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 003 March, 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:

Important Note.

This book is *wildly* incomplete. It is subject to rapid and ongoing change as Ember.js changes, and as my understanding of it changes. Good luck.

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 deal may not be available indefinitely.

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

Section 1.2

Who Are You? Who? Who?

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

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

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

JavaScript: I'm assuming that you have a basic familiarity with what JavaScript looks like. In other words, we're not going to explain what an `if` statement is or what a string is. Ideally, you're like I was 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, I'll provide a separate CoffeeScript version of the code repository. NOTE: This isn't ready yet; don't go looking for it.

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

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

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

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

Section 1.3

What to Expect When You Are Reading

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

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

- This book is focused on the current versions of the languages and libraries available. As I write this, that means ECMA Script 5 for JavaScript, jQuery 1.7.2, Jasmine 1.2.0, Backbone.js 0.9.2, Ember.js 1.0pre4 and Rails 3.2.x. Keeping up with current versions is hard enough, without worrying about the interactions among multiple versions.

Section 1.4

But is it finished?

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

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

- CoffeeScript versions of all the sample code in the book will be made available.
- Formatting for Kindle and ePub versions is still a little wonky in spots. I'm working on it.
- The directions for setting up the sample application probably need to be improved.
- Book 4 is very much in progress.
- 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

Important Note:

This chapter is **wildly incomplete**, and is being released because I'm trying to get on a regular release schedule. If you get to the end and feel like there's more to come, there is. Wait a few days. Follow `@noelrap` on Twitter to keep up to date. We are in **early, early beta** on this, mistakes are surely there.

Thanks,

The Management

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. Structurally, it occupies a similar space as Backbone, not to mention Angular, Knockout, and however many dozens of other frameworks that I don't plan on writing 100 pages about.

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 some Rails-style convention-over-configuration naming assumptions.

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 that will be called as part of the render but 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[1], 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.[2]:

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

2. 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.

As Ember core team member Trek Glowaki 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.".[3]

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, we'll take it in a little bit of a different direction, though.

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`[4] – we'll cover setup for non-Rails people in a bit.

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

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

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

Of course, that isn't enough. Since Ember is so fast moving, and in particular because we're using new features of the Ember router API that have not made it to the official release as of this writing, we want to use the cutting-edge Ember version.

Keep looking here to see what version of Ember the rest of the book uses: We're using Ember-Rails version 0.11.1 (revision 16fa749e), and ember version RC1, last commmit d4e6a5c on Feb 26.

Luckily, `ember-rails` gives us an easy way to update our local Ember to the current head on github. From the command line:[5]

```
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 should say that while earlier versions of the book did use this to get the most recent Ember, I'm currently running aginst the version tied to the current ember-rails.

One other piece of pure configuration. Ember.js has slightly different development and production versions. 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

4. 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)

5. 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.

On the Rails side, we need the same kind of minimal setup that we had for Backbone, basically just a route that does nothing but load up Ember.

We're going to do this the dirt-simple way. We'll keep our existing root route to `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 the generator is currently out-of-sync with Ember master, and because it generated some stuff we're not going to need.

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();

TimeTravel.Store = DS.Store.extend({
  revision: 11,
  adapter: DS.RESTAdapter.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. The other line creates an Ember DataStore, which Ember uses to define the communication between models and the storage mechanism. We need it here to make our application work, but the details aren't necessary for a while, we'll talk about it more in the future.

The application object in Ember mostly works in the background. For our purposes at the moment, creating the application is largely a declaration that we're entering Ember-land. The application object also serves as our namespace – all the other Ember things we create will be scoped to `TimeTravel`.[6]

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 defaults to expect 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.

At the moment, the important parts of this template are the `{{}}`, which is Handlebars 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, similar to the `yield` in a Rails layout.

We are now faced with the question of what goes into that `outlet`.

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

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

- **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 pair[7] state tends to be tied to an individual page. 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.

Ember maintains a pretty strict sense of what goes in the controller object and what goes in the model. 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. 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.

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 matching 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"
```

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

```
})

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

Sample 2-3-8: The simplest router

Please note that the router API has changed a few times, this is as of Feb 26 and 1.0RC1.

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 it helps to have an actual routing line to talk about.)

Okay, what does `index` point to?

Ember uses the router to connect a URL to a `Route` object, Ember uses string inflection based on the name of the route, so saying that the route is named `index` causes Ember to look for a route class called `IndexRoute`. Eventually, it will also default to 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`.[8]

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)

8. 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.

```
TimeTravel.IndexRoute = Ember.Route.extend({
  model: function() {
    return TimeTravel.Trip.find();
  }
});
```

Sample 2-3-9: The initial IndexRoute

"Okay", I hear you say... "I see a model, but no controller. You just said less than a page ago 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. It also allows 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 global 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.

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 referenced in the `IndexRoute` method `model`.

In Ember, Models come from the `ember-data` library, and have their own separate top-level namespace, `DS`. Ember models are designed to communicate with a data store and convert that data to 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 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'),
  startDate: DS.attr('date'),
  endDate: DS.attr('date'),
  imageName: DS.attr('string'),
  slug: DS.attr('string'),
  tagLine: 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 expects JSON, with a couple of peculiarities when we start talking about associations. Specifically, Ember is smart enough to know that a camel-case property like `startDate` can come from the server with underscores, as in `start_date`. Defining these properties makes them accessible from an Ember template, and also accessible to other objects that want to bind to or create properties derived from this model.

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.startDate}} - {{trip.endDate}}</div>
```

```
      <div class="price">{{trip.priceDisplay}}</div>
    </div>
  {{/each}}
</div>
<div class="detail_view span-12">
  <h2>Details</h2>
</div>
```

Sample 2-3-12: Initial template

This is a pretty simple set of HTML. We're using two features of the Handlebars/Ember combination. We've got an `{{#each trip in controller}}` declaration. Handlebars is less minimal then Mustache when it comes to control flow, actually using 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 the use the `content` property of the controller as the source of the loop.

Inside the loop, the normal `{{}}` syntax is used to reference properties of each trip in what appears to be 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:

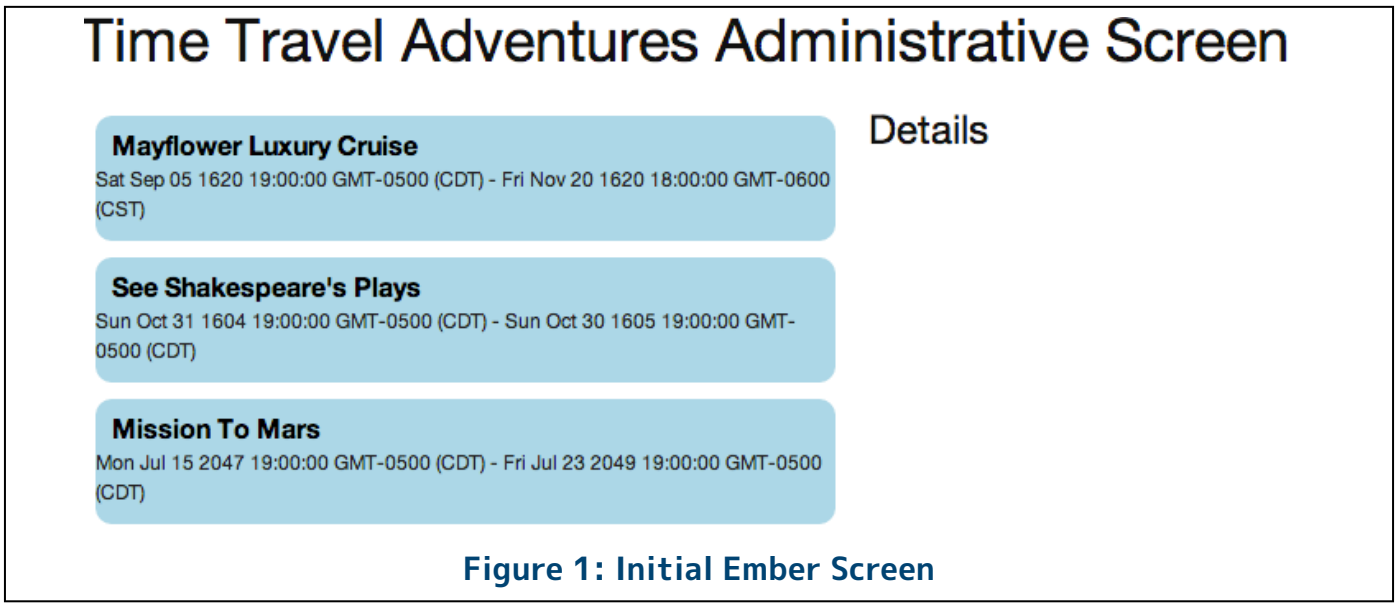

Figure 1: Initial Ember Screen

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 into it. All this does is change a property of a model being displayed.

```
TimeTravel.Trip.find(1).set("name", "A Different Trip");
```

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

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. In no way did we 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, 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.

More Elaborate Property Work

I want to do two things here. First, I want to clean up that date 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 this 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 this – parsing the incoming date string to 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 `startDateMoment` and `endDateMoment` 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('startDate'))
}.property("startDate"),

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

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

What we have here is a perfectly normal conversion function, with two little quirks that allow all 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.

At the end of the function definition, 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 dependent properties to be updated, even if the dependent property is in a Handlebars 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, than the `startMoment` property will also be marked as changed and any display parts that depend on `startMoment` will also be updated.

That's 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. Leaving us with two choices that don't quite work here. One choice is that the template could have just `{{startDateDisplay}}`, in which case a method on the controller would be invoked but wouldn't know which trip's date to display. This works great for calculated properties like a total price, 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.

Luckily, Ember has a feature that allows your array controller to specify a different controller class to use for individual items, 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'),

});
```

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 view 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` propagates to `startMoment` and then to `startDateDisplay`.

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

Select a Trip

One more quick thing to show you some of the power of Ember. We want for a click on one of the trip names 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 (well, it'll look in view object first, but we don't have our own view object in play at the moment), then it will look in the `Route`. Other arguments in the `{{action}}` are passed as arguments to the helper method.

So somebody clicks on one of those headers and Ember will look in the controller for 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,

  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 helper does is set the property. One thing to point out is that when you specify default behaviors as part of extending the class, as we've done here with `itemController` and `selectedTrip`, those become Ember properties when you create an instance of the class. Meaning that even those properties need to be accessed using Ember's `get` and `set`.

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?

If you are not working against a Rails back end.

Important Note I haven't written this yet. It's coming.

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 templates, but are a big part of the way controllers and views work.
3. Models manage persisted data and associated non-display business logic. They talk to a data store, often a RESTful server.
4. All Ember objects, including Models, are made up of properties that are read and written via the Ember object system. Properties can be defined in relation to other properties.
5. 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.
6. A controller wraps a model or an array of models, and associates its content with a template.
7. Controllers can have properties that reference it's associated model. In Ember, this is where you put display-specific logic.
8. Templates can reference properties. Ember looks to resolve these properties in the template's controller first, then automatically checks the controller's model.

9. These template references are bindings. 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 first checked against the enclosing view object, then it's enclosing view object, and so on until the parent view is reached, at which time, the controller object is checked for the handler method.

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 information 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 to display the total price taken in for each component, and the price for all the components summed up.

In the interest of keeping the data model simple, we're just going to have order count fields on the `trips`, `hotels`, and `extras` tables rather than having to process a bunch of orders.[9] (If

you got the code from an earlier 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 other options, that strongly implies that we need models representing the hotel and extra options. And we probably also need an association between a trip and 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 modes 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"),
  nightsOrdered: DS.attr("number"),
  trip: DS.belongsTo("TimeTravel.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("TimeTravel.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("TimeTravel.Trip")`. This declares that the `Hotel` and `Extra` classes have a one-to-many relationship with a `Trip`. On the other side, we augment the `Trip` class to declare the `hasMany` portion of the relationship.

9. 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.

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

```
  hotels: DS.hasMany('TimeTravel.Hotel'),
  extras: DS.hasMany('TimeTravel.Extra'),
```

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

Ember 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.

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

First up, Ember 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
  }
}
```

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 underscored names can be converted to camelCase.[10]

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 handle associations? The most important thing to start with is what Ember does *not* do. Namely, Ember does not embed the entire subordinate object inside the main object. Ember represents all relationships as an id or array of ids. On the trip side, listing many subordinate objects, that 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 objects class name (converted to underscores) with `_id` at the end:

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

10. If, for some reason, you have other custom conversions between server-side and client-side names, you can declare them as part of the `DS.Store` initialization.

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 property is requested by making an Ajax request for more JSON. If it's requesting a single object, the request will have a query parameter `id=3`, or whatever the number is. 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, for example, will convert that internally to `ids = [5, 6, 7]`. Ember will try to cache objects so as to minimize server trips.

There are some implications to the way Ember manages lazy loading. First off, it's a trip to the server and, as such, 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 maintain their own update state. 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.

Often, 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:

```
{
  "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 that it can match the data format we just described.[^fn_store] However, in our specific application we're using Rails, and I do want to cover exactly what I did. 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 back pocket if you are using Rails as a back end.

[^fn_store]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.

The basic idea is that you define a serializer for each of your ActiveRecord models. (Just as with RESTful resources, you don't have to have a one-to-one relationship between models and serializers, but it's the most common case.) 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 either the name of a method of the serializer itself, or the name of an attribute or getter method of the underlying model. You might write a method in the serializer to provide some derived data directly to the client, if you do so, you can access the model being serialized within the serializer as `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 the associated objects 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.[11] 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:

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

```
class TripSerializer < ActiveModel::Serializer
  embed :ids
```

11. I would not be surprised if this changes soon – it seems to be on the roadmap for `ember-data` to handle this case.

```
  attributes :name, :description, :start_date, :end_date,
      :image_name, :slug, :tag_line, :price, :location, :activity, :orders
  has_many :hotels, :key => "hotel_ids", :root => "hotels"
  has_many :extras, :key => "extra_ids", :root => "extras"
end
```

Sample 3-1-12: The full trip serializer

Results in the ID listing of associated objects being part of the JSON, as in example 3–1–6 above. When Ember receives that JSON, it will create the `Trip` objects and hold on to the associated ids, lazily calling the server again when the objects are needed.

For performance reasons, a sideloaded set of objects as in example 3–1–7 is sometimes preferred. When Ember receives a JSON package like that it creates the `Trip` object and all of the subordinate objects in one shot. You can create that behavior in `abstract_model_serializers` by changing the `embed` line to read `embed :ids, :include => true`, which will cause the serializer to include the associated objects as separate, non-nested elements of the JSON.

That's all we need to know about `abstract_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.

Wrapping 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 HotelSerializer < ActiveModel::Serializer
  attributes :id, :description, :price, :name, :trip_id, :nights_ordered
end
```

Sample 3-1-13: The full extra serializer

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

```
class HotelSerializer < ActiveModel::Serializer
  attributes :id, :description, :price, :name, :trip_id, :nights_ordered
end
```

Sample 3-1-14: The full hotel serializer

I've explicitly added `trip_id` as an attribute rather than declare it as `has_one: trip`, which is kind of a potato/potahto situation (honestly, I didn't even realize it as I wrote the code until this very moment.)

We also now need 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)

```
  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 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

Using Ember Data and Associations

I'd like to do a few things on our screen. The detail section that we have on the right hand side of the screen, I'd like it 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 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, we can unit test the sort logic, and we can also acceptance test that all that logic makes its way to the screen.

Lets start with the unit tests, mostly because they are the simplest structure, and will let us focus on the Ember-specific test features.

We have a very similar test structure 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: spec/javascripts/specs/models/hotel_spec.js (Branch: e_04)

```
describe("Hotels", function() {

  beforeEach(function() {
    Ember.testing = true;
  });

  describe("price calcuation", function() {
    it("calcuates the total revenue", function() {
      Ember.run(function() {
        hotel = TimeTravel.Hotel.createRecord({price: "100", nightsOrdered: "3"});
        expect(hotel.get("revenue")).toEqual(300);
      });
    });
  });
});
```

Sample 3-2-1: First test for hotel pricing

Okay, granted we're really just testing multiplication here. That said, I think I see three things that we haven't come across so far in this test.

Working bottom up, we create a new instance of `Hotel` using the Ember-data method `createRecord`, which takes as its 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 to do so would be `create`. Don't worry, if you use `created` to build a `DS.Model`, Ember will give you an error message.[12]

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

Now, in our `beforeEach` callback, we're setting an `Ember.testing` variable to `true`, and our actual test is a callback to a function called `Ember.run`. These two facts are related.

12. Ember also has a mechanism for creating fixture data using a `FixtureStore`. We're not going to talk about it here because a) it's super undocumented as far as I can tell and probably related, b) I couldn't get it to work consistently. If either changes, we'll discuss it.

One of the great things about Ember is that the framework controls all kinds of things that happen basically automatically – there's a rich set of automatic observers and change events that we've only scratched the surface of. 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 when we talk about views, the important point right now is that the run loop happens automatically and governs Embers auto-update and other change-event behaviors.

That's great, but when testing the run loop can sometimes be annoying since it keeps us from being able to control what happens in our tests and might make it hard to isolate unit tests.

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. Things like computed properties are really only accessible 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`.

After all 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('nightsOrdered') * this.get('price');
  }.property("nightsOrdered", "price")
});
```

Sample 3-2-2: 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 3-2-3: 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: spec/javascripts/specs/models/trip_spec.js (Branch: e_04)

```
describe("Trips", function() {

  beforeEach(function() {
    Ember.testing = true;
  });

  describe("price calcuation", function() {
    it("calcuates the total revenue", function() {
      Ember.run(function() {
        trip = TimeTravel.Trip.createRecord({price: 100, orders: 3});
        expect(trip.get("revenue")).toEqual(300);
      });
    });

    it("calculates total revenue for all hotels", function() {
      Ember.run(function() {
        trip = TimeTravel.Trip.createRecord({price: 100, orders: 3});
        trip.get('hotels').pushObject(
            TimeTravel.Hotel.createRecord({revenue: 500}));
        trip.get('hotels').pushObject(
            TimeTravel.Hotel.createRecord({revenue: 600}));
        expect(trip.get("totalHotelRevenue")).toEqual(1100);
      })
    });

    it("calculates total revenue for all extras", function() {
      Ember.run(function() {
        trip = TimeTravel.Trip.createRecord({price: 100, orders: 3});
        trip.get('extras').pushObject(
            TimeTravel.Extra.createRecord({revenue: 500}));
        trip.get('extras').pushObject(
            TimeTravel.Extra.createRecord({revenue: 600}));
        expect(trip.get("totalExtraRevenue")).toEqual(1100);
```

```
      });
    });

    it("calculates all of its revenue", function() {
      Ember.run(function() {
        trip = TimeTravel.Trip.createRecord({price: 100, orders: 3});
        trip.set('totalHotelRevenue', 500);
        trip.set('totalExtraRevenue', 500);
        expect(trip.get('totalRevenue')).toEqual(1300);
      });
    });
  });

});
```

Sample 3-2-4: Revenue tests for trips

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 got through defining as multiplication. 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`. And other cases might have 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 why should the test have to care.

In essence, we're mocking our own Ember-data 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 much more flexible as is.

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 3-2-5: Revenue methods for trip

I count two parts 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 of Ember-specific methods. 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.

In this example, reduce works similarly to how it works in other JS 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 `reduce` 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 it's revenue change. The `@each` is effectively a meta-property declaring exactly that 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 3.3

Accepting What You Can Control

So, here's the deal

NOTE

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

I've been fighting with writing acceptance-level tests in Ember for a some time now, and I haven't yet come up with something that is satisfying and works. I'm going to leave in some discussion of what I've been looking at, and I'm keeping my notes on some of what I've tried. But I'm definitely going to revisit this topic before the final book is released – one reason I've put the big notation here is so that it will be easy for you to tell when this section has changed.

That said, I need to move on and document other parts of Ember so that these betas continue to have value. We'll come back to acceptance tests later.

Okay, back to our book, already in progress...

END OF NOTE

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.

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 Jasmine, we could start up an instance of our Ember app, and simulate routing to a URL, then test the results using Jasmine matchers.
- Moving one more step in, from inside Jasmine, we could set up an Ember controller, associate it with a view, render it, and test the results using Jasmine matchers.

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 if I can't come up with a better way, we'll revisit it.

The router option seems feasible, and I know that other people have done it with other JS testing tools, I haven't completely integrated it into this one yet, but watch this space.

The controller option, which is probably what I'd be most likely to do on my own (in fact, I spent a very long time banging my head against it) seems to be either prohibitively difficult or impossible to do in Ember as Ember is currently configured – it seems to be kind of difficult to

create a controller outside the normal application and routing structure and have all of its necessary connections in place.[13].

Section 3.4

Anyway, Back in the View

Okay, then. We're going to pretend that we have an actual acceptance test and move on to what we need to do in the display to make it possible to actually, you know, 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.

Templates and Attribute Binding

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

13. Specifically, I got in trouble trying to render an index controller that deferred to item controllers, possibly because the item controller implementation is not 100% set as I write this.

First up, a little housekeeping. To prevent the handlebars templates from getting prohibitively long, I'm going to use the `{{template}}` helper to break the index template up into two pieces.[14]

Filename: app/assets/javascripts/templates/index.handlebars (Branch: e_04)

```
<div class="admin_trips span-12">
  {{#each trip in controller}}
    <div class="trip">
      <div class="header" {{action selectTrip trip}}>
        {{trip.name}}
        {{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}}
    {{template "details"}}
  {{/if}}
</div>
```

Sample 3-4-1: Index template with the details broken out

This is exactly what we had before, except we've replaced the detail panel with the directive `{{template "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.

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_04)

14. 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...

```
<h3 class="selected_name">{{selectedTrip.name}}</h3>
<h3>Hotels</h3>
<table>
  {{#each hotel in selectedTrip.hotels itemController="hotel"}}
    {{template "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}}
    {{template "extra_detail"}}
  {{/each}}
  <tr>
    <td>Total</td>
    <td>{{selectedTrip.totalExtrasOrdered}}</td>
    <td></td>
    <td>{{selectedTrip.totalExtraRevenue}}</td>
  </tr>
</table>
```

Sample 3-4-2: Detail template structure

All we have here is a couple of `each` loops, and a couple more `template` directives. One bit we haven't seen before is the `itemController` property of the each loop. This was added for Ember RC1, and allows us to explicitly do in any `each` loop what we previously implicitly did by setting `itemController` at the controller level, namely specify a controller that takes control of the template context for the loop and is merged with the current loop value. In this case, we'll have a `HotelController`, by way of naming inflection from declaring `itemController=hotel`, and we'll use that to store some values and do some event handling.

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_04)

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

Sample 3-4-3: 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_04)

```
<tr>
  <th>{{hotel.name}}</th>
  <td>
    <span {{bindAttr class="editing:hidden:normal"}}
          {{action startEditing}}>
      {{hotel.nightsOrdered}}
    </span>
    <span {{bindAttr class="editing:normal:hidden"}}>
      {{view Ember.TextField valueBinding="nightsOrderedInput"
          size="4"
          action="endEditing"}}
    </span>
  </td>
  <td>${{hotel.price}}</td>
  <td>${{hotel.revenue}}</td>
</tr>
```

Sample 3-4-4: 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.

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

To explain why Ember needs the `bindAttr` 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 associate property value. We've also mentioned that whenever you do 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, 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, `{{bindAttr}}` – a Handlebars helper that lets you create dynamic HTML attributes.[15]

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 may also have extensions to automate this process.

The simple form of `{{bindAttr}}` 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 `{{bindAttr 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.

Naturally, it's not quite that simple. Specifically, we are taking 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 `bindAttr` for a class is also a space-delimited list. Only the entries in the list are properties and not literal values.

So, if you do something like `<div {{bindAttr 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 `bindAttr` 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 specialer 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 `{{bindAttr 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 `{{bindAttr class="isVisible:visible"}}` or `{{bindAttr 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.

15. That said, I wouldn't be surprised if the Ember team figures out a way to make this work without `bindAttr` at some point in the future.

Finally, since `bindAttr` 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 `bindAttr`. 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.[16] In this example, we are reusing a pre-defined Ember component that manages a browser text field.

16. That said, the largest Ember application that I've worked on in anger actually does use Views extensively. All I'll say about that is a) it's clear we're fighting the framework a bit and b) we did it because Ember's ability to embed a controller to take over part of a display is not yet fully baked, and was even less so when we started.

Ember ships with four built-in views: `Ember.Checkbox`, `Ember.TextField`, `Ember.Select`, and `Ember.TextArea`[17]. 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, when one change, 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 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.[18]

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 3-4-5:

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

While that's not much of a benefit in this case, there are benefits to using bindings to easily share values across layers of the application. In our case, the `valueBinding` allows the controller easy access to the value entered in the form (updated on a keystroke-by-keystroke

17. 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.

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

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:

Filename: app/assets/javascripts/controllers/hotel_controller.js (Branch: e_04)

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

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

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

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

})
```

Sample 3-4-6: 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 `bindAttr` class for the text-only span to be `normal` and the `bindAttr` 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 `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 `bindAttr` 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 `bindAttr`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.

Important Note.

You've reached the end of what I've written so far. Luckily, both you and I know that there's more to come. I'm trying to release the book every Monday, which means you should expect more book by about March 18, 2012. Follow `@noelrap` on Twitter to keep up to date.

Thanks,

The Management

Chapter 4

Acknowledgements

A number of people helped in the creation of this book, whether they knew it or not.

As you may know, this book began life with a non-me publisher. Technical reviewers of the manuscript at that point included Pete Campbell, Kevin Gisi, Kaan Karaca, Evan Light, Chris Powers, Wes Reisz, Martijn Reuvers, Barry Rowe, Justin Searls, and Stephen Wolff. Kay Keppler and Susannah Pfalzer acted as editors. Thanks to all of them.

Trek Glowaki provided valuable technical sanity checks on Ember.js.

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

Avdi Grimm provided a lot of useful tools for self-publishing.

I've been lucky enough to work with people who were willing to share JavaScript experience with me, including, but not limited to: Dave Giunta, Sean Massa, Fred Polgardy, and Chris Powers.

Justin Searls and his Jasmine advocacy and toolkit have been a huge help.

One of the great things about self-publishing is that people take the time to help improve the book by pointing out mistakes and typos. Thanks to Pierre Nouaille-Degorce, Sean Massa, Vlad Ivanovic and Nicolas Dermine for particular efforts.

Emma Rosenberg-Rappin helped me design the noelrappin.com web site, pick cover fonts, and also did some copyediting.

Elliot Rosenberg-Rappin inspired the idea of a spaceship cover and laughed at some of my jokes.

This book, and many other wonderful things, would not exist without my wife Erin, who has been nothing but supportive through the ups and downs of this project. She's amazing.

Chapter 5

Colophon

Master Space and Time with JavaScript was written using the PubRx workflow, available at https://github.com/noelrappin/pub_rx. The initial text is written in MultiMarkdown, home page http://fletcherpenney.net/multimarkdown/. PubRx augments MultiMarkdown with extra descriptors allowing for page layout effects, such as sidebars, that Markdown does not manage on its own.

PDF conversion is managed by PrinceXML http://www.princexml.com, with a little bit of additional processing by PubRx. EPub and Mobi generation is managed by using the Calibre http://calibre-ebook.com/ command line interface, using a method described by Avdi Grimm's Orgpress https://github.com/avdi/orgpress.

For the PDF version, the header font is Museo and the body font is Museo Sans. Both are free fonts from the exljbris font foundry http://www.exljbris.com/. The code font is "M+ 1c" from M+ Fonts http://mplus-fonts.sourceforge.jp/. On the cover, the title font is Bangers by Vernon Adams http://code.google.com/webfonts/specimen/Bangers, and the signature font is Digital Delivery, from Comicraft http://www.comicbookfonts.com/, and designed by Richard Starkings & John 'JG' Roshell. Cover image credit is on the title page up front.

Chapter 6
Changelog

Release 001: 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