
describe("with a trip view", function() {

  beforeEach(function() {
    this.trip = new TimeTravel.Models.Trip({"description":"A cruise",
      "end_date":"1620-11-21", "id":13, "image_name":"mayflower.jpg",
      "name":"Mayflower Luxury Cruise", "price":1204.0,
      "start_date":"1620-05-17",
      "tag_line":"Enjoy The Cruise That Started It All"});
    this.tripView = new TimeTravel.Views.TripView({model: this.trip});
  });

  describe("display of individual trip values", function() {
```

```
    it("properly displays a date", function() {
      expect(this.tripView.formatDate("1620-05-15")).toEqual("May 15, 1620");
    });

  })

});
```

Sample 2-11-1: Our first test for display of the trip data

This is admirably straightforward. In the `beforeEach`, we create a `Trip` model and an associated `TripView` – we don't use the model in this test, but we will in the next one. In the actual spec, we call a new method `formatDate`, that appears to take a date in the original SQL format and convert it to a more human friendly display format. We're actually designing our API here a bit by passing `formatDate` the original date, rather than, say, passing it the entire `Trip` instance.

We're building this functionality up from the bottom – first, being able to generate the formatted date, then making sure that the formatted version is passed to the template, then potentially that the formatted version is used by the view.

To make this test pass, we're going to use the Moment.js library, available at http://momentjs.com. Rails people, we'll add `gem 'momentjs-rails'` to the Gemfile (per the Rails plugin at https://github.com/derekprior/momentjs-rails) and `//= require moment` to the `app/assets/javascripts/application.js` file. Non Rails people should download the file from http://momentjs.com and load it as part of their application.

Here's the passing code:

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_9_5)

```
  formatDate: function(badlyFormattedDate) {
    var momentDate = moment(badlyFormattedDate);
    return momentDate.format("MMMM D, YYYY");
  }
```

Sample 2-11-2: The first try at the formatDate function

The way the Moment.js library works is by converting strings or JavaScript dates to moment objects using the `moment` function. Here we have a string in a known format, so Moment will

handle the conversion. Then we can call `format` on the moment object, passing in a format string. Details on Moment's format strings and other docs are at http://momentjs.com/docs/. Also, we need to remember to add the `formatDate` method to the list of methods in the `_.bindAll` call when the `TripView` is initialized, otherwise our eventual `this.formatDate` call may fail.

Our tests are green. But, simple as this code is, I actually think there's a refactor that's worth doing. If the responsibility of the model object is to accurately represent the state of the model and the responsibility of the view is to accurately display the model, then it seems to me that the model is responsible for the data structure. Meaning that we want the Moment object to be stored in the model. Among other benefits, this structure means that the conversion from string to Moment only happens once.

We can add the data conversion to the `Trip` object automatically by adding initialize behavior to create the new attributes when a new model is built. That's a logic change, so we test-drive it:

Filename: spec/javascripts/tripSpec.js (Branch: section_9_6)

```
describe("with a trip", function() {

  beforeEach(function() {
    this.tripData = {"description":"A cruise",
      "end_date":"1620-11-21", "id":13, "image_name":"mayflower.jpg",
      "name":"Mayflower Luxury Cruise", "price":1204.0,
      "start_date":"1620-05-17",
      "tag_line":"Enjoy The Cruise That Started It All"};
  });

  describe("display of individual trip values", function() {

    it("stores begin and end dates as moments", function() {
      this.trip = new TimeTravel.Models.Trip(this.tripData);
      expect(this.trip.get("startMoment")).toEqual(moment("1620-05-17"));
      expect(this.trip.get("endMoment")).toEqual(moment("1620-11-21"));
    });

  })
```

```
});
```

Sample 2-11-3: Our test that moment objects are initialized

We've got the same old data – possibly it's time to make that a common bit – and a spec that creates a new `Trip` and verifies that new attributes are generated. To make the spec pass, all we need to do is create new attributes in the `initialize` method using the Backbone setter.

Filename: app/assets/javascripts/backbone/models/trip.js (Branch: section_9_6)

```
TimeTravel.Models.Trip = Backbone.Model.extend({

  initialize: function(attributes) {
    this.set({startMoment: moment(attributes.start_date),
        endMoment: moment(attributes.end_date)});
  }

});
```

Sample 2-11-4: Adding moment objects to the trip view

We're creating new attributes here, so the existing ones are still there should we need them for some reason.

On the view side, we need to adjust the `formatDate` method to expect a moment object rather than a string, which is easy enough:

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_9_6)

```
  formatDate: function(aMoment) {
    return aMoment.format("MMMM D, YYYY");
  },
```

Sample 2-11-5: Very minor tweak to our formatDate function

Next up, we need to ensure that the `TripView` actually uses the new values when it sends the data to the template. Meaning that we need to make sure that we prepare an object to be sent to the template that contains the correct attributes. Like this:

Filename: spec/javascripts/tripViewSpec.js (Branch: section_9_6)

```
  it("renders using the presented data", function() {
    result = this.tripView.presentTrip();
    expect(result.startDateDisplay).toEqual("May 17, 1620");
    expect(result.endDateDisplay).toEqual("November 21, 1620");
    expect(result.priceDisplay).toEqual("$1204.00")
  });
```

Sample 2-11-6: Testing for the use of the prepared test data.

Two quick things: in the interests of time and space, so to speak, I'm combining the `priceDisplay` piece into this test – normally, I'd consider doing a separate test for price as well. Also, we're not concerned with the error path, for the moment, we can assume that all trips will have proper, well-structured data.

That test fails, of course, because we don't have a `presentTrip` method. Let's fix this.

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_9_6)

```
formatPrice: function(aFloat) {
  return "$" + aFloat.toFixed(2).toLocaleString();
},

presentTrip: function() {
  var result = this.model.toJSON();
  result.startDateDisplay = this.formatDate(result.startMoment);
  result.endDateDisplay = this.formatDate(result.endMoment);
  result.priceDisplay = this.formatPrice(result.price);
  return result;
}
```

Sample 2-11-7: We create our trip presenter.

First off, we're using the `toJSON` method of our model rather than the `attributes` method. According to the Backbone docs, `attributes` gives you a direct reference to the model's attributes, while `toJSON` gives you a literal object copy. (And not, as the name might imply, a JSON string. Because of reasons. Mostly because it matches the API for the standard `JSON.stringify`.[9]

Anyway, once we have this copy, we use our existing `formatDate` and our new `formatPrice` methods to create our display versions, and return the augmented object.[10]

If you are having test failures at this point of the form "has no method 'formatPrice'", then you need to make sure that all the methods you've created are added to the `_bindAll` call when the view is initialized:

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_9_6)

```
initialize: function () {
  _.bindAll(this, 'render', 'formatDate', 'presentTrip', 'formatPrice');
},
```

Sample 2-11-8: Binding all our methods.

Okay, one more test. I'd like to verify that the render method actually uses the prepared data.

Filename: spec/javascripts/tripViewSpec.js (Branch: section_9_6)

```
  it("uses the prepared data when rendering", function() {
    spy = spyOn(this.tripView, 'presentTrip');
    this.tripView.render();
    expect(spy).toHaveBeenCalled();
  });
```

Sample 2-11-9: Testing differently for the use of the prepared test data.

And making that pass is a simple change to the `render` method to use the new method.

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_9_6)

```
render: function() {
  this.$el.html(TimeTravel.template(
      'tripViewTemplate').render(this.presentTrip()));
  return this;
```

9. Meaning, I assume, that `JSON.stringify` looks for a `toJSON` method. Of course, they could alias it to something actually comprehensible...

10. That `formatPrice` method is, I realize, super simplistic. People who need more realistic currency handling are directed to the accounting.js library at http://josscrowcroft.github.com/accounting.js/.

```
  },
```

Sample 2-11-10: We use our trip presenter.

Which should make things pass.

In order to make the new values show up, we also need to update the template to take the new display attributes. Under normal circumstances, I wouldn't consider chaining the template to be a logic change that would be test driven, however, you might want a view-level change like this to be part of an acceptance test, so I'll show you what a test for that display change might look like:

Filename: spec/javascripts/tripViewSpec.js (Branch: section_9_6)

```
    it("uses the prepared data in the template", function() {
      this.tripView.render();
      $el = this.tripView.$el;
      expect($el.find(".trip_dates")).toHaveText(
          "May 17, 1620 - November 21, 1620");
      expect($el.find(".trip_price")).toHaveText("$1204.00");
    });
```

Sample 2-11-11: Testing differently for the use of the prepared test data.

We're calling `render` again, but this time we're explicitly inspecting the result, rather than just checking to see if the correct method call is made. This test, then, acts as more of an acceptance test than a unit test.

To make this pass, and make the display properties show up in the actual page, we need some minor tweaks to the Mustache template. Specifically, we need to use our new display attributes.

Filename: app/assets/javascripts/backbone/templates/tripViewTemplate.mustache (Branch: section_9_6)

```
<div class="trip_header">
  <a href="/trips/detail/{{id}}" class="detail_page_link">{{name}}</a>
</div>
<div class="trip_tag">{{tag_line}}</div>
<div class="trip_dates">{{startDateDisplay}} - {{endDateDisplay}}</div>
```

```
<div style="text-align: center">
  <img src="assets/{{image_name}}" height="150" width="150" />
</div>
<div class="trip_price">{{priceDisplay}}</div>
<div class="trip_links">
  <a class="trip_detail_link" href="#">Show Details</a>
  <div class="hidden trip_details">{{description}}</div>
</div>
```

Sample 2-11-12: The template for a single trip, updated.

What have we done here?

On the Backbone side, we've shown how to adjust the data that you start off with to match your display needs. On the Jasmine side, we've shown how to test that change at four different levels:

- Specifying the change to an individual datapoint.
- Specifying that all data points are changed when view-specific data is needed.
- Specifying that the prepared data is requested when the view renders.
- Specifying that the actual rendered output uses the view-specific data.

In practice, I probably wouldn't use all four of these tests on the same feature – particularly if the data is as simple as these changes are. Which levels you want to test at is somewhat a matter of what you are trying to do, and somewhat a matter of style. The goals of writing the tests are to improve the structure of the resulting code and to make you more confident in the code's accuracy. Checking the rendered output is great for confidence (though it's probably the most fragile), but may not offer as much insight into the structure of the code. On the other hand, the spy version of the test doesn't cover the final output, but may be a better guide as to the code design.

Section 2.12

Making Things Happen, Revisited

The last little feature we're going to tackle on this page is bringing our old toggler friend to the Backbone land. We are going to run into a fundamental difference between the way we handled the process of using an event to update the screen. Before, we just changed the DOM directly as a result of the user click, and our code was a little vague about which part of that process was actually the source of truth as to what the state of the display should be.

Our Backbone event process will have two significant differences. First, we have model objects which act as the source of truth about the state of the system. Second, we don't change the view directly. Instead, we change the state of the system by changing the model object, and the view observes the change in the model object and updates itself appropriately. Here's an integration test of the entire click behavior.

Filename: spec/javascripts/tripViewSpec.js (Branch: section_9_7)

```
describe("toggle behavior", function() {

  it("shows the details on a click", function() {
    this.tripView.render();
    $el = this.tripView.$el;
    $el.find(".trip_detail_link").click();
    expect($el.find(".trip_details")).not.toHaveClass("hidden");
  });
});
```

Sample 2-12-1: Integration level test for toggler behavior

This is an integration test of the rendering and event functionality. We render the trip view, simulate a click on the appropriate link, and verify that the `hidden` CSS class is no longer attached to the DOM element we expect. As we did back in Book 1, we're using the CSS class `hidden` to manage shown and unshown elements.

In Backbone, to trigger a view change, we need to trigger a change to the model and let the view know about it. So if we want to change based on the display state of the model, then the

model needs to store that state and offer a mechanism to change it. Let's drive that logic with a test:

Filename: spec/javascripts/tripSpec.js (Branch: section_9_7)

```
it("Toggles Details", function() {
  m = new TimeTravel.Models.Trip({});
  expect(m.get('detailsDisplayed')).toBeFalsy();
  m.toggleDetails();
  expect(m.get('detailsDisplayed')).toBeTruthy();
  m.toggleDetails();
  expect(m.get('detailsDisplayed')).toBeFalsy();
});
```

Sample 2-12-2: A model test for detail toggle behavior

This spec is particularly easy because it doesn't depend on any pre-existing state. We start with a blank trip model, assert that it starts with the `detailsDisplayed` attribute having a false value, then go back and forth via a `toggleDetails` method and verify the expected change to the attribute.[11].

To make this work, there are two sections that we add to the `Trip` model class:

Filename: app/assets/javascripts/backbone/models/trip.js (Branch: section_9_7)

```
TimeTravel.Models.Trip = Backbone.Model.extend({

  defaults: {
    'detailsDisplayed': false
  },

  initialize: function(attributes) {
    this.set({startMoment: moment(attributes.start_date),
        endMoment: moment(attributes.end_date)});
  },
```

11. I realize that the idea that something like display status, which is, on some level, a view concern being stored in the model is possibly at odds with the earlier discussion of where display only properties should go. I have three responses: a) Backbone models are "view models" and a property like this, which represents state and is not displayed directly, is appropriate here, b) this seems to be how Backbone's creators want the separation to look, and c) I could be wrong about any of this.

```
  toggleDetails: function() {
    this.set({'detailsDisplayed': !this.get('detailsDisplayed')})
  },

});

TimeTravel.Collections.Trips = Backbone.Collection.extend({
  model: TimeTravel.Models.Trip,
  url: "/trips"
});
```

Sample 2-12-3: Our trip now has a toggle details method

Our `Trip` model has grown a `defaults` attribute, which is used by Backbone to set default values of a model if they are not set by the object passed to the model's constructor. In our case, we want a `detailsDisplayed` attribute to start at `false`.

Then we can add the `toggleDetails` method, which uses Backbone getters and setters to swap the value of that `detailsDisplayed` method. The API here is that the getter takes a single attribute name and returns the value of that attribute, while the setter takes a hash literal of one or more pairs where the key is the attribute name and the value is the new value of the attribute. Here, the toggle method gets the current value, logically negates it, and resets the attribute.

Now we need to turn to the view, and allow the view to take advantage of this new display property. What we want to do is have the `render` function know about this property and adjust the DOM element accordingly. Here are tests for the case where the model details are hidden and where the model details are shown:

Filename: spec/javascripts/tripViewSpec.js (Branch: section_9_8)

```
  describe("uses the display details behavior", function() {
    it("hides details given state", function() {
      this.tripView.model.set({'detailsDisplayed': false});
      this.tripView.render();
      var $el = this.tripView.$el;
      expect($el.find(".trip_details")).toHaveClass("hidden");
      expect($el.find(".trip_detail_link")).toHaveText("Show Details");
    });
```

```
  it("shows details given state", function() {
    this.tripView.model.set({'detailsDisplayed': true});
    this.tripView.render();
    var $el = this.tripView.$el;
    expect($el.find(".trip_details")).not.toHaveClass("hidden");
    expect($el.find(".trip_detail_link")).toHaveText("Hide Details");
  });
})
```

Sample 2-12-4: We create our trip presenter.

So, these are unit tests, where the previous test is an integration tests, because these tests explicitly set the state of the trip and call `render` directly, whereas the integration test tries to simulate a click and verify that the change happens as a result. It's the difference between testing only one structure in your code, versus testing the interaction between two different levels.

To make this pass, we'll add a call to a new `displayDetails` method into `render`.

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_9_8)

```
render: function() {
  this.$el.html(TimeTravel.template(
      'tripViewTemplate').render(this.presentTrip()));
  this.displayDetails();
  return this;
},
```

Sample 2-12-5: Render with display details.

And then we add the display details function itself. This logic should be very familiar...

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_9_8)

```
displayDetails: function() {
  if(this.model.get("detailsDisplayed")) {
    this.$el.find(".trip_detail_link").html("Hide Details");
    this.$el.find(".trip_details").removeClass("hidden");
  } else {
```

```
    this.$el.find(".trip_detail_link").html("Show Details");
    this.$el.find(".trip_details").addClass("hidden");
  }
},
```

Sample 2-12-6: Toggle the detail elements based on state.

These changes make the unit tests pass, but the integration test is still failing. We need to do two different things: register the event so that the click triggers a call to the models `toggleDetails` link, and then register the view as an observer of the resulting change event.

At this point, Backbone takes care of some of the work for us.

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_9_8)

```
events: {
  'click .trip_detail_link': 'toggleDetails'
},

initialize: function () {
  _.bindAll(this, 'render', 'formatDate', 'presentTrip', 'formatPrice',
      'displayDetails');
  this.model.bind('change:detailsDisplayed', this.displayDetails, this);
},

toggleDetails: function() {
  this.model.toggleDetails();
},
```

Sample 2-12-7: Register events and change methods

We've added another one of those magic attributes that Backbone uses to govern object behavior, namely `events`. The `events` object lets you associate events with view methods that should be called when they are triggered. The syntax of the event description is a string were the first part is the name of the event, the second part is the selector is a selector, and the value part of the pair is a method of the view. In this snippet, `'click .trip_detail_link': 'toggleDetails'` means that Backbone associates a click event on a `trip_detail_link` CSS class element and calls the method `toggleDetails` when that kind of event happens. Events are automatically scoped to be within the `$el` element of the view itself.

You'll notice that the `toggleDetails` method of the view immediately dispatches to the `toggleDetails` method of the model. That's enough for our simple case, a more complex case might do some processing of the view before dispatching.

The final new line `this.model.bind('change:detailsDisplayed', this.displayDetails, this);` attaches the change event to a method. The first argument is the event being bound, the `change` event automatically created by Backbone when a model changes via its setter. We're being more specific, and scoping the change event to a specific attribute via the syntax `change:detailsDisplayed`. With that specification, the event only fires when that attribute changes.

The second argument is the method that should be called when the event happens. We want to call our `displayDetails` method, which will update both the show/hide state of the details as well as the text in the label. We could call the entire `render` method, though it's considered a better practice to try to change as small a part of the view as you can get away with. (On some browsers, if you call `render` then the entire trip view will flicker briefly.) The third argument is the context that the method will be called in – almost always we want that to be the view, so that `this` within the method still refers to the view.

And with that, our integration test passes. Even better, if you hit the actual page in the browser, the show/hide detail link works.

Section 2.13

Whew!

We've thrown a lot at you so far. We've covered Backbone's Model/View structure, and we've used it to display our page. We've used events to trigger a model change that itself triggers a change to a view.

In the next chapter, available Real Soon Now, we'll go a bit further into business logic and event handling in Backbone, as we create an order page, that allows you to select options for purchasing an individual trip.

As always, you can comment on this book at http://noelrap.squarespace.com/mstwjs-forum/ or contact me directly at `noel@noelrappin.com`. In the extremely likely event there are typos here, please send those notices to `errata@noelrappin.com`. Also, if you'd like to Tweet about the book, please use the hashtag `#mstwjs`. Thanks!

Chapter 3

More Nerves on That Backbone

Once again, a request from the mysterious Doctor:

> Dear Person,
>
> Thank you for making our home page have some Backbone. Now, I've noticed a somewhat embarrassing oversight in our Time Travel Adventure site. There's no way for people to purchase a ticket. Which means that my existing customers must be coming from a future in which you have already solved this problem. But in order for that to work you must will have had to have solved this problem. So be having solved it already!
>
> From The Future,
>
> Doctor What

In this chapter, we're going to build a more complex page using Backbone, hopefully one which does a decent job of showing why Backbone is useful.

Specifically, we're going to build out a detail page for each of our trips. Thus far, we haven't seen many details of the trips, so we're going to add some. Each trip will have lodging options, plus extras and additional tours that our Time Traveling customers can purchase. On the page, the user will be able to select these options, and have the running cost of their selected trip update.

In order to make this work, our Time Travel Backbone application needs to do a few things:

- Respond to a click to move from the listing page to the detail page for an individual trip.
- Obtain the hotel, tour, and extra information for the trip in question
- Display all the information

- Respond to user selection by updating the price of the user's order
- Send the order back to the server when the user is ready to buy.

We've seen pieces of some of these before, but we've got a new set of things to build.

Section 3.1

Routing To Detail

The first piece of this detail page that we are going to build is simply the ability for the Backbone application to recognize that we've moved to the detail page and act accordingly. The trip views in the previous chapter were structured to provide a link to a detail page for each trip. In response to that link, we basically have three options:

1. Do the traditional thing, send the link back to the server and draw another page. For more information on this option, you are invited to peruse any other book on server-side web development.
2. Respond using jQuery to redraw the elements of the page.
3. Use Backbone's router to go through the Backbone stack and draw a new page within the Backbone app.

Since this is the Backbone chapter, we're going to try option 3.

We will start, naturally, by writing a test. Again, there are a few different reasonable ways to pry our way into this process via testing. I think the first thing I want to test is the actual event where I click on a link, and it stays in the Backbone system.

Program Note: If you are following along on the provided source code, when you go to branch `section_10_1`, you need to re-run the `rake mstwjs:setup` task, because new tables have been added to the database to manage the hotels, tours, and extras.

Filename: spec/javascripts/homePageSpec.js (Branch: section_10_1)

```
afterEach(function() {
  Backbone.history.stop();
});
```

```
describe("transition to detail page", function() {

  it("goes to a detail page on click", function() {
    affix("#container");
    TimeTravel.init(tripData);
    Backbone.history.stop();
    spyOn(TimeTravel.app, 'navigate');
    TimeTravel.app.index();
    $("#detail_page_link_13").click();
    expect(TimeTravel.app.navigate).toHaveBeenCalledWith('trips/detail/13', true);
  });

});
//##detail

//##detailDisplay
describe("display of detail page", function() {
  it("displays details", function() {
    affix("#container");
    TimeTravel.init(tripData);
    var $container = TimeTravel.app.tripDetail("13");
    expect($("#trip_detail_13").size()).toEqual(1);
  });
});
```

Sample 3-1-1: A first test to move to our detail page

This test has some bits we've already seen, combined with some tricky stuff to get around the way the Backbone router works. We're initializing our fixture and our `TimeTravel` global object. Then we're using a Jasmine spy to spy on the router. For the actual action part of the test, we're simulating starting on the index page, then clicking on one of the detail links. We're then using the spy to validate that a specific method of the Backbone router is called with the expected argument.[12]:

12. The `detail` part of the URL is to prevent this version of the trip show page from clashing with the server-side version with the rating widget that we did in Book 2.

A Brief, Necessary, Discussion of History and Routing

Let's talk about our interactions with the Backbone router and history objects here. We're interacting in three places – interacting with the history object by calling `Backbone.history.stop()` line in that `afterEach()` function, then another call to `stop()` in the test itself. Finally, we're spying on the `navigate` method of the router itself. The reason why we're doing these calls in this particular way has to do with how Backbone really, really wants to take over our page when we call it in our tests. Allow me to explain.

The Backbone history object, which manages Backbones's URL stack so that Backbone plays nicely with the browser back button, is a global object that is intended to only start itself up once. That's a problem when we're testing, because the Backbone history object is started in our `TimeTravel.init()` method, which may get called by the setup of any number of tests. If you take out that `afterEach()` function, you'll notice that Backbone fails our test because it's trying to start the history object a second time.

To prevent the error, and allow us to create test setups multiple times, Backbone provides the `Backbone.history.stop()` method, which, as you'd expect, stops Backbone from tracking history. Incidentally, stopping the history also allows it to be restarted. By placing the call in an `afterEach()` method that is outside our `describe` scope, we're ensuring that the history object will be stopped after every test, specifically that it will be stopped after the home page test that's already part of this file.

Which brings us to our actual test. In the test, we call `TimeTravel.init()`, which restarts the Backbone history object, then we immediately stop it again with another `Backbone.history.stop()`. Why do I do that? What do I have against the poor, downtrodden Backbone history object?

Well, if it isn't stopped, the history object and the router will hijack the Jasmine page. If that second `stop` method isn't there, then Backbone will, when the code is in place, respond to the `click` method be actually routing the page to the new URL. Very handy in production, somewhat irritating when the page is question is the Jasmine test runner. While there are some more complex mock-object incantations that can override normal behavior here, I think the clearest and easiest thing to do for now is just shut the history and router object off.

However, solving that problem leads to another problem, namely that with the Backbone history object stopped, our simulated click won't actually get matched against our router

object. All we can to is test that our response to the click sends the expected URL to the router's `navigate` method, which, in production, would compare it to our routes.

Back To The Code

In order to make the test's jQuery selector work, I needed to make a minor change to the `tripViewTemplate.mustache` file to add a DOM id to the link, which I neglected to do before, just add `id="detail_page_link_{{id}}"` to the anchor element in the template.

Now, running the tests leads to the failure message `Expected spy navigate to have been called with [ 'trips/detail/13', true ] but it was never called.` This is because we are expecting the spy navigate to have been called. But it isn't called. You probably figured that part out.

To make that call, we first have to catch the event, which we do by adding another key/value pair to the `events` attribute in the trip view.

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_10_1)

```
events: {
  'click .trip_detail_link': 'toggleDetails',
  'click .detail_page_link': 'goToDetailPage'
},
```

Sample 3-1-2: Trip View event object, now with the detail page click.

As you may recall from way back in the previous chapter, the key here `click .detail_page_link` indicates an event and a jQuery selector, while the value is the method to be invoked when the event is fired. Namely, this method:

Filename: app/assets/javascripts/backbone/views/tripView.js (Branch: section_10_1)

```
goToDetailPage: function(event) {
  event.preventDefault();
  TimeTravel.app.navigate("trips/detail/" + this.model.get("id"), true);
},
```

Sample 3-1-3: TripView response to click on detail link

Backbone ensures that the event object is the argument to an event handler, which is handy, because we want to do something to the event. Specifically, we want to call `preventDefault`, which, if you remember really way back in Book 1, keeps the actual default DOM event from happening – in this case, preventing a call to the Rails server.

Preventing a call is actually an issue in this case, because in Book 2, we really did write a page to be executed at that URL. Obviously, in a real production system, you wouldn't want totally different server side and client-side versions of the same page. Less obviously, even if all the code is client side, it's still possible that a user might enter the app via the detail link, and we need to make sure that link provides enough data for Backbone to start itself up at the detail URL.

The handler then calls the `navigate` method on our global router, which does two things. First, it updates the Backbone history to track the URL in the first argument as part of the Backbone history. Second, because we passed `true` as the second argument, the URL will be bounced through the Backbone router to see if it matches any routes. Well, at least that's what would happen in production – as mentioned above, we've muffled that Backbone feature during our tests in the name of test running sanity.

At this point the test passes, and it's time to turn our attention to the router and how it will respond to the new route.

Section 3.2

The Router, and How It Will Respond to the New Route

First thing the router needs is to define the pattern for the route:

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_10_1)

```
routes: {
  "": "index",
  "trips/detail/:id": "tripDetail",
},
```

Sample 3-2-1: TripView response to click on detail link

Our new route is the second one. The key part of that key/value pair is the pattern to match, namely `trips/:id`. As in Rails and other routing engines, the `:id` indicates a dynamic part of the route. The value, `tripDetail`, is the name of a method invoked when the route is matched. Any dynamic parts of the route pattern are passed, left to right, as arguments to the method. Backbone will also fire an event named `route:tripDetail`, which can be observed and reacted to by arbitrary other objects. Route patterns can also contain a splat part, which matches an arbitrary number of components, as in `document/*features`, which would match a URL like `document/name/fred/value/true` and pass `name/fred/value/true` as a single argument to the handler method.

Now that we have a matching route, we need to write the handler that will be called by the route. Here's a very simple acceptance-style test just verifying that whatever else happens when the method is triggered, something representing the trip gets drawn on the page.

Filename: spec/javascripts/homePageSpec.js (Branch: section_10_1)

```
describe("display of detail page", function() {
  it("displays details", function() {
    affix("#container");
    TimeTravel.init(tripData);
    var $container = TimeTravel.app.tripDetail("13");
    expect($("#trip_detail_13").size()).toEqual(1);
  });
});
```

Sample 3-2-2: A bare-bones acceptance test for the trip detail page

A couple of things worth mentioning. We're setting up something of a chain of test responsibility here – we've written a test to take us from the click to the router, and this test takes us from the router to the page. What we are missing is a test that the route matches the pattern we think it matches. I'm skipping that test on the grounds that a) the route logic is very simple, b) if it's wrong, that should be easy to spot in the actual app, and c) because of how Backbone routers work, the test would probably be ugly to write. So the cost/benefit of the test is not in favor of writing that test.

Also, this test may not be finished. For one thing, we haven't talked about downloading or displaying the hotel and extra information, in part because I want to get the skeleton page going first, and in part because I'm not 100% sure at this point how I want to deal with that data.

Meanwhile, On The Server

To make this a little easier to deal with on the server side, add the following line to the `config/routes.rb` file: `match "trips/detail/:id" => 'home#index'`. This is a little bit of a hack – if you come into the site from a detail URL, then the server will do exactly what it would do if you came in at the root URL, namely send down all the trip data and launch Backbone. If you come in via a detail URL, though, then the Backbone router will catch the pattern and load the detail version of the page.

This is a super simplistic version of a general point – if your "single-page" client-side app has multiple URL's, then you need to allow for the fact that somebody might use any of those URL's as their entry into the system.

From the router's view, the method we're going to write is extremely similar to the index page method – grab a view, populate it with the correct data, render it, put the rendered text into the page. We can't fully go "fast to green" here, because this is an end-to-end test, and the individual units aren't in place yet. We can, though, put in a quick implementation and come back to clean it up later.

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_10_1)

```
  tripDetail: function(id) {
    $container = $("#container");
    $container.append(this.topNavigationView.render().el);
    $container.append(this.sidebarView.render().el);
    var $content = $("<div/>").attr("id", "content");
    $container.append($content);
    var tripDetailView = new TimeTravel.Views.TripDetailView({
        model: TimeTravel.getTrip(id)});
    $content.append(tripDetailView.render().el);
```

```
    return $container;
  }
```

Sample 3-2-3: Quick and dirty implementation of detail page

Even the quickest glance at this method shows overwhelming similarities with the `index` method that draws the index page. There's a clear opportunity to refactor those two methods to remove duplication. But we should only refactor when the tests are passing.

With this code in place, the test now fails on the `TimeTravel.getTrip(id)` call, which is intended to retrieve the specific trip in question from the set of trips being maintained (or at least managed) by the global `TimeTravel` object. In more complex circumstances, we might need to go back to the server to retrieve this trip, but in this case, where we know we've already retrieved the trip to draw the index page and all we need to do is search that list of trips.

Still, lets test this out – the `getTrip` method may become more complex over time if we start using the server as a fallback if the data isn't already there.

Filename: spec/javascripts/timeTravelSpec.js (Branch: section_10_1)

```
describe("Global time travel app", function() {

  describe("finding trips", function() {
    it("finds a trip given an id", function() {
      TimeTravel.init(
          [{"id": "1", "name": "One"}, {"id": "2", "name": "Two"}]);
      expect(TimeTravel.getTrip("2").get("name")).toEqual("Two");
    });
  });

});
```

Sample 3-2-4: Test for gathering trip data

This test is pretty simple, since we don't really care about details of the trips beyond the fact that they are retrievable, we don't need full data on the trips, so we can use dummy data with realistic ids.

The implementation is similarly simple:

Filename: app/assets/javascripts/backbone/time_travel.js (Branch: section_10_1)

```
  getTrip: function(id) {
    return this.trips.find(function(trip) {
      return trip.get("id").toString() === id.toString();
    })
  }
```

Sample 3-2-5: The basic getTrip method

Though, again, this may become more complex over time.

At this point, we still fail, but now the message is `undefined is not a function`, referring, I suspect, to the fact that `TimeTravel.Views.TripDetailView` is being called, even though it doesn't exist. We can fix that with a boilerplate implementation.

Filename: app/assets/javascripts/backbone/views/tripDetailView.js (Branch: section_10_1)

```
TimeTravel.Views.TripDetailView = Backbone.View.extend({

  className: 'trip span-20',

  render: function() {
    $(this.el).html(TimeTravel.template('tripDetailViewTemplate').render(
        this.model.toJSON()));
    $(this.el).attr("id", "trip_detail_" + this.model.get("id"));
    return this;
  },

});
```

Sample 3-2-6: The boilerplate trip detail view

And an even boilerplate-er template:

Filename: app/assets/javascripts/backbone/templates/ tripDetailViewTemplate.mustache (Branch: section_10_1)

```
<h1>{{name}}</h1>

<div class="options" />
```

```
<div class="orders" />
```

Sample 3-2-7: That's one not-very-smart template

This is very much in keeping with other views we've seen. I'll just mention that the template is so logic free that we're even setting the DOM id in the JavaScript render method.

At this point, our tests actually pass – we asserted that the detail page would put up an element with a DOM id like `trip_detail_1`, and lo and behold, it does. It doesn't do much else, but it does that.

The next functional step is getting the associated hotel and tour options on the page. First, though, lets go through a refactor step on the router code.

Section 3.3

Cleanup, Aisle 5

When we last left our router code a couple pages back, we had some significant duplication between the `index` and `tripDetail` methods. In particular, the background layout for the top and side navigation was being duplicated.

Let's pull the layout into its own method and see where that leaves us:

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_10_2)

```
  container: function() {
    return $('#container');
  },

  content: function() {
    return $('#content');
  },

  pageHasContent: function() {
    return $.trim(this.container().html()) != "";
  },
```

```
layout: function() {
  if (this.pageHasContent()) {
    return;
  }
  this.container().append(this.topNavigationView.render().el);
  this.container().append(this.sidebarView.render().el);
  this.container().append($("<div/>").attr("id", "content"));
},

index: function() {
  var tripsView = new TimeTravel.Views.TripsView({
      collection: TimeTravel.trips});
  this.layout();
  this.content().html(tripsView.render().el);
  return this.container();
},

tripDetail: function(id) {
  var tripDetailView = new TimeTravel.Views.TripDetailView({
      model: TimeTravel.getTrip(id)});
  this.layout();
  this.content().html(tripDetailView.render().el);
  return this.container();
}
```

Sample 3-3-1: First pass at refactoring index and tripDetail

I've also added three convenience methods – one to get the outer `#container` element, one to get the inner `#content` element, and one to tell if the page has content for the purposes of knowing whether to redraw the layout. This allows us to skip having to continually redraw the top and side navigation if we don't need to.

Clearing out the noise in these methods makes another possible refactoring visible. These two methods have the same basic structure: create a view, render the layout, render the view, return the container. We can combine that functionality as follows:

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_10_2)

```
container: function() {
  return $('#container');
},

content: function() {
  return $('#content');
},

pageHasContent: function() {
  return $.trim(this.container().html()) != "";
},

layout: function() {
  if (this.pageHasContent()) {
    return;
  }
  this.container().append(this.topNavigationView.render().el);
  this.container().append(this.sidebarView.render().el);
  this.container().append($("<div/>").attr("id", "content"));
},

index: function() {
  var tripsView = new TimeTravel.Views.TripsView({
      collection: TimeTravel.trips});
  this.layout();
  this.content().html(tripsView.render().el);
  return this.container();
},

tripDetail: function(id) {
  var tripDetailView = new TimeTravel.Views.TripDetailView({
      model: TimeTravel.getTrip(id)});
  this.layout();
  this.content().html(tripDetailView.render().el);
  return this.container();
}
```

Sample 3-3-2: First pass at refactoring index and tripDetail

We'll leave it there for now, though if these methods get more complex, I'd probably start spinning them off into their own objects. At this point, though, more complexity in the router method seems unnecessary.

Section 3.4

Adding our Subordinate Object Data

At this point, we have a functional page, at least for very generous definitions of functional. We've got two more things to do, the first is that we need to display the hotel and extra options for the trip, and then we need to wire some actions together.

Really, the hard part of displaying the extra options is acquiring them in the first place – once we have the data in hand displaying it on the screen is very similar to the view structures we've already seen.

We have two basic options for gathering the extra data. The data can be included in our initial data download when the index page is first accessed, or we can make a return trip to the server to get the extra data as needed when we hit the detail page for a trip.

Adding the extras to the initial data fetch would require some digging into Rails associations and serialization, while getting the data as needed allows us to explore how Backbone interacts with a RESTful server. Since this is *Master Space and Time With JavaScript* and not *Master Space and Time With Rails Serialization Packages*, we'll go with the latter.

Adding Hotels and Extras to Backbone

The data associated with each trip comes from two database tables on the server, one for hotel options, and one for extras. There's not much difference in the structure of the two tables, but there's a significant difference in how they behave. A user can only pick one hotel option per booking, but can pick an arbitrary number of extras.[13]

13. I grant that's an oversimplification and that a real travel booking system would probably need to support multiple hotel options per trip. On the other hand, this is a booking system for *time travel*, so to paraphrase Joel Hodgson, "repeat to yourself it's just an example, I should really just relax"

Okay, that means we have two separate models, which in Backbone also implies two separate collection classes, like so:

Filename: app/assets/javascripts/backbone/models/hotel.js (Branch: section_10_3)

```
TimeTravel.Models.Hotel = Backbone.Model.extend({

});

TimeTravel.Collections.Hotels = Backbone.Collection.extend({
  model: TimeTravel.Models.Hotel,
  url: "/hotels"
});
```

Sample 3-4-1: Boilerplate hotel model and collection

And...

Filename: app/assets/javascripts/backbone/models/extra.js (Branch: section_10_3)

```
TimeTravel.Models.Extra = Backbone.Model.extend({

});

TimeTravel.Collections.Extras = Backbone.Collection.extend({
  model: TimeTravel.Models.Extra,
  url: "/extras"
});
```

Sample 3-4-2: Boilerplate hotel model and collection

The way that Backbone acquires data for a collection is via the `fetch` method. So, you'd create a new collection and then call `fetch`, something like this:

```
var aTrip = TimeTravel.getTrip(3);
var myHotels = new TimeTravel.Collections.Hotels(
                {trip: aTrip);
myHotels.fetch();
```

Sample 3-4-3:

When you call `fetch`, backbone uses the `url` property of the collection to drive a call to the server, which is assumed to be returning a JSON representation of the data. Backbone will

take that data, convert it to a set of Backbone model objects, and add all those objects into the collection.

But we have a problem with the code as written. As written (assuming that the server-side controller works as expected), the `fetch` method will return *all* the hotels, not just the ones associated with the trip. Luckily, we can set the `url` attribute to anything we want, we can even make it a function that is executed to give us a dynamic URL, based, say, on the value of a property of the associated trip.

This is actually testable logic:

Filename: spec/javascripts/hotelSpec.js (Branch: section_10_4)

```
describe("with a hotels collection", function() {

  it("correctly calculates URL ", function() {
    var trip = new TimeTravel.Models.Trip({id: 3});
    var hotels = new TimeTravel.Collections.Hotels([], {trip: trip});
    expect(hotels.url()).toEqual("/hotels?trip_id=3");
  });

});
```

Sample 3-4-4: Test for the URL to fetch hotel data

This test creates a trip model with a given ID, creates an empty hotel collection associated with that trip, and then expects the `url` property of the hotels collection to resolve to a URL containing the trip's id as a query parameter.[14]

We can make that pass by changing the `url` property to be a function, but there's one other thing we need to do to make the whole thing work.

Filename: app/assets/javascripts/backbone/models/hotel.js (Branch: section_10_4)

```
TimeTravel.Collections.Hotels = Backbone.Collection.extend({
  model: TimeTravel.Models.Hotel,

  initialize: function(models, options) {
```

14. Actually, on the Rails side, you could also make it a nested resource subordinate to `Trip`, which would have a slightly different URL. Not doing that here because this is still not *Master Space and Time With Rails Nested Routes*.

```
    if (options && options.trip) {
      this.trip = options.trip
    }
  },

  url: function() {
    return "/hotels?trip_id=" + this.trip.get("id");
  },
});
```

Sample 3-4-5: Generating the URL for hotels

When we created the new Hotels collection object in the test `new TimeTravel.Collections.Hotels([], {trip: trip});`, the second argument is technically a set of options. By default, Backbone doesn't do anything with the options, so if we want to use them, we need to write our own `initialize` function. Inside that `intialize` function we can do whatever the heck we want so we make the bold choice to hold on to the provided trip object by making it a property of the `Hotels` collection. With the trip in place, we can use the trip's `id` as part of the `url` function, returning exactly the URL you'd expect a server to respond to.

We need to do the same thing to the `Extras` collection, only the details of the URL change: (and yes, there's an associated test... I'll leave that one to your imagination.)

Filename: app/assets/javascripts/backbone/models/extra.js (Branch: section_10_4)

```
TimeTravel.Collections.Extras = Backbone.Collection.extend({
  model: TimeTravel.Models.Extra,

  initialize: function(models, options) {
    if (options && options.trip) {
      this.trip = options.trip
    }
  },

  url: function() {
    return "/extras?trip_id=" + this.trip.get("id");
  },

});
```

Sample 3-4-6: Generating the URL for extras

For completeness sake, here's the server side code that makes that work – even though this book is not *Master Space and Time Through Driving Jokes Into The Ground*. Those of you who are following along and not using the git repo, here you go:

Filename: app/controllers/hotels_controller.rb (Branch: section_10_4)

```
class HotelsController < ApplicationController

  respond_to :html, :json

  def index
    @hotels = Hotel.where(:trip_id => params[:trip_id]).all
    render :json => @hotels
  end

end
```

Sample 3-4-7: Hotel controller code

Filename: app/controllers/extras_controller.rb (Branch: section_10_4)

```
class ExtrasController < ApplicationController

  respond_to :html, :json

  def index
    @extras = Extra.where(:trip_id => params[:trip_id]).all
    render :json => @hotels
  end

end
```

Sample 3-4-8: Extra controller code

You also need to add the resources to the Rails `routes.rb` file:

```
resources :hotels
resources :extras
```

Sample 3-4-9:

Now that we know that we can grab the hotel and extra data for a given trip, we need to make sure that we actually do so. Because Backbone is not particularly opinionated[15], we have a few options as to how to structure the data.

- The router could ask for the hotel and extra collections to fetch themselves when the detail route is being processed.
- The trip could, on request, fetch the two collections and provide them to any views that need them.
- We're going to need some kind of hotel and extra collection-based views. Those views could grab their data when they are asked to render themselves.

The middle option arguably makes the most sense as a data model – the extra data belongs to the trip. That said, I've done a version of this code that basically used the last option on the grounds that the trip model doesn't necessarily reference the extra data and it's maximally lazy to wait until the view needs it to get the data. For now, we'll stick to the coherent data model and make sure that the trip model can load it's associated data.

One possibility for testing this logic looks like this:

Filename: spec/javascripts/tripSpec.js (Branch: section_10_4)

```
it("loads extra data", function() {
  this.trip = new TimeTravel.Models.Trip(this.tripData);
  spyOn(this.trip.hotels, "fetch").andReturn(null);
  spyOn(this.trip.extras, "fetch").andReturn(null);
  this.trip.fetchData();
  expect(this.trip.hotels.fetch).toHaveBeenCalled();
  expect(this.trip.extras.fetch).toHaveBeenCalled();
});
```

Sample 3-4-10: Can the trip fetch its data?

In this test, we're assuming that the initializer method of `Trip` will create empty collections for `hotels` and `extras`. If those collections exist, then it's easier to spy on them to check to see if

15. It only just now occurs to me that there's irony in the idea that a framework named Backbone doesn't have strong opinions.

`fetch` is invoked when requested. (Creating the `hotels` and `extras` collection in the `fetchData` method would be harder to test).

That test fails, initially because the `trip.hotels` property doesn't exist to be spied on. We can fix that be creating the properties in the initializer, and then that `fetchData` method becomes straightforward:

Filename: app/assets/javascripts/backbone/models/trip.js (Branch: section_10_4)

```
initialize: function(attributes) {
  this.set({startMoment: moment(attributes.start_date),
      endMoment: moment(attributes.end_date)});
  this.hotels = new TimeTravel.Collections.Hotels([], {trip: this});
  this.extras = new TimeTravel.Collections.Extras([], {trip: this});
},

fetchData: function() {
  this.hotels.fetch();
  this.extras.fetch();
},
```

Sample 3-4-11: Adding hotel and extras, and populating them

If we wanted to get really fancy we could check to see if the hotel and extras are filled before populating them. Alternately, we could cache the data locally in case we come back to the same detail page later. We're not going to be that fancy for the moment.

We also need to make sure that the trip in question fetches its data before the detail page is rendered. It seems like the logical place to do that is among the data gathering in the router:

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_10_4)

```
tripDetail: function(id) {
  var trip = TimeTravel.getTrip(id);
  trip.fetchData();
  this.basicPage(new TimeTravel.Views.TripDetailView({model: trip}));
}
```

Sample 3-4-12: Updating detail page method to fetch data

I don't have a test for this on the grounds that a) I'm not really adding untested logic at this point, b) there's no real design decision to be made, c) setting up the test is probably a little awkward at this point, and d) later on, we may add an acceptance test for the hotel display that would fail if the fetch doesn't happen.

Speaking of the hotel display, we've got the data, lets put it on screen.

Displaying the Hotel and Extras

In order to get our newly acquired data to display, we need to create some more views. Our `tripDetailView` template, which is drawing the individual trip page, is currently split into two `div` tags, one for "options" and one for "orders". I don't think we need a specific options view, we'll just create a view for the hotels and a view for the extras and render them into the options element.

We're going to start with an acceptance-style test that delineates the goal – that drawing the trip detail view renders hotel data. Again, we're working "outside-in" – this is a (somewhat limited) end-to-end test of the entire trip detail view, then we'll unit test any logic involved in the actual hotels view and template.

Filename: spec/javascripts/tripDetailViewSpec.js (Branch: section_10_5)

```
describe("with a trip detail view", function() {
  beforeEach(function() {
    this.trip = new TimeTravel.Models.Trip({"description":"A cruise",
      "end_date":"1620-11-21", "id":13, "image_name":"mayflower.jpg",
      "name":"Mayflower Luxury Cruise", "price": 1204.0,
      "start_date":"1620-05-17",
      "tag_line":"Enjoy The Cruise That Started It All"});
    this.tripDetailView = new TimeTravel.Views.TripDetailView(
        {model: this.trip});
  });

  describe("acceptance tests", function() {
    it("displays hotels", function() {
      this.trip.hotels = [
        new TimeTravel.Models.Hotel({"name": "Lux"}),
        new TimeTravel.Models.Hotel({"name": "Third class"})
      ];
```

```
      this.tripDetailView.render();
      expect(this.tripDetailView.$el.find(".hotel").length).toEqual(2);
    });
  });
});
```

Sample 3-4-13: Acceptance test for hotels in detail page

Similar to other view tests we've written, we're creating a test view, rendering it and evaluating output. The test, obviously, fails.

Now, lets work upwards to get there. We need a hotel view object, and we need the glue to get it attached to the existing `TripDetailView`. There's no real reason to do one before the other, let's do the glue first. Let's create a fully isolated `hotelRender` method inside our `TripDetailView` – by isolated I mean that it takes a renderer as an option, rather than creating one itself, which allows us a seam to place our own dummy renderer for testing purposes:

Filename: spec/javascripts/tripDetailViewSpec.js (Branch: section_10_6)

```
  describe("with a hotel", function() {
    it("renders a hotel", function() {
      var mockHotelView = {render: null}
      spyOn(mockHotelView, "render").andReturn({el: $("<div>test</div>")});
      this.tripDetailView.$el = $("<div><div class='options' /></div>");
      this.tripDetailView.hotelRender(mockHotelView);
      expect(mockHotelView.render).toHaveBeenCalled();
      expect(this.tripDetailView.$el.find(".options")).toHaveText("test");
    });
  });
```

Sample 3-4-14: Mock object test testing API

What the hell does that test do? This is a fairly strict isolated test that uses mock objects to isolate the method under test – in this case `hotelRender` from the rest of the world. (In my semi-occasional role as test museum curator for this book, I thought it was time to throw another test type at you.)

Instead of using a real `HotelView`, we are just going to use a pretend one, specifically a plain ordinary JavaScript literal that only defines a dummy `render` property.[16]. This is about as as simple as mock packages get, though if you want an actual mock library in JavaScript, check

out Sinon.js at http://sinonjs.org/, in the interest of not explaining *another* framework, I'll pass on including Sinon at this point.

Anyway, once we declare the trivial object, we immediately start spying on it like some kind of demented James Bond, replacing the render property with a spy that returns some fake HTML text that we can test for later. We also seed our actual `TripDetailView` with just enough HTML so that the `hotelRender` method has something to grab on to. We then call the `hotelRender` method with our mock as an argument.

Once the `hotelRender` method is called, we have two things that we want to make sure happened. The hotel view's `render` method needs to have been called, and the result of that method needs to be placed in the `.options` element of the parent `TripDetailView`.

Which brings us to some straightforward test-passing code, that pretty much just does those two things: calls `hotelRender` and uses the result.

Filename: app/assets/javascripts/backbone/views/tripDetailView.js (Branch: section_10_6)

```
TimeTravel.Views.TripDetailView = Backbone.View.extend({

  className: 'trip span-20',

  render: function() {
    this.basicRender();
    this.hotelRender(this.hotelsView());
    return this;
  },

  basicRender: function() {
    this.$el.html(TimeTravel.template('tripDetailViewTemplate').render(
        this.model.toJSON()));
    this.$el.attr("id", "trip_detail_" + this.model.get("id"));
  },

  hotelsView: function() {
```

16. Those of you familiar with mock object tools in other languages may be surprised that our little view object defines a `render` property – for better or worse, Jasmine spies raise an error if you attempt to spy on a property that does not exists.

```
    return new TimeTravel.Views.HotelsView({trip: this.model});
  },

  hotelRender: function(hotelsView) {
    this.$el.find(".options").append(hotelsView.render().el);
  }

});
```

Sample 3-4-15: The TripDetailView, with a passing hotelRender method

If you are following along super-strictly, you'll notice that a couple of other tests fail because `HotelsView` hasn't actually been defined yet – we'll get there.

But that brings up an important point – the test we've been working on passes, even though `HotelsView` hasn't been defined yet. That seems like a flaw. Actually, though, it's pretty neat. At least right up to the point where it turns out to be a flaw.

What's neat about the mock testing in this case is that we are limiting our test to exactly what this particular object and method is responsible for, no more and no less. This method is responsible for calling the `render` method on a hotel view and using that result. Whether that result is itself correct – even whether that result exists – is Somebody Else's Problem, or at least Some Other Test's Problem. Which is fine, because we're going to write Some Other Test in just a moment.

One benefit of this kind of mock testing style is that when a logic change breaks a test, it tends to break only one test, making it much easier to track the problem down. Also, mock testing tends to be very sensitive to the quality of isolation. What I mean is that if the code being tested has a lot of dependencies, a mock test will expose that quickly – you'll need to write a lot of mocks to make the test work. The proper reaction to this situation is not "mock testing sucks" but rather "maybe I need to restructure the code".

The down side, of course, is that it becomes possible for the tests to pass even though the code doesn't work end-to-end. Typically, if you are using a mock style of testing, you augment the mock tests with at least one end-to-end acceptance test that at the very minimum acts as a smoke test to ensure that all the parts of the code base play nicely together. In our case, the tests that are currently failing because the hotels view doesn't exist yet will probably serve, though you could also add a line of hotel display to the acceptance test in `homePageSpec.js`.

At the moment, though, we have a HotelsView to create. Lets start with a boilerplate view. Whether this needs to be tested or not basically depends on whether you consider this level of boilerplate to be a *de facto* part of the framework or not. We're going to have more interesting tests in a paragraph or two, so I'm fine to leave it right now.

Filename: app/assets/javascripts/backbone/views/hotelsView.js (Branch: section_10_6)

```
TimeTravel.Views.HotelsView = Backbone.View.extend({
  tagName:'section',
  className:'hotels span-20',

  initialize: function(options) {
    this.trip = this.options.trip;
    this.collection = this.trip.hotels;
    _.bindAll(this, 'render');
    this.collection.bind('reset', this.render, this);
  },

  //##render
  render: function() {
    this.$el.html(TimeTravel.template('hotelsViewTemplate').render());
    this.renderLengthOptions();
    return this;
  },

  renderLengthOptions: function() {
    var $lengthOption = this.$el.find("#length-select");
    for (i = 0; i <= this.trip.lengthInDays(); i++) {
      $lengthOption.append($("<option>").text(i));
    }
  }
  //##render
});
```

Sample 3-4-16: Boilerplate hotel view

We also need a template. Here is one:

Filename: app/assets/javascripts/backbone/templates/hotelsViewTemplate.mustache (Branch: section_10_6)

```
<h3>Hotels</h3>

<div>
  Number of Nights:
  <select id="length-select" />
</div>

<div id="hotels" />
```

Sample 3-4-17: Hotels template

Nothing here we haven't seen before, but it does have some gaps. Specifically, we've got a pulldown that will allow the user to choose the length of their stay, and we also have to include the actual information about each hotel.

Let's do the length of stay pulldown. We'll want to have one `option` tag for each possible length of stay. That data is stored in the `Trip` instance, so calculating the length should come from there. Test first:

Filename: spec/javascripts/tripSpec.js (Branch: section_10_6)

```
  it("knows how many days it is", function() {
    trip = new TimeTravel.Models.Trip(
        {start_date: "2012-11-11", end_date: "2012-11-20"});
    expect(trip.lengthInDays()).toEqual(9);
  });
```

Sample 3-4-18: Test for how many days long a trip is

And quick passing code, taking advantage of the fact that we've already parsed the string dates into JavaScript moment objects:

Filename: app/assets/javascripts/backbone/models/trip.js (Branch: section_10_6)

```
  lengthInDays: function() {
    var start = this.get("startMoment");
    var end = this.get("endMoment")
    return (end - start) / (1000*60*60*24);
```

```
  }
```

Sample 3-4-19: Determining length of a trip in days

With the model in place, we need to incorporate this into the view. Again, we start with a test.

Filename: spec/javascripts/hotelsViewSpec.js (Branch: section_10_6)

```
describe("with hotels view spec", function() {

  it("adds options for length of days", function() {
    var trip = new TimeTravel.Models.Trip({});
    var view = new TimeTravel.Views.HotelsView({trip: trip});
    spyOn(trip, "lengthInDays").andReturn("2");
    view.render();
    var options = view.$el.find("#length-select option");
    values = _.map(options, function(opt) { return $(opt).text() });
    expect(values).toEqual(['0', '1', '2']);
  });

});
```

Sample 3-4-20: A test to see if the option elements are created

This tests uses Jasmine spies to fake the trip length rather than force us to generate a start date and end date for the trip – I'd use a bare object in place of a real `Trip` here, except that Jasmine complains if you spy on a method that doesn't exist. In any case, we render the view and check the resulting `$el` element for evidence of the new `option` tags. Note that we need to explicitly find the `option` tags inside the `$el` rather than inside the DOM document – since we aren't actually rendering a page, there's nothing that inserts the rendered hotels view into the Jasmine spec runner's DOM.

The code that makes this pass goes in the `HotelsView` object:

Filename: app/assets/javascripts/backbone/views/hotelsView.js (Branch: section_10_6)

```
  render: function() {
    this.$el.html(TimeTravel.template('hotelsViewTemplate').render());
    this.renderLengthOptions();
    return this;
```

```
},

renderLengthOptions: function() {
  var $lengthOption = this.$el.find("#length-select");
  for (i = 0; i <= this.trip.lengthInDays(); i++) {
    $lengthOption.append($("<option>").text(i));
  }
}
```

Sample 3-4-21:

We've written a subordinate method `renderLengthOptions` that takes the length in days from the trip and simply inserts an `option` tag into the `#length-select` element, which we know is the `select` tag.[17]

Okay, we've got the length of stay select in place – don't worry, we'll add some event handling to it before we break out of this chapter. Now lets get the hotels in the display.

Each individual hotel doesn't have much data, they are pretty much just a name, a description, and a price. We have at least three options for display:

1. We can go the full backbone route of making a new view object and template for individual hotels.

2. At the other end of the spectrum, we can include a loop inside the `HotelsView` mustache template and loop inside the template.

3. In between, we can add the text for each hotel into the `HotelsView` in plain JavaScript without creating a dedicated view object.

I'm going to pick the first option, and not just because adding extra code samples pads the length of the book. Also not just because this is a book about Backbone. But also because keeping the hotel display data in the Backbone family, so to speak, will make it easier to integrate the view with the Backbone event model if and when we want the view to respond to events.

17. Strictly speaking, by specifying the DOM ID here in the render code, we probably have a tighter dependency between the code and the DOM than we need. You might want to break out the selector into some kind of `findLengthSelector` method.

There isn't a whole lot of unique logic here, just some structure, so I'm not feeling a strong need for a test (also, the hotel data showing up is covered by the original acceptance test, which is still failing at this point.)

The hotel view should look very familiar at this point:

Filename: app/assets/javascripts/backbone/views/hotelView.js (Branch: section_10_7)

```
TimeTravel.Views.HotelView = Backbone.View.extend({

  className: 'hotel',

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

});
```

Sample 3-4-22: Initial view for individual hotel data

With a related template:

Filename: app/assets/javascripts/backbone/templates/hotelViewTemplate.mustache (Branch: section_10_7)

```
<div class="hotel-name">{{name}}</div>
<div class="hotel-description">
  {{description}}
</div>
<div class="hotel-price">
  ${{price}}.00
</div>
```

Sample 3-4-23: The associated template for hotel data

And integration into the parent collection view:

Filename: app/assets/javascripts/backbone/views/hotelsView.js (Branch: section_10_7)

```
renderHotels: function() {
  var $hotels = this.$el.find("#hotels");
  this.collection.each(function(hotel) {
    var view = new TimeTravel.Views.HotelView(
          {model: hotel, collection: this.collection});
    $hotels.append(view.render().el);
  })
  return this;
}
```

Sample 3-4-24: Rendering each hotel inside the collection view

Finally, that `renderHotels()` method needs to be called from the `HotelsView` `render` method:

Filename: app/assets/javascripts/backbone/views/hotelsView.js (Branch: section_10_7)

```
render: function() {
  this.$el.html(TimeTravel.template('hotelsViewTemplate').render());
  this.renderLengthOptions();
  this.renderHotels();
  return this;
},
```

Sample 3-4-25: Rendering each hotel

That's quite a chunk of code. Worse, it's quite a chunk of what is very boilerplate code. Worse, we're going to have to do much the same thing to get the other tour options displayed.

So, let's talk about that for a second while I decide whether to put in all the code to display the events. Spoiler alert: I'm going to put it in. It's not like I'm worried about wasting paper, and inevitably somebody gets upset if I leave out coding steps.

What we're seeing here is a side effect of how Backbone is designed – it's meant to be a non-opionionated framework that offers support for an MVC style structure, but then allows you to do whatever you want on top of that structure. However, if you wind up doing the same thing a lot of times, then Backbone doesn't give you default support for handling that repitition, you have to build it in yourself.

In our case, we might start building our own parent classes for views, something like `TemplateView` to encapsulate the common structure we have of rendering a template and inserting the result into the view's element. Or possibly a `CollectionView` to handle the case of a view iterating over its members. I'm not going to spend time right now refactoring to a common pattern.

There's already a variety of community involvment in creating extensions for Backbone. Try https://github.com/documentcloud/backbone/wiki/Extensions,-Plugins,-Resources as a first place to look.

And here's what we need to handle the other trip-related extras. First off, the `TripDetailView` `render` method needs to call into our extras:

Filename: app/assets/javascripts/backbone/views/tripDetailView.js (Branch: section_10_7)

```
render: function() {
  this.basicRender();
  this.hotelRender(this.hotelsView());
  this.extrasRender(this.extrasView());
  return this;
},
```

Sample 3-4-26: The render method, now with a call to extrasRender

And a simple one-liner for the `extrasRender` method.

Filename: app/assets/javascripts/backbone/views/tripDetailView.js (Branch: section_10_7)

```
extrasView: function() {
  return new TimeTravel.Views.ExtrasView({trip: this.model});
},
```

Sample 3-4-27:

We have an extras view that is basically identical to `HotelsView`.

Filename: app/assets/javascripts/backbone/views/extrasView.js (Branch: section_10_7)

```
TimeTravel.Views.ExtrasView = Backbone.View.extend({
  tagName:'section',
  className:'extras span-20',

  initialize: function() {
    this.trip = this.options.trip;
    this.collection = this.trip.extras;
    _.bindAll(this, 'render');
    this.collection.bind('reset', this.render, this);
  },

  render: function() {
    this.$el.html(TimeTravel.template('extrasViewTemplate').render());
    this.renderExtras();
    return this;
  },

  renderExtras: function() {
    var $extras = this.$el.find("#extras");
    this.collection.each(function(extra) {
      var view = new TimeTravel.Views.ExtraView(
            {model: extra, collection: this.collection});
      $extras.append(view.render().el);
    })
    return this;
  }

});
```

Sample 3-4-28: The ExtrasView class

With a minimal template:

Filename: app/assets/javascripts/backbone/templates/extrasViewTemplate.mustache (Branch: section_10_7)

```
<h3>Extras</h3>
<div id="extras" />
```

Sample 3-4-29: The ExtrasView template

Now we need the view for an individual extra:

Filename: app/assets/javascripts/backbone/views/extraView.js (Branch: section_10_7)

```
TimeTravel.Views.ExtraView = Backbone.View.extend({

  className: 'extra',

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

});
```

Sample 3-4-30: The ExtraView class

And finally, the resulting template.

Filename: app/assets/javascripts/backbone/templates/extraViewTemplate.mustache (Branch: section_10_7)

```
<div class="extra-name">{{name}}</div>
<div class="extra-description">
  {{description}}
</div>
<div class="extra-price">
  ${{price}}.00
</div>
```

Sample 3-4-31: The ExtraView template

I realize that at least some of you are saying that's a lot of code for drawing, like, six lines of text on the screen. I don't disagree, but would like to point out that I'm being a little over-formal here for instructional purposes – in practice, I might start by loop over the hotel and extra data in the collection view rather than immediately create a separate individual view class. (I could always create a separate class later if needed) Also, as mentioned above, some judiciously created common parent classes would remove a lot of this code – it's worth taking a stab at if you want to play around. Finally, some of the additional class structure will come to play as we add events.

Which leads us to the next part of our show – adding the events to this page that will let our users actually place trip reservations.

Section 3.5

Creating Orders out of Chaos

Let's take a deep breath and examine our progress. We want our users to be able to select a trip, choose various options relating to that trip, and send the order back to the server. Thus far, after what must seem like a long hike through dark woods, we've been able to display the various options on the screen.

Now comes the part where we do things to the various parts of the trip, and display current data about what we've done.

The requirements are as follows:

- The user must select exactly one hotel option and a length of stay.
- The price of the hotel is simply calculated as the product of the hotel price and the length of stay. We'll leave the complications of cross-temporal sales tax for another day.
- The user may select more than one of the remaining extras. Their prices are just added to the hotel price.

Obviously, this is simplified, we're not worried about how many people might be going on our time travel adventure, we're not worried about other incidental charges or anything.

The easiest part of this is probably just the selection of a trip extra, we'll start there.

Section 3.6

I'll Have A Ham On Five Hold The Mayo

First off, we need an order model:

Filename: app/assets/javascripts/backbone/models/order.js (Branch: section_10_7)

```
TimeTravel.Models.Order = Backbone.Model.extend({

});
```

Sample 3-6-1: First order model. See, not everything is a ton of boilerplate

That was easy. (We aren't going to be dealing with groups of orders at this point, so we don't need a collection class.) Next, we need it to do something. I have a crazy idea. Let's have it be able to add and remove trip extras from a list. I know, it sounds wacky, but humor me.

Test first:

Filename: spec/javascripts/orderSpec.js (Branch: section_10_8)

```
describe("with an order", function() {

  beforeEach(function() {
    this.order = new TimeTravel.Models.Order();
  });

  it("can add an extra", function() {
    extra = new TimeTravel.Models.Extra();
    this.order.addExtra(extra);
    expect(this.order.get("extras").toArray()).toEqual([extra]);
  });

  it("can remove an extra", function() {
    extra = new TimeTravel.Models.Extra();
    this.order.addExtra(extra);
    this.order.removeExtra(extra);
    expect(this.order.get("extras").toArray()).toEqual([]);
  });

  it("can toggle an extra", function() {
    extra = new TimeTravel.Models.Extra();
    this.order.toggleExtra(extra);
    expect(this.order.get("extras").toArray()).toEqual([extra]);
    this.order.removeExtra(extra);
    expect(this.order.get("extras").toArray()).toEqual([]);
  });
```

```
...


});
```

Sample 3-6-2: Add and remove extra tests

Pretty basic, we're creating one of our shiny new order models, and adding and removing an extra to prove it can be done. We've added a toggle method that combines adding and removing for reasons that might become clear later.

We make that pass with an initializer and some functional methods:

Filename: app/assets/javascripts/backbone/models/order.js (Branch: section_10_8)

```
  initialize: function() {
    if(!this.get("extras")) {
      this.set("extras", new TimeTravel.Collections.Extras())
    }
  },

  addExtra: function(extra) {
    this.get("extras").add(extra);
  },

  removeExtra: function(extra) {
    this.get("extras").remove(extra);
  },

  hasExtra: function(extra) {
    return this.get("extras").contains(extra);
  },

  toggleExtra: function(extra) {
    if(this.hasExtra(extra)) {
      this.removeExtra(extra);
    } else {
      this.addExtra(extra);
```

```
    }
  },
```

Sample 3-6-3: The order model now handles extras

Our `intialize` method is setting the `extras` attribute of the order, it should either get an `extras` attribute when created (as in `new TimeTravel.Models.Order({extras: extrasCollection});`) or it will create an empty one. Then the `addExtra` and `removeExtra` methods just wrap the existing Backbone collection functionality.

We also want to calculate the price, or at least start to calculate the price. Price is an attribute of each extra, and we want to add them together.

Filename: spec/javascripts/orderSpec.js (Branch: section_10_8)

```
describe("it can calculate price", function() {
  beforeEach(function() {
    this.cheapExtra = new TimeTravel.Models.Extra({price: 100});
    this.pricyExtra = new TimeTravel.Models.Extra({price: 500});
    this.order = new TimeTravel.Models.Order();
  });

  it("initializes price", function() {
    this.order.calculatePrice();
    expect(this.order.get("price")).toEqual(0);
  });

  it("calculates price for one extra", function() {
    this.order.addExtra(this.cheapExtra);
    this.order.calculatePrice();
    expect(this.order.get("price")).toEqual(100);
  });

  it("calculates price for multiple", function() {
    this.order.addExtra(this.cheapExtra);
    this.order.addExtra(this.pricyExtra);
    this.order.calculatePrice();
    expect(this.order.get("price")).toEqual(600);
  });
```

```
});
```

Sample 3-6-4: Add and remove extra tests

Simple enough, we're testing the default case, and then we're testing the single item and multiple item cases. If you are doing a super-strict TDD, then a test asking you for the price of a single item just drives you to return the price of that item, it's only when you add a second element that you are driven to do a sum. We're also testing that the `calculatePrice` actually adds the price to the attribute set of the order object, rather than having price remain a standalone method.

I want to make one point about the test, which is the difference between writing the last line of the test as `expect(this.order.calculatePrice()).toEqual(600);` and writing it as `expect(this.order.calculatePrice()).toEqual( this.cheapTest.get("price") + this.pricyTest.get("price"));`. I generally prefer the first form because I think it's clearer, and in this case the necessary math is easy to follow. However, some people don't like having magic literals in their code and might prefer the second form. I'm a little leery of any time where I write a test where the test code seems very similar to the actual code-code. Makes me nervous that I could have a tautology and never realize it.

And the resulting method:

Filename: app/assets/javascripts/backbone/models/order.js (Branch: section_10_8)

```
calculatePrice: function() {
  var prices = this.get("extras").pluck("price");
  var price = _.reduce(prices,
      function(agg, item) { return agg + item },
      0);
  this.set({price: price})
  return price
}
```

Sample 3-6-5: The order model now handles extras

We're using two features of the Backbone/Underscore collection API. The `pluck` method on a collection takes an attribute name as an argument and returns a new array with the value of that attribute for each item in the list – it's a simplified version of `map` for a common use case.

If there are no prices, we exit with a default value. If there are prices, we use the Underscore `reduce` method to sum up the prices.

If you aren't familiar with `reduce` as a concept (also called `inject` in some languages), you should get familiar with it, because it's kind of useful. The `reduce` funcion is designed to convert an array into a single scalar value. I'll sum (no pun intended) it up in a paragraph, because this isn't *Master Space and Time With Enumeration Functions*. The `reduce` method takes a functional argument. That functional argument itself normally takes two arguments[18], often referred to as the *aggregator* and the *value*. When `reduce` executes, it executes its functional argument using the first two values of the array as arguments to that function. The result of the functional argument then becomes the first argument to the next iteration of the function, with the second argument coming from the next element of the array. And so on, with a running total value building in the first argument, and each successive element of the array taking a turn as the second. By the end, you've aggregated the array values into a scalar – in our case, by adding successive values to calculate the overall sum of all elements in the array. That's just the bare minimum of the kind of craziness you can get up to with `reduce`, however.

Enough advertisements for cool enumeration functions. The point for us is that now our price tests pass and we can actually take steps toward displaying the order on screen.

By this time, I suspect we all know the basic drill. We need a view class. Once again, we need to figure out what to do with a list of data, namely the set of selected extras. My fingers are threatening active rebellion at the thought of typing an entire set of view classes for what would turn out to be a single line template[19], so we'll try and do it in JavaScript. Test first:

Filename: spec/javascripts/orderViewSpec.js (Branch: section_10_8)

```
describe("with an order view", function() {

  describe("viewing extras", function() {

    beforeEach(function() {
      this.order = new TimeTravel.Models.Order();
      this.orderView = new TimeTravel.Views.OrderView({model: this.order});
    });
```

18. The Underscore version takes other arguments that we're not going to worry our pretty little heads about right now.

19. You'd think they'd have already rebelled at the thought of writing an entire book, but apparently not.

```
    it("can create a DOM element for an extra", function() {
      var extra = new TimeTravel.Models.Extra({name: "Extra", price: 100});
      var result = this.orderView.renderExtra(extra);
      expect(result.text()).toEqual("Extra: $100.00");
    });

    it("invokes render extra", function() {
      var extra = new TimeTravel.Models.Extra({name: "Extra", price: 100});
      this.order.addExtra(extra);
      var result = this.orderView.render();
      expect(result.$el.find("#ordered_extras").text()).toEqual("Extra: $100.00");
    });

  });

});
```

Sample 3-6-6: Our specs for printing extras as part of the order.

Two tests here, the first is more of a unit test, the second is more of an integration test. In the first, we're verifying that the `OrderView` can take a single extra object and convert it to a DOM element. In the second, we're verifying that it can take an entire order's worth of extras and convert them as part of a normal render – note that even though each test ends on the same assertion, the second test runs through a higher level function. Just as an aside, the technique of separating the body of a loop from the loop itself is a frequently useful TDD plan.

Anyway, code to make this test pass:

Filename: app/assets/javascripts/backbone/views/orderView.js (Branch: section_10_8)

```
TimeTravel.Views.OrderView = Backbone.View.extend({
  className: 'order span-8',

  render: function() {
    this.model.calculatePrice();
    this.$el.html(TimeTravel.template('orderViewTemplate').render(
        this.model.toJSON()));
    this.renderExtras();
```

```
    return this;
  },

  renderExtras: function() {
    self = this;
    this.model.get("extras").each(function(extra) {
      self.$el.find("#ordered_extras").append(self.renderExtra(extra));
    });
  },

  renderExtra: function(extra) {
    return $("<div>").text(
        extra.get("name") + ": $" + extra.get("price") + ".00");
  }

});
```

Sample 3-6-7: And the passing order view

I want to point out the call to `calculatePrice`, which is setting the price element of the order, in effect, it's telling the order to present itself. That may not be the final resting place of that `calculatePrice` call, but for now it's working. Also, I've got a call to `renderExtras` in the main `render` method, that ties the extras into the order display, but the fact that it's included as part of the overall render is not technically under test. Inside `renderExtras`, we're locating the view element to place our extra in using `self.$el.find("#ordered_extras")` rather than just `$("#ordered_extras")` because the `$el` element of this view may not have been placed in the larger page yet, and so we can't guarantee that an ordinary jQuery selector would find it.

Finally, to make this work we need to add it to the `TripDetailView` that controls the entire page. I'm actually going to back up a step and create the order at the router level. Conceptually, this page represents the merging of a trip and a potential order.

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_10_8)

```
  tripDetail: function(id) {
    var trip = TimeTravel.getTrip(id);
    var order = new TimeTravel.Models.Order({trip: this.model});
    trip.fetchData();
    this.basicPage(new TimeTravel.Views.TripDetailView({model: trip, order: order}));
```

```
}
```

Sample 3-6-8: Adding the order object to the page

Filename: app/assets/javascripts/backbone/views/tripDetailView.js (Branch: section_10_8)

```
initialize: function(options) {
  this.order = options.order || new TimeTravel.Models.Order({trip: this.model});
},

render: function() {
  this.basicRender();
  this.hotelRender(this.hotelsView());
  this.extrasRender(this.extrasView());
  this.ordersRender(this.ordersView());
  return this;
},
```

Sample 3-6-9: Adding the order render to the trip detail view

Filename: app/assets/javascripts/backbone/views/tripDetailView.js (Branch: section_10_8)

```
ordersView: function() {
  return new TimeTravel.Views.OrderView({model: this.order});
},

ordersRender: function(orderView) {
  this.$el.find(".orders").append(orderView.render().el);
},
```

Sample 3-6-10: Adding order view info to the trip detail view

It's probably worth it to add some of this order information to one of the existing `TripDetailView` tests, though frankly, we'll probably cover it in a later test. Also adding it exactly the kind of high setup cost, low design value, failure would be spectacularly visible anyway kind of test that gives TDD a bad name. Eventually, we're also going to have to make this order object available to some of the sub views.

Wiring it together

And so, we gaze upon our web page and we see that it is kind of goodish. All the pieces are there on the page, but they are inert and don't relate to each other. We need to add some events.

The underlying flow of Backbone events is:

1. The event happens.
2. Data changes as a result of the event
3. Views observe the event and update the screen.

It's tempting to want to update the view directly from the event.[20] It seems so easy and direct. I mean, we're already in the view, what's the harm of entangling our view and our data just a little teeny bit, just this once?

Like other code quality issues related to coupling, the problems with intertwining Backbone views and models may only become apparent later in a project when it's harder to fix the problems because everything is so coupled. But here's a quick way of thinking about the issue. Think about a piece of state in our system, for example, whether a particular trip extra is being purchased or not.

The information relating that state is effectively in our system at least twice – the order model tracks it to manage price calculations, and the view layer encodes it in the DOM so that the user can see a difference between selected and unselected options.[21]

The important part, though, is that even if the information is encoded different ways in various parts of the system, is that there is one, and only one, source of truth for the state. Which is the Don't Repeat Yourself (DRY) principle as applied to data. Either the model is the truth and the view draws itself based on the model, or (less likely) the view is the source of truth and the model makes calculations based on the view.

20. It is so tempting that the workshop exercise I based this chapter on actually does it, and I only just now realized it.

21. Technically, you could only have one representation by storing everything in the DOM. Good luck with that, let me know how it goes.

If they are both the source of truth at the same time, then changes need to always be made in both places, dramatically increasing the possibility of an error. If there's only one source of truth, though, it's easier to manage the individual pieces, an easier to test them because the data and view are less tightly coupled.

Okay, that's the theory. In practice we have the following workflow:

1. The user clicks on an trip extra.
2. The state of the extra is toggled in the order module.
3. The view recognizes the data change and re-draws the extras.

Recognizing the click is an easy first step, all we need to do is associate a click anywhere in the extra's view with a function. We've seen how to do this, we set an `events` object in our `ExtraView`:

Filename: app/assets/javascripts/backbone/views/extraView.js (Branch: section_10_9)

```
events: {
  "click": 'selectMe'
},
```

Sample 3-6-11: The events object suggesting that the event needs to be toggled

What needs to happen in that `selectMe` method? The view needs to inform the order that the extra has been changed. Perhaps a test would clear that up:

Filename: spec/javascripts/extraViewSpec.js (Branch: section_10_9)

```
describe("with an extra view", function() {

  beforeEach(function() {
    this.order = new TimeTravel.Models.Order();
    this.extra = new TimeTravel.Models.Extra();
    this.view = new TimeTravel.Views.ExtraView(
        {model: this.extra, order: this.order});
  });

  it("causes its extra to toggle", function() {
```

```
    spyOn(this.order, 'toggleExtra');
    this.view.selectMe();
    expect(this.order.toggleExtra).toHaveBeenCalledWith(this.extra)
  });

});
```

Sample 3-6-12: A test for the extra click action

This is a mock object test – all we're testing is that our click action calls the `toggleEvent` method of the order. Exactly what the `toggleEvent` method does isn't our concern at the moment, especially since it's already been tested.

In order for this test to pass, the `ExtraView` instance needs a reference to the order. This means that we need to make some minor tweaks to a couple of existing classes to make sure the order object is properly passed around. I'm going to also show some of the passing the order object to the hotel views, just so we don't need to show that code again in three pages. First we add the order to the `TripDetailView` data when the view is created:

Filename: app/assets/javascripts/backbone/views/tripDetailView.js (Branch: section_10_9)

```
  initialize: function(options) {
    this.order = options.order || new TimeTravel.Models.Order({trip: this.model});
  },
```

Sample 3-6-13: Adding the order to the creation of the TripDetailView

Filename: app/assets/javascripts/backbone/views/tripDetailView.js (Branch: section_10_9)

```
  hotelsView: function() {
    return new TimeTravel.Views.HotelsView({trip: this.model, order: this.order});
  },

  extrasView: function() {
    return new TimeTravel.Views.ExtrasView({trip: this.model, order: this.order});
  },
```

Sample 3-6-14: Adding the order to the creation of the TripDetailView

Inside the ExtrasView, we handle the initialization and pass it off to the individual ExtraView subviews.

Filename: app/assets/javascripts/backbone/views/extrasView.js (Branch: section_10_9)

```
initialize: function() {
  this.trip = this.options.trip;
  this.order = this.options.order;
  this.collection = this.trip.extras;
  _.bindAll(this, 'render');
  this.collection.bind('reset', this.render, this);
},

renderExtras: function() {
  var $extras = this.$el.find("#extras");
  var self = this;
  this.collection.each(function(extra) {
    var view = new TimeTravel.Views.ExtraView(
          {model: extra, collection: self.collection, order: self.order});
    $extras.append(view.render().el);
  })
  return this;
},
```

Sample 3-6-15: Adding the order to the extras view

And finally, we handle it in the actual ExtraView, including the actual event handler

Filename: app/assets/javascripts/backbone/views/extraView.js (Branch: section_10_9)

```
initialize: function(options) {
  this.order = options.order;
  //##event
  this.order.get("extras").on('add', this.renderSelected, this);
  this.order.get("extras").on('remove', this.renderSelected, this);
  //##event
},
```

```
  selectMe: function() {
    this.order.toggleExtra(this.model);
  },
```

Sample 3-6-16: Handling an event by toggling an order

At this point the test passes. I should mention that passing the order object around like some kind of digital hot potato is not the only design choice. We could have built on the idea that the order is a singleton for the purpose of this page, and made it globally accessible somehow, or accessible via the trip. Ultimately, I think this puts very little dependency on the `ExtraView` class, all it really needs is something that knows how to toggle an event, and I think this structure would be robust against what seems like a pretty real possibility we'd eventually be dealing with more than one order on a page (in an admin console or something).

Now, we need something to respond to the idea that the order has changed. In general, we want the smallest possible part of the screen to be affected. For the extra click, that's just this particular extra view, and also the summary display of the order and its price. Since extras aren't mutually exclusive, clicking on one event has no effect on any other event.

In Backbone, when we want something to respond to a Backbone-specific event, we use Backbone's `on` method[22] to associate a specific Backbone event to a method. What's a Backbone-specific event? Out of the box, it's typically a change to a Backbone model or collection. A Backbone model triggers a `change` event after every call to `set`, the event can be further specified by the attribute changes, as in `change:name`. In addition, when any model in a collection is modified, the collection also triggers a `change` event. Collections also trigger `add` and `remove` events when something is added or removed from them.

Well, we've now got a change in our model, so let's catch it to update our view. The following two lines in the `initialize` method of the `ExtraView` will catch the `add` and `remove` events of the order. While it's entirely possible that I've overlooked something extremely stupid, there doesn't seem to be a common event to catch that covers an add or a remove.

Filename: app/assets/javascripts/backbone/views/extraView.js (Branch: section_10_9)

```
s
  events: {
```

22. As with jQuery, the `on` method used to be called `bind`.

```
    "click": 'selectMe'
  },
  //##events

  //##initialize
  initialize: function(options) {
    this.order = options.order;
    //##event
    this.order.get("extras").on('add', this.renderSelected, this);
    this.order.get("extras").on('remove', this.renderSelected, this);
```

Sample 3-6-17: Handling an event by toggling an order

This event triggers a call to a new method called `renderSelected` which actually updates the view based on the data.

Filename: app/assets/javascripts/backbone/views/extraView.js (Branch: section_10_9)

```
  renderSelected: function() {
    this.$el.toggleClass("selected", this.order.hasExtra(this.model));
  }
```

Sample 3-6-18: Handling an event by toggling an order

The view display adds or removes a specific CSS class based on the data. So, the view display is completely dependent on the data in the model. It's the model that's the source of truth.

Now, we also want to display this information on the other side of the display, as part of the order summary. We're actually going to use the same event mechanism to force calculation of the price.

Filename: app/assets/javascripts/backbone/models/order.js (Branch: section_10_9)

```
TimeTravel.Models.Order = Backbone.Model.extend({
  //##initialCode
  initialize: function() {
    if(!this.get("extras")) {
      this.set("extras", new TimeTravel.Collections.Extras([]))
    }
```

```
    this.get("extras").on('add', this.calculatePrice, this);
    this.get("extras").on('remove', this.calculatePrice, this);
  },
  //##initialCode

  addExtra: function(extra) {
    this.get("extras").add(extra);
  },

  removeExtra: function(extra) {
    this.get("extras").remove(extra);
  },

  hasExtra: function(extra) {
    return this.get("extras").contains(extra);
  },

  toggleExtra: function(extra) {
    if(this.hasExtra(extra)) {
      this.removeExtra(extra);
    } else {
      this.addExtra(extra);
    }
  },
  //##initialCode


...
pricing
});
```

Sample 3-6-19: The new order initialize, forcing recalcuation of price

Same event capture, when we add or remove something to the extras list, the price automatically gets calculated.

Then, it'd be nice to update the order display on the page with the new extra and price. Here's the whole `OrderView`, now with a slightly refactored `render` method, and some event captures.

Filename: app/assets/javascripts/backbone/views/orderView.js (Branch: section_10_9)

```
TimeTravel.Views.OrderView = Backbone.View.extend({
  className: 'order span-8',

  initialize: function() {
    this.model.get("extras").on('add', this.renderExtras, this);
    this.model.get("extras").on('remove', this.renderExtras, this);
    this.model.on("change:price", this.renderTemplate, this)
  },

  render: function() {
    this.model.calculatePrice();
    this.renderTemplate();
    this.renderExtras();
    return this;
  },

  renderTemplate: function() {
    this.$el.html(TimeTravel.template('orderViewTemplate').render(
        this.model.toJSON()));
  },

  renderExtras: function() {
    self = this;
    self.$el.find("#ordered_extras").html("");
    this.model.get("extras").each(function(extra) {
      self.$el.find("#ordered_extras").append(self.renderExtra(extra));
    });
  },

  renderExtra: function(extra) {
    return $("<div>").text(
        extra.get("name") + ": $" + extra.get("price") + ".00");
  }

});
```

Sample 3-6-20: The order view responds to events

As far as the events go, we're catching the add and remove on the extras list and re-rendering the list of events. We're also catching the change in the price, and re-rendering the template, which includes the price display.

At this point, everything is wired up, clicking on the extra triggers a change to the extra display and the order display.

A diagram is probably in order here:

Which leads to the important question. We've added a bunch of features without really adding testable logic. So, how do we test it? All of the event callback and observer stuff is by definition integrating multiple pieces of the application, making it not amenable to unit testing by definition.

A multi-pronged approach works here. Unit test each individual part – we've done that so far, we've got a test for the select/unselect part of the view, and in the model we have tests for both managing the list of active extras and calculating the price. What we need is a test that runs end to end. The problem with end to end tests is that they tend to be slow and require more data and effort to set up, so we don't want to have two many of them (using mock objects on this kind of end-to-end test defeats the purpose – we're trying to test the actual interactions). If you write the end-to-end test first, that's sometimes called "Outside-In Testing".

Two visual metaphors that might be helpful are the Testing Pyramid and the Testing Chain of Responsibility. The Testing Pyramid has a lot of fast unit tests at the bottom, a much smaller number of slower integration tests in the middle and a very small number of really slow acceptance tests at the top. In many cases, the acceptance tests may need to be manual.

The Chain of Responsibility is how you manage the user tests – each test is one link in a chain and is limited to policing just that one link. The connections between the link are often mocked, so each test's responsibilities are clearly delineated.

What does an integration test look like here? It might look like this:

Filename: spec/javascripts/homePageSpec.js (Branch: section_10_9)

```
    it("prices a click", function() {
      TimeTravel.init(tripData);
```

```
    var trip = TimeTravel.getTrip(13);
    this.extra = new TimeTravel.Models.Extra({trip: trip, price: 100});
    trip.extras = new TimeTravel.Collections.Extras([this.extra], {trip: this})
    affix("#container");
    var $container = TimeTravel.app.detailPage(trip);
    $(".extra").click();
    expect($("#order_cost").text().trim()).toEqual("Total Price: $100.00")
  });
```

Sample 3-6-21: An integration test

There's a little bit of setup here, we're initializing the router, creating a trip, adding an extra and wiring it into the trip. Then we render the detail page, simulate a click on the extra, and verify that the order price changes. That simulates the entire path from click to display.

If you remember back to what the router functions looked like, you'll notice that there was no `detailPage` method. In fact, I've refactored the detail page methods a little bit:

Filename: app/assets/javascripts/backbone/routers/tripRouter.js (Branch: section_10_9)

```
tripDetail: function(id) {
  var trip = this.fetchTrip(id);
  this.detailPage(trip);
},

fetchTrip: function(id) {
  var trip = TimeTravel.getTrip(id);
  trip.fetchData();
  return trip;
},

detailPage: function(trip) {
  var order = new TimeTravel.Models.Order({trip: trip});
  this.basicPage(new TimeTravel.Views.TripDetailView({model: trip, order: order}));
}
```

Sample 3-6-22: A rearrainging of the router functions for the detail page

The issue here was that the original workflow made it impossible to render the page without calling `trip.fetchData()`, which triggers an Ajax call – a bad idea in a test. In the refactored version, by separating the trip fetching and page detailing functions, I've created a seam where I can drop in an already created trip so that I can run a test with trip data of my own choosing, getting the benefits of isolation without giving up much of the value of the integrated test.

While it may seem like this kind of functionality split is just making the code contort to the needs of the test, I guarantee that once you start splitting off functionality to give your code more seams, you'll find it a useful technique for integrating later changes into code smoothly.

Section 3.7

Hotelify

Displaying the hotel information is easier to manage on the order side, because each order only needs to manage one hotel, but a little bit more complex on the view side, since the hotel options behave like radio buttons, not checkboxes – you can only have one hotel, so selecting one deselects the others.

This time, we can start with the integration test:

Filename: spec/javascripts/homePageSpec.js (Branch: section_10_10)

```
    it("prices a hotel click", function() {
      TimeTravel.init(tripData);
      var trip = TimeTravel.getTrip(13);
      var cheapHotel = new TimeTravel.Models.Hotel({trip: trip, price: 100, id: 1});
      var expensiveHotel = new TimeTravel.Models.Hotel({trip: trip, price: 500, id: 2});
      trip.hotels = new TimeTravel.Collections.Hotels([cheapHotel, expensiveHotel],
          {trip: trip})
      affix("#container");
      var $container = TimeTravel.app.detailPage(trip);
      $("#length-select").val(3);
      $("#length-select").change();
      $("#hotel_" + expensiveHotel.get("id")).click();
      expect($("#hotel_" + expensiveHotel.get("id"))).toHaveClass("selected");
      expect($("#hotel_" + cheapHotel.get("id"))).not.toHaveClass("selected");
```

```
    expect($("#order_cost").text().trim()).toEqual("Total Price: $1500.00")
  });
```

Sample 3-7-1: The acceptance test for the hotel

This test is very similar to the integration test we just saw for extras. There are two additions. First, we're creating two hotels so that we can validate the behavior that only one is selected after a click. (This requires a slight change in the `hotelView.mustache` template to add an `id=hotel_{{id}}` field to the outer box, or alternately adding the id calculation to the view itself. Otherwise, we're still creating a bunch of data, clicking on something, and evaluating the results.

I'll be frank, I don't love the test, I think it's inelegant because we don't have well-developed mechanisms in this project to easily set up data, primarily because we haven't needed them thus far. A good system for creating objects and getting to the right page structure goes a long way in this kind of test between easy to work with, and dreadful.

Still, it's functional, and maybe we'll come back and refactor both integration tests in a little bit. Now we need to make it work. Thinking of the chain of responsibility... let's cover the responsibilities of the order model first.

Since we're dealing with pure data here, the unit tests are reasonably straightforward – this `describe` block is nested inside the other, so it has access to the `extra` models:

Filename: spec/javascripts/orderSpec.js (Branch: section_10_10)

```
  describe("and for hotels", function() {
    beforeEach(function() {
      this.hotel = new TimeTravel.Models.Hotel({price: 100});
    });

    it("calculates hotel price", function() {
      this.order.setHotel(this.hotel);
      this.order.setLengthOfStay(5);
      this.order.calculatePrice();
      expect(this.order.get("price")).toEqual(500);
    });

    it("calculates combined price", function() {
```

```
      this.order.setHotel(this.hotel);
      this.order.setLengthOfStay(5);
      this.order.addExtra(this.pricyExtra);
      expect(this.order.get("price")).toEqual(1000);
    });
  });
```

Sample 3-7-2: Unit tests for hotel price

And the resulting code:

Filename: app/assets/javascripts/backbone/models/order.js (Branch: section_10_10)

```
setHotel: function(hotel) {
  this.set("hotelName", hotel.get("name"));
  this.set("hotel", hotel);
},

setLengthOfStay: function(days) {
  this.set("lengthOfStay", days)
},

calculatePrice: function() {
  var prices = this.get("extras").pluck("price");
  var price = _.reduce(prices,
      function(agg, item) { return agg + item },
      0);
  if (this.get("hotel")) {
    price += this.get("hotel").get("price") * this.get("lengthOfStay")
  }
  this.set({price: price})
  return price
}
```

Sample 3-7-3: Code to make hotel pricing work

One quick comment: strictly speaking, I don't need to break off `setHotel` and `setLengthOfStay` as separate methods – they are just simple wrappers around already public behavior. Even though I don't need them as separate methods, I kind of like the idea of broadcasting, by

convention, a public API for this class that is more narrow than the infinite list of parameters that Backbone can handle. Of course, I can't stop anybody from using the `set` method directly, but I think it's a good general rule to minimize calling `set` on somebody else's data.

Moving on, we can now assume that the order behaves correctly, so now all we need to test is our interaction with it. That interaction has the same two parts that the extra interaction did – a click sets the value, and the set value determines the view status – plus one more, a change to the pulldown triggers the value change.

When we tested the `ExtraView`, we tested that the method called from the click triggered the interaction with the object. Just for the sake of showing something different, this time let's back up a step and test the click event itself.

Filename: spec/javascripts/hotelViewSpec.js (Branch: section_10_10)

```
describe("with a hotel view", function() {

  beforeEach(function() {
    this.order = new TimeTravel.Models.Order();
    this.hotel = new TimeTravel.Models.Hotel({id: 10});
    this.view = new TimeTravel.Views.HotelView(
        {model: this.hotel, order: this.order});
  });

  it("clicking on a hotel causes interaction with order", function() {
    spyOn(this.order, 'setHotel');
    var $el = this.view.render().$el
    $el.find(".hotel-name").click();
    expect(this.order.setHotel).toHaveBeenCalledWith(this.hotel)
  });

});
```

Sample 3-7-4: Testing the hotel view

If you're still thinking about the chain metaphor, this test is effectively two links – one between the click and the method call in the `HotelView`, and one between the `HotelView` and the `Order`. So that's the plus side, it's testing more of the system. On the down side, it's a little bit more complicated to set up, because the view actually has to `render` – we got a little lucky here, since there's not a lot of setup, but we still need to create a few objects before we go.

Also, the fact that it covers two links in the chain makes the test more brittle, since the test has more dependencies on the internal details of the `HotelView`.

Here's the whole `HotelView` class as it now stands, making this test pass:

Filename: app/assets/javascripts/backbone/views/hotelView.js (Branch: section_10_10)

```
TimeTravel.Views.HotelView = Backbone.View.extend({

  className: 'hotel',

  events: {
    "click": 'selectMe'
  },

  initialize: function(options) {
    this.order = options.order;
  },

  render: function() {
    this.$el.html(TimeTravel.template(
        'hotelViewTemplate').render(this.model.toJSON()));
    this.$el.attr("id", "hotel_" + this.model.get("id"));
    return this;
  },

  selectMe: function() {
    this.order.setHotel(this.model);
  },

});
```

Sample 3-7-5: Hotel View, now with click handler

Not much has changed, but we've added the `events` attribute triggering the call to `selectMe`, and we've also inserted the assignment of a DOM id inside the `render` method, which we're doing because if you set the ID the Backbone view's `id` attribute, then the ID has to be static.

We also have the interaction of the pull-down to test, and the status of the hotel views based on which hotel is set. Since the pull-down is actually in the collection `HotelsView`, and the status really needs to be tested as part of a group, both these tests are tests on the `HotelsViewSpec` object:

Filename: spec/javascripts/hotelsViewSpec.js (Branch: section_10_10)

```
it("interacts with order when length of days is set", function() {
  var order = new TimeTravel.Models.Order({});
  var trip = new TimeTravel.Models.Trip({});
  var view = new TimeTravel.Views.HotelsView({trip: trip, order: order});
  spyOn(trip, "lengthInDays").andReturn("2");
  spyOn(order, "setLengthOfStay");
  view.render();
  view.$el.find("#length-select option").val(1);
  view.$el.find("#length-select option").change();
  expect(order.setLengthOfStay).toHaveBeenCalledWith('1');
});

it("correctly determines select status of hotels", function() {
  var order = new TimeTravel.Models.Order({});
  var hotel_1 = new TimeTravel.Models.Hotel({id: 1});
  var hotel_2 = new TimeTravel.Models.Hotel({id: 2});
  var hotels = new TimeTravel.Collections.Hotels([hotel_1, hotel_2]);
  var trip = new TimeTravel.Models.Trip({});
  trip.hotels = hotels;
  var view = new TimeTravel.Views.HotelsView({trip: trip, order: order});
  view.render();
  order.setHotel(hotel_1);
  expect(view.$el.find("#hotel_1")).toHaveClass("selected")
  expect(view.$el.find("#hotel_2")).not.toHaveClass("selected")
});
```

Sample 3-7-6: Tests for the hotels view interactions

These tests are a little different – though to my mind they are both showing that it might be time to figure out a better way to generate test data, that's starting to be a lot of setup. The pulldown test renders the view, forces a `change` event, and uses mock objects to determine

that the order is called correctly. We're using the mock here to simulate our dependence between different layers of the application.

The status test, on the other hand, is using a real `Order` instance, and sends it a real `setHotel` message – we're not simulating the click here, for no real reason other than I thought it was clearer to write the test with explicitly setting the hotel. After we set, we determine that the `selected` status of each hotel view is as expected.

There are only a few changes we need to make to `HotelsView` to get this to work. First, we need to add some events.

Filename: app/assets/javascripts/backbone/views/hotelsView.js (Branch: section_10_10)

```
events: {
  "change #length-select": 'updateLength'
},

initialize: function(options) {
  this.trip = this.options.trip;
  this.collection = this.trip.hotels;
  this.order = this.options.order;
  _.bindAll(this, 'render');
  this.collection.bind('reset', this.render, this);
  if(this.order) {
    this.order.bind('change:hotel', this.updateSelectStatus, this);
  }
},
```

Sample 3-7-7: Initializing the hotels view

This snippet adds an `event` attribute pointing at the `change` event of the `#lenght-select` object, which is our length of stay pulldown. Inside the `initialize` method, we're binding a change to the order's hotel attribute by calling a different update method.

Those update methods look like this:

Filename: app/assets/javascripts/backbone/views/hotelsView.js (Branch: section_10_10)

```
  updateLength: function() {
    this.order.setLengthOfStay(this.$el.find("#length-select").val())
  },

  updateSelectStatus: function() {
    self = this
    this.collection.each(function(hotel) {
      self.$el.find("#hotel_" + hotel.get("id")).toggleClass(
          "selected", self.order.get("hotel") == hotel)
    });
  }
```

Sample 3-7-8: Updating the hotels view

Both straightforward. We update the length by sending the selected length to the order. We update the select status by going through the list of hotels, and changing the CSS class of the associated `HotelView` based on whether the hotel matches the one currently being held by the order.

Unless I'm missing something, there's just one more tiny piece we need:

Filename: app/assets/javascripts/backbone/models/order.js (Branch: section_10_10)

```
  initialize: function() {
    if(!this.get("extras")) {
      this.set("extras", new TimeTravel.Collections.Extras([]))
    }
    this.get("extras").on('add', this.calculatePrice, this);
    this.get("extras").on('remove', this.calculatePrice, this);
    this.on("change:hotel", this.calculatePrice, this);
    this.on("change:lengthOfStay", this.calculatePrice, this);
  },
  //##initialCode

  addExtra: function(extra) {
    this.get("extras").add(extra);
  },

  removeExtra: function(extra) {
```

```
    this.get("extras").remove(extra);
  },

  hasExtra: function(extra) {
    return this.get("extras").contains(extra);
  },

  toggleExtra: function(extra) {
    if(this.hasExtra(extra)) {
      this.removeExtra(extra);
    } else {
      this.addExtra(extra);
    }
  },
```

Sample 3-7-9: Adding more change events to the order

Inside the order, we need to bind changes to the hotel and length of stay attributes to automatically recalculate the price.

And with that, the original integration test also passes, and we have our hotels wired into the rest of the detail page.

One Last Thing

Just one more piece of functionality before we can do some wrap-up and call it a night. We need to be able to send a completed order to the server for, well, for whatever the server is going to do with it which, frankly is not our lookout for the purposes of this book.

First up, let's get a button up there. It's part of the `OrderView` template, and I assume we only want the button to be active if a hotel and length of stay have already been chosen. We've seen a few ways of implementing this conditional – we could make it a conditional in the template, or we could have the state of the button injected into the DOM from the order's `render` method. In either case, though, I want to make sure that the logic about whether an order is ready to be sent is contained in the `Order` model itself:

Filename: spec/javascripts/orderSpec.js (Branch: section_10_11)

```
it("knows when it can not be set", function() {
  this.hotel = new TimeTravel.Models.Hotel({price: 100});
  expect(this.order.isSendable()).toBeFalsy();
  this.order.setLengthOfStay(5);
  expect(this.order.isSendable()).toBeFalsy();
  this.order.setHotel(this.hotel);
  expect(this.order.isSendable()).toBeTruthy();
});
```

Sample 3-7-10: Quick test for whether an order is sendable

Which easily passes with:

Filename: app/assets/javascripts/backbone/models/order.js (Branch: section_10_11)

```
isSendable: function() {
  return this.get("hotel") && this.get("lengthOfStay");
},
```

Sample 3-7-11: And passing code

To incorporate the button in the view, we put the following line in the `orderViewTemplate.mustache` file:

```
<input type="button" value="Order..." class="hide" id="order_send">
```

Sample 3-7-12:

Then we augment the `renderTemplate` method in the `OrderView`:

Filename: app/assets/javascripts/backbone/views/orderView.js (Branch: section_10_11)

```
renderTemplate: function() {
  this.$el.html(TimeTravel.template('orderViewTemplate').render(
      this.model.toJSON()));
  this.$el.find("#order_send").toggleClass("hide", !this.model.isSendable());
},
```

Sample 3-7-13: Make the button appear when the order is sendable

So we're toggling the class of the button so that the `hide` class shows if the button is not sendable.

Nifty. We've got a button. What do we do with it?

Well, on the view side, we need to intercept a click action on the button and do something with it.

Filename: app/assets/javascripts/backbone/views/orderView.js (Branch: section_10_11)

```
events: {
  "click #order_send": 'orderSend'
},

orderSend: function() {
  this.model.sendToServer();
},
```

Sample 3-7-14: The event handler for our send button

All we're doing here is passing the click event along to the model. We're perfectly capable of actually sending the data from the view, it just makes more sense to continue to allow the model control over it's logic, in this case it's logic for saving itself.

The model side uses Backbone's existing `sync` mechanism.

Filename: app/assets/javascripts/backbone/models/order.js (Branch: section_10_11)

```
url: "/orders",

sendToServer: function() {
  Backbone.sync("create", this, {});
},
```

Sample 3-7-15: The event handler from the order

Just a couple of lines of code, but a few things to say:

- We need a `url` parameter because we don't have any concept of an orders collection in the app at the moment. Normally, Backbone models infer their url from their

collection. Obviously, the url parameter we have at the moment is limited to a create mechanism.

- This project uses the `rails-backbone` gem, which replaces the default behavior of `Backbone.sync` for reasons that are not completely spelled out for me.
- Current versions of Backbone specify `sync` as a method of the model (as in `this.sync`) rather than the global `Backbone.sync` I'm using in the code. The model version doesn't play nicely with `rails-backbone` for reasons that were not immediately obvious to this casual observer.

Anyway, the `sync` method takes three arguments. The first is the command being sent to the server, which can be "create", "read", "update", or "delete" – Backbone converts this to the appropriate HTTP verb given Rails-like RESTful conventions. The second argument is the model being sent, and the third argument is a bunch of options that don't really concern us here.

As a result of this method the orders model will be converted to JSON (including all subobjects) and sent as as parameters to the designated URL. Note that there is no root object to the JSON – the data comes in as `{id: 3, lengthOfStay: 1}` rather than `{order: {id: 3, lenghtOfStay: 1}}`. However, this behavior can be overridden – if the model defines a `paramRoot` attribute, then `rails-backbone`, at least, will honor that as the root name of the JSON sent back to the server.

Since this is not *Master Space and Time With Rails CRUD Methods*, sending the data to the server passes it out of our scope for the time being. Meaning that our Backbone app is feature-complete.

Section 3.8

What does it all Mean?

Chapter 4

Acknowledgements

A number of people helped in the creation of this book, whether they knew it or not.

As you may know, this book began life with a non-me publisher. Technical reviewers of the manuscript at that point included Pete Campbell, Kevin Gisi, Kaan Karaca, Evan Light, Chris Powers, Wes Reisz, Martijn Reuvers, Barry Rowe, Justin Searls, Stephen Wolff. Kay Keppler and Susannah Pfalzer acted as editors. Thanks to all of them.

Avdi Grimm provided a lot of useful tools for self-publishing.

I've been lucky enough to work with people who were willing to share JavaScript experience with me, including, but not limited to: Dave Giunta, Sean Massa, Fred Polgardy, and Chris Powers.

Justin Searls and his Jasmine advocacy and toolkit have been a huge help.

One of the great things about self-publishing is that people take the time to help improve the book by pointing out mistakes and typos. Thanks to Pierre Nouaille-Degorce, Sean Massa, Vlad Ivanovic and Nicolas Dermine for particular efforts.

Emma Rosenberg-Rappin helped me design the noelrappin.com web site, pick cover fonts, and also did some copyediting.

Elliot Rosenberg-Rappin inspired the idea of a spaceship cover and laughed at some of my jokes.

This book, and many other wonderful things, would not exist without my wife Erin, who has been nothing but supportive through the ups and downs of this project. She's amazing.

Chapter 5
Colophon

Master Space and Time with JavaScript was written using the PubRx workflow, available at https://github.com/noelrappin/pub_rx. The initial text is written in MultiMarkdown, home page http://fletcherpenney.net/multimarkdown/. PubRx augments MultiMarkdown with extra descriptors allowing for page layout effects, such as sidebars, that Markdown does not manage on its own.

PDF conversion is managed by PrinceXML http://www.princexml.com, with a little bit of additional processing by PubRx. EPub and Mobi generation is managed by using the Calibre http://calibre-ebook.com/ command line interface, using a method described by Avdi Grimm's Orgpress https://github.com/avdi/orgpress.

For the PDF version, the header font is Museo and the body font is Museo Sans. Both are free fonts from the exljbris font foundry http://www.exljbris.com/. The code font is "M+ 1c" from M+ Fonts http://mplus-fonts.sourceforge.jp/. On the cover, the title font is Bangers by Vernon Adams http://code.google.com/webfonts/specimen/Bangers, and the signature font is Digital Delivery, from Comicraft http://www.comicbookfonts.com/, and designed by Richard Starkings & John 'JG' Roshell. Cover image credit is on the title page up front.

Chapter 6

Changelog

Release 001: October 2, 2012 Initial release.

Release 002: November 19, 2012

- Second chapter started.
- Typos and errors discovered by Dutch Rapley, Sean Massa

Release 003: November 26, 2012

- Second chapter continued with display of detail view and subviews.
- URL of detail page changed to avoid conflict with book 2 show page.

Release 004: December 3, 2012

- Another eight pages or so on orders, trip details, and making them work together.

Release 005: December 10, 2012

- Another eight or so pages wiring up the events and calculating order price. Probably only one or two of these to go.

Release 006: December 18, 2012

- Calculating hotel prices. More pages.

Release 007: January 7, 2012

- Sending hotel information back to the server
- This book is now draft complete, pending a conclusion that will come after I do some editing.