MASTER SPACE AND TIME WITH JAVASCRIPT

BOOK 4: EMBER

Noel Rappin

a noelrappin.com publication

Master Space and Time With JavaScript

Book 4: Ember

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

Release 007 September, 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:

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.

Book 3: Backbone Is currently available for $7. It continues building the website using Backbone.js to create single-page interfaces for some more complex user interaction.

Book 4: Ember, is what you are reading right now. It is also available for $7.

You may purchase all four parts of the book for $15, a $6 savings over buying all three parts separately.

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 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 in Book 1.

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. At some point in the future, I may provide a separate CoffeeScript version of the code in this book.

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 2.0.3, Jasmine 1.3.1, Backbone.js 1.0.0, Ember.js 1.0.0 and Rails 3.2.x.[1] Keeping up with current versions is hard enough, without worrying about the interactions among multiple versions.

1. Rails 4.0 was released during the production of the book, and I haven't updated all the examples to it, but there should be minimal changes to support Rails 4.

Section 1.4

But is it finished?

This book is in very late beta. 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.

- CoffeeScript versions of all the sample code in the book may 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 4 needs a final copy edit, and may have some content additions. It will be kept up to date at least through Ember 1.1.
- 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

Ember Me

Section 2.1

Another Openin' Another Show

By now, I suspect you know the drill.

> Dear Friend, Excellent work on the Backbone project. I'm looking forward to your work on the nerves and muscles. In the meantime, our Time Travel Administrators are in an awful mess. All their work is done on paper, and even when you can travel back in time to get them done, that's quite a burden. Can you whip something up on your website for them? Sincerely, Doctor What

An admin panel. Okay. Looks like a good opportunity to try out that Ember thing we've been looking at.

Section 2.2

What's that Ember thing We've Been Looking At?

Ember bills itself on its website as "A framework for creating *ambitious* web applications". It is a framework that brings a Model/View/Controller structure to client-side JavaScript. In some ways, Ember shares the client-side framework space with Backbone, not to mention Angular, Knockout, and however many dozens of other frameworks that I don't plan on writing 100 pages about.[2]

2. I have gotten multiple requests to do Angular. (Yes, that means I've gotten two). No promises.

There is a fundamental philosophical difference between, specifically, Backbone and Ember. As we saw in Book 3, Backbone is a relatively small framework that makes minimal assumptions about the structure of your code and how your code and Backbone will interact. Backbone defines relatively little in the way of default behavior. Ember, on the other hand, is a much larger framework that makes a number of assumptions about the structure of your code, including many Rails-style convention-over-configuration naming assumptions. Ember will do a lot of things for you "magically".[3]

You can see one specific example of the difference between the two tools when it comes time to render objects to the display. In Backbone, view objects have a `render` method which is called when the view is expected to render itself. Backbone does not specify anything that happens in that method, by default it's a no-op. In Backbone, it is entirely your application's responsibility to insert elements in the DOM. In Ember, the render loop has a lot of moving parts, including automatically rendering templates for its display objects[4], and automatically updating displays as a result of data changes.

When model data changes, Backbone triggers an event, but does not require that anything in particular happen as a result. Ember will, more-or-less by default, automatically update places in the view where that data is displayed. Backbone doesn't specify a view-template structure. Ember uses Handlebars, a template tool with a fair amount of logic built in, and augments it with some Ember-specific helpers.[5]:

As Ember core team member Trek Glowacki said on Twitter, literally minutes before I wrote this first draft: "In Ember.js we're determined not to give you enough rope to hang yourself" https://twitter.com/trek/status/287566171803361280.

Ember is designed for "long running [applications] – [where] people will spend all day sitting and working in them. And usually, the application as you interact with it has deep view hierarchy changes in reaction to your data coming in or in reaction to user behavior."[6]

3. Programming magic, defined: functionality in a framework or system that is not obvious from inspecting application code that uses that framework. For example, Ember's automatic view updates.

4. Ember uses the term `view` slightly differently than Backbone does.

5. You get the point, I trust. In deference to any of you that haven't bought Book 3 on Backbone, I'll try to keep the Backbone comparisons down to a dull roar. Or, you know, you could consider buying Book 3. Just saying.

6. That is Trek Glowacki again, this time from the JavaScript Jabber podcast. http://javascriptjabber.com/034-jsj-ember-js/.

You'll see an effect of Ember's structure in this book. As compared to the Backbone example, the code samples will likely be much shorter. But we will spend a lot more time discussing the behavior and API of the framework itself.

Section 2.3

Fingers Ready, Let's Code!

We are going to build up the beginnings of our admin screen using Ember, using this as an excuse to map out Ember's most important concepts. Once that is done, we'll step back and look at how the concepts relate to each other. Then we'll add more features to our admin panel, specifically chosen to show off other Ember features.

The first thing we're going to do is just display an administrative version of our "show all the trips" screen. Yes, it's kind of similar to what we've done before, but we'll take it in a little bit of a different direction.

The inevitable annoying setup

In order to preserve my tenuous hold on sanity, I blew away all the previous JavaScript in the application from books 1 - 3. All the old branches still exist in the GitHub repo, but the branches that start with `e_01` have none of that previous JavaScript. If you are following along and paying close attention, you'll also note I've refreshed the gemset a bit.

Since we're working inside a Rails application, I started my Ember.js setup by adding the `ember-rails` gem to my `Gemfile`[7] – we'll cover setup for non-Rails people in a bit.

```
gem 'ember-rails'
```

Sample 2-3-1: Adding ember-rails to the Gemfile

We're using ember-rails primarily because it automatically puts handlebars templates into the Rails asset pipeline. We're not depending on it to have the current version of Ember (though it probably will be better for this once Ember settles down.) It also provides some boilerplate generators – we won't be using those either.

7. As of this writing, I'm using the git head for `ember-rails` as well: `gem 'ember-rails', git: "git://github.com/emberjs/ember-rails.git"`. (Previous versions of the book used a custom fork of `ember-rails`. You don't need that anymore)

Ember-rails has actually delegated the actual container of the Ember source to different gems called `ember-source` and `ember-data-source`. In theory, this allows Ember itself to be updated without having to cut a new version of Ember-rails.

Keep looking here to see what version of Ember the rest of the book uses: We're using Ember-Rails version 0.13 (revision `37f7d126`), Ember 1.0 (downloaded from http://builds.emberjs.com.s3.amazonaws.com/tags/v1.0.0/ember.js), and Ember-data 1.0 beta 2 (downloaded from http://builds.emberjs.com.s3.amazonaws.com/tags/v1.0.0-beta.2/ember-data.js).

If you need to use the cutting edge Ember, `ember-rails` gives us an easy way to update our local Ember to the current head on github. From the command line:[8]

```
rails g ember:install --head
```

Sample 2-3-2: Invoking a generator to grab the Ember.js master

This will run through some git command line stuff, and eventually populate the `vendor/assets` directory with `ember.js` and `ember-data.js`. Note that when you rerun this command you'll be prompted to overwrite the existing files.

I have to say, in practice, I've bypassed this mechanism in favor of grabbing the Ember files directly from the Ember build site at http://emberjs.com/builds/#/tagged and sticking them in the `vendor/assets/ember/development` and `venor/assets/ember/production` directories my own self.

There is one other piece of pure configuration. Ember.js has slightly different development and production versions (namely, the production version is minimized). In Rails, we can piggy-back on the existing environment initialization to tell Ember which version to use. For the moment, we're only concerned with the development environment, so:

```
config.ember.variant = :development
```

Sample 2-3-3: Add this to your development config file

That's the Ember side, we also need some setup in our Rails application itself. To start, we don't need much more than a route that does nothing but load up Ember.

8. Ember-Rails keeps a clone of the Ember git repo in `~/.ember`. I've had that kind of get stuck and not give me the current master when asked, but deleting `~/.ember` and re-running the Rails generator works.

We're going to do this the dirt-simple way. We'll keep our existing root route as `home#index`.

But we're going to make the `HomeController` a nearly empty controller:

Filename: app/controllers/home_controller.rb (Branch: e_01)

```
class HomeController < ApplicationController

  def index
  end

end
```

Sample 2-3-4: Boring controller

Which needs a nearly-empty layout

Filename: app/views/layouts/home.html.erb (Branch: e_01)

```
<!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 2-3-5: Boring layout

And a completely empty `app/views/home/index.html.erb`. Totally empty. In the Backbone case, the analogous controller and view loaded a JSON representation of our trips. We could do that here, but we're not going to because we're going to let Ember do that for us.

A Basic Ember Display

Back to Ember. The `ember-rails` gem can generate a lot of boilerplate setup. We're not going to use that, because it generates some stuff we're not going to need. Instead, we'll do the minimal setup by hand.

We need to start by making sure all the Ember code gets loaded and then we need to create an `Application` object.

Filename: app/assets/javascripts/application.js (Branch: e_01)

```
//= require jquery
//= require jquery_ujs
//= require jquery-ui
//= require moment
//= require handlebars
//= require ember
//= require ember-data
//= require_self
//= require_tree .

TimeTravel = Ember.Application.create();
```

Sample 2-3-6: Ember in the manifest, and creating the application object

We've added a few lines to the asset pipeline manifest to load `handlebars`, `ember` and `ember-data`. Ember splits the framework into two parts, where `ember-data` handles model logic and data storage, and the `ember` library handles the controller and view logic.

Our first actual line of Ember code creates an `Ember.Application` object of our very own. It's probably worth mentioning that most Ember tutorials or guides will recommend that you name your application object `App`, in part to make it easier to copy code from one application to another. You'll notice that I've manifestly failed to do so, out of a possibly idiosyncratic dislike for generic variable names. Don't let me stop you, though, if you want to use `App`.

Past versions of Ember needed to create a data store here, but the 1.0 versions of Ember and Ember Data do that behind the scenes, so we don't need to explicitly create it.

The application object in Ember generally works in the background, and for the most part, we won't deal with it directly. For our purposes, creating the application is primarily a declaration

that we're entering Ember-land, and Ember will have control over routing and the like. The application object also serves as our namespace – all the other Ember things we create will be scoped to `TimeTravel`.[9]

As the application object initializes itself, it sets in motion the most basic beginnings of an Ember application – the *application template* and the *router*.

The application template in an Ember application serves a very similar purpose to the layout in a Rails application. It is the background part of the page that is always there, no matter what state the application is in. Ours is going to start pretty simple:

Filename: app/assets/javascripts/templates/application.handlebars (Branch: e_01)

```
<h1>Time Travel Adventures Administrative Screen</h1>

{{outlet}}
```

Sample 2-3-7: Our application template

This is a Handlebars template. Handlebars is a templating tool that is an extension of Mustache. Handlebars is more accepting of logic in the template than Mustache was, and you'll find that Ember style tends to put more logic in the template than we've seen so far.

The `ember-rails` gem, by default, expects all Ember templates to be in the directory `app/assets/javascripts/templates` with an extension of `.js.hjs`, `.hbs`, or `.handlebars`. Ember will automatically find the template named `application` with any of those extensions and use it as the application template.[10]

At the moment, the important parts of this template are the `{{}}`, which in Handlebars, as in Mustache, is syntax to indicate dynamic content, and `outlet`, which is an Ember keyword that is a placeholder for some other object that is going to have content to be placed at the point of the outlet. The Ember `outlet` is roughly similar to the `yield` in a Rails layout. By default, an application template is expected to have one outlet. We'll see more about what outlets are good for when we get deeper into Ember routes and nested routes.

Now we have to put something in that `outlet`.

9. Remember, we can use `TimeTravel` as a global name again, because the old code that used that name has been removed.

10. Non Rails users can embed the template in a `script` tag using an attribute called `data-template-name` to name the template. All Ember template names are in the same namespace.

Let's back up and think about this problem generically for a second. At any point, your web application state consists of the following:

- **Data**. Information you need to convey to the user.
- **Display logic**. A pattern for showing that information.

In a server-side web application, a particular data and display combination represents a state of the application that usually maps an individual web page.[11] Each page has a URL, which causes the server-side to round up some data, and merge it with some display logic. You'll have an index page, then maybe a detail page, then maybe an edit page, and so on.

In a client-side Ember application, the difference between states is a bit more fluid. Nevertheless, there is still a need to say "the user clicked on this link, and now I would like to show this data arranged like so." As far as the user is concerned it may or may not still look like a completely different page, but as far as Ember is concerned, you've switched to a different context.

In Ember, each such context is called a *route*. Each route unifies a particular state of the application with a *controller* that handles display logic, and a *model* that handles data. The routes are paired with URLs by the Ember `Router` object. URL's are important to Ember, every state in the system is associated with one, and conversely, coming to an Ember application with a particular URL should take you to the same state.

Ember maintains a pretty strict sense of what goes in the controller object and what goes in the model. If you are using Ember Data, the model is backed by an Ember `DS.Store` object and contains data that is either directly backed by persistent storage, or is a calculated property based on persistent data.[12] If you aren't using Ember Data, you can roll your own model layer.

The controller manages the Handlebars template and any data that is specific to display logic. Ember also has `View` objects that have a more limited scope than Backbone views, and which are also managed by the controller. In most places where you have a model in Ember, you will have a controller mediating access to it.[13]

11. I'm deliberately avoiding the MVC magic words here so as not to step on Ember's usage of those concepts.

12. Strictly speaking, that's an overstatement – the model can be any Ember object, but it's often backed by a data store.

13. I realize I'm approaching David Foster Wallace levels of footnoting here, but... As Ember has moved from beta to release, it's gotten much easier to attach a controller to a particular model in a particular view.

That's MVC, Ember Style.

Code will make this clearer. We want our administrative screen to display a list of all trips, vaguely similar to the user screen, though eventually we'll be adding different information.

We need to unify three things – the state of the application, a set of display logic, and a set of data. In Ember, this unification is handled by the `Router` object. Here's ours:

Filename: app/assets/javascripts/routes/app_router.js (Branch: e_01)

```
TimeTravel.Router.reopen({
  location: "history"
});

TimeTravel.Router.map(function() {
  this.route("index", {path: "/"});
});
```

Sample 2-3-8: The simplest router

Please note that the router API has changed a few times, this is the 1.0 version.

The most important line of this router is `this.route("index", {path: "/"});`, which tells Ember that when it detects the root route, to pass information on to `index`. Strictly speaking, this line is redundant – Ember matches this route by default. I'm including it here because a) it wasn't redundant when I started writing the book and b) it helps to have an actual routing line to talk about.

Okay, so the route passes information to a route named `index`. What does that actually mean?

Ember uses the router to connect a URL to a `Route` object, using string inflection based on the name of the route. Saying that the route is named `index` causes Ember to look for a route class called `IndexRoute`. Eventually, it will also look for a controller class called `IndexController`, and a template file called `index.handlebars`.

Before I show `IndexRoute`, two quick points. First, notice that we never actually create a `Router` – Ember takes care of that when the application is started. Also, that business with `reopen` and `location` – Ember allows you to reopen already created classes in much the same way that Ruby does, using, you guessed it, the `reopen` method. The argument to `reopen` is a literal object that is merged into the class. The `location: "history"` causes Ember to use `pushState`

to manage history on browsers that allow it, If you don't set location to `history`, then routes will be of the form `http://localhost/#/admin`, rather than `http://localhost/admin`.[14]

If you are keeping score, we've got a `Router` that uses the URL to pass control to a `Route` object. The `Route` object's goal in life is to associate a controller and a model.

Here's our initial `Route` object:

Filename: app/assets/javascripts/routes/index_route.js (Branch: e_01)

```
TimeTravel.IndexRoute = Ember.Route.extend({
  model: function() {
    return this.store.find('trip');
  }
});
```

Sample 2-3-9: The initial IndexRoute

"Okay", I hear you say... "I see a model, but no controller. You just said that a `Route` unifies a model and a controller. Where's my controller?"

Good question. The answer, perhaps unsurprisingly, is another Ember default. When the route is invoked, it calls it's `model` method, which in our case goes and creates a bunch of models. Specifically, the `model` method calls `find` on the Ember Data `store` with an argument `trip`.

A very quick overview of Ember Data terminology: the `store` is Ember's local copy of all model instances that the client-side part of the application knows about. (Ember Data documentation uses the term *record* as a synonym for model instance.) When you ask the store to find data, it first looks locally – the argument to the `find` method is the string-inflected name of the model class being requested. If the data isn't in the store already, the store requests the data from an external source via an *adapter*. The default adapter uses a standard REST API to communicate with an application server. Later in the book, we'll go into more detail about what is actually being requested from the server, and what is actually being returned. For now, it's enough to know that we're getting a list of all the `Trip` models in the system.

14. Remember that whatever your route is, your server-side solution also needs to be able to handle it, you want there to be no difference to the user whether they traverse your site via internal links or enter via an arbitrary URL.

Ember routes also allow you to create your controller using a `setupController` method. If `setupController` is not called, then the default is to use string inflection to find the controller, and assign the result of the `model` method to the `content` property of said controller.

Which means our controller is going to be called `IndexController` in our application namespace. Here it is:

Filename: app/assets/javascripts/controllers/index_controller.js (Branch: e_01)

```
TimeTravel.IndexController = Ember.ArrayController.extend({

});
```

Sample 2-3-10: Our initial index controller

Complex, no? We'll be adding more to it in a bit. Ember controllers come in two flavors based on the model data. An `ObjectController` is used if the model is scalar data, and an `ArrayController` is used if the model is a list.

At this point, we have one more piece of inflected magic to discuss – the template – and one class that we've referenced but haven't talked about – the model. Which to do first? Actually, talking about the model will make it easier to understand the template. So, model first.

When we last saw our model class, `Trip`, it was being obliquely referenced in the `IndexRoute` method `model`.

The model classes that we'll be talking about get their behavior from two different parent classes. The basic behavior of properties with observers and calculated properties comes from the Ember core library and the `Ember.Object` class. This behavior is shared among all Ember framework objects.

Models that use Ember Data to manage persistence are subclasses of the class `DS.Model`, which is itself a subclass of `Ember.Object`. Ember Data models are designed to communicate with a source of information and convert the raw data from that source to instances with a set of properties that can be referenced by other objects. By wrapping data access in Ember properties, Ember simplifies two key components of a web application, namely a) converting incoming data to a useful object and b) allowing the display or other objects to update when data changes.

Here's our initial model:

Filename: app/assets/javascripts/models/trip.js (Branch: e_01)

```
TimeTravel.Trip = DS.Model.extend({
  name: DS.attr('string'),
  description: DS.attr('string'),
  start_date: DS.attr('date'),
  end_date: DS.attr('date'),
  image_name: DS.attr('string'),
  slug: DS.attr('string'),
  tag_line: DS.attr('string'),
  price: DS.attr('number'),
  location: DS.attr('string'),
  activity: DS.attr('string')
});
```

Sample 2-3-11: Our initial ember model

And you'll note that we have some code here, even if we don't have any logic.

What we are doing here is defining the properties of a trip as we expect to receive them from the server. Ember Data expects a specific JSON format, which is intended to be compatible with the format described at [15] It uses the `attr` method to declare a persisted property, and the list of properties defined here are used to convert the JSON data from the server to a model instance and vice-versa.

You'll note that we're using Ruby underscore naming rather than the more standard JavaScript camelCase for the attributes. By default, Ember Data expects the names of the keys in the JSON data to exactly match the properties in the model.[16] Having the attributes also be underscore is the easiest road right now, we'll see some mitigation techniques in the next chapter.

The type argument to `attr` is optional, but if given Ember Data will attempt to convert the data. The default set of types is `boolean`, `date`, `number`, and `string`. Once defined, these attributes behave like any other Ember object property.

15. Though both Ember Data and JSON API are in flux right now, so whether the two actually line up on any given day... who knows?

16. Versions of Ember Data pre–1.0 did do some string inflection, for the reboot, the team seems to have made a deliberate choice to make the default explicit and exact, but make it easy to have custom inflection. See https://github.com/emberjs/data/blob/master/TRANSITION.md for more details.

We now have a route which has retrieved a set of models and associated that set with a controller. Now that controller needs a template to define how to display the data to the browser.

Filename: app/assets/javascripts/templates/index.handlebars (Branch: e_01)

```
<div class="admin_trips span-12">
  {{#each trip in controller}}
    <div class="trip">
      <div class="header">{{trip.name}}</div>
      <div class="dates">{{trip.start_date}} - {{trip.end_date}}</div>
      <div class="price">{{trip.price}}</div>
    </div>
  {{/each}}
</div>
<div class="detail_view span-12">
  <h2>Details</h2>
</div>
```

Sample 2-3-12: Initial template

We're using two features of the Handlebars/Ember combination. We've got an `{{#each trip in controller}}` declaration. Handlebars is less minimal than Mustache when it comes to control flow, meaning it actually uses textual helpers that define what's going on. Here, the `#each` helper, as you might expect, defines a loop. We're using the more verbose form of the `#each` helper, which gives us a variable name, `trip`, to use inside the loop for each looped element. The `in controller` part is the source of the loop. In this case, the use of the variable `controller` causes Ember to automatically use the `content` property of the controller as the source of the loop (the content is effectively synonymous with the model).

Inside the loop, the normal `{{}}` syntax is used to reference properties of each trip in what is superficially similar to the way we might access those properties in Mustache.

At this point, we have a working Ember application. If I haven't left anything out, and you hit `http://localhost:3000/`, you'll get an actual screen:

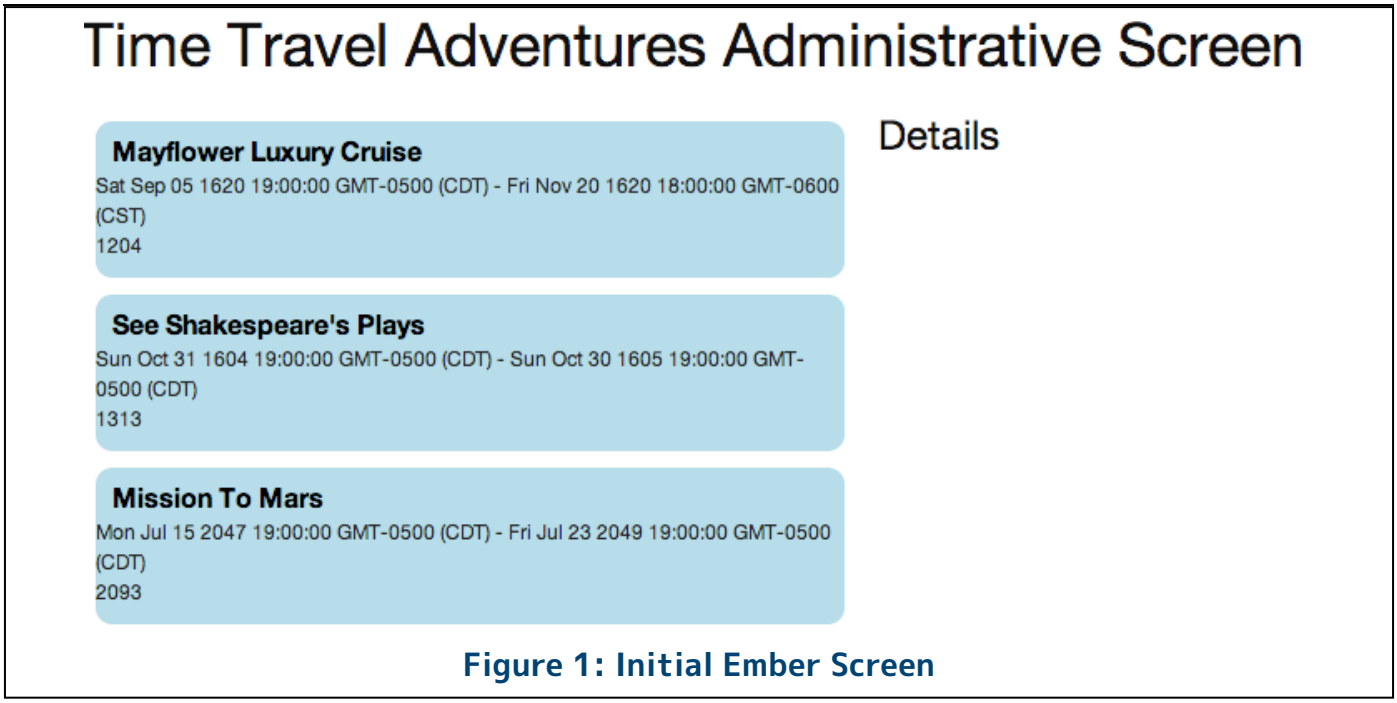

Figure 1: Initial Ember Screen

There's something weird going on though. Here's a snippet of the HTML generated by one iteration of that loop:

```
<div class="trip">
  <div class="header">
   <script id="metamorph-15-start" type="text/x-placeholder"></script>
   Mayflower Luxury Cruise
   <script id="metamorph-15-end" type="text/x-placeholder"></script>
  </div>
  <div class="dates">
    <script id="metamorph-16-start" type="text/x-placeholder"></script>
    Sat Sep 05 1620 19:00:00 GMT-0500 (CDT)
    <script id="metamorph-16-end" type="text/x-placeholder"></script>
    -
    <script id="metamorph-17-start" type="text/x-placeholder"></script>
    Fri Nov 20 1620 18:00:00 GMT-0600 (CST)
    <script id="metamorph-17-end" type="text/x-placeholder"></script>
  </div>
  <div class="price">
    <script id="metamorph-18-start" type="text/x-placeholder"></script>
    <script id="metamorph-18-end" type="text/x-placeholder"></script>
```

```
  </div>
</div>
```

Sample 2-3-13: HTML as emitted by Ember

You don't have to be especially observant to realize that Ember has placed a lot of its own markers in the output.

Why?

To answer that question, try this... Open the console, and type the following three lines into it. All this does is change a property of a model being displayed.

```
window.store = TimeTravel.__container__.lookup('store:main');
trip = window.store.find('trip', 1);
trip.set("name", "A Different Trip");
```

Sample 2-3-14: Type this in the console

The first line of this console snippet is basically an egregious hack because Ember Data 1.0 no longer puts a `store` instance in the global namespace, so we're putting it there. One assumes the Ember community will create an easier way of doing this eventually. Line two actually grabs an individual trip from the store, contacting the server to do so, if needed (if you are playing at home, your ID number may not match). In the third line, we set a property name to a new value.[17]

If your data is set up the same as mine (specifically, if you have a database `Trip` with an ID of 1), you will see the browser change, the trip on the top of the list will have its name changed to, you guessed it, `A Different Trip`.

But all we did was change the property of a model. We didn't touch the view layer or the template or anything like that.

What's happening here is that when the Handlebars template accesses the property of a model, as in `{{trip.name}}`, it not only retrieves the data but it creates a binding between the model property and that particular part of the DOM. Ember uses those `script` tags to keep track of the binding – tags because it can use those tags to mark dynamic sections,

17. Another change for those of you transitioning from pre 1.0 Ember Data. You can't chain the `find` and the `set` calls effectively, because the `find` call now returns a promise. It's fine here because by the time you type in the third line, the object is in place. If this all sounds like gibberish, we'll explain it when we dive deeper into Ember Data.

specifically `script` tags because unlike `div` or `span`, `script` doesn't affect the page layout. Because of that binding, when the model property changes, any associated part of any template that depends on that property automatically changes, as we just saw when we typed into the console.

You have questions about this, no doubt. And we will hopefully get to all of them. But take a moment here to appreciate this. Think of all the time you've spent building JavaScript apps and made this kind of update happen by hand. If you've read book 3, think of all the time we spent there handling `change` events.

All done for you by Ember.

Now, lets use that power to do something a little more elaborate.

We Interrupt This Program For A Surprise Inspection

Before we go on, I'd like to introduce you to something that will make your Ember experience a little more awesome: the Ember Inspector.

The Ember Inspector is a Chrome extension that will give you some visibility into the structure of your Ember application as it appears on the page.

You install the Ember Inspector just like any other Chrome extension, head to the Chrome webstore at https://chrome.google.com/webstore and search for "Ember Inspector", click on the appropriate search result and install it to Chrome. The Inspector will show up in your developer tools, right next to the console.

When you use it, you'll see that it has three panels. The left panel, right now, consists of "View Tree", "Routes", and "Data. The center panel shows results based on which left panel you pick, and the right panel shows details based on a selected left panel.

Each tab provides useful information about the live state of your Ember application. Here's what it looks like with the "Routes" tab selected:

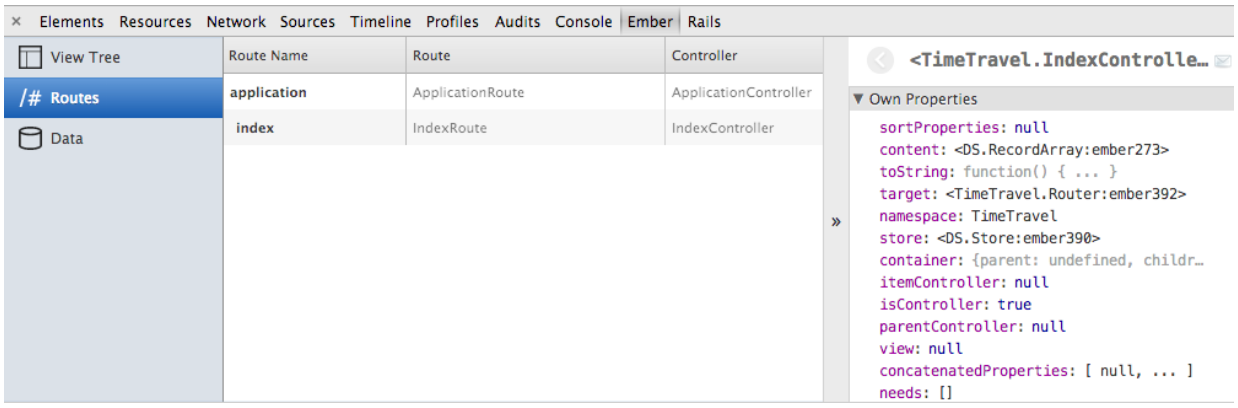

Figure 2: Inspecting Routes

The "View Tree" tab shows a nested list of all the Ember templates used to draw the current display. As you mouse over or click on them, the main display overlays to show what portion of that display is drawn by the selected template (similar to the way the Elements tab shows what part of the screen comes from a given DOM element). A little bit of extra information is included about each template.

The "Routes" tab shows a list of each named route in your application along with the class of that route, and the class of the controller associated with the route. (Though not, seemingly, the pattern that the route matches). Clicking on any of the route or controller names causes that object to show up on the rightmost panel, which displays properties for that object and for any class it inherits from. Clicking on the value of properties that are themselves Ember objects allows you to drill down through the values of the object.

The "Data" tab shows the contents of the data store. It starts showing a list of all model types that have data in the store at that moment. Clicking on one of the types causes a list of all the items of that type to show up, along with a search bar and the ability to filter based on the state of the object (new, modified, or clean). Again, clicking on an individual entry puts that object in the right hand detail pane, allowing you to view and edit the properties of the object. Editing a property triggers the normal Ember updating loops, so you can trigger changes on the screen based on updates in the Ember Inspector.

The Ember Inspector is also new as a finished, public object, so it should continue to get more features over time. Even now, the ability to view and change the running data for you application is powerful.

More Elaborate Property Work

I want to do two things to the admin screen. First, I want to clean up that date and price display to a better format. Second, I want to set this application up so that clicking on a trip title makes that trip the selected element in the right hand side of the page. In both cases, Ember properties are going to make the task easier.

First up is improving the date formatting. We're going to use the Moment.js library, just as we did in Book 3. There are two steps to improving the formatting – converting the incoming date string into a moment object and formatting the moment object for display. As you may recall, we had what was either a spirited debate or a long digression about exactly where to put those display logic methods in Backbone.

In Ember, though, the framework has a definite opinion. Model logic goes in the model, display logic goes in the controller. I'm making an executive decision that converting the string to a Moment.js object is a model thing, but converting the moment to output display is a view thing. So, we're going to create properties called `startMoment` and `endMoment` in the model, but we're going to make the properties `startDateDisplay` and `endDateDisplay` part of the controller.

The model part is pretty straightforward. Oh – we're skipping tests for the moment because we've already seen this logic in past chapters and so we can focus on Ember. Tests will come back with a vengeance, though.

Filename: app/assets/javascripts/models/trip.js (Branch: e_02)

```
startMoment: function() {
  return moment(this.get('start_date'))
}.property("start_date"),

endMoment: function() {
  return moment(this.get('end_date'))
}.property("end_date")
```

Sample 2-3-15: Properties to convert dates to moments

What we have here is a perfectly normal conversion function, with two little quirks. These quirks will allow the great automatic update and binding features of the Ember object model to work.

First up, the `get` method is being used to access the properties. In Ember, anything that has been declared as a property is accessible via `get` (if you try to `get` a property that hasn't been defined, the return value will just be `undefined`.) The argument to `get` is the name of the property. On the flipside, the method `set`, with a property name and a value as arguments, sets a property.

As a quick side note, Ember getters and setters can take compound property names. In other words, instead of typing `trip.get("hotels").get("firstObject").get("name")`, you can type the shorter `trip.get("hotels.firstObject.name")`.[18] If any object along the chain is undefined, then the result of the compound get will be undefined – Ember won't throw an error. This works for setters as well.

At the end of the function definition in the snippet, we decorate the function object with a call to Ember's `property` method. This lets Ember treat what would normally be a method call (as in `trip.startMoment()`) as a property call (as in `trip.get("startMoment")`. Big whoop, you say.

Well, it kinda is. Any method declared to be a `property` is accessible via Handlebars templates. Even better, any property set via the `set` method automatically causes references to that property in Handlebars templates to be updated. Any properties that are themselves dependent on the property that has just been set may also be recalculated, if needed by the template.

Which brings us to the argument in our `property` calls in the code sample. You can call `property` without an argument, which just makes it part of the Ember observed property world. If you call `property` with one or more arguments then you are declaring that the property is derived from those other properties. Again, within the confines of this object, not that big a deal. But it means that if a Trip's `startDate` is modified, then the `startMoment` property will also be marked as changed and any display parts that depend on `startMoment` will also be updated.

There's a subtle distinction here. Ember caches property values and maintains a list of which views and properties are dependent on particular properties. When a property is updated, Ember updates view items that depend on it. Other properties that depend on the initial

18. This is cool, but I'll also play wet blanket and say that if you are doing a lot of chained properties, you probably should look up the Law of Demeter.

property are marked as "dirty" – basically, they have their cache cleared – and are only explicitly recalculated if there's part of the display that is dependent.[19]

Defining `startMoment` is the first part of our dual property. We also need to get a display in. Our initial instinct is to change the template to point at a `startDateDisplay` property:

Filename: app/assets/javascripts/templates/index.handlebars (Branch: e_02)

```
<div class="admin_trips span-12">
  {{#each trip in controller}}
    <div class="trip">
      <div class="header">{{trip.name}}</div>
      <div class="dates">{{trip.startDateDisplay}} - {{trip.endDateDisplay}}</div>
      <div class="price">{{trip.priceDisplay}}</div>
    </div>
  {{/each}}
</div>
<div class="detail_view span-12">
  <h2>Details</h2>
</div>
```

Sample 2-3-16: Index template, pointing at display

Which is a good instinct. But then we're going to want to put the `startDateDisplay` in the `IndexController`, and that's a little bit of a problem.

The problem is context. The `IndexController` is a wrapper around the entire list of trips, not any one individual trip. This seems to leave us with two choices that aren't quite optimal. One choice is that the template could have just `{{startDateDisplay}}`, in which case a `startDateDisplay` property on the controller would be invoked but wouldn't know which trip's date to display. This is fine for properties of the entire trip collection as a whole, like total sales, but doesn't help us for a property of an individual element. Alternately, we could have `{{trip.startDateDisplay}}`, which goes straight to the model without touching the `IndexController` (since, again, the `IndexController` wraps the entire list of trips). Granted, we could put `startDateDisplay` in the model, but that defeats the purpose of trying to keep display logic out of the model.

19. Later, we'll see a couple ways to force recalculation or otherwise trigger action based on a property change.

Luckily, Ember has a feature that allows your array controller to specify a completely different controller class to use for managing the individual items in the array, like so:

Filename: app/assets/javascripts/controllers/index_controller.js (Branch: e_02)

```
TimeTravel.IndexController = Ember.ArrayController.extend({
  itemController: "indexTrip"
});
```

Sample 2-3-17: Specifying the item controller

The specification of an `itemController` tells Ember that individual items of the array should point to an instance of that class when they are referenced individually. Ember does string inflection here, so our naming the `itemController` as `indexTrip`, causes us to be directed to an `IndexTripController`:

Filename: app/assets/javascripts/controllers/index_trip_controller.js (Branch: e_02)

```
TimeTravel.IndexTripController = Ember.ObjectController.extend({

  startDateDisplay: function() {
    return this.get('startMoment').format("MMM D, YYYY");
  }.property('startMoment'),

  endDateDisplay: function() {
    return this.get('endMoment').format("MMM D, YYYY");
  }.property('endMoment'),

  priceDisplay: function() {
    return "$" + this.get('price')
  }.property('price')

});
```

Sample 2-3-18: Our Index Trip Controller defines our properties

And here we get the `startDateDisplay` and `endDateDisplay` we so richly deserve. We've also declared `startDateDisplay` and `endDateDisplay`to be properties, so they are accessible from the template. The properties are dependent on `startMoment` and `endMoment`. The controller has direct access to those properties since it wraps the model – meaning that a change to `startDate` marks `startMoment` as changed and propagates to `startDateDisplay`.

At this point, reloading the page will give us trip summaries with properly formatted dates.

Select a Trip

I've got one more quick thing that will show you some of the power of Ember. When we click on one of the trip names, we want to give that trip control over the detail view on the right hand side. Eventually, we'll put form and other admin stuff there, but for now let's just display the name of the trip.

There are a couple of things we need here. We need a click handler to fire when we click on the header, and we need a way to tell the template to display the newly selected trip's title.

I'm pretty sure this will take fewer lines of code than you expect.

Here's the update to the template:

Filename: app/assets/javascripts/templates/index.handlebars (Branch: e_03)

```
<div class="admin_trips span-12">
  {{#each trip in controller}}
    <div class="trip">
      <div class="header" {{action selectTrip trip}}>{{trip.name}}</div>
      <div class="dates">{{trip.startDateDisplay}} - {{trip.endDateDisplay}}</div>
      <div class="price">{{trip.priceDisplay}}</div>
    </div>
  {{/each}}
</div>
<div class="detail_view span-12">
  <h2>Details</h2>
  {{#if selectedTrip}}
    <h3>{{selectedTrip.name}}</h3>
  {{/if}}
</div>
```

Sample 2-3-19: Index template with action and selected hooks

There are two things to notice in this template. Toward the bottom, we have an `{{#if selectedTrip}}` section. As you might expect, in the case where `selectedTrip` is falsy, the entire block is skipped, if `selectedTrip` is truthy, the block is rendered. Inside the block, we access `selectedTrip.name`.

We've got one other new Handlebars directive in the template, `{{action selectTrip trip}}`. Ember defines `action` as a Handlebars helper, and it sets up an event handler for the tag the `{{action}}` is embedded in. By default the action being watched for is `click`, but you can specify other actions by including an argument of the form `on=doubleclick`.

The first argument to the `{{action}}` directive, in this case `selectTrip`, indicates the handler method to look for. Ember will look in the controller first, then it will look in the `Route` – if you want it to look someplace else you can specify with an argument of the form `target=view`. Any other arguments in the `{{action}}` are passed as arguments to the handler method.

One new twist in recent versions of Ember is that when it looks in the controller or route for a particular action, that action must be defined as part of a nested `actions` object that itself contains the specified action handler. So when somebody clicks on one of those headers, Ember will look in the controller for an `actions` object that defines a `selectTrip` method. Which might look like this:

Filename: app/assets/javascripts/controllers/index_controller.js (Branch: e_03)

```
TimeTravel.IndexController = Ember.ArrayController.extend({
  itemController: "indexTrip",
  selectedTrip: false,

  actions: {
    selectTrip: function(trip) {
      this.set('selectedTrip', trip)
    }
  }

});
```

Sample 2-3-20: Allowing trips to be selected

This couldn't be simpler, we added a `selectedTrip` property, defaulted it to `false`, and all the event handler does is set the property. The rest is handled by Ember, which automatically updates the view in response to the property change.

One thing to point out is that when you specify attributes as part of extending the class, as we've done here with the attributes `itemController` and `selectedTrip`, those become Ember properties of each new instance when you create an instance of the class, the value of the class attribute acting as a default. In this case, each new instance of `IndexController` would

get its own `selectedTrip` property, with the default value `false`. That `selectedTrip` property can then be accessed using Ember's normal `get` and `set` property methods.

Amazingly, that's it. This is working code, and if you click on a handler the name will show up on the right. What happens is the change to the `selectedTrip` property propagates to the view, causing the entire `if` block to redisplay automatically.

That should give you a good quick grasp of how Ember works and what is cool about it. Next, let's go deeper into what Ember can do.

But, Noel, I'm not on Rails, What About Me?

Setting up Ember without a Rails back end is easy – essentially all you need to do is make sure the `ember.js` and (optionally) `ember-data.js` files are loaded as part of your page. You'll also need jQuery and Handlebars as dependencies. Load jQuery and Handlebars before Ember, and Ember Data last.

If you aren't using a build tool that manages the Handlebars templates, you can include the templates in a regular `html` file as a `script` tag with a type `type="text/x-handlebars"` and using the attribute `data-template-name="index"` or whatever the internal Ember name of the template should be.

The Ember Guide on getting started at http://emberjs.com/guides/getting-started/ walks you through this process en route to a standalone application with no server component. You might also be interested in Lineman https://github.com/testdouble/lineman, a tool which manages a number of different setup and management functions for rich JavaScript and CoffeeScript applications.

If you are using a non-Rails server side, the only dependency is that the server needs to respond to a Rails-like RESTful structure and return JSON that matches what Ember Data expects. Over the course of the next few chapters, we'll cover exactly what that means in more detail.

Section 2.4

The Zen of Ember

Here's a quick concept list for Ember, similar to the one I did for Backbone.

1. Ember has five important concepts: routers, controllers, models, views, and bindings.

2. Plus there are templates, which are basically just templates, but are a big part of the way controllers and views work.

3. All Ember objects are made up of properties that are read and written via the Ember object system. Properties can be defined as being dependent on other properties. Methods can also be set to automatically fire when a property changes.

4. In Ember, models manage non-display business logic. The Ember Data package allows you to create subclasses of Ember Object that can manage persisted data. These models can talk to a data store, often a RESTful server.

5. A controller can wrap a model or an array of models, and associates that content with a template.

6. In Ember, controllers are where you put display-specific logic. The are the default location for properties and actions referenced in a template, and properties not found in the controller automatically delegate to the associated model.

7. Templates can reference properties. Ember looks to resolve these properties in the template's context, which is usually a controller.

8. A binding creates a link between properties in two different objects, such that when one property changes, the other changes in sync. A property name that ends in `binding` takes a string value that points to a property and automatically binds the two properties. So, a property defined as `targetBinding` = `content.name` binds the property `target` and the property `content.name`.

9. When a template references a property, that reference is a binding. When the underlying model value changes, the template automatically updates.

10. A route connects a state of the application with a URL, a controller, a model (or array of models), and a template.

11. Internal links within the application are resolved by the router object that matches the URL pattern with the route.

12. A view in ember is used to either encapsulate a reusable widget, allow for custom event handling, or split a long template.

13. Templates can define handlers for actions. These handler methods are assumed to be in the controller object, but can be targeted at any object that is visible to the template.

Ember in CoffeeScript

You can write Ember in CoffeeScript, however I think it's fair to say that some of Ember's idioms are awkward in CoffeeScript. A couple of tips:

- Your `Application` object must be explicitly placed in the global namespace, as in `@TimeTravel = Ember.Application.create()`. Otherwise, other classes won't be able to be added to the application's namespace
- To the best of my knowledge, you cannot extend an Ember class using CoffeeScript's `class` mechanism (though I think some people have created hacked versions of CoffeeScript that can do it), you must still use `Ember.Object.extend`.
- Creating a property or binding is a little tricky, since the JavaScript version in Ember depends on being able to use dot-notation at the end of a function. CoffeeScript's whitespace-significant syntax doesn't give you a place to hang that extension, so you have do to it yourself by surrounding the function call in parenthesis:

```
fullName: (->
        "#{this.get('firstName')} #{this.get('lastName')}"
).property('firstName', 'lastName')
```

Sample 2-4-1: Ember properties in CoffeeScript

Chapter 3

Admin Stuff

By this time, you know what is coming next. You count to ten, but even before you finish, there's a new email...

> Dear you,
>
> Don't get me wrong, I really like what you've done with the admin site. But I'd really like it if we could get some, you know, admining going on. Showing totals, letting people edit things. Please? We have a bunch of administrators in the 22nd Century that can't wait forever.
>
> See ya,
>
> Doctor What

This time around, we're going to augment our page by displaying some calculated totals. We're also going to take that detail sidebar that we created and put some form editing functionality in it. Along the way, we're going to check out Ember's array calculation properties, see some Views, and we are also going to start testing this stuff.

Section 3.1

Calculating totals

The first thing we want to do is display an overview of the financial totals of each time travel trip. As you may recall from earlier books, each trip has a price, plus there are hotel options that are priced per night, plus there are tour extras that are optional. We want each trip detail panel to display the total amount of each component that was purchased, as well as a total for the trip as a whole.

In the interest of keeping the data model simple, we're just going to place a single order count field on the `trips`, `hotels`, and `extras` tables rather than having to process a bunch of

orders.[20] (If you got the code from an early version of this book, you'll want to do a `rake db:reset`, since I've changed the schema and the seed file.)

Data and Associations

If we want to display the total revenue of a trip with all its hotel and extra options, that strongly implies that we need models representing the hotel and extras. And we probably also need an association between a trip and each of those parts.

The Ember Data library allows us to create associations and automatically build the client side data structures based on an expected JSON data structure. First, of course, we need to define client models for the hotel and extra objects. You may notice a certain similarity between the two definitions:

Filename: app/assets/javascripts/models/hotel.js (Branch: e_04)

```
TimeTravel.Hotel = DS.Model.extend({
  name: DS.attr("string"),
  description: DS.attr("string"),
  price: DS.attr("number"),
  nights_ordered: DS.attr("number"),
  trip: DS.belongsTo("trip"),
```

Sample 3-1-1: The attribute definition of the hotel model

Filename: app/assets/javascripts/models/extra.js (Branch: e_04)

```
TimeTravel.Extra = DS.Model.extend({
  name: DS.attr("string"),
  description: DS.attr("string"),
  price: DS.attr("number"),
  orders: DS.attr("number"),
  trip: DS.belongsTo("trip"),
```

Sample 3-1-2: The attribute definition of the extra model

Most of this we saw last chapter, the only new bit is that last declaration that the `trip` attribute is of type `DS.belongsTo("trip")`. This declares that the `Hotel` and `Extra` classes have a one-to-many relationship with a `Trip` – note that we are using the shortcut name of the class,

20. That processing would most likely take place on the server, and therefore isn't interesting to us here. If you like, you can assume that in a parallel universe, somebody reading *Master Space and Time With Rails* has generated this data for us.

Ember will inflect to the real class definition.[21] On the other side, we augment the `Trip` class to declare the `hasMany` portion of the relationship.

Filename: app/assets/javascripts/models/trip.js (Branch: e_04)

```
  hotels: DS.hasMany('hotel'),
  extras: DS.hasMany('extra'),
```

Sample 3-1-3: The declarations of the has many relationships in a trip

Ember Data only offers `belongsTo` and `hasMany` as association options – to declare a many-to-many relationship, you simply declare the `hasMany` attribute on both sides of the relationship.[22]

The actual mechanics of the relationship within the Ember world are straightforward. From the point of view of the hotel, saying `hotel.get('trip')` returns the associated `Trip` object. From the point of view of the trip, saying `trip.get('hotels')` returns an array of `Hotels`.

Worth noting in passing: Ember augments the JavaScript array prototype with a bunch of helper methods – hey, it's the zillion and first implementation of `map` – including some helper methods that are aware of Ember properties.

Where the Ember Data association gets interesting is in how Ember Data derives the associated objects from the JSON data it receives from the server. Ember Data expects to receive JSON data in a particular structure.

First up, Ember Data expects all data for an object to be inside a root property whose name matches the class of the object being described. So, part of a response defining a single trip might look like this:

```
{
  "trip": {
          "name":"Mayflower Luxury Cruise",
          "description":"Blah",
          "end_date":"1620-11-21",
          "price":1204.0
```

21. This is a change in Ember Data 1.0 as compared to previous versions.

22. Rails programmers note that, unlike Rails, the hasMany declaration in Ember takes the singular version of the class name.

```
  }
}
```

Sample 3-1-4: Partial JSON for a single trip

The root, `trip` is the name of the class being created, and the key/value pairs inside it represent attributes and values. The keys must match the attribute names as listed in the class declaration, although you can write code Ember side to munge the JSON payload before conversion to Ember objects.

If the JSON payload is an array, then the root name is plural, and the internals are an array – again, most of the attributes are not shown here, so that you can more clearly see the structure.

```
{
  "trips": [
        {"name": "Mayflower Luxury Cruise"},
        {"name": "See Shakespeare's Plays"}
  ]
}
```

Sample 3-1-5: Partial JSON for multiple objects

How does Ember Data handle associations? The most important thing to start with is what Ember Data does *not* do. Namely, Ember Data does not embed the entire subordinate object inside the main object. Ember Data represents all relationships as an id or array of ids. On the trip side, listing many subordinate objects might look like this:

```
{
  "trip": {
          "name":"Mayflower Luxury Cruise",
          "extras": [1, 2, 3],
          "hotels": [4, 5, 6]
  }
}
```

Sample 3-1-6: Partial JSON for a single trip with extras

From the other side, where there's only one object, the convention is the other object's class name (converted to underscores) with `_id` at the end:

```
{
  "hotel": {
          "name":"Deluxe Room",
          "trip_id": 1
  }
}
```

Sample 3-1-7: Partial JSON for a single hotel with a trip

When Ember receives the JSON with an association, it does not immediately retrieve the associated object. It will do so, lazily, when the the associated object, or a property of the associated object is requested. When that happens, Ember Data will make an Ajax request to the server for more JSON[23]. If Ember Data is requesting a single object, the request will have a query parameter `id=3`, or whatever the number is. Rails will convert that RESTful request to the appropriate controller's `show` action.

If the request is for multiple objects, the request will be of the form `ids[]=5&ids[]=6&ids[]=7` (only, you know, URL Encoded). Rails will convert that internally to `ids = [5, 6, 7]` and the request will be routed to the controller's `index` action, which you need to be aware of, as there's a good chance your `index` action isn't checking for `ids` as a parameter.

Ember Data will try to cache objects on the client so as to minimize server trips.

There are some implications to the way Ember manages lazy loading for incoming data. First off, there's a trip to the server involved, and that is not exactly instantaneous even for a time-travel application. Luckily, Ember's data objects are aware that they are initialized with a delay, and know when they are between a request and the data having been returned. Technically, the request to Ember Data returns a *promise* object that allows you to define your own response when the data actually shows up above and beyond what Ember Data will do. We'll talk more about promises in Ember in a few chapters.

When the object's data arrives from the server, all properties fire, and any bindings or templates that depend on the data will automatically update. Once we get all this wired together, if you have a slow server for the Time Travel sample app, you might see the hotel and extra information flicker in after page load because Ember is waiting to update until the data is in place.

23. Strictly speaking, we're talking about the behavior of the RESTAdapter for Ember-data. Other adapters will respond differently.

Usually, you'd rather not go back to the server to get the subordinate object's data, you want a parent object and it's entire object tree to enter your application at once. Ember will do that for you if you sideload the other objects underneath their own top-level element, the resulting JSON looks something like this. (In fact, if you don't side load object data, you may need to explicitly declare the association to be asynchronous, by using `DS.hasMany('hotels', { async: true })` – I admit, though, that I'm not fully clear on how the new Ember Data is managing the asynchronous part).

```
{
  "trip": {
          "name":"Mayflower Luxury Cruise",
          "extras": [1, 2, 3],
          "hotels": [4, 5, 6]
  },
  "hotels": [
          {"id": 4, "name": "Deluxe"},
          {"id": 5, "name": "Average"},
          {"id": 6, "name": "Horrid"},
  ]
}
```

Sample 3-1-8: Partial JSON for a single trip with sideloaded data

When Ember gets this data, it is able to take the `Hotel` objects and associate them all with the given `Trip`. If, however, this data was all there is, then Ember would still go back to the server if asked to return the `Extra` objects.

Creating JSON Data In Rails

Ember doesn't care about the specifics of the server-side application beyond it's ability to match the data format we just described.[24] However, in our specific application we're using Rails, and I do want to cover exactly what I did on the Rails side to make this work. For one thing, it's an important part of how this particular application works, and for another, it's a good general tool to have in your pocket if you are using Rails as a back end.

24. The mapping between the format and the eventual Ember objects is governed by Ember's `DS.RESTAdapter` object. We're not going to get into it too much, but you can write your own adapter if your data is coming from a different source.

We're using a gem called `active_model_serializers`, which you can find online at https://github.com/rails-api/active_model_serializers. The goal of the `active_model_serializers` gem is to provide a consistent way to define JSON output for a Rails application that is acting as a web service providing data to clients. Which – what are the odds – is exactly what we need.

The basic idea is that you define a serializer for each of your ActiveRecord models. (Just as with RESTful resources in general, you don't have to have a one-to-one relationship between models, database tables, and serializers, but that is the most common relationship.) The serializers go in the `app/serializers` directory. Each one is named after the model it is serializing and inherits from `ActiveModel::Serializer`. Once the serializer is in place, it intercepts any attempt to convert the object to JSON using Rails `to_json` or `render :json => obj` mechanisms.

At its simplest, the serializer contains just a list of attributes that are part of the JSON payload:

```
class TripSerializer < ActiveModel::Serializer
  attributes :name, :description, :start_date, :end_date,
   :image_name, :slug, :tag_line, :price, :location,
   :activity, :orders
end
```

Sample 3-1-9:

A symbol in that `attributes` list is assumed to be the name the name of a method of the serializer itself. If the serializer doesn't define a method by that name, then the process delegates to a method by that name in the underlying model. This allows you to decorate the model with derived data that is only needed by the client. If you need to access an attribute of the underlying model from the serializer, you do so using the property `object`. For example:

```
def length_in_days
  object.end_date.to_date - object.start_date.to_date
end
```

Sample 3-1-10: This is just an example, we're not really going to do this

If we want objects associated with the model to be part of the serialized download, we can just add them using the serializer methods `has_one` and `has_many`:

```
class TripSerializer < ActiveModel::Serializer
  attributes :name, :description, :start_date, :end_date,
   :image_name, :slug, :tag_line, :price, :location,
   :activity, :orders
  has_many :hotels
  has_many :extras
end
```

Sample 3-1-11:

The serializer doesn't care whether a `has_one` relationship is a Rails `belongs_to` or a Rails `has_one`. In either case it will behave the same: embedding the associated object as a nested value in the JSON. A `has_many` will embed the associated objects as an array of nested values in the JSON.

That's not quite what we want, however. As I mentioned earlier, Ember doesn't like nested objects, and would rather have a list of IDs.[25] Luckily, `active_model_serializer` is flexible enough to manage this case using the `embed` method. You call `embed` as part of the serializer, the argument is either `objects`, which is the default nesting behavior, or `ids`, which replaces the array of nested objects with a list of IDs. So the serializer that we're actually going to use looks like this:

Filename: app/serializers/trip_serializer.rb (Branch: e_04)

```
class TripSerializer < ActiveModel::Serializer
  embed :ids, include: true
  attributes :name, :description, :start_date, :end_date, :id,
      :image_name, :slug, :tag_line, :price, :location, :activity, :orders
  has_many :hotels, key: :hotels
  has_many :extras, key: :extras
end
```

Sample 3-1-12: The full trip serializer

This serializer results in a list of the IDs of the associated objects being part of the JSON, as in example 3–1–8 above, along with all the associated `Hotel` and `Extra` records, which are side loaded into the JSON – the `key` option ensures they go in to the JSON as `hotels` and not

25. Ember Data 1.0 has an API for converting incoming JSON to the format you want before creating objects. You can use this to handle embedded objects if you are determined. Again, the transition guide at https://github.com/emberjs/data/blob/master/TRANSITION.md will help.

hotel_ids[26]. When Ember receives that JSON, it will create the Trip objects and all of the subordinate objects in one shot. This behavior is triggered in active_model_serializers by the embed line which reads embed :ids, :include => true, which will cause the serializer to include the associated objects as separate, non-nested elements of the JSON, while still including the list of IDs in the original model.

We could also fix up the underscore vs. camelCase issue by explicitly declaring camelCase attributes in the serializer, but honestly, I think that'll just be confusing at the moment.

That's all we need to know about active_model_serializers for our purposes. The gem has a couple of other tricks up its sleeve, check out https://github.com/rails-api/active_model_serializers for more details.

Let's wrap up the Rails code for those of you that are following along. The serializers for Hotel and Extra are very straightforward:

Filename: app/serializers/extra_serializer.rb (Branch: e_04)

```
class ExtraSerializer < ActiveModel::Serializer
  embed :ids, include: true
  attributes :id, :description, :price, :name, :orders
  has_one :trip, key: :trip
end
```

Sample 3-1-13: The full extra serializer

Filename: app/serializers/hotel_serializer.rb (Branch: e_04)

```
class HotelSerializer < ActiveModel::Serializer
  embed :id
  attributes :id, :description, :price, :name, :nights_ordered
  has_one :trip, key: :trip
end
```

Sample 3-1-14: The full hotel serializer

We also now need Rails controllers that that will send this JSON data back to the client. The TripController, for the moment at least, only sends back a set of all trips, and therefore can just have an index method like so:

Filename: app/controllers/trips_controller.rb (Branch: e_04)

26. Ember Data will probably accept the form hotel_ids at some point in the near future.

```
  def index
    @trips = Trip.all
    render json: @trips
  end
```

Sample 3-1-15: TripsController sending all trips as JSON

When we render the output as JSON using `render json: trips`, Rails will use the `active_model_serializers` gem to create the JSON for each `Trip` using the `TripSerializer`.

The `HotelsController` and `ExtrasController` are a little different, since they may need to return a restricted set of items based on ids:

Filename: app/controllers/hotels_controller.rb (Branch: e_04)

```
class HotelsController < ApplicationController

  respond_to :html, :json

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

end
```

Sample 3-1-16: HotelsController sending all trips as JSON

Filename: app/controllers/extras_controller.rb (Branch: e_04)

```
class ExtrasController < ApplicationController

  respond_to :html, :json

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

end
```

Sample 3-1-17: ExtrasController sending all trips as JSON

Okay, with that server-side code in place, we can get back to Ember and actually using the data.

Section 3.2

QUnit. Because Apparently We Need Another Testing Library

Until now, this book has been happily moving along using the Jasmine library for testing. At the time I started writing the book, Jasmine seemed reasonably popular, and I personally like the syntax.

However, one of the risks involved in writing a technical book over a two year time cycle is that occasionally a tool you choose to feature will not be forever applicable to all the places you want to use it. And that's happened here.

The Ember core team has mentioned on several occasions that QUnit (http://www.qunitjs.com) is their preferred testing library. That was fine as long as it just meant that they used QUnit for their internal testing. But as the Ember Testing tools have come into being as a means of integration testing Ember code, Ember Testing clearly prefers the use of QUnit – right now, just as a first among equals kind of thing, but eventually with the goal of writing functionality specific to QUnit.

And so, we'll use QUnit to test Ember in this book. In this section, we'll set up QUnit, first using Rails, then showing how to set up for non-Rails projects. In the next section, we'll show our first QUnit model tests, and in the next chapter, we'll use the Ember Testing library to write integration tests that actually interact with our Ember controllers and views.

QUnit Setup

Happily, QUnit is relatively drama-free to set up. Rails version first. Because.

There's a gem. We include it in the `Gemfile`. In the interests of sanity, you can assume we pulled all the Jasmine gems from the list.

```
gem 'qunit-rails'
```

Sample 3-2-1: Including qunit-rails in the gem file

After installing the gem with `bundle install`, we generate some files:

```
$ rails generate qunit:install
```

Sample 3-2-2: Command line to install qunit

This gives us two files, one of which is at `test/stylesheets/test_helper.css`. If you read and remember the Jasmine setup discussed somewhere back in book 1, this is similar, it just makes sure that the application's styles are visible to QUnit.

Filename: test/stylesheets/test_helper.css (Branch: e_04)

```
/*
 *= require application
 *= require_tree .
*/
```

Sample 3-2-3: The test helper css for our QUnit setup

The other file is at `test/javascripts/test_helper.js` and consistently with the various `spec_helper` files we've seen, contains code that is run when tests are started. At the moment, we're just going to plug it in to the asset pipeline:

Filename: test/javascripts/test_helper.js (Branch: e_04)

```
//= require application
//= require_tree .
//= require_self
```

Sample 3-2-4: Asset pipleline setup for our QUnit tests

Adding the asset pipeline just makes our existing JavaScript code available to the test.

In order to write tests, we include testing code inside the JavaScript files loaded in `test_helper.js`, which in our case means anywhere in the `test/javascripts` subdirectory.

Non-Rails QUnit setup

The general use of QUnit is actually really simple, and the QUnit home page http://www.qunit.com lays out the basics like so:

```
<!DOCTYPE html>
<html>
<head>
```

```
  <meta charset="utf-8">
  <title>QUnit Example</title>
  <link rel="stylesheet" href="/resources/qunit.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="/resources/qunit.js"></script>
  <script src="/resources/tests.js"></script>
</body>
</html>
```

Sample 3-2-5: A minimal QUnit setup

All you really need for your test runner HTML page is the following:

- Include a link to the `qunit.css` stylesheet
- Include a link to the `qunit.js` library code
- Have `div` elements with ids `qunit` and `qunit-fixture`
- Link in the files with your tests.

Then launch the page in a browser and the screen will show up as in the screenshot following shortly.

Running QUnit

Although the non-Rails version of QUnit can be just loaded into a browser, with our Rails server running, the `qunit-rails` gem gives us access to our tests at `http://localhost:3000/qunit` (or whatever your development setup server is...) After we add in the tests we'll write in the next little bit, that screen looks a little something like this:

Figure 3: QUnit Screen

All these tests are currently passing – if there was a failure, the failing tests would display with an error message and a stack trace.

The default QUnit runner has some nice UI features along the top.

- *Hide passed tests*, if checked will hide passed tests from the list display.
- *Check for globals*, if checked will cause any test that puts a variable in global scope to fail.
- *No try-catch*, if checked will not catch test failures inside QUnit, but will pass them along to the browser itself.

On the right of the top bar is a pulldown that allows you to limit the test running to a specific module. Which I realize would be a more useful piece of information if we had talked about modules yet, but bear with me, we can't talk about everything at once.

Anyway, each individual test has a little "Rerun" link next to it that causes only that test to be run. Pro tip: if you do start rerunning individual tests, you can get back to running the entire test suite by clicking on the QUnit logo in the upper left.[27]

27. Knowing that would have saved me a few minutes of frantic Googling when suddenly I was only seeing one test executing.

QUnit in Some Kind of Small Package

We're going to cover the very basics of QUnit here – the next section will show simple examples, and any more advanced feature will be covered if and when we find them useful. You are invited to checkout http://api.qunitjs.com for more detail.

The basic test method in QUnit is aptly named `test`. In its most commonly used form, it takes a string argument which is a description, and a functional argument that is the test itself, as in:

```
test("basic arithmetic should work", function() {
  equal(2 + 2, 4);
});
```

Sample 3-2-6: Basic addition in QUnit

Inside the `test` function, QUnit defines eight assertion methods, of which the most basic and widely used are `equal`, and `ok`.

The `equal` assertion, as shown in our little snippet compares two values, the first of which is the actual value, and the second of which is the actual value. As with many test packages, you have an optional third argument for an error message – in practice I find that the stack trace is usually enough of me to find the failing test.

QUnit's `equal` uses the JavaScript `==` operator to compare its two arguments, failing if the are not equal according to that operator. QUnit also defines some related assertions, such as `notEqual`, which is the reverse of `equal` and `strictEqual`, which uses the JavaScript triple-equal `===` operator.[28]

More generically, there's the `ok` assertion which takes one argument and passes if that argument resolves to a truthy value. An optional second argument can be used as an error message. For whatever reason, there is no `notOk` operator.

In QUnit, tests can be combined into modules. Unlike Jasmine, tests are not subordinate to modules. Rather, the `module` function is called, and any call to `test` after that `module` call belongs to that module until `module` is called again. Typical practice is to start a new test file with a single call to `module` that then places all the tests in that file together in a single module.

28. And yes, there's a `notStrictEqual` as well.

Module takes one required argument, a string which is the name of the module and is used to identify the module in the test runner. The optional second argument is a normal JavaScript object expected to have two properties, `setup` and `teardown`. If those properties exist, then their values are expected to be functions. The function value of `setup` is called before any test in the module is run, the value of `teardown` is called after.

That's the basics of QUnit, there are some more features for dealing with asynchronous tests that we'll get to at some point, but first lets see some real examples of TDD with QUnit.[29]

Section 3.3

Using Ember Data and Associations

We now return to our actual Ember book, already in progress...

We'd like to do a few things on our screen. We want the detail section on the right hand side of the screen to list the individual hotels and extras, their price, orders and revenue.

In the main section, I'd like the orders to be sorted high-revenue to low revenue, and have that data change automatically when we enter a new order (or at least when we fake entering a new order).

And now we actually have some testable logic. We can unit test the revenue calculations of the models, and we can also acceptance test that all that logic, plus the sort logic, makes its way to the screen.

Lets start with the unit tests.

We have a very similar test for our `Hotel` and `Extra` models, we want them to be able to calculate their total revenue from the order count and unit price. In deference to the limited number of pixels available to me, I'll only show one of the tests – here's the `Hotel` version:

Filename: test/javascripts/model/hotelTest.js (Branch: e_04)

```
module("Hotels", {
  setup: function() {
```

29. If you like Jasmine's more BDD-style syntax, the Pavlov extension to QUnit, http://github.com/mmonteleone/pavlov provides a similar syntax. I do like BDD-style syntax, but the last thing either of us need right now is another library to discuss.

```
    store = TimeTravel.__container__.lookup('store:main');
  },

  teardown: function() {
    store = null;
  }
});

test("price calculation", function() {
  Ember.run(function() {
    hotel = store.createRecord('hotel', {price: "100", nights_ordered: "3"})
  });
  equal(hotel.get("revenue"), 300);
});
```

Sample 3-3-1: First test for hotel pricing

Okay, granted we're really just testing multiplication here. That said, there are a couple of setup issues worth pointing out.

The file starts with a QUnit `module` function, declaring this to be the `Hotels` module. We've actually got a little bit of setup and teardown here. In the `setup` function, we're accessing the Ember Data store using the same incantation that we used back in Chapter 2 to get a global store in the console. We need this, because Ember Data 1.0 no longer exposes the store in the application namespace directly. I'll be frank – this looks hack-ish to me, but it works, and I suspect I'll find a more idiomatic version before long. In the `teardown`, we just clean up the store variable.

In the `test` itself, we create a new instance of `TimeTravel.Hotel` using the Ember Data method `store.createRecord`, which takes as its first argument the Ember Data short name of the model to create an instance of, and as the second argument a set of key value pairs corresponding to the attributes of the object to be created.

One important quirk is that `createRecord` is specific to `DS.Model` and its subclasses. If we were creating an ordinary `Ember.Object` that is not backed by a persistent store, the method we would call is `create`. Don't worry, if you use `create` to build a `DS.Model`, Ember will give you an error message.[30]

Another thing to notice is that the `createRecord` calls are wrapped inside calls to `Ember.run`. Strictly speaking, that's not necessary yet, though it will be when we explicitly turn on testing mode in a few pages for our integration test. The short form is that `Ember.run` is an explicit trigger for Ember to go through it's dependency management loop, and that during testing, it's nice to be able to control that execution. We'll have a longer explanation when we talk about integration testing.

Anyway, the nice feature here is that we can easily build partial models for ease of testing and without creating Jasmine mock objects – specifically, we can override calculated properties of Ember objects. Hold that thought, we'll come back to it later.

After that, the actual code is largely just multiplication, though we're implementing `revenue` as a derived property...

Filename: app/assets/javascripts/models/hotel.js (Branch: e_04)

```
  revenue: function() {
    return this.get('nights_ordered') * this.get('price');
  }.property("nights_ordered", "price")
});
```

Sample 3-3-2: Hey, our code is multiplying!

Filename: app/assets/javascripts/models/extra.js (Branch: e_04)

```
  revenue: function() {
    return this.get('orders') * this.get('price');
  }.property("orders", "price")
});
```

Sample 3-3-3: Hey, our code is multiplying!

We also want our trip to be able to calculate revenue. In this case, the trip has revenue for itself – it's own price times orders, plus the sum of the revenue from all of its hotels and extras. That's four tests, here they are in one shot...

Filename: test/javascripts/model/tripTest.js (Branch: e_04)

```
module("Trips", {
  setup: function() {
    store = TimeTravel.__container__.lookup('store:main');
```

30. Ember also has a mechanism for creating fixture data using a `FixtureStore`. We'll also get to that when we talk about integration testing.

```
    Ember.run(function() {
      trip = store.createRecord('trip', {price: 100, orders: 3});
    });
  },

  teardown: function() {
    store = null;
  }
});

test("calculate the total revenue", function() {
  equal(trip.get("revenue"), 300)
});

test("calculate total revenue for all hotels", function() {
  Ember.run(function() {
    trip.get('hotels').pushObject(
        store.createRecord('hotel', {revenue: 500}));
    trip.get('hotels').pushObject(
        store.createRecord('hotel', {revenue: 600}));
  });
  equal(trip.get("totalHotelRevenue"), 1100);
});

test("calculate total revenue for all extras", function() {
  Ember.run(function() {
    trip.get('extras').pushObject(
        store.createRecord('extra', {revenue: 500}));
    trip.get('extras').pushObject(
        store.createRecord('extra', {revenue: 600}));
  });
  equal(trip.get("totalExtraRevenue"), 1100);
});

test("calculate all revenue", function() {
  trip.set('totalHotelRevenue', 500);
  trip.set('totalExtraRevenue', 500);
```

```
  equal(trip.get('totalRevenue'), 1300);
});
```

Sample 3-3-4: Revenue tests for trips

In this case, we're using our `module` function to declare a `setup` in which we create a `TimeTravel.Trip` object from the store with a given price and number of orders.

There's one important feature of these tests worth pointing out. In the last three tests, we're shortcutting the Ember calculated property system and directly setting a derived property. So in the `Hotel` test, we have `TimeTravel.Hotel.createRecord({revenue: 600})`, which directly sets the `revenue` property that we just defined a page or so ago. In the last test, we set derived properties of `Trip` directly with lines like `trip.set('totalHotelRevenue', 500);`.

Why?

A few reasons to do it, and one reason not to. On the plus side, test setup is easier – in this case, we're setting one property, `revenue`, rather than two, `price` and `nights_ordered`. More complicated properties would have even more dependent variables. We're also isolating the test from the details of the calculation – the `Trip` doesn't really care how the `Hotel` counts revenue, so the test shouldn't care either.

In essence, we're mocking our own Ember object by creating a stub value for what would otherwise be a calculation. Which means that the downside is similar to the downside of using a mock object directly: there's no guarantee that the mocked value actually exists on the client. For example, the Ember `createRecord` method does not test for the existence of a properties that are included in its options argument – and rightly so, because Ember is better with that flexibility.

In practice, this means that if you are using this mechanism as an effective stub of an associated object, you need to test the property on the associated object, and also have an end-to-end test that covers the relationship. We'll show the end-to-end test in a moment.

Meantime, here's the relevant part of the `Trip` class that enables those revenue calculations. There's two new features here:

Filename: app/assets/javascripts/models/trip.js (Branch: e_04)

```
  revenue: function() {
    return this.get('orders') * this.get('price');
  }.property("orders", "price"),
```

```
totalHotelRevenue: function() {
  return this.get('hotels').reduce(function(runningTotal, item) {
    return runningTotal + item.get('revenue');
  }, 0);
}.property("hotels.@each.revenue"),

totalExtraRevenue: function() {
  return this.get('extras').reduce(function(runningTotal, item) {
    return runningTotal + item.get('revenue');
  }, 0);
}.property("extras.@each.revenue"),

totalRevenue: function() {
  return this.get('revenue') +
      this.get('totalHotelRevenue') + this.get('totalExtraRevenue');
}.property('revenue', 'totalHotelRevenue', 'totalExtraRevenue'),
```

Sample 3-3-5: Revenue methods for trip

I count two elements of this code that we haven't seen before. First is the call to `reduce` in the two methods that sum total revenue over the hotels and extras. The `reduce` method is one of a number of enumeration-style methods that Ember mixes into the JavaScript prototype for Arrays, meaning that they can be used with any Array in an Ember system.

The general scope of these methods is similar to the list of enumeration methods defined by, say Underscore.js, but there are a couple methods that are Ember-specific. For example, you have `map`, which takes a function argument and returns a new array with the function applied to each element of the original array but you also have `mapProperty`, which takes the name of a property as an argument and returns a new array containing the value of that property for each element of the original array (simlar to the Underscore.js `pluck` method).

In this example, `reduce` works similarly to how it works in other JavaScript libraries (or in Ruby). It takes one functional argument, which itself takes two arguments. Those two arguments are the running value of the reduction and the next item. In turn, each item is passed to the inner function and the returned value becomes the first argument to the next iteration. Finally, the `0` used as a second argument to the `reduce` method itself is the initial value of the first argument of the inner function. Which is a long way to go to basically say

that we're adding all the values together and returning the sum. But, you know, with extra Ember-ocity.

The other new feature is that weird description of the property dependency as `hotels.@each.revenue`. This syntax is Ember's way of solving the problem of how to manage a dependency against a list of properties. In other words, the value of the `totalHotelRevenue` property changes if you add a new hotel, but it also changes if an existing hotel has a change in its individual revenue. The `@each` is effectively a meta-property declaring exactly that multifaceted dependency. Saying the property is dependent on `hotels.@each.revenue` means that the the property will be marked as dirty if any of the hotels in the list have their revenue updated.

There's one similar meta-property that applies to changes to the list itself rather than properties of individual elements. If we declared the property as dependent on `hotels.[]`, then any change to the composition of the list would mark the property as changed, the `[]` being our meta-property.

Section 3.4

Moving On

That covers the model data side of our admin panel. In the next chapter, we'll talk about the view and integration testing.

Chapter 4

Ember testing with Ember-testing

Just this once, we'll spare you the Dr. What letter, okay?

So, here's the deal

NOTE

Hi, it's Noel here, breaking out of author-explain-y voice, and into just plain me voice.

If you've been following along on previous versions, you know that this section has a history of being awkward and strange. With this, version, and with the ongoing help of Toran Billups[^1], Eric Kidd[31], Jo Liss, and Erik Bryn – who, by the way, have no idea they were helping me, so thanks for putting great stuff online where I can find it – we are finally using the Ember Testing library for these tests. Erik is suggesting that the Ember Testing api is still a work in progress, so I'll be keeping this up to date for a while.

END OF NOTE

Section 4.1

Integration testing

With the data out of the way, let's turn our attention to the view layer. I'd like to start with an integration test, by which I mean a test that attempts to test Ember's logic against actual DOM elements being placed in an actual testing DOM.

31. http://gist.github.com/emk

I can think of three strategies for integration testing Ember. Probably there are more.

- Use a completely external tool like Capybara. In this scenario, we would hit our application completely externally in a headless JavaScript engine, simulate user actions and test the results.
- Moving one step in, from inside QUnit, we could start up an instance of our Ember app, and simulate routing to a URL, then test the results using QUnit matchers.
- Moving one more step in, from inside QUnit, we could set up an Ember controller, associate it with a view, render it, and test the results using jQuery matchers and QUnit assertions.

The first option is possibly a little outside the scope of this book, since it would require introducing another testing tool. That said, external end-to-end testing is a good part of your complete testing breakfast, and is worth some study. Perhaps in a different book.

Ember-testing is somewhere close to the second option. We point our test to a particular start of the application by passing it a URL. Ember-testing defines a handful of helpers to allow us to interact programmatically with the application, and then we can use QUnit to assert facts about the state of the application.

Section 4.2

Ember-testing setup

To make this work, we need to add some code to our QUnit `test_helper.js` file for setup.[32]

Filename: test/javascripts/test_helper.js (Branch: e_04)

```
//= require application
//= require_tree .
//= require_self

document.write('<div id="ember-testing-container"><div id="ember-testing">' +
    '</div></div>');
```

32. For a little bit of a bibliography, check out the Ember-testing guide at http://emberjs.com/guides/testing/integration/ and Erik Bryn's sample testing application https://github.com/ebryn/bloggr-client-rails.

```
document.write('<style>#ember-testing-container { position: absolute; ' +
    'background: white; bottom: 0; right: 0; width: 640px; height: 384px; ' +
    'overflow: auto; z-index: 9999; border: 1px solid #ccc; } ' +
    '#ember-testing { zoom: 50%; }</style>');

TimeTravel.rootElement = '#ember-testing';

TimeTravel.ApplicationAdapter = DS.FixtureAdapter.extend();

TimeTravel.setupForTesting();
TimeTravel.injectTestHelpers();

function exists(selector) {
  return !!find(selector).length;
}
```

Sample 4-2-1: Setup, Ember-testing and QUnit style

The `test_helper.js` file starts with the manifest elements we had before, but we now add a few setup features.

First up, we add a pair of `div` elements to the document (which, in this case, is the QUnit test-runner page). There's some CSS boilerplate in there that will place those DOM elements in the lower right corner of the screen. The following line sets the `rooteElement` of the Ember app itself to the DOM element, effectively giving us a small window in our browser containing a fully runnable, interactive version of our application. This is probably most useful when you are only running one test, it gets a little muddied when you run the entire suite.

Next up, we mess with the data store. The store is the Ember Data abstraction for dealing with a data source. We saw back when we were setting up the application that our real app uses a `DS.Store` based on a `RESTAdapter` that allows it to communicate with a REST based server.

For testing, we want our `DS.Store` to be backed by the `DS.FixtureAdapter`, which allows us to set test data up locally, in memory, and not call the server. Not only does this give us control over what data is available for our tests, but it keeps our test from touching our development environment, which is great.

Moving on, we then call two setup methods, `setupForTesting`, and `injectTestHelpers`. All `injectTestHelpers` does is make Ember Testing's helper methods available in the test.

The `setupForTesting` method does a few boilerplate things to put Ember in test mode. It sets a variable called `Ember.testing` to `true`, sets the Ember router back to the base application, and also delays the startup of your Ember application until your test begins. Let's take a second to talk about that...

One of the great things about Ember is that the framework does all kinds of things for you behind the scenes – there's a rich set of automatic observers and change events that we've only scratched the surface of so far. In particular, Ember has the concept of a `run loop`, which allows Ember to bundle up a bunch of changes and execute them together at the end of the loop so as to minimize expensive DOM interactions. We'll talk more about the run loop later, the important point right now is that the run loop happens automatically and governs Ember's auto-update and other change-event behaviors.

That's great under normal circumstances, but the run loop can sometimes be annoying during tests since it can make it hard to isolate unit tests, because multiple events are not executed separately.

When you place Ember in testing mode, all of that automatic stuff – all of Ember's Ember-ness, if you will – gets placed on hold. Instead, we can manually start and stop the run loop. Or, conveniently, we can use `Ember.run` to delimit a single lap through the run loop. When `Ember.run` is called, Ember will set up a run loop, execute the callback function, then close the loop, updating changes. Features based on Ember's data dependency model, such as computed properties, are really only manageable inside a run loop. If you are testing outside a run loop and you try to access something that requires Ember-data dependencies, Ember will throw an error and suggest you enclose the code in a call to `Ember.run`, as we already saw in our earlier tests.

Finally, we (or actually Erik Bryn) define a helper function called `exists` that takes a jQuery selector and returns true if that selector exists on the page. The `find(selector)` call in that method automatically limits the search for the selector to the Ember application itself, without looking at anything the test runner might be displaying.

Section 4.3

Ember-testing testing

With our setup in place, we are free to write our first Ember-testing integration test. Ordinarily, we'd have written this first, to get a failing test. But this time, I decided to get the

Ember concepts in place first, so this test covers the trip and detail display functionality we've already created.

Filename: test/javascripts/integration/indexControllerTest.js (Branch: e_04)

```
module("Index Controller", {
  setup: function() {
    TimeTravel.Trip.FIXTURES = [
      { id: 1, name: "Mayflower", description: "DESC",
        start_date: "1620-09-06", end_date: "1620-11-21" }
    ];
    Ember.run(TimeTravel, TimeTravel.advanceReadiness);
  },
  teardown: function() {
    TimeTravel.reset();
  }
});

test("it displays trips", function() {
  visit("/").then(function() {
    ok(exists(".admin_trips"));
    equal($(".trip").length, 1);
    equal($(".trip .dates").text(), "Sep 6, 1620 - Nov 21, 1620")
  });

});

test("it manages clicks", function() {
  visit("/").then(function() {
    return click(".trip .header");
  }).then(function() {
        equal($(".selected_name").text(), "Mayflower");
      });
});
```

Sample 4-3-1: Our actual integration test

We start by defining a QUnit `module` called `Index Controller`, and we also pass the second argument defining `setup` and `teardown` functions. The `setup` function will be called before each test, and the `teardown` function will be called after.

Let's take this in three parts – the setup, the teardown, and the actual tests.

Ember test setup

Our setup does two things – creates fixture data and starts the application.

The Ember `DSFixtureAdapter` assumes that each model class will define a property called `FIXTURES`. That property is expected to define an array of objects. Each object specifies the Ember-properties of one instance of that model – explicitly including the `id` property. The `DSFixtureAdapter` wires that into the application so that if you call `find` on the model, it searches the `FIXTURE` array for the item with the matching id, converting those properties to an actual instance of the Ember model (essentially calling `createRecord` using the data in the `FIXTURE` array).

In our case, we are defining one set of data for `TimeTravel.Trip`, so that later, when the application calls `TimeTravel.Trip.find`, we'll get one Trip with the name set to `Mayflower` and all the other data in place.

You can also set fixture data in the `test_helper.js` (really, you can set it anywhere). I'm experimenting with setting the fixtures in the actual module setup on the grounds that the test is easier to understand if the data is in the same file, as well as the forlorn hope that Ember will someday allow me to set this on a test-by-test basis.

The last line of the setup is the flip side of the `deferReadiness` method call we put in the `test_helper.js`. The `advanceReadiness` call in each test setup gives the application an "Ladies and Gentlemen, start your engines" warning, causing Ember-related application initialization to be performed.

Ember test teardown

Our teardown does one thing: calls the `reset` method of our `TimeTravel` app.

Ember uses the `reset` method of an Ember application to, well, reset the application. A running Ember application creates instances of the controller and view classes in the application – note that we never explicitly create them in our code. Ember stores these in a data structure called a `container`. Calling `reset` clears the container and returns the application to the index route, `/`.

What `reset` allows us to do is create our own controllers and views in a test with the confidence that the controllers and views will not bleed into other tests.

Ember test helpers

We've written two tests in this module.

The first tests visits our root URL, then verifies that it displays our one trip. We also check the date property to show that the formatted display version is being placed in the DOM.

We're seeing our first Ember-testing helper in action here, namely `visit`. The `visit` helper takes in a URL corresponding to a route known to the Ember application – in this case `\`. It returns... a *promise*. That sounds poetic, doesn't it?

In the interest of keeping this thread on track, lets just say that functionally, what happens is the URL passed to visit goes through the Ember

You Made Me Promises

A *promise* is a relatively new JavaScript coding construct that has become popular as a way of managing a very common pattern of an asynchronous method and a callback.[33] Ember uses promises in at least three places: here in Ember Testing, as part of the transition between routes, and when Ember data loads from or saves to it's storage location.

You can see the pattern that promises are managing pretty clearly in something like the jQuery `$.ajax` method, where the method takes options like `success` and `failure` the values of which are methods to be called asynchronously when the Ajax call completes. The problem with this model is that if you need to have a callback inside, say, your `success` callback, the resulting code quickly becomes deeply nested and complicated. And, as a bonus, it's also hard to test.[34]

A promise is an object that encapsulates the state of an asynchronous call. Specifically a promise is an object that defines one method, `then`. You call `then` with at least one argument, which is a function. The implementation of the `then` function must be such that when the asynchronous call completes successfully, then the functional argument is invoked, in exactly the same way that a success callback would be invoked in a traditional call. So, you can do `$.ajax().then(successHandler)`, and it would behave exactly like `$.ajax(success: successHandler)`. An optional second argument to `then` acts as the error handler.

Big deal. Just some syntactic sugar. Maybe, except for one critical fact about the

routing system, and then the functional argument to `then` is evaluated. There's a sidebar somewhere near here with more information about promises.

For our purposes, the point is that the functional argument to `then` is where we put any test logic that we want to execute after our application visits the URL. In this case, we're testing that the `.admin_trips` element is there, which basically just tests that we're getting the template we expect. Then we test that the output contains the one trip we created in our fixture, via the test for one instance of the `.trip` element. Finally, a test to make sure that we're using the controller's date formatter.

implementation of `then`: it returns another promise object. This specifically is meant to allow promises to be chained, replacing the deep nesting of a series of callbacks, with a much easier to follow list of `then` calls with responses.

The second test is a little more interesting. We visit `\` as before, but inside the function we return another Ember Testing header, `click`. The `click` helper takes as an argument a jQuery selector, it triggers a click event on the corresponding element, and returns a promise. The promise object evaluates after all Ember activity related to the click event ends.

Since the click event returns a promise, we can put any assertions we want to evaluate inside the function called when the promise is complete, in this case checking `equal($(".selected_name").text(), "Mayflower");`, or whether the click makes the name of the selected trip change.

Now, that continually having to return helpers is a little awkward, so Ember-testing provides a shortcut where you can chain the helpers directly, like so:

Filename: test/javascripts/integration/indexControllerTest.js (Branch: e_04a)

```
test("it manages clicks", function() {
  visit("/").click(".trip .header").then(function() {
    equal($(".selected_name").text(), "Low");
  });
});
```

Sample 4-3-2: Chaining Ember-testing helpers directly

33. There's actually a draft standard for what a promise does, you can see it at http://wiki.commonjs.org/wiki/Promises/A.

34. Yes, jQuery's `$.ajax` method can also return an object that is consistent with the promise API.

Ember Testing defines a couple more helper methods – we've already discussed `visit` and `click`. We've also seen `find`, which is used in `test_helper.js`, and which takes a selector and an optional context argument and finds a matching element within the Ember application.

There are two more form filling helpers, `fillIn`, which takes a selector and some text. It finds a matching form element based on the selector and sets its value to the second argument using the jQuery `val` method. It returns a promise that triggers when Ember is done processing any updates related to changing that form field. The helper `keyEvent` usually takes three arguments, a selector, an event type, as in `keydown`, `keyup`, `keypress` and the third argument is the `keyCode` for the key you are simulating. It also returns a promise that triggers when Ember is done processing the key event.

Finally, the `wait` helper just returns a promise that is fulfilled when all ongoing asynchronous behavior completes – it's used most often within other test helpers.

Section 4.4

A Failing Test

Let's push this a little and get a failing test then make something pass.

We haven't done sorting yet. From an integration test standpoint, sorting means that the display of the trips is in revenue order, highest to lowest. We can set that up in a test.

First off, sorry, but it seems like you can only set up fixtures globally. We need at least two items to make sorting testable, so the existing fixture won't do, and we need to move the fixture definition back to `test_helper.js`. Ugh. I'm hopeful that someday this can be more flexible.

Anyway, new fixtures:

Filename: test/javascripts/test_helper.js (Branch: e_04a)

```
TimeTravel.Trip.FIXTURES = [
  { id: 1, name: "Low", orders: 1, price: 100,
    start_date: "1620-09-06", end_date: "1620-11-21", hotels: []},
  { id: 2, name: "High", orders: 5, price: 100,
    start_date: "1600-09-06", end_date: "1600-09-07", hotels: [1]}
];
```

```
TimeTravel.Hotel.FIXTURES = [
  { id: 1, name: "Hotel", price: 10, nights_ordered: 0, trip: 1}
]
```

Sample 4-4-1: New fixtures for testing sorting

Note that the data is in reverse order of the intended sort, which enables us to tell if things are working.[3536]

Making these changes requires some changes to the existing `indexControllerTest.js` file, to remove the fixture there and update the assertions to match the new data – I'm not going to show those changes here.[37]

The test for sorted data is really simple, we just go to the page and check that the trip names show up in the sorted order.

Filename: test/javascripts/integration/sortedIndexControllerTest.js (Branch: e_04a)

```
module("Sorted Index Controller", {
  setup: function() {
    Ember.run(TimeTravel, TimeTravel.advanceReadiness);
  },
  teardown: function() {
    TimeTravel.reset();
  }
});

test("displays the trips sorted by revenue", function() {
  visit("/").then(function() {
    names = $(".trip .name").map(function() {
      return $(this).text();
    });
    equal(names.toArray()[0], "High");
```

35. Sometimes, when I test sorting displays, I'll create three objects that are not sorted in either direction.

36. Double footnote! If you've been playing along at home in previous versions, you'll notice this fixture has changed. Seems like you can not set calculated properties from a fixture in Ember Data 1.0.

37. There's a longer discussion about what's good and bad about global fixtures that I'm also going to pass on for now. Still hopeful that the fixtures will be mutable or that I'll figure out a way to make them so.

```
    equal(names.toArray()[1], "Low");
  });
});
```

Sample 4-4-2: An integration test for sorting

The setup is as in our other integration test. The test itself uses jQuery to grab the `.name` elements from the resulting view, extract their `text()`, and convert to an array for comparison to our expected value, which is that the `High` name should display first.

This test will fail.

But making the test pass is really easy. The content array in the controller uses the `Ember.SortableMixin` class to provide sort behavior. All we need to do to take advantage of this behavior is set a couple of properties on the controller, like so:

Filename: app/assets/javascripts/controllers/index_controller.js (Branch: e_04a)

```
TimeTravel.IndexController = Ember.ArrayController.extend({
  itemController: "indexTrip",
  selectedTrip: false,
  sortProperties: ['totalRevenue'],
  sortAscending: false,
```

Sample 4-4-3: Making our ArrayController sortable

The `sortProperties` property is an array of the properties that the array is sorted by, you can specify an arbitrary amount of properties. The `sortAscending` property is a boolean that specifies the direction of the sort, `false` as we have it, means a descending, high to low sort.

At this point, the test passes, even better, if you reload the site in the browser, the trips on the left are now sorted by their revenue.

This may seem a long way to go for some tests, but over time the setup for these tests in Ember is only going to get easier, and the ability to specify integration behavior in tests is really valuable in keeping your Ember application correct and clean.

Section 4.5

Debugging Ember

One quick thing before we get back to the view – Ember has a bunch of debugging flags that you can either set when the application starts up or enter in code or in the console of a running application. Here's a few – there's a more complete list at [38]

You can view some global data. The list of all known route names is available via `Ember.keys(App.Router.router.recognizer.names)` – replace `App` with your application's name, and the list of all registered templates is available using `Ember.keys(Ember.TEMPLATES)`. You can log object bindings as they are created with `Ember.LOG_BINDING = true` and route transitions with `Ember.LOG_TRANSITIONS = true`.

Given an Ember Data record, you can get a list of all it's state transitions – that is, whether it has been loaded or saved or updated via something like this:

```
t = TimeTravel.Trip.find(13);
t.stateManager.get('currentPath');
```

Sample 4-5-1: Getting state transitions

And then you can log further state transitions to the console with `t("stateManager.enableLogging", true)`. If you have an entire property path to a property, you can get a list of all objects observing changes to that property with `Ember.observersFor(comments, keyName)`.

Inside a Handlebars template you can use the directive `{{log object}}` to write to the console, and the directive `{{debugger}}` to enter a debugging session in the developer tools. You can also determine the view object that is backing a given DOM element, given it's DOM id with `Ember.View.views['DOM_ID']`.

The data structure `TimeTravel.__container__` contains a lot of the Ember data and structure (again, replace `TimeTravel` with the name of your application). The method `TimeTravel.__container__.lookup` can give you a view into specific data. The argument to

38. Thanks to Eric Berry for making this an Ember guide, and Akshay Rawat for an initial blog post at http://www.akshay.cc/blog/2013–02–22-debugging-ember-js-and-ember-data.html.

`lookup` is something of the form `controller:trips` or `route:index`. Using the argument `store:main` gives you access to the Ember Data data store, as we've seen in our test setups.
[^1]: http://toranbillups.com/blog/archive/2013/04/09/Unit-testing-your-emberjs-templates-with-jasmine-part–2/

Chapter 5

Enjoying The View

Section 5.1

Now Back in the View

Let's move on to what we need to do in the display to make it possible to actually administer things in this administration panel.

What we are going to do is use that detail sidebar on the right. We'll set it up so that when you click on a trip, the sidebar will show the number of orders, price, and revenue for each hotel and extra. Further, we'll make some of those columns editable, replacing the static view with a text field where you can enter new numbers.

In order to make this happen, we'll be using some of the template building blocks we've already seen. We'll also be adding one or two other Handlebar helpers.

Most importantly, we'll be using the critical Ember concept of *bindings*, leading to the more general concept of *observers*. A binding is a linkage between variables in two different objects, such that when one changes, the other changes to match. An observer is a more generic connection, causing a method to be automatically executed when a property changes.

Bindings and observers cause a fundamental change in the way your application interacts with the DOM. So far, when we've wanted to update the DOM in response to a change in our data, we've needed to explicitly make those changes. In Ember, we do something more like defining our data and then defining the pattern by which the DOM should reflect that data. When our data changes, Ember automatically updates the DOM in keeping with the pattern.

Bindings are used internally in Ember, and, in general, are not part of your application code.

Templates and Attribute Binding

Let's see what we need to do in our Handlebars templates to make this admin screen work.

First up, a little housekeeping. To prevent the handlebars templates from getting prohibitively long, I'm going to use the `{{partial}}` helper to break the index template up into two pieces.[39]

Filename: app/assets/javascripts/templates/index.handlebars (Branch: e_04a)

```
<div class="admin_trips span-12">
  {{#each trip in controller}}
    <div class="trip">
      <div class="header" {{action selectTrip trip}}>
        <span class="name">{{trip.name}}</span>
        {{trip.totalRevenueDisplay}}
      </div>
      <div class="dates">{{trip.startDateDisplay}} - {{trip.endDateDisplay}}</div>
      <div class="price">{{trip.priceDisplay}}</div>
    </div>
  {{/each}}
</div>
<div class="detail_view span-12">
  <h2>Details</h2>
  {{#if selectedTrip}}
    {{partial "details"}}
  {{/if}}
</div>
```

Sample 5-1-1: Index template with the details broken out

This is exactly what we had before, except we've replaced the detail panel with the directive `{{partial "details"}}`, which causes Ember to replace the template associated with that name inline to the parent template without changing the context. In our case, `ember-rails` will have loaded the file `app/assets/javascripts/templates/details.handlebars`, to make it accessible to Ember.

39. I think it's a good idea to keep template size small. As further incentive, it's hard for me to display partial handlebars files in my publishing setup...

Now the `details` template contains the contents of the sidebar. Here's most of it, looping over the hotel and extras, but I've extracted those templates as well.

Filename: app/assets/javascripts/templates/details.handlebars (Branch: e_04a)

```
<h3 class="selected_name">{{selectedTrip.name}}</h3>
<h3>Hotels</h3>
<table>
  {{#each hotel in selectedTrip.hotels itemController="hotel"}}
    {{partial "hotel_detail"}}
  {{/each}}
  <tr>
    <td>Total</td>
    <td>{{selectedTrip.totalHotelNights}}</td>
    <td></td>
    <td>{{selectedTrip.totalHotelRevenue}}</td>
  </tr>
</table>
<h3>Extras</h3>
<table>
  {{#each extra in selectedTrip.extras}}
    {{partial "extra_detail"}}
  {{/each}}
  <tr>
    <td>Total</td>
    <td>{{selectedTrip.totalExtrasOrdered}}</td>
    <td></td>
    <td>{{selectedTrip.totalExtraRevenue}}</td>
  </tr>
</table>
```

Sample 5-1-2: Detail template structure

All we have here is a couple of `each` loops, and a couple more `partial` directives. One bit we haven't seen before is the `itemController` property of the each loop. This was added for Ember RC1. The `itemController` property allows us to explicitly do in any `each` loop what we previously did implicitly by setting `itemController` at the controller level. We can specify a controller for the items of the loop that acts as the template context for the duration of the loop and is merged with each current loop value in turn[40].

In this case, we'll have a `HotelController`, by way of naming inflection from declaring `itemController=hotel`, and we'll use that to compute some properties and do some event handling. Having specific item controllers allows us to keep controllers and models smaller by allowing controllers to focus on specific portions of the template.

Just to get a tiny little thing out of the way, the `extra_detail` template is simple.

Filename: app/assets/javascripts/templates/extra_detail.handlebars (Branch: e_04a)

```
<tr>
  <th>{{extra.name}}</th>
  <td>{{extra.orders}}</td>
  <td>${{extra.price}}</td>
  <td>${{extra.revenue}}</td>
</tr>
```

Sample 5-1-3: Simple table for the extras display

Now, the reason why the `extra_detail` is so simple is that we haven't actually added any of the editing feature to it, all we're doing is displaying already calculated properties. Eventually, we might want to add edits to all these table cells, but in the name of not cluttering up these templates, I haven't yet.

I have made exactly one of the table cells in the `hotel_extra` template editable. See if you can guess which one.

Filename: app/assets/javascripts/templates/hotel_detail.handlebars (Branch: e_04a)

```
<tr>
  <th>{{hotel.name}}</th>
  <td>
    <span {{bind-attr class="editing:hidden:normal"}}
          {{action startEditing}}>
      {{hotel.nights_ordered}}
    </span>
    <span {{bind-attr class="editing:normal:hidden"}}>
      {{view Ember.TextField valueBinding="nightsOrderedInput"
          size="4"
          action="endEditing"}}
```

40. Technically, you get a unique controller instance for each loop value, which allows them to maintain separate state.

```
    </span>
  </td>
  <td>${{hotel.price}}</td>
  <td>${{hotel.revenue}}</td>
</tr>
```

Sample 5-1-4: Simple table for the hotel display, showing an editable cell

If you said the editable cell is the first cell for nights orders, just because it's grown all kinds of `span` tags and Ember handlebars directives, then you are right.

Let's go over how this edit feature works. I'll tell you right now that this edit is supported by fewer than ten lines of JavaScript in our controller, which includes automatically updating all the values that are dependent on that change – the revenue for that hotel, the total number and revenue for the trip, and the overall total revenue for the trip as displayed on the left side of the screen, not to mention updating the sort order of the trips to reflect the new revenue number.

There are two new Ember handlebars directives in this template, `bind-attr` and `view`. Since `bind-attr` is both first on the screen and conceptually simpler, let's talk about that first.

To explain why Ember needs the `bind-attr` directive, let's point out two things that I've said so far that are potentially contradictory or at least problematic when combined. First off, we've seen that Ember replaces any Handlebars `{{}}` directive with the associated property value. We've also mentioned that whenever you place a property value in your template, Ember surrounds the property with some HTML markup so that it can automatically update the DOM when the property value changes.

Given those two facts, you might wonder what happens when you want to have a dynamic value that is actually inside an HTML tag. You might want an item's DOM class to be dynamically changed – if you've been reading any of this book, you are familiar with the show/hide toggle capabilities that are implemented as dynamic DOM classes. Or you might want the `href` of a tag to be set based on other data. Or any of jillions of other reasons to have dynamic HTML attributes.

What happens if you put a bare `{{}}` directive inside an HTML tag? Wacky hijinks, at least if your idea of wacky hijinks includes HTML tags inside other HTML tags and the resulting browser chaos. Hence, `{{bind-attr}}` – a Handlebars helper that lets you create dynamic HTML attributes.[41]

The simple form of `{{bind-attr}}` takes one or more key/value pairs where the key is an HTML attribute and the value is the path to an Ember property, as in `{{bind-attr href="hotel.url" target="currentTarget"}}`. When the template is resolved, Ember creates regular HTML attributes based on the property (equivalent to doing `controller.get('property.path')`), but also adds additional `data` attributes to the tag as hooks for the automatic update of the values. And it's all nice and HTML legal.

Ember Templates

Ember stores all its templates in a global variable conveniently called `Ember.TEMPLATES`. If you are using `ember-rails`, files with the extension `.handlebars` or `.hbs` will automatically be precompiled into this variable as part of the asset pipeline. If you are in asset pipeline developer mode and you look in the resources tab of the Chrome or Safari web inspector, you'll see JavaScript files corresponding to each template.

If you aren't using `ember-rails`, you have a couple of options. First off, you can put Handlebars templates inside HTML `script` tags, setting their `data-template-name` attribute to the name you want to refer to the template, and Ember will automatically compile it as part of starting the application.

Alternately, you can compile the templates manually using something like `Ember.TEMPLATES['index'] = Ember.Handlebars.compile("template string")`. You can also use Handlebars precompilation to give you JavaScript files to place in your own tool. Common JavaScript build tools like Grunt and Lineman may also have extensions to automate this process.

Naturally, it's not quite that simple. Specifically, we need to advantage of the special way that Ember handles the `class` attribute, and only the `class` attribute. First off, the regular HTML class element is a space-delimited list. Similarly, the Ember `bind-attr` for a class is also a space-delimited list. Only the entries in the list are properties and not literal values.

41. I expect the Ember team to have a way to make this work without `bind-attr` at some point in the future. And I say that because Yehuda Katz has said so publicly, and I believe him.

So, if you do something like `<div {{bind-attr class="visibilityClass spaceClass"}}`, then Ember effectively does a `get('visibilityClass)` and a `get(spaceClass)` on the template target. The template target is the same as it would be for an ordinary `{{property}}` lookup – by default it's the controller delegating to the model. You can change the template target by enclosing part of the template in a `{{with}}` directive.

Using `bind-attr` on a list of properties, then, gives you a `class=` attribute with the associate list of values. If, however, a value is falsy, then Ember just skips it.

There are a couple of even more special cases. Often, you want to mix literal DOM classes with dynamically generated ones. Ember allows you to do that by prefixing the literal names with a colon, as in `{{bind-attr class=":span6 visibleClass"}}`, which would result in something like `class="span6 hidden"`, depending on the value of `visibleClass`. Another common case is having the property value be true or false, while the DOM class is some tag-specific value. This is useful, for example, in the case where the same property needs to drive different DOM classes in different parts of the application. In the template, we can do that by adding one or more literal values to the property name, separated by colons, as in `{{bind-attr class="isVisible:visible"}}` or `{{bind-attr class="isVisible:visible:hidden"}}`. If there's only one such literal, then it is used if the property is true with false being disregarded, if there are two literals, then the first is used with a true property and the second with a false one.

Finally, since `bind-attr` is creating an Ember binding, just like any other place the template looks up a property, if the underlying property changes, then the bound HTTP attribute is automatically changed in unison.

In this particular example, we have two `span` elements being modified by `bind-attr`. In the span that is just plain text, the declaration is `class=editing:hidden:normal`, in the second span, which contains the `Ember.TextField`, the declaration is `editing:normal:hidden`. Both of these elements are using the `editing` property of our controller to drive their visibility. If the `editing` property is true, the the text element takes the DOM class `hidden`, and the text field takes the DOM class `normal`. If editing is false, then vice-versa. In other words, when the controller thinks we are editing, the editable field is displayed, otherwise the plain text is displayed.

Some of What You Need To Know About Views

Our other new template directive is `view`. The `view` directive inserts an Ember view object into the template. What's an Ember view object, you ask?

We'll start here with the generic definition, then we'll talk specifically about the predefined `Ember.TextField` view that we are using in this code. Eventually, we'll talk about views in general, (but possibly not until next chapter).

You'll notice that we have gone pretty far in our Ember application without explicitly creating any views. This is by design – Ember is structured so that most of your display functionality is either captured by the template, which has the DOM layout, the model, which has persistent data, or the controller and router, which manage state. In general, Ember's view objects exist to handle browser events, like `click` and convert them to semantic events that are defined by your application, like `addTrip`. In practice, a lot of this defining can be done in your Handlebars template via `action` directives, so Ember views are typically only needed when you need particularly unusual event handling, or if you have a component you want to reuse.[42] In this example, we are reusing a pre-defined Ember component that manages a browser text field.

Ember ships with four built-in views: `Ember.Checkbox`, `Ember.TextField`, `Ember.Select`, and `Ember.TextArea`[43]. All of them wrap an HTML form element and allow you to bind the value of the element and events on the element to the rest of your application.

Let's talk about how we can use one of the built-in views. The first argument to the `{{view}}` directive is the name of the view class being used, here it's `Ember.TextField`. We're passing in three attributes to the pre-defined view: `size`, which gets passed onward to the underlying HTML `input`, `action`: which is defined by the `Ember.TextField` and is the name of a method to trigger when the user hits return inside the text field, and `valueBinding`, which is interesting.

Bindings are a very important topic in Ember. Ember allows you to link properties on two different objects, such that when one changes, the other updates to the new value. This is somewhat similar to the mechanism by which template property references are automatically updated when the property changes, but can be applied to any two arbitrary objects.

In our specific case, we have set a property on our view called `valueBinding` and given it the value `nightsOrderedInput`. What that does is tie `value` property of the view with the `nightsOrderedInput` property of the controller.[44]

42. Recent versions of Ember allow you to define a reusable component solely with a template and maybe a subclass of `Ember.Component`, and not create a bare view object.

43. Radio button and radio button group support is in progress, but the exact structure of the Ember view is, I think, still under debate as I write this.

More generally, If you have any Ember object, and you give it a property with a name that ends in `Binding`, then the value of that property is a path string referencing a property on the other object. And just like that, the two properties are bound. Something like this:

```
TimeTravel.Hotel = DS.Model.extend({
  tripNameBinding: "trip.name"
});
```

Sample 5-1-5:

sets up a binding where `hotel.get('tripName')` automatically shares a value with `hotel.get('trip.name')`.

Okay, there's not much benefit in having hotel and trip bind their names, since there are other ways of getting the data. There are benefits to using bindings to easily share values across layers of the application.[45] In our case, the `valueBinding` allows the controller easy access to the value entered in the form (updated on a keystroke-by-keystroke basis, no less), without having to add any code to either the view or the controller, just by adding the property declaration when the view is used. In the controller, we can access the property `nightsOrderedInput` and get the currently entered value. We can also set the property `nightsOrderedInput`, and update the value on the form.

Using the property in our code is simple. The controller in question is our `HotelController` because we are inside the loop declared with `{{#each hotel in selectedTrip.hotels itemController="hotel"}}`. The code for the controller looks like this – remember that action handlers now go inside an `actions` object:

Filename: app/assets/javascripts/controllers/hotel_controller.js (Branch: e_04a)

```
TimeTravel.HotelController = Ember.ObjectController.extend({
  editing: false,

  init: function() {
    this._super();
  },
```

44. It's a property of the controller because the property string is evaluated in the current context of the template.

45. That said, over time the Ember core team has made it more and more clear that they consider bindings like this to be in internal mechanism, and that generic application code should use computed properties and observers.

```
  actions: {
    startEditing: function() {
      this.set("nightsOrderedInput", String(this.get("nights_ordered")));
      this.toggleProperty("editing");
    },

    endEditing: function() {
      this.set("nights_ordered", parseInt(this.get("nightsOrderedInput")));
      this.toggleProperty("editing");
      this.get("content").save();
    }
  }

});
```

Sample 5-1-6: Editing code for our hotel controller

Let's follow the bouncing code and track what happens as this code gets executed. We will pick up the action after the page has already been loaded, and a user has clicked on a trip to display it on the right side of the browser window.

1. The `HotelController` instance has its `editing` property default to `false`. This causes the `bind-attr` class for the text-only span to be `normal` and the `bind-attr` class for the span with the text field to be `hidden`.

2. The user clicks on the text span, which has an `{{action startEditing}}` directive.

3. The `startEditing` method of the `actions` object inside the `HotelController` is invoked.

4. The `nightsOrderedInput` property is set to the string version of the current number of nights ordered. Since this property is bound to the value of the text field, the value will automatically change to that string.

5. The `editing` property is toggled. This causes both `bind-attr` directives to be recalculated, meaning that the text goes into hiding and the form field is now displayed, with the newly set value on display.

6. The user edits away. Eventually they hit `return` to finish editing, triggering the `action=endEditing` property of the view.

7. The `endEditing` method of the controller is invoked.

8. The `nightsOrdered` property of the model is set to the integer value of the text field. This will automatically cause the text field and all the other totals derived from this value to recalculate and update their display.

9. The editing property is toggled back, causing the `bind-attr`s to be calculated, and hiding the form field and replacing it with the plain text, whose value has just been updated.

That seems like a lot of steps because I'm walking through a lot of what Ember does behind the scenes. From our perspective, we get that edit field updating all kinds of places on our page for two two-line methods and some glue.

To tie up one loose end – we could bind the value of the text field directly to `nightsOrdered` but because the underlying value is an integer, you'll get some weird updates when you do something like delete the last digit for editing. It's a little more user friendly to buffer the value in a separate property and only swap the values when the user is done.

Section 5.2

Back to the server

There's one other bit of functionality we probably want our admin panel to have, which is the ability to actually send our new data change back to the server. We're going to use Ember Data and the `RESTAdapter` for this. However, there are a few things worth pointing out or reiterating:

- Ember Data is still under rapid development as I write this, although core Ember is considered to be under an API freeze, Ember Data is not. Saving stuff back to the server changed dramatically in Ember Data 1.0

- You don't actually need to use Ember Data to use Ember. If you are using Ember Data, you don't necessarily need to use the `RESTAdapter`.

- You can, in fact, fall back to the jQuery `$.ajax` method to manage data back and forth if you like. We're not going to do that here, but it's worth being aware of as an option, if Ember-data is giving you problems. Discourse, which is the first Ember application to

make a big public splash, does not use Ember Data, and does manage server communication at the jQuery level.

Still with me? Good.

After all that, actually saving something back to the server is quite easy in Ember Data 1.0 – we've actually already seen the code in the previous snippet, it's in the `HotelController`'s `endEditing` method. The relevant line of code is:

```
this.get("content").save();
```

Sample 5-2-1: Saving the model context

All we are doing here is getting the controller's model via `this.get("content")`, and saving it. Ember Data does the rest, so to speak.[46] In the common case of using a `RESTStore`, `save` triggers a `PUT` request to the server, passing the object and its properties as parameters. On my machine, this triggers an HTTP `PUT` call to `http://localhost:3000/hotels/62`, with a JSON payload like so:

```
{"hotel":
  {"name":"Bunk with the Emperor",
  "description":"Live like a king. Literally.",
  "price":1250,"nights_ordered":65,
  "trip_id":20}}
```

Sample 5-2-2:

It's incumbent on our server-side code to handle the HTTP request. Luckily for us, a boilerplate Rails server-side `update` controller method will handle it.

The `save` method has one other trick up its sleeve, which is a clear improvement for Ember Data 1.0 over earlier versions (as if the simpler save API wasn't enough). The `save` method returns a promise that fires when the save request returns from the server. The promise makes it much easier to manage testing for whether the save was successful, you can do something like this:

```
this.get("content").save().then(
              function() {
```

46. Yes, pre Ember Data 1.0, there was a whole dance involving transaction objects and commits. Seems to be all gone now.

```
    // successful case
  },
  function() {
    //failure case
  });
```

Sample 5-2-3: An Ember Data save with promises

In the success case, the object having been saved is guaranteed to have had an ID assigned. In the failure case, you can do whatever you want, including retry. And, as with any promise, the promises can be chained to build up complex behavior.

Section 5.3

The Components of Success

Looking at what we've go here, it seems like we have the beginnings of a UI pattern that might be common to many aspects of this application: click on a number, edit it in the browser, have the item update and save itself. We don't want to have to repeat the multiple lines of Ember every time we want to edit a number. So some way of encapsulating the entire – let's call it a component – might be valuable.

Ember allows you to easily convert snippets of Handlebars into *components*, which consist of a Handlebars template and an optional Ember object to manage action logic. The Ember component is basically there for two things: the first is to make it easy to reference a common chunk of Handlebars template, while the second is to make it easy to convert a series of raw browser events into a more meaningfully named event that the rest of your application might be able to handle.

Here's how we might convert that hotel number editor to a more generic Ember component.

All it takes to define a component in Ember is to have a template with a name beginning with `components/`. In our Rails application, that will happen automatically to any template that we put in the `app/assets/javascripts/templates/components` directory. In a non-Rails application, o if you are doing Handlebars templates inside `script` tags, that means having the `id` of the template be something like `components/number-editor`.

You invoke the component by using the name of the component as if it were a Handlebars directive.

Filename: app/assets/javascripts/templates/hotel_detail.handlebars (Branch: e_04b)

```
<tr>
  <th>{{hotel.name}}</th>
  <td>
    {{number-editor emberObject=hotel
        propName="nights_ordered"
        action="itemChanged"}}
  </td>
  <td>${{hotel.price}}</td>
  <td>${{hotel.revenue}}</td>
</tr>
```

Sample 5-3-1: Hotel Detail template, now calling the number-editor component

We've replaced the text edit view with a direct call to our new `number-editor` component, which Ember will expect to be defined as `components/number-editor`, or in our case, `app/assets/javascripts/templates/components/number-editor.handlebars`.

The attributes that make up the remainder of the call to the component are key/value pairs that become properties of the component. We've got three: the `hotel` in question, which the component will refer to as its `emberObject`, in an attempt to be more generic, the `propName`, which is the property being edited, we're setting that to `nights_ordered` – the quotation marks around `nights_ordered` are critical here. With them, Ember will send the literal string as the value, without them, Ember would attempt to evaluate `nights_ordered` as a property and send the resulting value. Needless to say, this would not work out well. Finally, the `action` is `itemChanged`, that's the action in our controller that will be invoked by the component.

Our component template is just the relevant snippet of Handlebars, with the names generalized.

Filename: app/assets/javascripts/templates/components/number-editor.handlebars (Branch: e_04b)

```
<span {{bind-attr class="editing:hidden:normal"}}
  {{action startEditing}}>
  {{propValue}}
</span>
<span {{bind-attr class="editing:normal:hidden"}}>
  {{view Ember.TextField valueBinding="numericalInput"
  size="4"
```

```
  action="endEditing"}}
</span>
```

Sample 5-3-2: The Number-Editor component template

There's almost literally nothing there we haven't seen before, it's the same snippet with a couple of names changed.

In this case, we also have some logic that this component needs. Previously, we kept that logic in the `HotelController`. But now we can move it into the component. Ember looks for an object whose name is inflected from the name of the component template, in our case, that means a class named `NumberEditorComponent`, which we can define like so:

Filename: app/assets/javascripts/components/number_editor.js (Branch: e_04b)

```
TimeTravel.NumberEditorComponent = Ember.Component.extend({

  editing: false,

  propValue: function() {
    return this.get("emberObject").get(this.get("propName"));
  }.property('emberObject', 'propName', 'editing'),

  actions: {
    startEditing: function() {
      this.set("numericalInput", String(this.get('propValue')));
      this.toggleProperty("editing");
    },

    endEditing: function() {
      newValue = parseInt(this.get("numericalInput"))
      this.get("emberObject").set(this.get("propName"), newValue);
      this.toggleProperty("editing");
      this.sendAction();
    }
  }
});
```

Sample 5-3-3: The JavaScript side of the number editor component

Most of this code is taken directly from the `HotelController`, but it's been made more generic. First off, the `propValue` property is a shortcut for accessing the exact property of the object being edited, making it easier to reference that property from the template. (Making the property dependent on the `editing` toggle ensures that it will not stay cached when you are done editing).

As for the rest, we've changed some names to make them more generic – the internal placeholder value is now called `numericalInput`. But the next really new thing is the `sendAction` call in the very last line.

At the beginning of this section, I mentioned that components play a role in converting browser events to events that are meaningful within your application domain. A component can define one or more (but by default, one) action to be emitted in response to component logic. The `sendAction` method is the way this mechanism is invoked from the component side.

When we defined the component, we called it with an `action` attribute, `action="itemChanged"`. Inside the component, calling `sendAction` triggers that action within the object that invoked the component.

In this particular snippet, `sendAction` is called at the end of the editing process. Since the component was invoked with `action="itemChanged"`, calling `sendAction` triggers the `itemChanged` method on the controller, which looks like this:

Filename: app/assets/javascripts/controllers/hotel_controller.js (Branch: e_04b)

```
TimeTravel.HotelController = Ember.ObjectController.extend({

  init: function() {
    this._super();
  },

  actions: {
    itemChanged: function() {
      this.get("content").save();
    }
  }

});
```

Sample 5-3-4: The Hotel Controller event handler

The hotel controller performs the actual save.

Giving the controller a chance to respond to the edit action gives us a nice separation of concerns. The component is responsible for managing the display of the editing form and updating the value of the resulting object, while the controller is responsible for any repercussions of that update. Which might just be saving the item back to the server, or might involve further logic.

To clean up a couple more advanced cases, if you want to send parameters back as part of the action, you need to call `sendAction` with the first argument `action`, and then any arbitrary parameters following, as in `sendAction("action", 3, "banana")`. The parameters will be sent to the handler for the action on the controller side. This syntax is a little weird, but I think it's related to the next case.

If your component needs to send multiple actions for some reason, you specify that on the component side with `sendAction` and an arbitrary first argument that names the action, as in `sendAction("delete")` or `sendAction("explode")`. When the component is invoked, you don't use `action` as an attribute, instead you use the action names, as in `{{number-editor delete="deleteHandler" explode="explodeHandler"}}`. Ember will map the `sendAction` calls to the handlers based on the name.

Chapter 6

Links and Detail Stuff

By now you know what happens next.

> Great work, old chum. This is undeniably a view that our administrators can use to administer things. We're very excited.
>
> However, one of our employees in the 23rd Century has a complaint (you know those 23rd century types, always whining over something...). It seems that this administrator would really like to give all the trip detail screens there own URL, so they can be bookmarked.
>
> Also, if you could give us a real edit form, that would be fantastic,
>
> Yours in ephemera,
>
> Doctor What

As always, the Doctor's command is our wish.[47] We'll learn more about Ember's routing system, and how it is used, and we'll also talk about pushing logic into views and Handlebar helpers.

Section 6.1

Get Your Kicks On Route /trips/66

Earlier, we said that an Ember route codifies the state of the application in a URL and associates that URL with a particular combination of controllers and models that are used to create the display in the browser.

47. Actually, the idea to revisit the structure came from a comment by Shane Mingins, who pointed out that the `link-to` structure would be appropriate here. See also http://ember101.com/videos/004-master-detail-router-outlet-link-to/

Then I showed you an example that used no routing at all. Because we had other stuff to cover first, that's why.

Let's go back and replay our trip example using routing to manage the display of the detailed view. By using routing, we are able to easily give each detail trip its own URL that can be shared and which uniquely causes the associated trip detail to be displayed.

What we want is for each trip detail to have a URL of the form `trips/1`, where the number is the ID of the trip. We'll also make the list of all the trips `trips/`, rather than `/`, which makes the Ember routing a little more consistent.

Making this work involves changing our routes to the new structure, then moving our existing code around to match Ember's expected naming conventions – we actually change relatively little code, but we do rename some objects and files.

The advantages of moving to this route pattern are:

- Each state has a unique URL that can be shared
- In the previous version, we were managing the identity of the currently selected trip in our application code, in the routed version, Ember will manage that for us.
- The ability to have more focused controllers, which makes the classes smaller and easier to manage.

In order to make this work, we need to talk a bit more about Ember's routing system.

Starting with a code sample, here's the updated route file that we will need for our new scheme:

Filename: app/assets/javascripts/routes/app_router.js (Branch: e_05)

```
TimeTravel.Router.reopen({
  location: "history"
});

TimeTravel.Router.map(function() {
  this.resource("trips", {path: "/trips"}, function() {
    this.resource("trip", {path: ":trip_id"});
  });
});
```

Sample 6-1-1: Router with nested resources

We've changed a couple of things. We've removed the explicit reference to an index route, which is fine because, as we'll see, Ember implicitly creates an index route for us. We've also added two new routes using Ember's `resource` method rather than the `route` method – for the moment, the two are basically identical, we'll see the difference in a page or so.

The first new route is named `trips` and matches the URL `/trips`. The second is named `trip`, and matches any URL of the form `/trips/3`, where 3 can be any id. As with many other web framework routing systems, the syntax `:trip_id` is a dynamic segment that matches whatever is placed in that part of the route.

The second route, which displays an individual trip is declared by being nested inside the functional argument to the general trips route. This nesting indicates that the show route is itself nested underneath the general route. Nesting a route in Ember indicates two different things:

- The URL pattern for the route is concatenated on to the parent pattern, so the full pattern for the singular `trip` route is `trips/:trip_id`.
- The controller and template for the inner route have a specific relationship and placement with respect to the outer route.

The URL mapping is probably broadly familiar to you if you've worked in other web frameworks. It is, for example, roughly similar to how Rails nested resources work. (Okay, very roughly).

That said, The relationship between the outer controller and the inner controller in Ember is likely to be a little different than what you are used to.[48] In Ember, all of the various steps of the nested URL collaborate to create the final page – the Chrome Ember inspector gives a great view of that process.

So if the URL is `trips/3`, the plural `TripsController` gets a shot at drawing part of the page, then the singular `TripController` inserts its content at a specified location within the template used by the `TripsController`. (It's actually a little more complicated than that, but we'll get there step by step).

48. By which, of course, I mean that it was different from what *I* was used to, and I'm assuming you are all at least a little bit like me.

The magic word to connect these pieces together is the `{{outlet}}` handlebars helper defined by Ember. The `{{outlet}}` helper tells the routing system to move to the next part of the route, and insert its content at the point of the `outlet` helper. In these respect it's similar to, but more flexible than, the Rails `yield` call inside a layout. In essence, Ember allows any template to act as the layout for the next template that needs to be rendered.

Ember's default naming convention is that the names of the route object, the controller, and the template will be based on the name you give the route in the `Router.map` function. Therefore our `trips` route would look for a route object named `TripsRoute`, a controller named `TripsController`, and a template named `trips`.

We would expect our `application.handlebars` template to have an `outlet` which defers to a `trips` controller and template pair, which itself has an `outlet` that defers to a singular `trip` controller and template pair.

Let's walk through what that looks like in Ember's routing to see how `trips`, and then `trips/1` are rendered. Please note that the full router path has a couple more complications – we're going to discuss a basic path first, then talk about where Ember makes things more... interesting.

Okay, we start, as we did originally, with the top level application. Ember looks for an `ApplicationRoute`, an `ApplicationController`, and a template named `application`. The only one of these things that we actually have is the template, and it has not changed:

Filename: app/assets/javascripts/templates/application.handlebars (Branch: e_05)

```
<div class="container">
  <h1>Time Travel Adventures Administrative Screen</h1>
  {{outlet}}
</div>
```

Sample 6-1-2: Our application template

This template is rendered first on any page that Ember controls. In this case, it's simple enough not to need any router or controller support – Ember will create dummy routes and controllers if we don't define our own. That covers the case where the user just hits the route `/`.

In the case where the user has hit the URL `/trips`, that URL matches the route we've named `trips`, so we're looking for `TripsRoute`, `TripsController`, and a `trips` template. We'll actually define all of these.

If you compare this code to the code in the previous chapter, you'll see that the objects that we're now calling `Trip` route and controller are very, very similar to the objects we were calling `Index` route and controller. That makes sense, because we've basically moved the same behavior that in previous chapters came from a URL that was matching the `index` route, to a URL that matches the `trips` route.[49]

The route still needs to load all the `Trip` objects, so that we can display them:

Filename: app/assets/javascripts/routes/trips_route.js (Branch: e_05)

```
TimeTravel.TripsRoute = Ember.Route.extend({
  model: function() {
    return this.store.find('trip');
  }
});
```

Sample 6-1-3: The TripsRoute gathers all the trips

This route is still using the Ember `model` method to tell the controller associated to this route where it is getting its data.

The controller is now named `TripsController`, and it's similar to, but slightly simpler than, the older `IndexController` version, see if you can spot how:

Filename: app/assets/javascripts/controllers/trips_controller.js (Branch: e_05)

```
TimeTravel.TripsController = Ember.ArrayController.extend({
  itemController: "indexTrip",
  sortProperties: ['totalRevenue'],
  sortAscending: false,

  overallRevenue: function() {
    return this.get('content').reduce(function(runningTotal, item) {
      return runningTotal + item.get('totalRevenue');
    }, 0);
```

49. If you are wondering what happened to the `index` route, hold that thought, we're coming back to it in a little bit.

```
  }.property("content.@each.totalRevenue"),

  overallRevenueDisplay: function() {
    return accounting.formatMoney(this.get('overallRevenue'));
  }.property("overallRevenue")
});
```

Sample 6-1-4: The TripsController no longer tracks the selected trip

Our friendly neighborhood caption gives us a subtle hint. In the previous version, the controller explicitly tracked which trip was currently selected for the purpose of displaying the detailed information for that trip. In this version, it's none of our beeswax, we'll defer it to another controller/router pair. We're also no longer responding to a `selectTrip` action here. The `Trips` controller still manages its relationship with an `itemController` and with the sorting properties, same as before.

The handlebars template, in `templates/trips.handlebars` has only two minor changes.

Filename: app/assets/javascripts/templates/trips.handlebars (Branch: e_05)

```
<h2>Trips</h2>
<div class="admin_trips span-11">
  {{#each trip in controller}}
    <div class="trip">
      <div class="header">
        {{#link-to "trip" trip}}
          <span class="name">{{trip.name}}</span>
        {{/link-to}}
        {{trip.totalRevenueDisplay}}
      </div>
      <div class="dates">{{trip.startDateDisplay}} - {{trip.endDateDisplay}}</div>
      <div class="price">{{trip.priceDisplay}}</div>
    </div>
  {{/each}}
</div>
<div class="detail_view span-11">
  <h2>Details</h2>
  {{outlet}}
</div>
```

Sample 6-1-5: The template now manages link_to and outlet

Once again, our friendly caption has beaten me to the punch and described the change. First off, we are now using the Ember `link-to` helper to describe a route change. Secondly, where the previous version explicitly used `partial` to defer to the detail template, this version just has an `outlet` helper. Just as an aside, notice that the index trip display inside each loop is still managed by the controller identified as the `itemController` property of `TripsConroller`, that hasn't changed.

The `link-to` helper allows you to move between routes within your Ember application – if you are going outside your Ember application, you'd just use an ordinary HTML anchor tag. The `link-to` helper takes one required argument, which is the name of the route you are transitioning to, and one optional argument, which is the model that will be used as the context for the route, controller, and template at the end of the transition. The text between the open and close of the `link-to` is the caption of the link – the text of the resulting HTML anchor tag.

You have a couple of options with `link-to`. If you don't want the resulting tag to be an HTML anchor, you can add a `tagName=` property to the helper with the name of the tag you want surrounding the transition click target. Also, by default, Ember will automatically add the CSS class `active` to the link if the link matches the route currently being rendered. You can customize this CSS class with the keyword property `activeClass=`.

The `outlet` helper is the location in the page where Ember places the next template to be rendered. By default, a route/controller/template set expects to have a single `outlet`. You can override the default behavior for what should be placed in the `outlet` by defining a method on the route object called `renderTemplate`, and inside that method calling `this.render()` with the name of a template as an argument.

In the template, You can optionally give an `outlet` helper a second argument, which is a name of the outlet, as in `{{outlet individualTrip}}`. If you want to fill one of those outlets, then you must define that behavior in the route's `renderTemplate` method, with something like `this.render('tripTemplate', {outlet: 'individualTrip'});`. Also, you can override the default behavior of `outlet` by declaring a different view for event handling using the keyword property `viewClass`.

If we're just drawing `/trips`, we stop there, because there is no further part of the route to place in the `outlet`.

If we want to draw an individual trip route that places the detail data on the right side, then we need to continue onward.

In order to draw the individual `trip` route, first you'd click on a `link-to` link – since the `link-to` is being drawn inside a loop, each trip gets its own. Ember uses the second argument of the `link-to` to attach the link to a specific trip, so the `href` arguments wind up in the form `trips/2` and so on.

Clicking one of those links causes Ember to update the window URL and render the route. First Ember draws the first part of the URL, meaning `trips`, and does the exact same thing we've already seen.

This time, though, there's more to the URL. The route is defined as a resource named `trip`, singular. So we're looking for a `TripRoute`, like so:

Filename: app/assets/javascripts/routes/trip_route.js (Branch: e_05)

```
TimeTravel.TripRoute = Ember.Route.extend({
  model: function(params) {
    return this.store.find('trip', params.trip_id)
  }
});
```

Sample 6-1-6: The trip route finds the trip we need

It simply finds the trip associated with the route's ID, which is identifiable as `params.trip_id` because the definition of the route used `:trip_id` to identify the dynamic part of the route – as in many other web frameworks the name in the route is used to identify that part of the URL throughout the render process.

We actually don't need a specific controller here, and the template is now called `trip.handlebars`, but it's almost the exact same thing we used to call `detail.handlebars`:

Filename: app/assets/javascripts/templates/trip.handlebars (Branch: e_05)

```
<h3 class="selected_name">{{name}}</h3>
<h3>Hotels</h3>
<table>
  {{#each hotel in hotels itemController="hotel"}}
    {{partial "hotel_detail"}}
  {{/each}}
```

```
  <tr>
    <td>Total</td>
    <td>{{totalHotelNights}}</td>
    <td></td>
    <td>{{totalHotelRevenue}}</td>
  </tr>
</table>
<h3>Extras</h3>
<table>
  {{#each extra in extras}}
    {{partial "extra_detail"}}
  {{/each}}
  <tr>
    <td>Total</td>
    <td>{{totalExtrasOrdered}}</td>
    <td></td>
    <td>{{totalExtraRevenue}}</td>
  </tr>
</table>
```

Sample 6-1-7: Our Trip display template

The only significant difference is that before, all of our property references were of the form `selectedTrip.name` because the template was being rendered in the context of the index route. This time, it's being rendered in the context of an individual trip route, so the trip model is itself the context, so property references are of the form `name`. Notice that the loop through hotels and extras remain the same, at least for the moment. Though if you were looking longingly at those `template` helpers and wondering if they might be able to become `outlets`, well hold on to your hat for a little bit, we're getting there.

Section 6.2

Ember Routing Details

That's the basic story of Ember routing – a URL is paired with a router object, controller object, and template, with a default naming convention tying them together.

The router's job is to fetch the data model and also potentially to tie outlets to other controllers. The controller's job is to provide display specific logic and event handling, and the view/template provides the pattern for the eventual DOM output.

Naturally, it's not always that simple. There are a few Ember features that complicate the routing story. Ironically, the base impulse of these features is to simplify, at least in the sense of allowing you to use common patterns with a minimum of typing. Like many of Ember's kind-of-magic-seeming implicitness, these features make sense and are valuable once you get the hang of them, but since the features often depend on Ember inferring things that aren't there, figuring out what's going on can be challenging.[50]

Let's talk about some different features of Ember routing that may be difficult to get your head around.

First, *implicit controllers*. And implicit routes, for that matter. If Ember does not find a controller or a route object that matches the name associated with the URL in the router, then it implicitly creates an empty placeholder object and continues on its merry way.

We're actually already seeing this in our application – we have an `appication.handlebars`template, but not an associated `ApplictationRoute` or `ApplicationController`, both of which would be used by Ember to draw the background template if they existed. We might need a `ApplicationRoute` and `ApplicationController` if our application template needed some data to draw – for example, if we wanted the background to change based on the currently logged in user.

That said, in general, your routes will need route objects, if only to associate models with the templates. You can get by without controllers if you don't have any display-specific logic and the part of the screen being controlled doesn't have event handling. Simple templates, in other words, can be made simpler by not requiring basically superfluous objects.

Somewhat more weirdly, Ember automatically injects an index route for each level of nesting in the router's `map` method. We alluded to this behavior back toward the beginning of the book, but now let's explore it in more depth.

Our current routing table looks like this:

50. Have I spent nearly an hour trying to debug a method when it turned out that the actual behavior was managed by a different method? No comment. Though, to be fair, that had nothing to do with routing, it was an observer relationship that I hadn't tracked down.

Filename: app/assets/javascripts/routes/app_router.js (Branch: e_05)

```
TimeTravel.Router.reopen({
  location: "history"
});

TimeTravel.Router.map(function() {
  this.resource("trips", {path: "/trips"}, function() {
    this.resource("trip", {path: ":trip_id"});
  });
});
```

Sample 6-2-1: Current routing table

But the two routes defined there are not all the routes that are available. Ember implicitly defines a top level `this.route('index', {path: '/'})` which resolves to `TimeTravel.IndexRoute`, `TimeTravel.IndexController`, and an `index.handlebars` file, none of which we are using at the moment.

Ember also defines the same route inside the `trips` declaration, at the same level as the `trip` route. This one is a little weirder, it looks for objects named `TimeTravel.TripsIndexRoute`, `TimeTravel.TripsIndexController`, and a template at `trips/index.handlebars`, meaning we'd need a `templates/trips` directory to place the template in.

We're not using that route either, although we could. In particular, these routes allow us to distinguish between "content that is on all trip-related pages" (in the `trips` route) and "content that is only in the trips index page" (in the `trips/index` implicit route).

What's the practical difference between putting content in the `TripsController`, as we are doing, versus using a `TripsIndexController`?

The way we have our page laid out, the left side is the list of all the known trips, and the right side is the detail view for the currently selected trip. Because the left-side listing is tied to the `TripsController` it will be part of the rendering path for any page in the `trips` realm. So that listing will show up as the background, so to speak, of any trips page.

Try this:

- In `app/assets/javascripts/routes/trip_route.rb`, change the name of the route class to `TimeTravel.TripsIndexRoute`.

- In app/assets/javascripts/controllers/trips_controller.rb, change the name of the controller class to TimeTravel.TripsIndexController.

- Take the template in app/assets/javascripts/templates/trips.handlebars, and move it to app/assets/javascripts/templates/trips/index.handlebars. If you are feeling extra persnickety, you can take out the detail dive in the last four lines of the template.

By making these simple moves, we've now taken the trip listing data out of the common trips routing path and moved it into code that is only executed when the URL trips is requested, not when a URL like trips/2 is requested.

What does that mean in reality? If you make those changes trips will result in just the listing of the trips, as before, but if you click on one of the trips to render the trips/3 detail link, then you still see the detail info, but the listing info that got moved to trips/index is now gone, and the entire page is the trip-specific detail.[51]

This is a consequence of the difference between placing code in the Trips slot of the routing path, which makes it part of all the trip-related pages, and placing code in the index slot of the routing path, which makes it specific to the index page.

There's one subtlety to the naming conventions that you may have noticed. We have an (implicit, but real) index route that uses naming conventions like TripsIndexRoute and TripsIndexController. If we were to create an edit route inside the trips resource using something like this.route("edit", {path: "/edit"});, then the objects would be named TripsEditRoute and TripsEditController (and if you ever needed to refer to the route directly, you'd call it trips.edit).

However, when we declare the show route using this.resource("trip", {path: ":trip_id"});, the resting objects are named TripRoute and TripController, not TripsTripRoute and TripsTripController. What gives? And it's not just that TripsTripRoute is a ridiculous name.

The difference is because of the way Ember treats routing elements that are declared using route versus elements that are declared using resource. Conceptually, a nested resource is a noun, like trip, while a nested route is a verb, like edit, or a page description, like index.

51. I probably shouldn't admit that I discovered this kind of accidentally as I was playing around trying to implement some of the router docs, and having the listing disappear freaked me out until I figured out why it happened.

There's an actual distinction to go along with the conceptual one. First off, you can nest additional routes and resources beneath a `resource`, but you can't nest anything beneath a `route`.

Additionally, If you declare the URL handler using `resource`, then the naming conventions make that resource the beginning of the object names even if the resource is nested inside another route. So, `TripRoute` rather than `TripsTripRoute`, because the inner trip is a resource. Inner items declared using `route` just append their names, so `TripsIndexRoute`. As we'll see in a bit, when we declare a `hotel` resource inside `trip`, it'll be `HotelRoute`, and not `TripsTripHotel` route.

As far as I can tell,[52] there's no super-compelling reason to have the distinction between a `route` and a `resource` besides the aesthetic reason of trying to avoid long object names and deeply nested template folders. Which is fine by me.

Section 6.3

Multiple Controllers, Same Object

As you look over the code that's writing out these trip pages, you may also notice potential duplication of effort. Specifically, we now have two different parts of the code dedicated to drawing individual trips:

- We have the individual trips being rendered in each row of our index table, this display is governed by the what now seems unfortunately named `IndexTripController` (which is declared as the `item_controller` of `TripsController`). The template for this part is currently inside the loop of the `trips.handlebars` template.
- We also have the right hand side of the page, essentially drawing `trip#show`. At the moment, this render doesn't have an explicit controller, but the template is in `trip.handlebars`

There's no problem at the moment because the detail page is showing completely different information than the listing segment. However, a few weeks time and traffic down the road, maybe the detail page needs an explicit controller with some of that information and

52. Not necessarily very far...

suddenly you have logic being duplicated between the two controllers. Seems bad. What to do?

There are a couple of ways to manage this issue, which is likely to come up increasingly as you have multiple Ember controllers managing small patches of screen.

One method would be to give up, and put the display logic back in the `Trip` model. On the plus side, at least the logic wouldn't be duplicated, but this would seem to miss the point of separating display logic into the controllers in the first place.

Another option is to create a common parent for all the trip controllers, declared with something like `TimeTravel.AbstractTripController = Ember.ObjectController.extend`, and then have the individual trip controllers extend that (`TimeTravel.IndexTripController = TimeTravel.AbstractTripController.extend`). Any common display logic would go in the parent, and logic separate to each individual instance would go in the children. This is better than giving up, and would give you a little more structure to hang common behavior on.

Section 6.4

Nested Nesting That Nests

Let's pull this nesting one level deeper, and assume that we're going to actually put some detail info about an individual hotel or extras on the page, accessible via a URL like `trips/3/hotels/2`. Mostly this is a straightforward extension of what we just showed, but there's one little piece that might seem new.

What do we need for this new URL to work? We definitely need routes. We need templates. We might need controllers if there's display specific information.

Start with the routes. We'll declare `hotel` and `extra` to be resources on their own, because they are nouns.[53]

Filename: app/assets/javascripts/routes/app_router.js (Branch: e_06)

53. By the way, I've been using this time travel trip example in one form or another for about four years, and it only just occurred to me that the real word for the thing I call an "extra" is probably "activity". If I get ambitious, I'll go back and change it everywhere.

```
TimeTravel.Router.reopen({
  location: "history"
});

TimeTravel.Router.map(function() {
  this.resource("trips", {path: "/trips"}, function() {
    this.resource("trip", {path: ":trip_id"}, function() {
      this.resource("hotels", {path: "/hotels"}, function() {
        this.resource("hotel", {path: ":hotel_id"})
      });
      this.resource("extras", {path: "/extras"}, function() {
        this.resource("extra", {path: ":extra_id"})
      });
    });
  });
});
```

Sample 6-4-1: More Nested Routes!

Granted, that seems like rather a lot of nesting, especially if you are used to Rails (in Rails, declaring, say `hotels` as a resource, implies the existence of `hotels/:hotel_id`, whereas in Ember, it does not. Yet.)

This is going to get a little weird – we already have a `HotelController` that supports the editing of the number of orders that a hotel has.

We're going to add another wrinkle. Clicking on the name of the hotel will add some detail information about the hotel – this is mostly a placeholder right now, but you could see were me might add more detail in the future.

At minimum, we need a `link-to` and `outlet` pairing. Here it is in the `hotel_detail.handlebars` template, which was originally extracted from the `trip.handlebars` template. As with the edit features, I'll show this for the hotels and you can add it to the extras on your own. Here's the new template:

Filename: app/assets/javascripts/templates/hotel_detail.handlebars (Branch: e_06)

```
<tr>
  <th>
    {{#link-to "hotel" hotel}}
```

```
      {{hotel.name}}
    {{/link-to}}
  </th>
  <td>
    {{number-editor emberObject=hotel
        propName="nights_ordered"
        action="itemChanged"}}
  </td>
  <td>${{hotel.price}}</td>
  <td>${{hotel.revenue}}</td>
</tr>
```

Sample 6-4-2: Hotel Template with additional nested link

Right up there at the top, we've surrounded the `hotel.name` directive with another `link-to`, which we can just pass the hotel. Ember is smart enough to extract the entire URL from the hotel, given the nested route relationship that we've defined.

We've placed the `{{outlet}}` to get to this in the `trip.handlebars` template itself:

Filename: app/assets/javascripts/templates/trip.handlebars (Branch: e_06)

```
<h3 class="selected_name">{{name}}</h3>
<h3>Hotels</h3>
<table>
  {{#each hotel in hotels itemController="hotel"}}
    {{partial "hotel_detail"}}
  {{/each}}
  <tr>
    <td>Total</td>
    <td>{{totalHotelNights}}</td>
    <td></td>
    <td>{{totalHotelRevenue}}</td>
  </tr>
</table>
{{outlet}}
<h3>Extras</h3>
<table>
  {{#each extra in extras}}
    {{partial "extra_detail"}}
```

```
  {{/each}}
  <tr>
    <td>Total</td>
    <td>{{totalExtrasOrdered}}</td>
    <td></td>
    <td>{{totalExtraRevenue}}</td>
  </tr>
</table>
```

Sample 6-4-3: Trip Template with outlet to hotel details

We need to put the `{{outlet}}` in the trip template, because putting the outlet inside the hotel detail template, which is rendered multiple times, will confuse Ember by placing multiple `{{outlet}}` directives on the page.

To make this work against our Rails app, we have to make to make some small changes. In order to be able to enter the application at a `trips/3` or `trips/3/hotels/1` URL, we need to do the same thing we did for `trips#index`, namely pass through the HTML requests to start up Ember.js.

Filename: app/controllers/trips_controller.rb (Branch: e_06)

```
class TripsController < ApplicationController

  def index
    @trips = Trip.all
    respond_to do |format|
      format.html { render text: "", layout: "home" }
      format.json { render json: @trips }
    end

  end

  def show
    @trip = Trip.find(params[:id])

    respond_to do |format|
      format.html { render text: "", layout: "home" }
      format.json { render json: @trip }
    end
```

```
  end

end
```

Sample 6-4-4: TripsController, Rails side

Filename: app/controllers/hotels_controller.rb (Branch: e_06)

```
class HotelsController < ApplicationController

  respond_to :html, :json

  def index
    @hotels = Hotel.where(:id => params[:ids]).all
    respond_to do |format|
      format.html { render text: "", layout: "home" }
      format.json { render json: @hotels }
    end
  end

  def show
    @hotel = Hotel.find(params[:id])
    respond_to do |format|
      format.html { render text: "", layout: "home" }
      format.json { render json: @hotel }
    end
  end

end
```

Sample 6-4-5: HotelsController, Rails side

We also need to add a nested route for hotels if we want to be able to enter the application at the nested hotel URL.

```
resources :trips do
  resources :hotels
end
```

Sample 6-4-6: routes.rb declaration for hotel nested routes

At this point, the code works but doesn't do much. If you hover over the hotel names, you see that URL's of the form `trips/20/hotels/62` show up in the status bar. Clicking on one changes

the URL, but not the page. This is because we haven't added a template to place inside that new `{{outlet}}`.

Technically, Ember will be happier if we create two templates. Remember, how when we were working at the level of trips, we created the `trips.handlebars` (plural) template that was the backdrop for all trip-related URLs, as well as the `trip.handlebars` (singular) template that was used just for the single trip route? We have the same issue here. Ember wants us to create a plural `hotels.handlebars` template to be the background for the hotel URL's and also a singular `hotel.handlebars` to display a single hotel.[54]

We don't have anything in the layout, though we could have some kind of header, I guess... For now, all we'll put there is the outlet.

Filename: app/assets/javascripts/templates/hotels.handlebars (Branch: e_06)

```
{{outlet}}
```

Sample 6-4-7: Our hotels layout that passes through

Inside that outlet, we'll put the detail for the specific hotel being drawn. Right now, we'll just elaborate on the hotel with its description.

Filename: app/assets/javascripts/templates/hotel.handlebars (Branch: e_06)

```
<tr>
  <td>{{description}}</td>
</tr>
```

Sample 6-4-8: Hotel detail template

This `hotel.handlebars` template is going to be backed by the existing `HotelController` that we wrote to hand the editing of hotel bookings. That's kind of an accident of Ember naming, since we declared the controller in the line `each hotel in hotels itemController="hotel"`, and then declared the route using the same `hotel` tag. There's no reason at the moment to have separate controllers, so we'll leave it.

At this point, the full URL passage thorough a hotel looks like this:

54. Strictly speaking, you can get away without the plural hotels template here – Ember will let you off with a warning. Literally, Ember puts a scary warning in the console about how an outlet isn't going where it thinks it should. I *think* the warning is ignorable – everything seems to work, and there's some discussion over whether it's necessary.

On my system, the URL for that page is `trips/20/hotels/62`. Let's follow the bouncing ball through the outlets – you can do this with the Chrome Ember Inspector. This page passes through the following templates:

- `application.handlebars` – the layout for the whole app. In this case, it draws the `h1` title at the top of the screen.
- `trips.handlebars` – covers the `trips` part of the URL and anything subordinate to it. In this case, draws the left side of the screen plus the "Details" header on the right.
- `trip.handlebars` – covers the trip id and renders an individual trip. In this case, most of the hotel and extra tables are there. We also split pieces of that template into partials, but that's somewhat orthogonal to the route path.
- `hotels.handlebars` – covers the `hotels` part of the URL and anything subordinate to it. Basically empty right now.
- `hotel.handlebars` – covers the 62 id for the individual hotel. Right now, it draws the "Live like a king. Literally" description.

On the one hand, that's kind of a lot of route parts for one screen. On the other hand, there's a nice pattern where each part of the URL in turn draws part of the screen hand hands control to the next part.

Section 6.5

Need Me

Suppose we want to include information about the trip itself inside the hotel part of the display. Easy enough, we can just refer to the hotel that is the content of the hotel controller, and use its Ember-data association to get at the trip.

Fine. Now let's say we want one of the display-logic specific properties that we've only defined in the `TripController`. Well, we could... no, that wouldn't work... Neither would that. Okay, what can we do?

It would seem that we have a similar dilemma to the one that we had before, when discussing how to manage the potential of multiple controllers over the same data – either we need to figure out how to access the controller in Ember, or we need to move this display-specific functionality back into the model.

Luckily, it won't come to that. We can access the trip controller by making one simple change to the hotel controller:

Filename: app/assets/javascripts/controllers/hotel_controller.js (Branch: e_06)

```
TimeTravel.HotelController = Ember.ObjectController.extend({
  needs: "trip",
  trip: null,
  tripBinding: "controllers.trip",
```

Sample 6-5-1: We really, need a vacation. I mean, a TripController

That `needs` line in the hotel controller gives you access to the associated trip controller by using the property string `controllers.trip`.[55] Or, you can do what we've done in the next line and use that property string to create a binding – `tripBinding: "controllers.trip"` – which allows us to access the `TripController` using just the property string `trip` (because the `Binding` suffix on a property name creates an Ember binding, as discussed in the previous chapter).

55. Internally, Ember maintains the set of all controllers and the objects they apply to – that's what you are getting limited access to.

With that, we have what we want, access to the display properties defined on the controller, and we can still get at the model transparently through the controller just as we'd be able to normally. Cool.

Section 6.6

You Promised Me A Route!

Before this chapter goes off into that good night, I did want to mention how you can hook into the Ember router to insert your own logic. For example, if you are authenticating a user, you probably want to be able to check whether the user is authorized to see the content *before* the route transitions to the new state.[56]

You get three hooks in to the Ember routing system, which you can implement as methods on your route objects: `beforeModel`, `afterModel`, and `willTransition`.

The first one called is `willTransition`, which is called on the route you are transitioning from – strictly speaking, Ember expects `willTransition` to be inside an `actions` object of the route, similar to the `actions` objects we have discussed for controllers. If you define `willTransition` for your route, you are letting the route decide if there is a condition that needs to be fulfilled for the transition to the new route to happen. You define `willTransition` as a function that itself takes a single argument, which is an object encapsulating the transition between routes. So you might do something like this:

Filename: app/assets/javascripts/routes/trip_route.js (Branch: e_07)

```
TimeTravel.TripRoute = Ember.Route.extend({
  model: function(params) {
    return this.store.find('trip', params.trip_id)
  }

  actions: {
    willTransition: function(transition) {
      if(model.wontLetYouGo()) {
```

56. The discussion of authentication patterns was greatly informed by the EmberCast screencast at http://www.embercasts.com/episodes/client-side-authentication-part–1 by Erik Brin and Alex Matchneer, and the jsBin at http://jsbin.com/eQOZoGe/3/edit by Yehuda Katz, and, of course, the relevant Ember Guide.

```
        transition.abort();
      } else {
        return true;
      }
    }
  }
});
```

Sample 6-6-1: A ridiculous implementation of willTransition

The idea here is that inside the `willTransition` function, you check for whatever state is interesting, and optionally block the transition by calling `abort` on the transaction object. (Transaction objects also respond to `retry`, should you want to restart an interrupted transition.) Alternately, you could call the route's `transitionTo` method, which takes the name of a route (not the URL) and an optional model and, well, transitions to that route. If you return `true` from `willTransition`, the transition will bubble up to any parent routes that might also want control over the route – so putting a `willTransition` in the index route means it will be invoked on all transitions in the application.

You wouldn't necessarily use `willTransition` to manage authentication state – you'd normally want the route receiving control to make the gateway decision. Instead, you might use `willTransition` to determine if the user has unfinished business on the route being left. For example, it might be a place to validate form data.

In contrast, `beforeModel` and `afterModel` are called on the route receiving the transition. Those are actual methods of the route (meaning, not buried in the `actions` item), and they take the same transition object as an argument.

The `beforeModel` method is called, as you might expect, before the `model` method. As you may recall, the `model` method is called by the route to determine which model object or array is passed on to be the content of the resulting controller. Normally, the `model` method itself is bypassed if the route is accessed in such a way that a model is already provided. For example, when the route transition is triggered via a `link-to` helper that provides a model. However, even when the `model` method would be bypassed, `beforeModel` and `afterModel` are still called.

The `beforeModel` hook is a great place to check to see if the user has been authenticated. The emerging pattern is to have the user receive an authentication token from the server after a successful login, which you then store someplace convenient (possibly in the application namespace, or attached to the index route).

Then the `beforeModel` hook checks for the token, and if it doesn't exist, takes steps to acquire it, which might mean using `transitionTo` to go to another route that displays a login form (holding on to the transition so it can be `retry`ed if login is successful). It might look something like this – assuming that we're using the application namespace to handle some local storage.

Filename: app/assets/javascripts/routes/trips_route.js (Branch: e_07)

```
TimeTravel.TripsRoute = Ember.Route.extend({
  model: function() {
    return this.store.find('trip');
  },

  beforeModel: function(transition) {
    if (!TimeTravel.isLoggedIn()) {
      TimeTravel.setStalledTransition(transition);
      this.transitionTo('login');
    }
  }
});
```

Sample 6-6-2: A ridiculous implementation of willTransition

This is scratch code, by the way, it's not in any way integrated into our project.

The `afterModel` method is slightly different. The argument to `afterModel` is the model that is headed for the controller. Which means you normally wouldn't use it for authentication. Except maybe for the case where the route in question is actually retrieving the authentication information. You can use `afterModel` to provide any post-processing you might want on the incoming model.

You can use these features to add arbitrary asynchronous processing to your routing.

Chapter 7

Done, Done, Done, Done…. Done!

I'm calling that a wrap.

It's hard. Unlike some of the other topics covered in this book, Ember is so big and growing so fast that I feel like there's still so much to cover. I hope this book guides you through getting started with Ember and helps you grapple with it's initial learning curve and really access the power it has.

I've been really impressed with Ember as a tool, coming back to it just before the beta releases started. The core team seems to me to have done an amazing job responding to feedback, making tweaks where needed, admitting that some ideas weren't fully baked and replacing them with better ones. And keeping up documentation, which has been fantastic. Also thanks to the Ember community, which is showing every sign of becoming large, enthusiastic, helpful, and inclusive.

I'll be checking back in to this book for a while to keep it up to date with the latest and greatest.

In the meantime, though, I hope this book helps you do awesome wonderful things with JavaScript.

The end of this chapter also means the end of my writing on *Master Space and Time With JavaScript* (with of course, some editing, corrections, and updates yet to come). One way or another, I've been working on this book for over two years, or about 18 months longer than the original plan. It's been interesting, to say the least. Also fun. Thanks to all of you who have taken part by reading and by asking questions and correcting my mistakes.

At one point, I was going to end the book with a brief paragraph or two where you would find whatever time mechanism was being used by the mysterious Dr. What that's provided all the

silly openers to chapters, and travel back in time, revealing yourself to be Dr. What. And I guess I kind of have done that.

Thanks for your time and attention.

Noel Rappin

September, 2013

Chapter 8

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, and Stephen Wolff. Kay Keppler and Susannah Pfalzer acted as editors. Thanks to all of them.

Trek Glowacki provided valuable technical sanity checks on Ember.js. Peter Wagenet also answered Ember questions for me. The rest of the Ember core team, including Yehuda Katz and Tom Dale, have also been friendly and available for my silly questions. A huge thank you to all the people who have maintained and improved the Ember API documentation and Guides, especially Trek for organizing it. Fantastic work, all of them.

A few people provided feedback on a very early Alpha of this book, and were nice enough to do it on a quick turnaround. Thanks to Brandon Hays, Kyle Stevens, Sean Massa, David Burrows and Chris Stump.

Many people have sent me errata, including: Grant Defayette.

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 Grant Defayette, Pierre Nouaille-Degorce, Sean Massa, Vlad Ivanovic and Nicolas Dermine for particular efforts.

Shane Mingins made a comment that basically led to Chapter 4.

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 9

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.

For the record, text editors used on the book's content at some point or another included SublimeText, Byword, iaWriter, and FoldingText, before I finally settled on Ulysses 3 for the Markdown text. I even did a little of the writing on iPad, using Byword, iaWriter, Textastic, and Editorial, in more or less ascending order of usefulness for me.

Chapter 10

Changelog

Release 001: January, 2013

Initial release.

Release 002: February, 2013

- Moving forward on attributes
- Errata fixes from Grant Defayette

Release 003: March, 2013

- Totally goofy section on acceptance testing
- Bindings, views, and the like
- Update to Ember 1.0RC1

Release 004: April, 2013

- Overall edit, clarify distinction between Ember and Ember-data
- Update to Ember 1.0RC3
- Much less goofy section on acceptance testing
- Section on updating via Ember-data

Release 005, May, 2013

- Errata from Adam Ferguson, Raul Brito, Géraud Mathe, Bob Hanson, Dele Omotosho, Andrew Davis, Nicholas Rowe, Augusto H. Teixeira, Erik Trom, and Shane Mingins
- New chapter on Ember routing and nested routing.

Release 006, July, 2013

- Replaced all testing code with QUnit and Ember-testing
- Added discussion of promises.
- Brief section on Ember debug flags thanks to Eric Berry and Akshay Rawat.

Release 007, Sep, 2013

- Updated all code to Ember 1.0 and Ember Data 1.0 beta 2
- Added information about Ember components
- Updated all Ember Data text to reflect new Ember Data structures
- Added information about route promises