Backbone.js *on* Rails

Build snappier, more interactive apps with cleaner code and better tests in less time

Backbone.js on Rails

thoughtbot Jason Morrison Chad Pytel
Nick Quaranto Harold Giménez Joshua Clayton
Gabe Berke-Williams Chad Mazzola

June 22, 2012

Contents

Chapter 1

Introduction

The shift to client-side web applications

Modern web applications have become increasingly rich, shifting their complexity onto the client side. While there are very well-understood approaches embodied in mature frameworks to organize server-side code, frameworks for organizing your client-side code are newer and generally still emerging. Backbone is one such library that provides a set of structures to help you organize your JavaScript code.

Libraries like jQuery have done a great deal to help abstract inconsistencies across browsers and provide a high-level API for making AJAX requests and performing DOM manipulation, but larger and richer client-side applications that lack decoupled and modular organizational structures often fall victim to the same few kinds of technical debt.

These apps are often highly asynchronous and the "path of least resistance" implementation is often to have deeply nested callbacks to describe asynchronous behavior, with nested Ajax calls and success/failure conditional concerns going several layers deep.

Rich client-side applications almost always involve a layer of state and logic on the client side. One way to implement this is to store domain objects or business logic state in the DOM. However, storing state in the DOM, stashing your application's data in hidden `<div>` elements that you clone, graft, and toggle into and out of view, or reading and writing to lengthy sets of HTML `data-*` attributes can quickly get cumbersome and confusing.

A third common feature in rich client-side apps is the presentation of multiple views on a single domain object. Consider a web conferencing application with multiple views on the members of your contact list - each contact is rendered in brief inside a list view, and in more specificity in a detail view. Additionally, your

conference call history includes information about the people who participated. Each time an individual contact's information changes, this information needs to cascade to all the view representations.

This often leads to a tight coupling of persistence and presentation: invoking `$.ajax` to save a user's update and then updating several specific DOM elements upon success.

By separating business logic, persistence, and presentation concerns, and providing a decoupled, event-driven way to cascade changes through a system of observers, each module of code is more well-encapsulated and expresses a cohesive set of responsibilities without being coupled to outside concerns. Your application code becomes easier to test, modify, and extend, and you can better manage its complexity while its feature set grows.

Granted, you can thoughtfully organize your code in a clean, coherent manner without using an external library. However, using a library like Backbone helps you get started more quickly, reduces the number of decisions to make, and provides a common vocabulary for your team members or open source contributors.

If you're coming from a Rails background, you understand that a large part of Rails' value is expressing and implementing highly-opinionated conventions that guide development decisions. Backbone doesn't do this. Instead of trying to serve as "the one way," or an opinionated framework like Rails, Backbone provides a set of structures that help you organize your application by building your own framework with its own set of conventions.

Goals for this book

This book aims to cover topics that are of interest when integrating Backbone into a Rails application. The primary Backbone documentation is quite good, and concisely readable. While we'll touch on introductory topics when they are critical to understand the points at hand, this book does not aim to provide an introduction to Backbone, and generally assumes the reader can lean on the Backbone documentation to explain the details of some concepts.

This book also does not aim to provide a comprehensive mapping of all possible solutions to problem domains, but rather to describe the best approaches we have found for solving problems and organizing applications using both Rails and Backbone.

Alternatives to Backbone

Web applications are pushing an increasing amount of responsibility to the client. The user experience can be quite enjoyable, but deeply nesting callbacks

and relying on the DOM for app state are not. Fortunately, there is a host of new JavaScript client-side frameworks blossoming, and you have no shortage of options.

Knockout and Angular support declarative view-bindings and the Model-View-View Model (MVVM) pattern. Cappuccino and SproutCore deliver a rich library of UI controls for building desktop-like applications. JavaScriptMVC provides quite a bit of structure, including dependency management and build tools. Spine is perhaps the most similar to Backbone, but takes an opinionated stance to emphasize completely asynchronous client-server interactions for a faster user experience. Ember, originally a SproutCore rewrite, provides a host of conventions including two-way bindings, computed properties, and auto-updating templates.

Backbone favors a pared-down and flexible approach. The code you write ends up feeling very much like plain JavaScript. Although you will need to write some of your own conventions, Backbone is built to be easy to change: the source is small, well annotated, and modularly designed. It is small and flexible enough to smoothly introduce into an existing application, but provides enough convention and structure to help you organize your JavaScript. Additionally, a growing community of users brings with it a rich ecosystem of plugins, blog articles, and support.

The example application

The example application is a classic todo item manager. This is a popular example, and for good reason: The concepts and domain are familiar, and room is left to explore interesting implementations like deferred loading and file attachment.

The application uses Rails 3.2.6 and Ruby 1.9.2. We provide an `.rvmrc`.

The included JavaScript libraries are non-minified for readability. This is a general good practice, and the Rails asset pipeline will properly package the assets for production.

While Rails provides the ability to write in CoffeeScript, we have decided to make all of the example code normal JavaScript so as to reduce the number of new things introduced at once.

The example application comes with a full test suite. The README in the `example_app` root directory has instructions for bootstrapping the app and running all the tests.

Chapter 2

Getting up to speed

Backbone.online resources

This book is not an introduction, and assumes you have some knowledge of Javascript and of Backbone. Fortunately, there is solid documentation available to get you up to speed on Backbone.

The online documentation for Backbone is very readable:

http://documentcloud.github.com/backbone/

The GitHub wiki for Backbone links to a large number of tutorials and examples:

https://github.com/documentcloud/backbone/wiki/Tutorials%2C-blog-posts-and-example-sites

PeepCode is producing a three-part series on getting up to speed on Backbone:

http://peepcode.com/products/backbone-js

Finally, Nick Gauthier and Chris Strom have published an ebook with several real-world examples they have extracted from production code:

http://recipeswithbackbone.com/

JavaScript resources

JavaScript: The Good Parts by Douglas Crockford is highly recommended for any JavaScript developer.

Test-Driven JavaScript Development by Christian Johansen teaches not only the ins and outs of how to test-drive your code, but also covers good fundamental JavaScript development practices and takes a deep dive on language fundamentals.

Finally, *JavaScript Web Applications* by Alex MacCaw provides broad and thorough coverage on developing well-organized client side applications.

Chapter 3

Organization

Backbone and MVC

Model–View–Controller (MVC) is a software architectural pattern used in many applications to isolate domain or business logic (the application logic for the user) from the user interface (input and presentation).

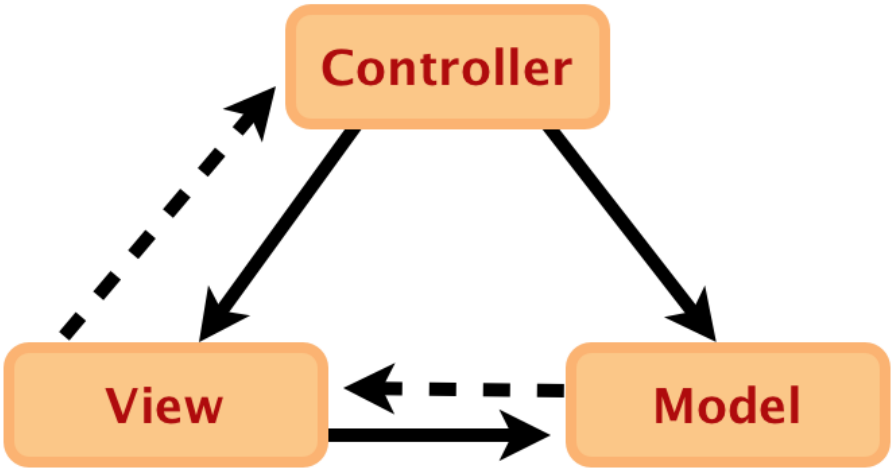

Figure 3.1: Model-View-Controller concept

In the above diagram, a solid line represents a direct association and a dashed line represents an indirect association, such as one mediated by an observer.

As a user of Rails, you're likely already familiar with the concept of MVC and the benefits that the separation of concerns can provide. However, Rails

itself is, technically, not traditional MVC, but a pattern called Model2. A traditional MVC is event-based, and views are bound directly to models as observers, updating themselves when the model changes.

Given that Javascript has events, and that much of the interactions between the different components of Backbone in the browser are not limited to request/response, Backbone can be structured as an actual MVC architecture.

That said, technically speaking, Backbone is *not* MVC, and Backbone acknowledged this when it renamed "Controllers" to "Routers" in version 0.5.0.

What is Backbone then, if not MVC? Classically, views handled the presentation of information, and controllers would take the user input and decide what to do with it. In Backbone, these two concerns are merged into view classes, which are responsible for presentation as well as both establishing and responding to UI event bindings.

What goes where

Part of the initial learning curve of Backbone can be figuring out what goes where, and mapping it to expectations set by working with Rails. In Rails we have Models, Views, Controllers, and Routers. In Backbone, we have Models, Collections, Views, Templates, and Routers.

It's important to note that, although Rails and Backbone share several concept names, several of which have significant overlap, you shouldn't try to map your understanding of one directly onto the other. That said, it's valuable to draw similarities to help ease the learning curve.

The models in Backbone and Rails are fairly analogous - each represent objects in your domain, and both mix the concerns of domain logic with persistence. In Rails, the persistence is usually made to a database, and in Backbone.js it's generally made to a remote HTTP JSON API.

Backbone collections are just ordered sets of models. Because it lacks controllers, Backbone routers and views work together to pick up the functionality provided by Rails controllers. Finally, in Rails, when we say "views," we actually mean "templates," as Rails does not provide for view classes out of the box. In Backbone, however, you have a separation between the view class and the HTML templates that they use.

Once you introduce Backbone into your application, you grow the layers in your stack by four levels. This can be daunting at first, and frankly, at times it can be difficult to keep straight everything that's going on in your application. Ultimately, though, the additional organization and functionality of Backbone outweighs the costs - let's break it down.

| Rails | |————| | Model | | Controller | | View |

| Backbone | |——————-| | Model and Collection | | Router | | View | | Template |

In a typical Rails and Backbone application, the initial interaction between the layers will be as follows:

- A request from a user comes in; the **Rails router** identifies what should handle the request, based on the URL
- The **Rails controller action** to handle the request is called; some initial processing may be performed
- The **Rails view template** is rendered and returned to the user's browser
- The **Rails view template** will include **Backbone initialization**; usually this is populating some *Backbone collections* as sets of **Backbone models** with JSON data provided by the **Rails view**
- The **Backbone router** determines which of its methods should handle the display, based on the URL
- The **Backbone router** calls that method; some initial processing may be performed, and one or more **Backbone views** are rendered
- The **Backbone view** reads **templates** and uses **Backbone** models to render itself onto the page

At this point, the user will see your application in their browser and be able to interact with it. The user interacting with elements on the page will trigger actions to be taken at any level of the above sequence: **Backbone model**, **Backbone views**, **Backbone routers**, or requests to the remote server.

Requests to the remote server may be any one of the following:

- Normal requests that don't hit Backbone and trigger a full page reload
- Normal Ajax requests, not using Backbone at all
- Ajax requests from the **Backbone model** or **Backbone collection**, communicating with Rails via JSON

Generally speaking, by introducing Backbone into our application we'll reduce the first two types of requests, moving the bulk of client/server interaction to requests encapsulated inside domain objects like Backbone models.

Namespacing your application

You will want to create an object in JavaScript in which your Backbone application will reside. This variable will serve as a namespace for your Backbone application. Namespacing all of the JavaScript is desirable to avoid potential collisions in naming. For example, it's possible that a JavaScript library you want to use might also create a task variable. If you haven't namespaced your task model, this would conflict.

This variable includes a place to hold models, collections, views, and routes, and an `initialize` function which will be called to initialize the application.

Typically, initializing your application will involve creating a router and starting Backbone history to route the initial URL fragment. This app variable will look like the following:

```
var ExampleApp = {
  Models: {},
  Collections: {},
  Views: {},
  Routers: {},
  initialize: function() {
    new ExampleApp.Routers.Tasks();
    Backbone.history.start();
  }
};
```

You can find a more fully fleshed-out version of this file in the example app in `app/assets/javascripts/example_app.js`.

Mixins

Backbone provides a basic mechanism for inheritance. Sometimes, you'll want to build a collection of related, reusable behavior and include that in several classes that already inherit from a Backbone base class. In these cases, you'll want to use a mixin.

Backbone includes Backbone.Events as an example of a mixin.

Here, we create a mixin named `Observer` that contains behavior for binding to events in a fashion that can be cleaned up later:

```
var Observer = {
  bindTo: function(source, event, callback) {
    source.on(event, callback, this);
```

```
    this.bindings = this.bindings || [];
    this.bindings.push({ source: source, event: event, callback: callback });
  },

  unbindFromAll: function() {
    _.each(this.bindings, function(binding) {
      binding.source.off(binding.event, binding.callback);
    });
    this.bindings = [];
  }
};
```

We can mix `Observer` into a class by using Underscore.js' `_.extend` on the prototype of that class:

```
SomeCollectionView = Backbone.View.extend({
  initialize: function() {
    this.bindTo(this.collection, "change", this.render);
  },

  leave: function() {
    this.unbindFromAll(); // calling a method defined in the mixin
    this.remove();
  }
});

_.extend(SomeCollectionView.prototype, Observer);
```

Chapter 4

Rails Integration

Organizing your Backbone code in a Rails app

When using Backbone in a Rails app, you'll have two kinds of Backbone-related assets: classes and templates.

Rails 3.0 and prior

With Rails 3.0 and prior, store your Backbone classes in `public/javascripts`:

```
public/
  javascripts/
    jquery.js
    jquery-ui.js
    collections/
      users.js
      todos.js
    models/
      user.js
      todo.js
    routers/
      users_router.js
      todos_router.js
    views/
      users/
        users_index.js
        users_new.js
        users_edit.js
```

```
      todos/
        todos_index.js
```

If you are using templates, we prefer storing them in `app/templates` to keep them separated from the server views:

```
app/
  views/
    pages/
      home.html.erb
      terms.html.erb
      privacy.html.erb
      about.html.erb
  templates/
    users/
      index.jst
      new.jst
      edit.jst
    todos/
      index.jst
      show.jst
```

On Rails 3.0 and prior apps, we use Jammit for packaging assets and precompiling templates:

http://documentcloud.github.com/jammit/

http://documentcloud.github.com/jammit/#jst

Jammit will make your templates available in a top-level JST object. For example, to access the above todos/index.jst template, you would refer to it as:

```
JST['todos/index']
```

Variables can be passed to the templates by passing a Hash to the template, as shown below.

```
JST['todos/index']({ model: this.model })
```

Jammit and a JST naming gotcha

One issue with Jammit that we've encountered and worked around is that the JST template path can change when adding new templates. Let's say you place templates in `app/templates`. You work for a while on the "Tasks" feature, placing templates under `app/templates/tasks`. So, `window.JST` looks something like:

```
JST['form']
JST['show']
JST['index']
```

Now, you add another directory under `app/templates`, say `app/templates/user`. Now, templates with colliding names in JST references are prefixed with their parent directory name so they are unambiguous:

```
JST['form'] // in tasks/form.jst
JST['tasks/show']
JST['tasks/index']
JST['new']  // in users/new.jst
JST['users/show']
JST['users/index']
```

This breaks existing JST references. You can work around this issue by applying the following monkeypatch to Jammit, in `config/initializers/jammit.rb`:

```
Jammit::Compressor.class_eval do
  private
  def find_base_path(path)
    File.expand_path(Rails.root.join('app','templates'))
  end
end
```

As applications are moving to Rails 3.1 or above, they're also moving to Sprockets for the asset packager. Until then, many apps are using Jammit for asset packaging. We have an open issue and workaround:

https://github.com/documentcloud/jammit/issues/192

Rails 3.1 and above

Rails 3.1 introduced the asset pipeline, which uses the Sprockets library for preprocessing and packaging assets.

To take advantage of the built-in asset pipeline, organize your Backbone templates and classes in paths available to it: classes go in `app/assets/javascripts/`, and templates go alongside, in `app/assets/templates/`:

```
app/
  assets/
    javascripts/
      collections/
```

```
      todos.js
    models/
      todo.js
    routers/
      todos_router.js
    views/
      todos/
        todos_index.js
  templates/
    todos/
      index.jst.ejs
      show.jst.ejs
```

In Rails 3.1 and above, jQuery is provided by the `jquery-rails` gem, and no longer needs to be included in your directory structure.

Using Sprockets' preprocessors, we can use templates as before. Here, we're using the EJS template preprocessor to provide the same functionality as Underscore.js' templates. It compiles the `*.jst` files and makes them available on the client side via the `window.JST` object. Identifying the `.ejs` extension and invoking EJS to compile the templates is managed by Sprockets, and requires the `ejs` gem to be included in the application Gemfile.

> Underscore.js templates: http://documentcloud.github.com/underscore/#template
>
> EJS gem: https://github.com/sstephenson/ruby-ejs
>
> Sprockets support for EJS: https://github.com/sstephenson/sprockets/blob/master/lib/sprockets/ejs_template.rb

To make the `*.jst` files available and create the `window.JST` object, require them in your application.js Sprockets manifest:

```
//  other application requires
//= require_tree ../templates
//= require_tree .
```

Load order for Backbone and your Backbone app is very important. jQuery and Underscore must be loaded before Backbone, then the Rails authenticity token patch must be applied. Then your models must be loaded before your collections (because your collections will reference your models) and then your routers and views must be loaded.

Fortunately, Sprockets can handle this load order for us. When all is said and done, your application.js Sprockets manifest will look as shown below:

```
//= require jquery
//= require jquery_ujs
//= require jquery-ui-1.8.18.custom.min
//
//= require underscore
//= require json2
//= require backbone
//= require backbone-support
//
//= require backbone-forms.js
//= require jquery-ui-editors.js
//= require uploader.js
//
//= require example_app
//
//= require_tree ./models
//= require_tree ./collections
//= require_tree ./views
//= require_tree ./routers
//= require_tree ../templates
//= require_tree .
```

The above is taken from the example application included with this book. You can view it at `example_app/app/assets/javascripts/application.js`.

An overview of the stack: connecting Rails and Backbone

By default, Backbone communicates with your Rails application via JSON HTTP requests. If you've ever made a JSON API for your Rails app, then for the most part, this will be very familiar. If you have not made a JSON API for your Rails application before, lucky you! It's pretty straightforward.

This section will briefly touch on each of the major parts of an application using both Rails and Backbone. We'll go into more detail in later chapters, but this should give you the big picture of how the pieces fit together.

Setting up models

In our example application, we have a Task model, exposed via a JSON API at `/tasks`. The simplest Backbone representation of this model would be as shown below:

```
var Task = Backbone.Model.extend({
  urlRoot: '/tasks'
});
```

The `urlRoot` property above describes a base for the server-side JSON API that houses this resource. Collection-level requests will occur at that root URL, and requests relating to instances of this model will be found at `/tasks/:id`.

It's important to understand that there is no need to have a one-to-one mapping between Rails models and Backbone models. Backbone models instead correspond with RESTful resources. Since your Backbone code is in the presentation tier, it's likely that some of your Backbone models may end up providing only a subset of the information present in the Rails models, or they may aggregate information from multiple Rails models into a composite resource.

In Rails, it's possible to access individual tasks, as well as all tasks (and query all tasks) through the same `Task` model. In Backbone, models only represent the singular representation of a `Task`. Backbone splits out the plural representation of `Tasks` into `Collections`.

The simplest Backbone collection to represent our `Tasks` would be the following.

```
var Tasks = Backbone.Collection.extend({
  model: Task
});
```

If we specify the URL for `Tasks` in our collection instead, then models within the collection will use the collection's URL to construct their own URLs, and the `urlRoot` no longer needs to be specified in the model. If we make that change, then our collection and model will be as follows.

```
var Tasks = Backbone.Collection.extend({
  model: Task,
  url: '/tasks'
});

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

Notice in the above model definitions that there is no specification of the attributes on the model. As in ActiveRecord, Backbone models get their attributes from the data used to populate them at runtime. In this case, this schema and data are JSON responses from the Rails server.

The default JSON representation of an ActiveRecord model is an object that includes all the model's attributes. It does not include the data for any related models or any methods on the model, but it does include the ids of any

`belongs_to` relations as those are stored in a `relation_name_id` attribute on the model.

The JSON representation of your ActiveRecord models will be retrieved by calling `to_json` on them, which returns a string of JSON. Customize the output of `to_json` by overriding the `as_json` method in your model, which returns a Ruby data structure like a Hash or Array which will be serialized into the JSON string. We'll touch on this more later in the section, "Customizing your Rails-generated JSON."

Setting up Rails controllers

The Backbone models and collections will talk to your Rails controllers. The most basic pattern is one Rails controller providing one family of RESTful resource to one Backbone model.

By default, Backbone models communicate in the normal RESTful way that Rails controllers understand, using the proper verbs to support the standard RESTful Rails controller actions: index, show, create, update, and destroy. Backbone does not make any use of the new action.

Therefore, it's just up to us to write a *normal* RESTful controller. The newest and most succinct way to structure these is to use the `respond_with` method, introduced in Rails 3.0.

When using `respond_with`, declare supported formats with `respond_to`. Inside individual actions, you then specify the resource or resources to be delivered using `respond_with`:

```
class TasksController < ApplicationController::Base
  respond_to :html, :json

  def index
    respond_with(@tasks = Task.all)
  end
end
```

In the above example tasks controller, the `respond_to` line declares that this controller should respond to requests for both the HTML and JSON formats. Then, in the index action, the `respond_with` call will build a response according to the requested content type (which may be HTML or JSON in this case) and provided resource, `@tasks`.

Validations and your HTTP API

If a Backbone model has a `validate` method defined, it will be validated on the client side, before its attributes are set. If validation fails, no changes to the

model will occur, and the "error" event will be fired. Your `validate` method will be passed the attributes that are about to be updated. You can signal that validation passed by returning nothing from your `validate` method. You signify that validation has failed by returning something from the method. What you return can be as simple as a string, or a more complex object that describes the error in all its gory detail.

The amount of validation you include on the client side is essentially a tradeoff between interface performance and code duplication. It's important for the server to make the last call on validation.

So, your Backbone applications will likely rely on at least some server-side validation logic. Invalid requests return non-2xx HTTP responses, which are handled by error callbacks in Backbone:

```
task.save({ title: "New Task title" }, {
  error: function() {
    // handle error from server
  }
});
```

The error callback will be triggered if your server returns a non-2xx response. Therefore, you'll want your controller to return a non-2xx HTTP response code if validations fail.

A controller that does this would appear as shown in the following example:

```
class TasksController < ApplicationController::Base
  respond_to :json

  def create
    @task = Task.new(params[:task])
    if @task.save
      respond_with(@task)
    else
      respond_with(@task, :status => :unprocessable_entity)
    end
  end
end
```

The default Rails responders will respond with an unprocessable entity (422) status code when there are validation errors, so the action above can be refactored:

```
class TasksController < ApplicationController::Base
  respond_to :json
```

```
  def create
    @task = Task.new(params[:task])
    @task.save
    respond_with @task
  end
end
```

Your error callback will receive both the model as it was attempted to be saved and the response from the server. You can take that response and handle the errors returned by the above controller in whatever way is fit for your application.

A few different aspects of validations that we saw here are covered in other sections of this book. For more information about validations, see the "Validations" section of the "Models and Collections" chapter. For more information about reducing redundancy between client and server validations, see the "Duplicating business logic across the client and server" section of the "Models and Collections" chapter. For more information about handling and displaying errors on the client side, see the "Forms" section of the "Routers, Views and Templates" chapter.

Setting Up Views

Most Backbone applications will be a single-page app, or "SPA." This means that your Rails application handles two jobs: First, it renders a single page which hosts your Backbone application and, optionally, an initial data set for it to use. From there, ongoing interaction with your Rails application occurs via HTTP JSON APIs.

For our example application, this host page will be located at `Tasks#index`, which is also routed to the root route.

You will want to create an object in JavaScript for your Backbone application. Generally, we use this object as a top-level namespace for other Backbone classes, as well as a place to hold initialization code. For more information on this namespacing see the "Namespacing your application" section of the Organization chapter.

This application object will look like the following:

```
var ExampleApp = {
  Models: {},
  Collections: {},
  Views: {},
  Routers: {},
  initialize: function(data) {
```

```
    var tasks = new ExampleApp.Collections.Tasks(data.tasks);
    new ExampleApp.Routers.Tasks({ tasks: tasks });
    Backbone.history.start();
  }
};
```

You can find this file in the example app in `app/assets/javascripts/example_app.js`.

IMPORTANT: You must instantiate a Backbone router before calling `Backbone.history.start()` otherwise `Backbone.history` will be `undefined`.

Then, inside `app/views/tasks/index.html.erb` you will call the `initialize` method. You will often bootstrap data into the Backbone application to provide initial state. In our example, the tasks have already been provided to the Rails view in an `@tasks` instance variable:

```
<%= content_for :javascript do -%>
  <%= javascript_tag do %>
    ExampleApp.initialize({ tasks: <%== @tasks.to_json %> });
  <% end %>
<% end -%>
```

The above example uses ERB to pass the JSON for the tasks to the `initialize` method, but we should be mindful of the XSS risks that dumping user-generated content here poses. See the "Encoding data when bootstrapping JSON data" section in the "Security" chapter for a more secure approach.

Finally, you must have a Router in place that knows what to do. We'll cover routers in more detail in the "Routers, Views and Templates" chapter.

```
ExampleApp.Routers.Tasks = Backbone.Router.extend({
  routes: {
    "": "index"
  },

  index: function() {
    // We've reached the end of Rails integration - it's all Backbone from here!

    alert('Hello, world!  This is a Backbone router action.');

    // Normally you would continue down the stack, instantiating a
    // Backbone.View class, calling render() on it, and inserting its element
    // into the DOM.

    // We'll pick back up here in the "Converting Views" section.
  }
});
```

The example router above is the last piece needed to complete our initial Backbone infrastructure. When a user visits `/tasks`, the `index.html.erb` Rails view will be rendered, which properly initializes Backbone and its dependencies and the Backbone models, collections, routers, and views.

Customizing your Rails-generated JSON

There are a few common things you'll do in your Rails app when working with Backbone.

First, it's likely that you'll want to switch from including all attributes, which is the default, to delivering some subset.

This can be done by specifying explicitly only the attributes that are to be included (whitelisting), or specifying the attributes that should *not* be included (blacklisting). Which one you choose will depend on how many attributes your model has and how paranoid you are about something important appearing in the JSON when it shouldn't be there.

If you're concerned about sensitive data unintentionally being included in the JSON when it shouldn't be, then you'll want to whitelist attributes into the JSON with the `:only` option:

```
def as_json(options = {})
  super(options.merge(:only => [ :id, :title ]))
end
```

The above `as_json` override will make it so that the JSON will *only* include the id and title attributes, even if there are many other attributes on the model.

If instead you want to include all attributes by default and just exclude a few, you accomplish this with the `:except` option:

```
def as_json(options = {})
  super(options.merge(:except => [ :encrypted_password ]))
end
```

Another common customization you will want to do in the JSON is include the output of methods (say, calculated values) on your model. This is accomplished with the `:methods` option, as shown in the following example:

```
def as_json(options = {})
  super(options.merge(:methods => [ :calculated_value ]))
end
```

The final thing you'll most commonly do with your JSON is include related objects. If the `Task` model `has_many :comments`, include all of the JSON for comments in the JSON for a Task with the `:include` option:

```
def as_json(options = {})
  super(options.merge(:include => [ :comments ]))
end
```

As you may have guessed, you can then customize the JSON for the comments by overriding the `as_json` method on the `Comment` model.

While this is the most common set of `as_json` options you'll use when working with Backbone, it certainly isn't all of them. The official, complete documentation for the `as_json` method can be found here: http://apidock.com/rails/ActiveModel/Serializers/JSON/as_json

ActiveRecord::Base.include_root_in_json

Depending on the versions, Backbone and Rails may have different expectations about the format of JSON structures; specifically, whether or not a root key is present. When generating JSON from Rails, this is controlled by the ActiveRecord setting `ActiveRecord::Base.include_root_in_json`.

```
 > ActiveRecord::Base.include_root_in_json = false
 > Task.last.as_json
=> {"id"=>4, "title"=>"Enjoy a three mile swim"}

 > ActiveRecord::Base.include_root_in_json = true
 > Task.last.as_json
=> {"task"=>{"id"=>4, "title"=>"Enjoy a three mile swim"}}
```

In Rails 3.0, `ActiveRecord::Base.include_root_in_json` is set to "true." Starting with 3.1, it defaults to "false." This reversal was made to simplify the JSON returned by default in Rails application, but it is a fairly big change from the default behavior of Rails 3.0.

Practically speaking, this change is a good one, but take particular note if you're upgrading an existing Rails 3.0 application to Rails 3.1 or above and you already have a published API; you may need to expose a new version of your API.

From the Backbone side, the default behavior expects no root node. This behavior is defined in a few places: `Backbone.Collection.prototype.parse`, `Backbone.Model.prototype.parse`, and `Backbone.Model.prototype.toJSON`:

```
_.extend(Backbone.Collection.prototype, Backbone.Events, {
  // http://documentcloud.github.com/backbone/#Collection-parse
  parse : function(resp, xhr) {
    return resp;
  },

  // snip...
});

_.extend(Backbone.Model.prototype, Backbone.Events, {
  // http://documentcloud.github.com/backbone/#Model-toJSON
  toJSON : function() {
    return _.clone(this.attributes);
  },

  // http://documentcloud.github.com/backbone/#Model-parse
  parse : function(resp, xhr) {
    return resp;
  },

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

If you need to accept JSON with a root node, you can override `parse` in each of your models, or override the prototype's function. You'll need to override it on the appropriate collection(s), too.

If you need to send JSON back to a server that includes a root node, you can override `toJSON`, per model or across all models. When you do this, you'll need to explicitly specify the name of the root key. We use a convention of a `modelName` function on your model to provide this:

```
Backbone.Model.prototype.toJSON = function() {
  var hashWithRoot = {};
  hashWithRoot[this.modelName] = this.attributes;
  return _.clone(hashWithRoot);
};

var Task = Backbone.Model.extend({
  modelName: "task",

  // ...
});
```

Converting an existing page/view area to use Backbone

This section is meant to get you started understanding how Backbone views work by illustrating the conversion of a Rails view to a Backbone view.

Its important to note that a Rails view is not directly analogous to a Backbone view. In Rails, the term "view" usually refers to an HTML template, where Backbone views are classes that contain event handling and presentation logic.

Consider the following Rails view for a tasks index:

```
<h1>Tasks</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Completed</th>
  </tr>

  <% @tasks.each do |task| %>
    <tr>
      <td><%= task.title %></td>
      <td><%= task.completed %></td>
    </tr>
  <% end %>
</table>
```

So far, we have the Backbone `Task` model and collection and the Rails `Task` model and controller discussed above, and we're bootstrapping the Backbone app with all the tasks. Next, we will create a Backbone view which will render a corresponding Backbone template.

A Backbone view is a class that is responsible for rendering the display of a logical element on the page. A view also binds to DOM events occurring within its DOM scope that trigger various behaviors.

We'll start with a basic view that achieves the same result as the Rails template above, rendering a collection of tasks:

```
ExampleApp.Views.TasksIndex = Backbone.View.extend({
  render: function () {
    this.$el.html(JST['tasks/index']({ tasks: this.collection }));
    return this;
  }
};
```

The `render` method above renders the `tasks/index` JST template, passing the collection of tasks into the template.

Each Backbone view has an element that it stores in `this.$el`. This element can be populated with content, although it's a good practice for code outside the view to actually insert the view into the DOM.

We'll update the Backbone route to instantiate this view, passing in the collection for it to render. The router then renders the view, and inserts it into the DOM:

```
ExampleApp.Routers.Tasks = Backbone.Router.extend({
  routes: {
    "": "index"
  },

  index: function() {
    var view = new ExampleApp.Views.TasksIndex({ collection: ExampleApp.tasks });
    $('body').html(view.render().$el);
  }
});
```

Now that we have the Backbone view in place that renders the template, and it's being called by the router, we can focus on converting the above Rails view to a Backbone template.

Backbone depends on Underscore.js which, among many things, provides templating. The delimiter and basic concepts used for Underscore.js templates and ERB are the same. When converting an existing Rails application to Backbone, this similarity can help ease the transition.

The `tasks/index` JST template does two things:

- Loops over all of the tasks
- For each task, it outputs the task title and completed attributes

Underscore.js provides many iteration functions that will be familiar to Rails developers such as `_.each`, `_.map`, and `_.reject`. Backbone also proxies to Underscore.js to provide these iteration functions as methods on `Backbone.Collection`.

We'll use the `each` method to iterate through the `Tasks` collection that was passed to the view, as shown in the converted Underscore.js template below:

```
<h1>Tasks</h1>
```

```
<table>
  <tr>
    <th>Title</th>
    <th>Completed</th>
  </tr>

  <% tasks.each(function(model) { %>
    <tr>
      <td><%= model.escape('title') %></td>
      <td><%= model.escape('completed') %></td>
    </tr>
  <% }); %>
</table>
```

In Rails 3.0 and above, template output is HTML-escaped by default. In order to ensure that we have the same XSS protection as we did in our Rails template, we access and output the Backbone model attributes using the `escape` method instead of the normal `get` method.

Breaking out the TaskView

In Backbone, views are often bound to an underlying model, re-rendering themselves when the model data changes. Consider what happens when any task changes data with our approach above; the entire collection must be re-rendered. It's useful to break up these composite views into two separate classes, each with their own responsibility: a parent view that handles the aggregation, and a child view responsible for rendering each node of content.

With each of the `Task` models represented by an individual `TaskView`, changes to an individual model are broadcast to its corresponding `TaskView`, which re-renders only the markup for one task.

Continuing our example from above, a `TaskView` will be responsible for rendering just the individual table row for a `Task`:

```
<tr>
  <td><%= model.escape('title') %></td>
  <td><%= model.escape('completed') %></td>
</tr>
```

And the Task index template will be changed to appear as shown below:

```
<h1>Tasks</h1>

<table>
```

```
  <tr>
    <th>Title</th>
    <th>Completed</th>
  </tr>

  <!-- child content will be rendered here -->

</table>
```

As you can see above in the index template, the individual tasks are no longer iterated over and rendered inside the table, but instead within the `TasksIndex` and `TaskView` views, respectively:

```
ExampleApp.Views.TaskView = Backbone.View.extend({
  render: function () {
    this.$el.html(JST['tasks/view']({ model: this.model }));
    return this;
  }
});
```

The `TaskView` view above is very similar to the one we saw previously for the `TasksIndex` view. It is only responsible for rendering the contents of its own element, and the concern of assembling the view of the list is left to the parent view object:

```
ExampleApp.Views.TasksIndex = Backbone.View.extend({
  render: function () {
    var self = this;

    this.$el.html(JST['tasks/index']()); // Note that no collection is needed
                                         // to build the container markup.

    this.collection.each(function(task) {
      var taskView = new ExampleApp.Views.TaskView({ model: task });
      self.$('table').append(taskView.render().el);
    });

    return this;
  }
});
```

In the new `TasksIndex` view above, the `tasks` collection is iterated over. For each task, a new `TaskView` is instantiated, rendered, and then inserted into the <table> element.

If you look at the output of the `TasksIndex`, it will appear as follows:

```
<div>
  <h1>Tasks</h1>

  <table>
    <tr>
      <th>Title</th>
      <th>Completed</th>
    </tr>

    <div>
      <tr>
        <td>Task 1</td>
        <td>true</td>
      </tr>
    </div>
    <div>
      <tr>
        <td>Task 2</td>
        <td>false</td>
      </tr>
    </div>
  </table>
</div>
```

Unfortunately, we can see that there is a problem with the above rendered view: the surrounding div around each of the rendered tasks.

Each of the rendered tasks has a surrounding div because this is the element that each view has that is accessed via `this.el`, and what the view's content is inserted into. By default, this element is a div and therefore every view will be wrapped in an extra div. While sometimes this extra div doesn't really matter, as in the outermost div that wraps the entire index, other times this produces invalid markup.

Fortunately, Backbone provides us with a clean and simple mechanism for changing the element to something other than a div. In the case of the `TaskView`, we would like this element to be a tr, then the wrapping tr can be removed from the task view template.

The element to use is specified by the `tagName` member of the `TaskView`, as shown below:

```
ExampleApp.Views.TaskView = Backbone.View.extend({
  tagName: "tr",

  initialize: function() {
  },
```

```
  render: function () {
    this.$el.html(JST['tasks/view']({ model: this.model }));
    return this;
  }
};
```

Given the above `tagName` customization, the task view template will appear as follows:

```
<td><%= model.escape('title') %></td>
<td><%= model.escape('completed') %></td>
```

And the resulting output of the `TasksIndex` will be much cleaner, as shown below:

```
<div>
  <h1>Tasks</h1>

  <table>
    <tr>
      <th>Title</th>
      <th>Completed</th>
    </tr>

    <tr>
      <td>Task 1</td>
      <td>true</td>
    </tr>
    <tr>
      <td>Task 2</td>
      <td>false</td>
    </tr>
  </table>
</div>
```

We've now covered the basic building blocks of converting Rails views to Backbone and getting a functional system. The majority of Backbone programming you will do will likely be in the views and templates, and there is a lot more to them: event binding, different templating strategies, helpers, event unbinding, and more. Those topics are covered in the "Routers, Views, and Templates" chapter.

Chapter 5

Routers, Views, and Templates

View explanation

A Backbone view is a class that is responsible for rendering the display of a logical element on the page. A view can also bind to events which may cause it to be re-rendered.

Again, it's important to note that a Rails view is not directly analogous to a Backbone view. A Rails view is more like a Backbone *template*, and Backbone views are often more like Rails *controllers*, in that they are responsible for deciding what should be rendered and how, and for rendering the actual template file. This can cause confusion with developers just starting with Backbone.

A basic Backbone view appears as follows.

```
ExampleApp.Views.ExampleView = Backbone.View.extend({
  tagName: "li",
  className: "example",
  id: "example_view",

  events: {
    "click a.save": "save"
  },

  render: function() {
    this.$el.html(JST['example/view']({ model: this.model }));
    return this;
  },
```

```
  save: function() {
    // do something
  }
};
```

Initialization

Backbone views could also include an `initialize` function which will be called when the view is instantiated.

You only need to specify the initialize function if you wish to do something custom. For example, some views call the `render()` function upon instantiation. It's not necessary to immediately render that way, but it's relatively common to do so.

You create a new view by instantiating it with `new`. For example, `new ExampleView()`. It is possible to pass in a hash of options with `new ExampleView(options)`. Any options you pass into the constructor will be available inside of the view in `this.options`.

There are a few special options that, when passed, will be assigned as properties of view. These are `model`, `collection`, `el`, `id`, `className`, and `tagName`. For example, if you create a new view and give it a model option using `new ExampleView({ model: someTask })`, then inside of the view `someTask` will be available as `this.model`.

The View's element

Each Backbone view has an element which it stores in `this.el`. This element can be populated with content, but isn't on the page until placed there by you. Using this strategy it is then possible to render views outside of the current DOM at any time, and then later, in your code, insert the new elements all at once. In this way, high performance rendering of views can be achieved with as few reflows and repaints as possible.

A jQuery or Zepto object of the view's element is available in `this.$el`. This is useful, in that you don't need to repeatedly call `$(this.el)`. This jQuery or Zepto call is also cached, so it should be a performance improvement over repeatedly calling `$(this.el)`.

It is possible to create a view that references an element already in the DOM, instead of a new element. To do this, pass in the existing element as an option to the view constructor with `new ExampleView({ el: existingElement })`.

You can also set this after the fact with the `setElement()` function:

```
var view = new ExampleView();
view.setElement(existingElement);
```

Customizing the View's Element

You can use `tagName`, `className`, and `id` to customize the new element created for the view. If no customization is done, the element is an empty `div`.

`tagName`, `className`, and `id` can either be specified directly on the view or passed in as options at instantiation. Since `id` will usually to correspond to the `id` of each model, it will likely be passed in as an option rather than declared statically in the view.

`tagName` will change the element that is created from a `div` to something else that you specify. For example, setting `tagName: "li"` will result in the view's element being an `li` rather than a `div`.

`className` will add an additional class to the element that is created for the view. For example, setting `className: "example"` in the view will result in that view's element having that additional class like `<div class="example">`.

Rendering

The `render` function above renders the `example/view` template. Template rendering is covered in depth in the "Templating strategy" chapter. Suffice to say, nearly every view's render function will render some form of template. Once that template is rendered, other actions to modify the view may be taken.

In addition to rendering a template, typical responsibilities of the `render` function could include adding more classes or attributes to `this.el`, or firing or binding other events.

Backbone, when used with jQuery (or Zepto) provides a convenience function of `this.$` that can be used for selecting elements inside of the view. `this.$(selector)` is equivalent to the jQuery function call `$(selector, this.el)`

A nice convention of the render function is to return `this` at the end of render to enable chained calls on the view - usually fetching the element. For example:

```
render: function() {
  this.$el.html(this.childView.render().el);
  return this;
}
```

Events

The view's `events` hash specifies a mapping of the events and elements that should have events bound, and the functions that should be bound to those events. In the example above, the `click` event is being bound to the element(s) that match the selector `a.save` within the view's element. When that event fires, the `save` function will be called on the view.

When event bindings are declared with the `events` hash, the DOM events are bound with the `$.delegate()` function. Backbone also takes care of binding the event handlers' `this` to the view instance using `_.on()`.

Event binding is covered in great detail in the "Event binding" chapter.

Templating strategy

There's no shortage of templating options for JavaScript. They generally fall into three categories:

- *HTML with JavaScript expressions interpolated.* Examples: `_.template`, EJS
- *HTML with other expressions interpolated, often logic-free.* Examples: Mustache, Handlebars, `jQuery.tmpl`
- *Selector-based content declarations.* Examples: PURE, just using jQuery from view classes

To quickly compare the different approaches, we will work with creating a template that renders the following HTML:

```
<ul class="tasks">
  <li><span class="title">Buy milk</span> Get the good kind </li>
  <li><span class="title">Buy cheese</span> Sharp cheddar </li>
  <li><span class="title">Eat cheeseburger</span> Make with above cheese </li>
</ul>
```

Assuming we have a TasksCollection instance containing the three elements displayed in the above HTML snippet, let's look at how different templating libraries accomplish the same goal of rendering the above. Since you're already familiar with Underscore.js templates, let's start there.

An Underscore.js template may look like this:

```
<ul class="tasks">
  <% tasks.each(function(task) { %>
    <li>
        <span class="title"> <%= task.escape("title") %> </span>
        <%= task.escape("body") %>
    </li>
  <% }) %>
</ul>
```

Here, we interpolate a bit of JavaScript logic in order to iterate through the collection and render the desired markup. Also note that we must fetch escaped values from the task objects, as Underscore.js templates do not perform any escaping on their own.

This is probably the path of least resistance on a Rails Backbone app. Since Backbone depends on Underscore.js, it is already available in your app. As has already been shown in earlier chapters, its usage is very similar to ERB. It has the same <%= and %> syntax as ERB, and you can pass it an options object that is made available to the template when it's rendered.

While we've found Underscore.js' templating to be useful and sufficient to build large backbone applications, there are other templating libraries that are worth mentioning here because they either provide richer functionality or take a different approach to templating.

Handlebars is one such example. One major distinction of Underscore.js is that it allows you to define and register helpers that can be used when rendering a template, providing a framework for writing helpers similar to those found in ActionView::Helpers, like `domID` or other generic rendering logic. It also allows you to write what are called "block helpers," which are functions that are executed on a different, supplied context during rendering. Handlebars itself exploits this functionality by providing a few helpers out of the box. These helpers are `with`, `each`, `if` and `unless`, and simply provide control structures for rendering logic.

The above template would look like this in Handlebars:

```
<ul class="title">
  {{#each tasks}}
    <li>
        <span class="title"> {{ this.get("title") }} </span>
        {{ this.get("body") }} %>
    </li>
  {{/each}}
<ul>
```

Of note:

- Use of {{`#each`}}, which iterates over the collection
- Within the {{`#each`}} block, the JavaScriptJavaScript context is the task itself, so you access its properties via `this`
- There's no need to escape HTML output, as Handlebars escapes by default

A similar library is Mustache.js. Mustache is a templating system that has been ported to a number of languages including JavaScript. The promise of Mustache is "logic-less templates." Instead of requiring you to write logic in pure JavaScript, using `if`, for example, Mustache provides a set of tags that take on different meanings. They can render values or not render anything at all.

Like Handlebars, Mustache HTML escapes rendered values by default.

You can learn more about Handlebars at the project's home on the web, and Mustache at the project's man page and [javascript implementation(https://github.com/janl/mustache.js)

Choosing a strategy

Like any technology choice, there are trade-offs to evaluate and external factors to consider when choosing a templating approach.

One of the common questions we've found ourselves asking is: Do I already have server-side templates written that I'd like to "Backbone-ify," or am I writing new Backbone functionality from scratch? Both of these scenarios are described in more detail in the next two sections.

Adding Backbone to existing Rails views

If you are replacing existing Rails app pages with Backbone, you are already using a templating engine, and it's likely ERB. When making the switch to Backbone, change as few things as possible at a time, and stick with your existing templating approach.

If you're using ERB, give `_.template` a shot. It defaults to the same delimiters as ERB for interpolation and evaluation, <%= %> and <% %>, which can be a boon or can be confusing. If you'd like to change them, you can update `.templateSettings` - check the Underscore.js docs.

If you're using Haml, check out the `jquery-haml` and `haml-js` projects.

If you're using Mustache.rb or Handlebars.rb, you're likely aware that JavaScript implementations of these both exist, and that your existing templates can be moved over much like the ERB case.

Ultimately, you should choose a templating strategy that your entire team is comfortable with, while minimizing the cost of rewriting templates. Make sure that designers' considerations are taken into account, because it will affect how they work with that area of the app as well.

Writing new Backbone functionality from scratch

If you're not migrating from existing server-side view templates, you have more freedom of choice. Strongly consider the option of no templating at all, but rather using plain HTML templates, and then decorating the DOM from your view class.

You can build static HTML mockups of the application first, and pull these mockups directly in as templates, without modifying them.

```
<!-- snip -->
<div id="song-player">
  <nav>
    <a class="home"    href="#/">Home</a>
    <a class="profile" href="/profile.html">My Profile</a>
  </nav>
  <h2>Song title</h2>

  <audio controls="controls">
    <source src="/test.ogg" type="audio/ogg" />
    Your browser does not support the audio element.
  </audio>
</div>
<!-- snip -->
```

```
MyView = Backbone.View.extend({
  render: function() {
    this.renderTemplate();
    this.fillTemplate();
  },

  renderTemplate: function() {
    this.$el.html(JST['songs/index']();
  },

  fillTemplate: function() {
    this.$('nav a.profile').text(App.currentUser().fullName());
    this.$('h2').html(this.model.escape('title'));

    var audio = this.$('audio');
```

```
    audio.empty();
    this.model.formats.each(function(format) {
      $("<source></source>")
        .attr("src",  format.get('src'))
        .attr("type", format.get('type'))
        .appendTo(audio);
    });
  }
});
```

You can see an example of this in the example application's `TaskItem` view class, at `app/assets/javascripts/views/task_item.js`.

The only disadvantage of this is that your view's `render()` functions become more coupled to the structure of the HTML. This means that a major change in the markup may break the rendering because the selectors used to replace parts of the DOM may no longer find the same elements, or may not find any elements at all.

Routers

Routers are an important part of the Backbone infrastructure. Backbone routers provide methods for routing application flow based on client-side URL fragments (`yourapp.com/tasks#fragment`).

Routes are meant to represent serializable, bookmarkable entry points into your Backbone application. This means that the pertinent application state is serialized into the route fragment and that, given a route, you can completely restore that application state. They serve as navigational checkpoints, and determine the granularity at which users navigate "Back" in your application.

As an aside, it's worth pointing out that there may well be many states in your application that you don't want represented by a route - modes in your application that users don't really care about returning exactly to, or where the cost of building the code that reconstructs the state is too expensive to justify it. For example, you may have a tabbed navigation, expandable information panes, modal dialog boxes, or resizable display ports that all go untracked by routes.

Anecdotally, one recent client application we developed has around 100 Backbone view classes, but fewer than twenty routes. Additionally, many of the view classes are displayed in parallel and have multiple internal states of their own, providing for much more than 100 different interface states.

A note about `pushState`

Backbone now includes support for pushState, which can use real, full URLs instead of URL fragments for routing.

However, pushState support in Backbone is fully opt-in due to lack of browser support and that additional server-side work is required to support it.

pushState support is currently limited to the latest versions of Firefox, Chrome, Safari, and Mobile Safari. For a full listing of support and more information about the History API, of which pushState is a part, visit http://diveintohtml5.info/history.html#how.

Thankfully, if you opt-in to pushState in Backbone, browsers that don't support pushState will continue to use hash-based URL fragments, and if a hash URL is visited by a pushState-capable browser, it will be transparently upgraded to the true URL.

In addition to browser support, another hurdle to seamless use of pushState is that because the URLs used are real URLs, your server must know how to render each of the URLs. For example, if your Backbone application has a route of +/tasks/1+, your server-side application must be able to respond to that page if the browser visits that URL directly.

For most applications, you can handle this by just rendering the content you would have for the root URL and letting Backbone handle the rest of the routing to the proper location. But for full search-engine crawlability, your server-side application will need to render the entire HTML of the requested page.

For the reasons and complications above, the examples in this book all currently use URL fragments and not pushState.

Example router

A typical Backbone router will appear as shown below:

```
ExampleApp.Routers.ExampleRouter = Backbone.Router.extend({
  routes: {
    ""         : "index"
    "show/:id" : "show"
  },

  index: function() {
    // Instantiate and render the index view
  }

  show: function(id) {
    // Instantiate and render the show view
```

```
  }
});
```

The routes hash

The basic router consists of a routes hash, which is a mapping between URL fragments and methods on the router. If the current URL fragment, or one that is being visited, matches one of the routes in the hash, its method will be called.

Like Rails routes, Backbone routes can contain parameter parts, as seen in the `show` route in the example above. In this route, the part of the fragment after `show/` will then be based as an argument to the `show` method.

Multiple parameters are possible, as well. For example, a route of `search/:query/p:page` will match a fragment of `search/completed/p2` passing `completed` and `2` to the action.

In the routes, `/` is the natural separator. For example, a route of `show/:id` will not match a fragment of `show/1/2`. To allow you to match fragments like this, Backbone provides the concept of splat parts, identified by `*` instead of `:`. For example, a route of `show/*id` would match the previous fragment, and `1/2` would be passed to the action as the `id` variable.

Routing occurs when the browser's URL changes. This can occur when a link is clicked, when a URL is entered into the browser's URL bar, or when the back button is clicked. In all of those cases, Backbone will look to see if the new URL fragment matches an existing route. If it does, the specified function will be called with any parameters extracted from the URL fragment.

In addition, an event with the name of "route" and the function will be triggered. For example, when the router's `show` function above is triggered, an event of `route:show` will be fired. This is so that other objects can listen to the router, and be notified when the router responds to certain routes.

Initializing a router

It is possible to specify an `initialize` function in a Router which will be called when the router is instantiated. Any arguments passed to the router's constructor will be passed to this `initialize` function.

Additionally, it is possible to pass the routes for a router via the constructor such as `new ExampleRouter({ routes: { "" : "index" }}`. But note that this will override any routes defined in the routes hash on the router itself.

Event binding

A big part of writing snappy rich client applications is building models and views that update in real-time with respect to one another. With Backbone, you accomplish this with events.

Client-side applications are asynchronous by nature. Events binding and triggering are at the heart of a Backbone application. Your application is written using event-driven programming where components emit and handle events, achieving non-blocking UIs.

With Backbone, it's very easy to write such applications. Backbone provides the `Backbone.Events` mixin, which can be included in any other class.

Here's a quick example of a very simple game engine, where things happen in the system and an event is triggered, which in turn invokes any event handlers that are bound to that event:

```
var gameEngine = {};
_.extend(gameEngine, Backbone.Events);

gameEngine.on("user_registered", function(user) {
  user.points `= 10
});

gameEngine.trigger("user_registered", User.new({ points: 0 }));
```

In the example above, `on` subscribes the gameEngine to listen for the "user_registered" event, then `trigger` broadcasts that event to all subscribed listeners, which invokes the function that adds points to the user. Any arguments passed to `trigger` after the name of the event are in turn passed to the event handler. So in this case the output of `User.new()` is received as `user` in the handler.

`Backbone.Views`, `Backbone.Model` and `Backbone.Collection` are all extended with `Backbone.Events`. There are some events that are triggered by Backbone at particularly convenient moments. These are common events to which many user interface flows need to react. For example, when a Backbone model's attributes are changed, that model will trigger the `change` event. It is still up to you to bind a handler on those events. (More on that later.)

As you can see from the example though, it is possible to bind and trigger arbitrary events on any object that extends `Backbone.Events`. Additionally, if an event handler should always trigger regardless of which event is fired, you can bind to the special `all` event.

There are three primary kinds of events that your views will bind to:

- DOM events within the view's `this.el` element
- Events triggered by closely associated objects, such as the view's model or collection
- Events your view itself publishes

Event bindings declared on your view will need to be cleaned when your view is disposed of. Events that your view publishes will need to be handled a different way. Each of these three categories of events is discussed in more detail below.

Binding to DOM events within the view element

The primary function of a view class is to provide behavior for its markup's DOM elements. You can attach event listeners by hand if you like:

```
<!-- templates/soundboard.jst -->
<a class="sound">Honk</a>
<a class="sound">Beep</a>
```

```
var SoundBoard = Backbone.View.extend({
  render: function() {
    $(this.el).html(JST['soundboard']());
    this.$("a.sound").bind("click", this.playSound);
  },

  playSound: function() {
    // play sound for this element
  }
});
```

But Backbone provides an easier and more declarative approach with the `events` hash:

```
var SoundBoard = Backbone.View.extend({
  events: {
    "click a.sound": "playSound"
  },

  render: function() {
    this.$el.html(JST['soundboard']());
  },

  playSound: function() {
```

```
    // play sound for this element
  }
});
```

Backbone will bind the events with the `Backbone.View.prototype.delegateEvents()` function. It binds DOM events with `$.delegate()`, whether you're using the jQuery or Zepto `.delegate()` function.

It also takes care of binding the event handlers' `this` to the view instance using `_.on()`.

Events observed by your view

In almost every view you write, the view will be bound to a `Backbone.Model` or a `Backbone.Collection`, most often with the convenience properties `this.model` or `this.collection`.

Consider a view that displays a collection of `Task` models. It will re-render itself when any model in the collection is changed or removed, or when a new model is added:

```
var TasksIndexView = Backbone.View.extend({
  template: JST['tasks/tasks_index'],
  tagName: 'section',
  id: 'tasks',

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

    this.collection.bind("change", this.render);
    this.collection.bind("add",    this.render);
    this.collection.bind("remove", this.render);
  },

  render: function() {
    this.$el.html(this.template({tasks: this.collection}));
  }
});
```

Note how we bind to the collection's `change`, `add` and `remove` events. The `add` and `remove` events are triggered when you either `add()` or `remove()` a model from that collection as expected. The `change` event requires special mention; it will trigger when any of the underlying models' `change` event triggers. Backbone just bubbles up that event to the containing collection for convenience.

While the most common view bindings will be to events from its associated models and collections, your view can bind to any events to which it wants to listen. The life-cycle for the binding and unbinding, and the handling of these events will be the same as those for models and collections.

Events your view publishes

With sufficiently complex views, you may encounter a situation where you want one view to change in response to another. This can be accomplished with events. Your view can trigger an event to which the other view has bindings.

Consider a simple example with a table of users and a toggle control that filters the users to a particular gender:

```
GenderPicker = Backbone.View.extend({
  render: {
    // render template
  },
  events: {
    "click .show-male":   "showMale",
    "click .show-female": "showFemale",
    "click .show-both":   "showBoth"
  },

  showMale: function()   { this.trigger("changed", "male");   },
  showFemale: function() { this.trigger("changed", "female"); },
  showBoth: function()   { this.trigger("changed", "both");   }
});

UsersTable = Backbone.View.extend({
  initialize: function() {
    this.genderPicker = new GenderPicker();
    this.genderPicker.on("changed", this.filterByGender);
    this.collectionToRender = this.collection;
    this.render();
  },

  render: {
    this.genderPicker.render();
    this.$el.html(JST['users']({ users: this.collectionToRender }));
  }

  filterByGender: function(gender) {
    this.collectionToRender = this.collection.byGender(gender);
    this.render();
```

```
  }
});
```

In the above snippet, the `GenderPicker` is responsible for the filter control. When the appropriate elements are clicked, a custom `changed` event is triggered on itself. Note how it is also possible to pass arbitrary parameters to the `trigger()` function.

On the other hand, we have a `UsersTable` which is responsible for rendering a collection of users. It also observes this event via the call to `on()`, where it invokes the `filterByGender` function.

While your views will generally bind to events on models and collections, a situation like the above may arise where it is handy to trigger and bind to custom events at the view layer. However, it's always a good idea to consider whether you should, instead, be binding to events on the underlying components.

Cleaning up: unbinding

In the last section, we discussed three different kinds of event binding in your `Backbone.Views` classes: DOM events, model/collection events, and custom view events. Next, we'll discuss unbinding these events: why it's a good idea, and how to do it.

Why unbind events?

Consider two views in a todo app: an index view, which contains all the tasks that need to be done:

. . . and a detail view that shows detail on one task:

The interface switches between the two views.

Here's the source for the aggregate index view:

```
var TasksIndexView = Backbone.View.extend({
  template: JST['tasks/tasks_index'],
  tagName: 'section',
  id: 'tasks',

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

    this.collection.bind("change", this.render);
    this.collection.bind("add",    this.render);
```

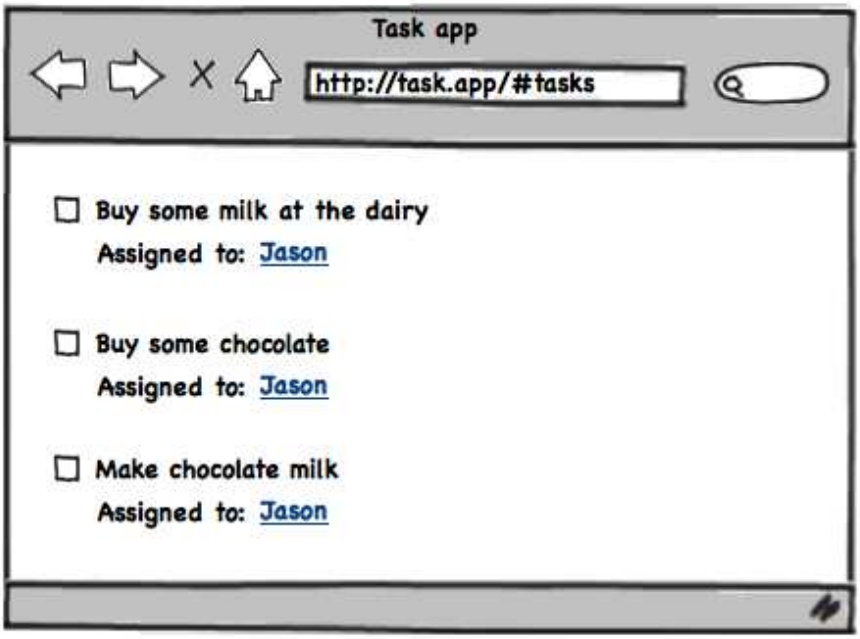

Figure 5.1: Tasks index view

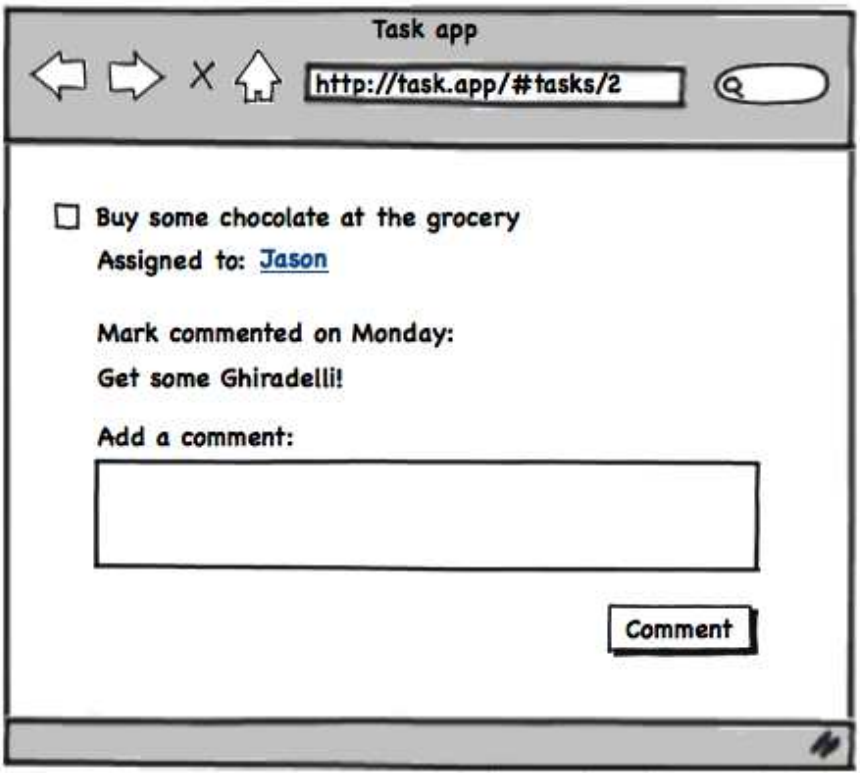

Figure 5.2: Tasks detail view

```
    this.collection.bind("remove", this.render);
  },

  render: function() {
    this.$el.html(this.template({tasks: this.collection}));
  }
});
```

. . . and the source for the individual task detail view:

```
var TaskDetail = Backbone.View.extend({
  template: JST['tasks/task_detail'],
  tagName: 'section',
  id: 'task',

  events: {
    "click .comments .form-inputs button": "createComment"
  },

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

    this.model.on("change", this.render);
    this.model.comments.on("change", this.render);
    this.model.comments.on("add",    this.render);
  },

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

  createComment: function() {
    var comment = new Comment({ text: this.$('.new-comment-input').val() });
    this.$('.new-comment-input').val('');
    this.model.comments.create(comment);
  }
});
```

Each task on the index page links to the detail view for itself. When a user follows one of these links and navigates from the index page to the detail page, then interacts with the detail view to change a model, the `change` event on the `TaskApp.tasks` collection is fired. One consequence of this is that the index view, which is still bound and observing the `change` event, will re-render itself.

This is both a functional bug and a memory leak: not only will the index view re-render and disrupt the detail display momentarily, but navigating back and

forth between the views without disposing of the previous view will keep creating more views and binding more events on the associated models or collections.

These can be extremely tricky to track down on a production application, especially if you are nesting child views. Sadly, there's no "garbage collection" for views in Backbone, so your application needs to manage this itself. Luckily, it's not too hard to keep track of and correctly maintain your bindings.

Let's take a look at how to unbind three kinds of events: DOM events, model and collection events, and events you trigger in your views.

Unbinding DOM events

DOM events are the simplest case - they more or less get cleaned up for you. When you call `this.remove()` in your view, it delegates to `jQuery.remove()` by invoking `$(this.el).remove()`. This means that jQuery takes care of cleaning up any events bound on DOM elements within your view, regardless of whether you bound them with the Backbone `events` hash or by hand; for example, with `$.bind()`, `$.delegate()`, `live()` or `$.on()`.

Unbinding model and collection events

If your view binds to events on a model or collection with `on()`, you are responsible for unbinding these events. You do this with a simple call to `this.model.off()` or `this.collection.off()`; the http://documentcloud.github.com/backbone/#Events-off[`Backbone.Events.off()` function] removes all callbacks on that object.

When should you unbind these handlers? Whenever the view is going away. This means that any pieces of code that create new instances of this view become responsible for cleaning up after it's gone. That isn't a very cohesive approach, so it's best to include the cleanup responsibility on the view itself.

To do this, you'll write a `leave()` function on your view that wraps `remove()` and handles any additional event unbinding that's needed. As a convention, when you use this view elsewhere, you'll call `leave()` instead of `remove()` when you're done:

```
var SomeCollectionView = Backbone.View.extend({
  // snip...

  initialize: function() {
    this.collection.bind("change", this.render);
  },

  leave: function() {
    this.collection.unbind("change", this.render);
```

```
    this.remove();
  }

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

Keep track of `on()` calls to unbind more easily

In the example above, unbinding the collection change event isn't too much hassle; since we're only observing one thing, we only have to unbind one thing. But even the addition of one line to `leave()` is easy to forget, and if you bind to multiple events, it only gets more verbose.

Let's add a step of indirection in event binding, so that we can automatically clean up all the events with one call. We'll add and use a `bindTo()` function that keeps track of all the event handlers we bind, and then issue a single call to `unbindFromAll()` to unbind them:

```
var SomeCollectionView = Backbone.View.extend({
  initialize: function() {
    this.bindings = [];
    this.bindTo(this.collection, "change", this.render);
  },

  leave: function() {
    this.unbindFromAll();
    this.remove();
  },

  bindTo: function(source, event, callback) {
    source.on(event, callback, this);
    this.bindings.push({ source: source, event: event, callback: callback });
  },

  unbindFromAll: function() {
    _.each(this.bindings, function(binding) {
      binding.source.off(binding.event, binding.callback);
    });
    this.bindings = [];
  }
});
```

These functions, `bindTo()` and `unbindFromAll()`, can be extracted into a reusable mixin or superclass. Then, we just have to use `bindTo()` instead of `model.on()` and be assured that the handlers will be cleaned up during `leave()`.

Unbinding view-triggered events

With the first two kinds of event binding that we discussed, DOM and model/collection, the view is the observer. The responsibility to clean up is on the observer, and here the responsibility consists of unbinding the event handler when the view is being removed.

But other times, our view classes will trigger (emit) events of their own. In this case, other objects are the observers, and are responsible for cleaning up the event binding when they are disposed. See "Events your view publishes" in the earlier "Event binding" section for more details.

Finally, when the view itself is disposed of with `leave()`, it should clean up any event handlers bound on *itself* for events that it triggers.

This is handled by invoking `Backbone.Events.off()`:

```
var FilteringView = Backbone.View.extend({
  // snip...

  events: {
    "click a.filter": "changeFilter"
  },

  changeFilter: function() {
    if (someLogic()) {
      this.trigger("filtered", { some: options });
    }
  },

  leave: function() {
    this.off(); // Clean up any event handlers bound on this view
    this.remove();
  }

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

Establish a convention for consistent and correct unbinding

There's no built-in garbage collection for Backbone's event bindings, and forgetting to unbind can cause bugs and memory leaks. The solution is to make sure you unbind events and remove views when you leave them. Our approach to this is two-fold: write a set of reusable functions that manage cleaning up a view's bindings, and use these functions wherever views are instantiated -

in `Router` instances, and in composite views. We'll take a look at these concrete, reusable approaches in the next two sections about `SwappingRouter` and `CompositeView`.

Swapping router

When switching from one view to another, we should clean up the previous view. Earlier, we discussed a convention of writing a `view.leave()`. Let's augment our view to include the ability to clean itself up by "leaving" the DOM:

```
var MyView = Backbone.View.extend({
  // ...

  leave: function() {
    this.off();
    this.remove();
  },

  // ...
});
```

The `off()` and `remove()` functions are provided by `Backbone.Events` and `Backbone.View` respectively. `Backbone.Events.off()` will remove all callbacks registered on the view, and `remove()` will remove the view's element from the DOM, equivalent to calling `this.$el.remove()`.

In simple cases, we replace one full page view with another full page (less any shared layout). We introduce a convention that all actions underneath one `Router` share the same root element, and define it as `el` on the router.

Now, a `SwappingRouter` can take advantage of the `leave()` function, and clean up any existing views before swapping to a new one. It swaps into a new view by rendering that view into its own `el`:

```
Support.SwappingRouter = function(options) {
  Backbone.Router.apply(this, [options]);
};

_.extend(Support.SwappingRouter.prototype, Backbone.Router.prototype, {
  swap: function(newView) {
    if (this.currentView && this.currentView.leave) {
      this.currentView.leave();
    }
```

```
    this.currentView = newView;
    this.currentView.render();
    $(this.el).empty().append(this.currentView.el);
  }
});

Support.SwappingRouter.extend = Backbone.Router.extend;
```

Now all you need to do in a route function is call `swap()`, passing in the new view that should be rendered. The `swap()` function's job is to call `leave()` on the current view, render the new view appending it to the router's `el`, and, finally, store what view is the current view, so that the next time `swap()` is invoked, it can be properly cleaned up as well.

SwappingRouter and Backbone internals

If the code for `SwappingRouter` seems a little confusing, don't fret: it is, thanks to JavaScript's object model! Sadly, it's not as simple to just drop the `swap` method into `Backbone.Router`, or call `Backbone.Router.extend` to mixin the function we need.

Our goal here is essentially to create a subclass of `Backbone.Router`, and to extend it without modifying the original class. This gives us a few benefits: first, `SwappingRouter` should work with Backbone upgrades. Second, it should be *obvious* and *intention-revealing* when a controller needs to swap views. If we simply mixed in a `swap` method and called it from a direct descendant of `Backbone.Router`, an unaware (and unlucky) programmer would need to go on a deep source dive in an attempt to figure out where that was coming from. With a subclass, the hunt can start at the file where it was defined.

The procedure used to create `SwappingRouter` is onerous, thanks to a mix of Backbone-isms and just how clunky inheritance is in JavaScript. Firstly, we need to define the constructor, which delegates to the `Backbone.Router` constructor with the use of `Function#apply`. The next block of code uses Underscore.js' `Object#extend` to create the set of functions and properties that will become `SwappingRouter`. The `extend` function takes a destination - in this case, the empty prototype for `SwappingRouter` - and copies the properties into the `Backbone.Router` prototype along with our new custom object that includes the `swap` function.

Finally, the subclass cake is topped off with some Backbone frosting, by setting `extend`, which is a self-propagating function that all Backbone public classes use. Let's take a quick look at this function, as seen in Backbone 0.5.3:

```
var extend = function (protoProps, classProps) {
  var child = inherits(this, protoProps, classProps);
```

```
  child.extend = this.extend;
  return child;
};

// Helper function to correctly set up the prototype chain, for subclasses.
// Similar to 'goog.inherits', but uses a hash of prototype properties and
// class properties to be extended.
var inherits = function(parent, protoProps, staticProps) {
  // sparing our readers the internals of this function... for a deep dive
  // into the dark realms of JavaScript's prototype system, read the source!
}
```

This is a function that calls `inherits` to make a new subclass. The comments reference `goog.inherits` from Google's Closure Library, which contains similar utility functions to allow more class-style inheritance.

The end result here is that whenever you make a custom controller internally in Backbone, you're making *another* subclass. The inheritance chain for `TasksRouter` would then look like:

Phew! Hopefully this adventure into Backbone and JavaScript internals has taught you that although it entails learning and employing more code, it can (and should) save time down the road for those maintaining your code.

You can find an example of a `SwappingRouter` on the example app under `app/assets/javascripts/routers/tasks.js`. Note how each action in that router uses `SwappingRouter.swap()` to invoke rendering of views, freeing itself from the complexities of cleaning them up.

Composite views

The `SwappingRouter` above calls `leave()` on the view it currently holds. This function is not part of Backbone itself, and is part of our extension library to help make views more modular and maintainable. This section goes over the Composite View pattern, the `CompositeView` class itself, and some concerns to keep in mind while creating your views.

Refactoring from a large view

One of the first refactorings you'll find yourself doing in a non-trivial Backbone app is splitting up large views into composable parts. Let's take another look at the `TaskDetail` source code from the beginning of this section:

```
var TaskDetail = Backbone.View.extend({
```

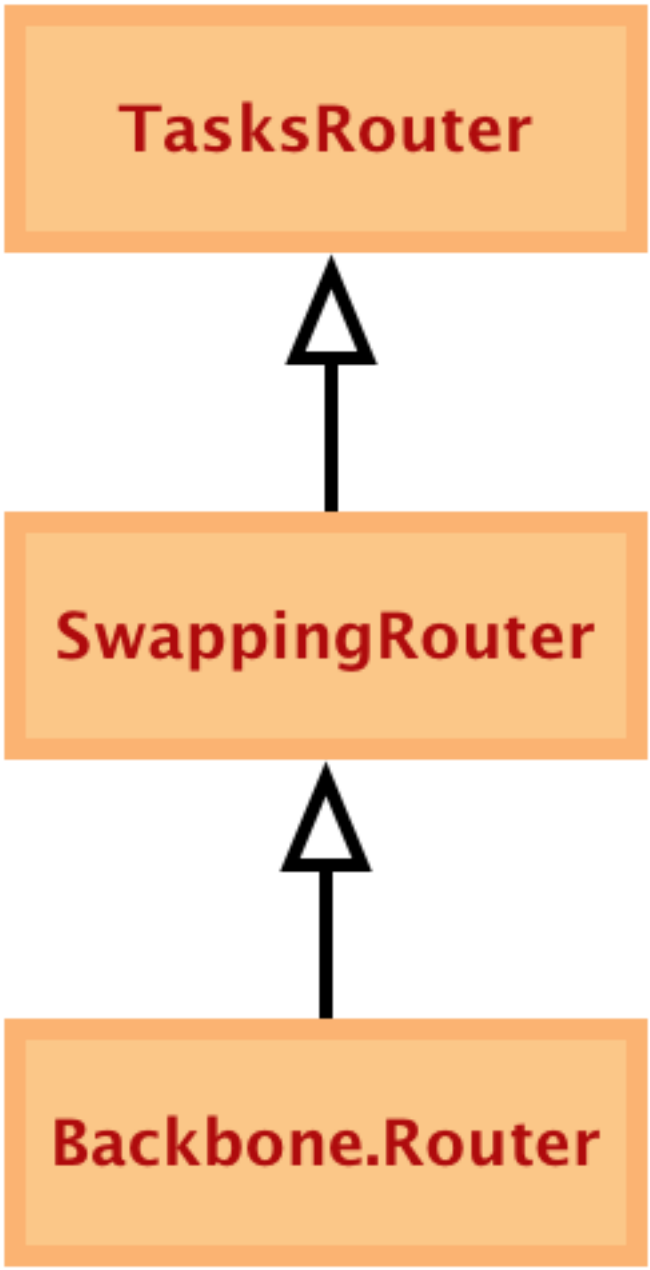

Figure 5.3: Router class inheritance

```
  template: JST['tasks/task_detail'],
  tagName: 'section',
  id: 'task',

  events: {
    "click .comments .form-inputs button": "createComment"
  },

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

    this.model.on("change", this.render);
    this.model.comments.on("change", this.render);
    this.model.comments.on("add",    this.render);
  },

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

  createComment: function() {
    var comment = new Comment({ text: this.$('.new-comment-input').val() });
    this.$('.new-comment-input').val('');
    this.model.comments.create(comment);
  }
});
```

The view class references a template, which renders out the HTML for this page:

```
<section class="task-details">
  <input type="checkbox"<%= task.isComplete() ? ' checked="checked"' : '' %> />
  <h2><%= task.escape("title") %></h2>
</section>

<section class="comments">
  <ul>
    <% task.comments.each(function(comment) { %>
      <li>
        <h4><%= comment.user.escape('name') %></h4>
        <p><%= comment.escape('text') %></p>
      </li>
    <% } %>
  </ul>

  <div class="form-inputs">
```

```
    <label for="new-comment-input">Add comment</label>
    <textarea id="new-comment-input" cols="30" rows="10"></textarea>
    <button>Add Comment</button>
  </div>
</section>
```

There are clearly several concerns going on here: rendering the task, rendering the comments that folks have left, and rendering the form to create new comments. Let's separate those concerns. A first approach might be to just break up the template files:

```
<!-- tasks/show.jst -->
<section class="task-details">
  <%= JST['tasks/details']({ task: task }) %>
</section>

<section class="comments">
  <%= JST['comments/list']({ task: task }) %>
</section>
```

```
<!-- tasks/details.jst -->
<input type="checkbox"<%= task.isComplete() ? ' checked="checked"' : '' %> />
<h2><%= task.escape("title") %></h2>
```

```
<!-- comments/list.jst -->
<ul>
  <% task.comments.each(function(comment) { %>
    <%= JST['comments/item']({ comment: comment }) %>
  <% } %>
</ul>

<%= JST['comments/new']() %>
```

```
<!-- comments/item.jst -->
<h4><%= comment.user.escape('name') %></h4>
<p><%= comment.escape('text') %></p>
```

```
<!-- comments/new.jst -->
<div class="form-inputs">
  <label for="new-comment-input">Add comment</label>
  <textarea id="new-comment-input" cols="30" rows="10"></textarea>
  <button>Add Comment</button>
</div>
```

But this is really only half the story. The `TaskDetail` view class still handles multiple concerns, such as displaying the task and creating comments. Let's split that view class up, using the `CompositeView` base class:

```
Support.CompositeView = function(options) {
  this.children = _([]);
  Backbone.View.apply(this, [options]);
};

_.extend(Support.CompositeView.prototype, Backbone.View.prototype, {
  leave: function() {
    this.unbind();
    this.remove();
    this._leaveChildren();
    this._removeFromParent();
  },

  renderChild: function(view) {
    view.render();
    this.children.push(view);
    view.parent = this;
  },

  appendChild: function(view) {
    this.renderChild(view);
    $(this.el).append(view.el);
  },

  renderChildInto: function(view, container) {
    this.renderChild(view);
    $(container).empty().append(view.el);
  },

  _leaveChildren: function() {
    this.children.chain().clone().each(function(view) {
      if (view.leave)
        view.leave();
    });
  },

  _removeFromParent: function() {
    if (this.parent)
      this.parent._removeChild(this);
  },

  _removeChild: function(view) {
```

```
    var index = this.children.indexOf(view);
    this.children.splice(index, 1);
  }
});

Support.CompositeView.extend = Backbone.View.extend;
```

Similar to the `SwappingRouter`, the `CompositeView` base class solves common housekeeping problems by establishing a convention. See the "SwappingRouter and Backbone internals" section for an in-depth analysis of how this subclassing pattern works.

Now our `CompositeView` maintains an array of its immediate children as `this.children`. With this reference in place, a parent view's `leave()` method can invoke `leave()` on its children, ensuring that an entire tree of composed views is cleaned up properly.

For child views that can dismiss themselves, such as dialog boxes, children maintain a back-reference at `this.parent`. This is used to reach up and call `this.parent.removeChild(this)` for these self-dismissing views.

Making use of `CompositeView`, we split up the `TaskDetail` view class:

```
var TaskDetail = CompositeView.extend({
  tagName: 'section',
  id: 'task',

  initialize: function() {
    _.bindAll(this, "renderDetails");
    this.model.on("change", this.renderDetails);
  },

  render: function() {
    this.renderLayout();
    this.renderDetails();
    this.renderCommentsList();
  },

  renderLayout: function() {
    this.$el.html(JST['tasks/show']());
  },

  renderDetails: function() {
    var detailsMarkup = JST['tasks/details']({ task: this.model });
    this.$('.task-details').html(detailsMarkup);
  },
```

```
  renderCommentsList: function() {
    var commentsList = new CommentsList({ model: this.model });
    var commentsContainer = this.$('comments');
    this.renderChildInto(commentsList, commentsContainer);
  }
});

var CommentsList = CompositeView.extend({
  tagName: 'ul',

  initialize: function() {
    this.model.comments.on("add", this.renderComments);
  },

  render: function() {
    this.renderLayout();
    this.renderComments();
    this.renderCommentForm();
  },

  renderLayout: function() {
    this.$el.html(JST['comments/list']());
  },

  renderComments: function() {
    var commentsContainer = this.$('comments-list');
    commentsContainer.html('');

    this.model.comments.each(function(comment) {
      var commentMarkup = JST['comments/item']({ comment: comment });
      commentsContainer.append(commentMarkup);
    });
  },

  renderCommentForm: function() {
    var commentForm = new CommentForm({ model: this.model });
    var commentFormContainer = this.$('.new-comment-form');
    this.renderChildInto(commentForm, commentFormContainer);
  }
});

var CommentForm = CompositeView.extend({
  events: {
    "click button": "createComment"
  },
```

```
  initialize: function() {
    this.model = this.options.model;
  },

  render: function() {
    this.$el.html(JST['comments/new']);
  },

  createComment: function() {
    var comment = new Comment({ text: $('.new-comment-input').val() });
    this.$('.new-comment-input').val('');
    this.model.comments.create(comment);
  }
});
```

Along with this, remove the <%= JST(...) %> template nestings, allowing the view classes to assemble the templates instead. In this case, each template contains placeholder elements that are used to wrap child views:

```
<!-- tasks/show.jst -->
<section class="task-details">
</section>

<section class="comments">
</section>

<!-- tasks/details.jst -->
<input type="checkbox"<%= task.isComplete() ? ' checked="checked"' : '' %> />
<h2><%= task.escape("title") %></h2>

<!-- comments/list.jst -->
<ul class="comments-list">
</ul>

<section class="new-comment-form">
</section>

<!-- comments/item.jst -->
<h4><%= comment.user.escape('name') %></h4>
<p><%= comment.escape('text') %></p>

<!-- comments/new.jst -->
<label for="new-comment-input">Add comment</label>
<textarea class="new-comment-input" cols="30" rows="10"></textarea>
<button>Add Comment</button>
```

There are several advantages to this approach:

- Each view class has a smaller and more cohesive set of responsibilities
- The comments view code, extracted and decoupled from the task view code, can now be reused on other domain objects with comments
- The task view performs better, since adding new comments or updating the task details will only re-render the pertinent section, instead of re-rendering the entire task ‘ comments composite

In the example app, we make use of a composite view on `TasksIndex` located at `app/assets/javascripts/views/tasks_index.js`. The situation is similar to what has been discussed here. The view responsible for rendering the list of children will actually render them as children. Note how the `renderTasks` function iterates over the collection of tasks, instantiates a `TaskItem` view for each, renders it as a child with `renderChild`, and finally appends it to table's body. Now, when the router cleans up the `TasksIndex` with `leave`, it will also clean up all of its children.

Cleaning up views properly

You've learned how leaving lingering events bound on views that are no longer on the page can cause both UI bugs or, what's probably worse, memory leaks. A slight flickering of the interface is annoying at best, but prolonged usage of your "untidy" app could, in fact, make the user's browser start consuming massive amounts of memory, potentially causing browser crashes, data loss and unhappy users and angry developers.

We now have a full set of tools to clean up views properly. To summarize, the big picture tools are:

- A *Swapping Router* that keeps track of the current view and tells it to clean up before it swaps in a new view
- A *Composite View* that keeps track of its child views so it can tell them to clean up when it is cleaning itself up

The `leave()` function ties this all together. A call to `leave()` can either come from a `SwappingRouter` or from a parent `CompositeView`. A `CompositeView` will respond to `leave()` by passing that call down to its children. At each level, in addition to propagating the call, `leave()` handles the task of completely cleaning up after a view by removing the corresponding element from the DOM via jQuery's `remove()` function, and removing all event bindings via a call to `Backbone.Events.off()`. In this way a single call at the top level cleans the slate for an entirely new view.

Forms

Who likes writing form code by hand? Nobody, that's who. Rails' form builder API greatly helps reduce application code, and we aim to maintain a similar level of abstraction in our Backbone application code. Let's take a look at what we need from form building code to achieve this.

We have a few requirements when it comes to handling forms. We need to:

- Build form markup and populate it with model values
- Serialize a form into a model for validation and persistence
- Display error messages

Additionally, it's nice to:

- Reduce boilerplate
- Render consistent and stylable markup
- Automatically build form structure from data structure

Let's look at the requirements one by one and compare approaches.

Building markup

Our first requirement is the ability to build markup. For example, consider a Rails model `User` that has a username and password. We might want to build form markup that looks like this:

```
<form>
  <li>
    <label for="email">Email</label>
    <input type="text" id="email" name="email">
  </li>
  <li>
    <label for="password">Password</label>
    <input type="password" id="password" name="password">
  </li>
</form>
```

One approach you could take is writing the full form markup by hand. You could create a template available to Backbone via JST that contains the raw HTML. If

you took the above markup and saved it into `app/templates/users/form.jst`, it would be accessible as `JST["users/form"]()`.

You *could* write all the HTML by hand, but we'd like to avoid that.

Another route that might seem appealing is reusing the Rails form builders through the asset pipeline. Consider `app/templates/users/form.jst.ejs.erb` which is processed first with ERB, and then made available as a JST template. There are a few concerns to address, such as including changing the EJS or ERB template delimiters `<% %>` to not conflict and mixing the Rails helper modules into the `Tilt::ERbTemplate` rendering context. However, this approach still only generates markup; it doesn't serialize forms into data hashes or Backbone models.

Serializing forms

The second requirement in building forms is to serialize them into objects suitable for setting Backbone model attributes. Assuming the markup we discussed above, you could approach this manually:

```
var serialize = function(form) {
  var elements = $('input, select, textarea', form);

  var serializer = function(attributes, element) {
    var element = $(element);
    attributes[element.attr('name')] = element.val();
  };

  return _.inject(elements, serializer, []);
};

var form = $('form');
var model = new MyApp.Models.User();
var attributes = serialize(form);
model.set(attributes);
```

This gets you started, but has a few shortcomings. It doesn't handle nested attributes, doesn't handle typing (consider a date picker input; ideally it would set a Backbone model's attribute to a JavaScript Date instance), and will include any `<input type="submit">` elements when constructing the attribute hash.

A Backbone forms library

If you want to avoid writing form markup by hand, your best bet is to use a JavaScript form builder. Since the model data is being read and written by

Backbone views and models, it's ideal to have markup construction and form serialization implemented on the client side.

One solid implementation is https://github.com/powmedia/backbone-forms[`backbone-forms` by Charles Davison]. It provides markup construction and serialization, as well as a method for declaring a typed schema to support both of those facilities. It offers a flexible system for adding custom editor types, and supports configuring your form markup structure by providing HTML template fragments.

Display server errors

We are assuming, with a hybrid Rails/Backbone application, that at least some of your business logic resides on the server. Let's take a look at the client/server interaction that takes place when a user of the example application creates a task.

The client side interface for creating a new task is structured similarly to a traditional Rails form. Although moderated by Backbone views and models, essentially there is a form whose contents are submitted to the Rails server, where attributes are processed and a response is generated.

Let's add a validation to the Task Rails model, ensuring each task has something entered for the title:

```
  validates :title, :presence => true
```

Now, if you create a task without a title, the Rails `TasksController` still delivers a response:

```
def create
  respond_with(current_user.tasks.create(params[:task]))
end
```

but the response now returns with an HTTP response code of 422 and a JSON response body of `{"title":["can't be blank"]}`.

Establishing a few conventions, we can display these per-field errors alongside their corresponding form inputs. We'll establish a few conventions that, when we can adhere to them, allow us to render the Rails validation errors inline on the form. Depending on how you structure markup in your application, you can employ a variation on this approach.

For an example, let's examine a form field modeled after Formtastic conventions:

```
<form id="example_form">
  <ol>
    <li id="task_title_input">
      <label for="task_title">Title</label>
      <input id="task_title" name="title" type="text">
      <!--
        <p class="inline_errors">
          The error for this field will be rendered here.
        </p>
      -->
    </li>
  </ol>
</form>
```

Elsewhere, likely in a view class, when a user triggers a save action in the interface, we save the form's corresponding model. If the `save()` fails, we'll parse the model attributes and corresponding error(s) from the server's response and render an `ErrorView`.

```
var formField = $('form#example_form');

model.on('error', function(model, response, options) {
  var attributesWithErrors = JSON.parse(response.responseText);

  new ErrorView({
    el: formField,
    attributesWithErrors: attributesWithErrors
  }).render();
});

model.save();
```

The `ErrorView` iterates over the response attributes and their errors (there may be more than one error per model attribute), rendering them inline into the form. The `ErrorView` also adds the `error` CSS class to the `<li>` field container:

```
ErrorView = Backbone.View.extend({
  initialize: function(options) {
    this.attributesWithErrors = this.options.attributesWithErrors;
    _.bindAll(this, "clearErrors", "renderErrors", "renderError", "fieldFor");
  },

  render: function() {
    this.clearOldErrors();
```

```
    this.renderErrors();
  },

  clearOldErrors: function() {
    this.$(".error").removeClass("error");
    this.$("p.inline_errors").remove();
  },

  renderErrors: function() {
    _.each(this.attributesWithErrors, this.renderError);
  },

  renderError: function(errors, attribute) {
    var errorString = errors.join(", ");
    var field = this.fieldFor(attribute);
    var errorTag = $('<p>').addClass('inline_errors').text(errorString);
    field.append(errorTag);
    field.addClass("error");
  },

  fieldFor: function(attribute) {
    return this.$('li[id*="_' ‘ attribute ‘ '_input"]');
  }
});
```

Internationalization

When you move your application's view logic onto the client, such as with Backbone, you quickly find that the library support for views is not as comprehensive as what you have on the server. The http://guides.rubyonrails.org/i18n.html[Rails internationalization (i18n) API], provided via the https://rubygems.org/gems/i18n[i18n gem], is not automatically available to client-side view rendering. We'd like to take advantage of that framework, as well as any localization work you've done, if you are adding Backbone into an existing app.

There is a JavaScript library, available with Rails support as a Ruby gem https://github.com/fnando/i18n-js[`i18n-js`], that provides access to your i18n content as a JavaScript object, similar to the way the JST object provides access to your templates.

From the documentation, you can link the client-side locale to the server-side locale:

```
<script type="text/javascript">
  I18n.defaultLocale = "<%= I18n.default_locale %>";
```

```
  I18n.locale = "<%= I18n.locale %>";
</script>
```

. . . and then use the `I18n` JavaScript object to provide translations:

```
// translate with your default locale
I18n.t("some.scoped.translation");

// translate with explicit setting of locale
I18n.t("some.scoped.translation", {locale: "fr"});
```

You can use the `I18n.t()` function inside your templates, too:

```
<nav>
  <a href="#/"><%= I18n.t("nav.links.home") %></a>
  <a href="#/projects"><%= I18n.t("nav.links.projects") %></a>
  <a href="#/settings"><%= I18n.t("nav.links.settings") %></a>
</nav>
```

Number, currency, and date formatting is available with `i18n.js` as well - see the documentation for further usage information.

Chapter 6

Models and collections

Filters and sorting

When using our Backbone models and collections, it's often handy to filter the collections by reusable criteria, or sort them by several different criteria.

Filters

To filter a `Backbone.Collection`, as with Rails named scopes, first define functions on your collections that filter by your criteria, using the `select` function from Underscore.js; then, return new instances of the collection class. A first implementation might look like this:

```
var Tasks = Backbone.Collection.extend({
  model: Task,
  url: '/tasks',

  complete: function() {
    var filteredTasks = this.select(function(task) {
      return task.get('completed_at') !== null;
    });
    return new Tasks(filteredTasks);
  }
});
```

Let's refactor this a bit. Ideally, the filter functions will reuse logic already defined in your model class:

```
var Task = Backbone.Model.extend({
  isComplete: function() {
    return this.get('completed_at') !== null;
  }
});

var Tasks = Backbone.Collection.extend({
  model: Task,
  url: '/tasks',

  complete: function() {
    var filteredTasks = this.select(function(task) {
      return task.isComplete();
    });
    return new Tasks(filteredTasks);
  }
});
```

Going further, notice that there are actually two concerns in this function. The first is the notion of filtering the collection, and the second is the specific filtering criteria (`task.isComplete()`).

Let's separate the two concerns here, and extract a `filtered` function:

```
var Task = Backbone.Model.extend({
  isComplete: function() {
    return this.get('completed_at') !== null;
  }
});

var Tasks = Backbone.Collection.extend({
  model: Task,
  url: '/tasks',

  complete: function() {
    return this.filtered(function(task) {
      return task.isComplete();
    });
  },

  filtered: function(criteriaFunction) {
    return new Tasks(this.select(criteriaFunction));
  }
});
```

We can extract this function into a reusable mixin, abstracting the `Tasks` collection class using `this.constructor`:

```
var FilterableCollectionMixin = {
  filtered: function(criteriaFunction) {
    return new this.constructor(this.select(criteriaFunction));
  }
};

var Task = Backbone.Model.extend({
  isComplete: function() {
    return this.get('completed_at') !== null;
  }
});

var Tasks = Backbone.Collection.extend({
  model: Task,
  url: '/tasks',

  complete: function() {
    return this.filtered(function(task) {
      return task.isComplete();
    });
  }
});

_.extend(Tasks.prototype, FilterableCollectionMixin);
```

Propagating collection changes

The `FilterableCollectionMixin`, as we've written it, will produce a filtered collection that does not update when the original collection is changed. To do so, bind to the change, add, and remove events on the source collection, reapply the filter function, and repopulate the filtered collection:

```
var FilterableCollectionMixin = {
  filtered: function(criteriaFunction) {
    var sourceCollection = this;
    var filteredCollection = new this.constructor;

    var applyFilter = function() {
      filteredCollection.reset(sourceCollection.select(criteriaFunction));
    };

    this.bind("change", applyFilter);
    this.bind("add",    applyFilter);
    this.bind("remove", applyFilter);
```

```
    applyFilter();

    return filteredCollection;
  }
};
```

Sorting

The simplest way to sort a `Backbone.Collection` is to define a `comparator` function. This functionality is built in:

```
var Tasks = Backbone.Collection.extend({
  model: Task,
  url: '/tasks',

  comparator: function(task) {
    return task.dueDate;
  }
});
```

If you'd like to provide more than one sort order on your collection, you can use an approach similar to the `filtered` function above, and return a new `Backbone.Collection` whose `comparator` is overridden. Call `sort` to update the ordering on the new collection:

```
var Tasks = Backbone.Collection.extend({
  model: Task,
  url: '/tasks',

  comparator: function(task) {
    return task.dueDate;
  },

  byCreatedAt: function() {
    var sortedCollection = new Tasks(this.models);
    sortedCollection.comparator = function(task) {
      return task.createdAt;
    };
    sortedCollection.sort();
    return sortedCollection;
  }
});
```

Similarly, you can extract the reusable concern to another function:

```
var Tasks = Backbone.Collection.extend({
  model: Task,
  url: '/tasks',

  comparator: function(task) {
    return task.dueDate;
  },

  byCreatedAt: function() {
    return this.sortedBy(function(task) {
      return task.createdAt;
    });
  },

  byCompletedAt: function() {
    return this.sortedBy(function(task) {
      return task.completedAt;
    });
  },

  sortedBy: function(comparator) {
    var sortedCollection = new Tasks(this.models);
    sortedCollection.comparator = comparator;
    sortedCollection.sort();
    return sortedCollection;
  }
});
```

. . . And then into another reusable mixin:

```
var SortableCollectionMixin = {
  sortedBy: function(comparator) {
    var sortedCollection = new this.constructor(this.models);
    sortedCollection.comparator = comparator;
    sortedCollection.sort();
    return sortedCollection;
  }
};

var Tasks = Backbone.Collection.extend({
  model: Task,
  url: '/tasks',
```

```
  comparator: function(task) {
    return task.dueDate;
  },

  byCreatedAt: function() {
    return this.sortedBy(function(task) {
      return task.createdAt;
    });
  },

  byCompletedAt: function() {
    return this.sortedBy(function(task) {
      return task.completedAt;
    });
  }
});

_.extend(Tasks.prototype, SortableCollectionMixin);
```

Just as with the `FilterableCollectionMixin` before, the `SortableCollectionMixin` should observe its source if updates are to propagate from one collection to another:

```
var SortableCollectionMixin = {
  sortedBy: function(comparator) {
    var sourceCollection = this;
    var sortedCollection = new this.constructor;
    sortedCollection.comparator = comparator;

    var applySort = function() {
      sortedCollection.reset(sourceCollection.models);
      sortedCollection.sort();
    };

    this.on("change", applySort);
    this.on("add",    applySort);
    this.on("remove", applySort);

    applySort();

    return sortedCollection;
  }
};
```

Validations

The server is the authoritative place for verifying whether data being stored is valid. Even though Backbone.js exposes an API for performing client-side validations, when it comes to validating user data in a Backbone.js application, we want to continue to use the very same mechanisms on the server side that we've used in Rails all along: the ActiveModel validations API.

The challenge is tying the two together: letting your ActiveRecord objects reject invalid user data, and having the errors bubble all the way up to the interface for user feedback - and keeping it all seamless to the user and easy for the developer.

Let's wire this up. To get started, we'll add a validation on the task's title attribute on the ActiveRecord model, like so:

```
class Task < ActiveRecord::Base
  validates :title, presence: true
end
```

On the Backbone side of the world, we have a Backbone task called `YourApp.Models.Task`:

```
YourApp.Models.Task = Backbone.Model.extend({
  urlRoot: '/tasks'
});
```

We also have a place where users enter new tasks - just a form on the task list:

```
<form>
  <ul>
    <li class="task_title_input">
      <label for="title">Title</label>
      <input id="title" maxlength="255" name="title" type="text">
    </li>
    <li>
      <button class="submit" id="create-task">Create task</button>
    </li>
  </ul>
</form>
```

On the `NewTask` Backbone view, we bind the button's click event to a new function that we'll call `createTask`:

```
YourApp.Views.NewTask = Backbone.View.extend({
  events: {
    "click #create-task": "createTask"
  },

  createTask: {
    // grab attribute values from the form
    // storing them on the attributes hash
    var attributes = {};
    _.each(this.$('form input, form select'), function(element) {
      var element = $(element);
      if(element.attr('name') != "commit") {
        attributes[element.attr('name')] = element.val();
      }
    });

    var self = this;
    // create a new task and save it to the server
    new YourApp.Models.Task(attributes).save({
        success: function() { /* handle success */ }
        error:   function() { /* validation error occurred, show user */ }
      });
    return false;
  }
})
```

This gets the job done, but let's introduce a new class to handle extracting attributes from the form so that it's decoupled from this view and is therefore easier to extend and reuse.

We'll call this the `FormAttributes`, and its code is as follows:

```
FormAttributes = function(form) {
  this.form = form;
}

_.extend(FormAttributes.prototype, {
  attributes: function() {
    var attributes = {};
    _.each($('input, select', this.form), function(element) {
      var element = $(element);
      if(element.attr('name') != "commit") {
        attributes[element.attr('name')] = element.val();
      }
    });
    return attributes;
```

```
  }
});
```

With this class in place, we can rewrite our form submit action to:

```
YourApp.Views.NewTask = Backbone.View.extend({
  events: {
    "click #create-task": "createTask"
  },

  createTask: {
    var attributes = new FormAttributes(this.$('form')).attributes();

    var self = this;
    // create a new task and save it to the server
    new YourApp.Models.Task(attributes).save({
        success: function() { /* handle success */ }
        error:   function() { /* validation error occurred, show user */ }
      });
    return false;
  }
})
```

When you call `save()` on a Backbone model, Backbone will delegate to `.sync()` and create a POST request on the model's URL, where the payload is the attributes that you've passed onto the `save()` call.

The easiest way to handle this in Rails is to use `respond_to`/`respond_with`, available in Rails 3 applications:

```
class TasksController < ApplicationController
  respond_to :json
  def create
    task = Task.create(params)
    respond_with task
  end
end
```

When the task is created successfully, Rails will render the show action using the object that you've passed to the `respond_with` call, so make sure the show action is defined in your routes:

```
resources :tasks, only: [:create, :show]
```

When the task cannot be created successfully because some validation constraint is not met, the Rails responder will render the model's errors as a JSON object, and use an HTTP status code of 422, which will alert Backbone that there was an error in the request and it was not processed.

The response from Rails in that case looks something like this:

```
{ "title": ["can't be blank"] }
```

That two-line action in a Rails controller is all we need to talk to our Backbone models and handle error cases.

Back to the Backbone model's `save()` call: Backbone will invoke one of two callbacks when it receives a response from the Rails app, so we simply pass in a hash containing a function to run for both the success and the error cases.

In the success case, we may want to add the new model instance to a global collection of tasks. Backbone will trigger the add event on that collection, which is a chance for some other view to bind to that event and re-render itself so that the new task appears on the page.

In the error case, however, we want to display inline errors on the form. When Backbone triggers the `error` callback, it passes along two parameters: the model being saved and the raw response. We have to parse the JSON response and iterate through it, rendering an inline error on the form corresponding to each of the errors. Let's introduce a couple of new classes that will help along the way.

First is the `ErrorList`. An `ErrorList` encapsulates parsing of the raw JSON that came in from the server and provides an iterator to easily loop through errors:

```
ErrorList = function (response) {
  if (response && response.responseText) {
    this.attributesWithErrors = JSON.parse(response.responseText);
  }
};

_.extend(ErrorList.prototype, {
  each: function (iterator) {
    _.each(attributesWithErrors, iterator);
  },

  size: function() {
    return _.size(attributesWithErrors);
  }
});
```

Next up is the `ErrorView`, which is in charge of taking the `ErrorList` and appending each inline error in the form, providing feedback to the user that their input is invalid:

```
ErrorView = Backbone.View.extend({
  initialize: function() {
    _.bindAll(this, "renderError");
  },

  render: function() {
    this.$(".error").removeClass("error");
    this.$("p.inline-errors").remove();
    this.options.errors.each(this.renderError);
  },

  renderError: function(errors, attribute) {
    var errorString = errors.join(", ");
    var field = this.fieldFor(attribute);
    var errorTag = $('<p>').addClass('inline-errors').text(errorString);
    field.append(errorTag);
    field.addClass("error");
  },

  fieldFor: function(attribute) {
    return $(this.options.el).find('[id*="_' + attribute + '_input"]').first();
  }
});
```

Note the `fieldFor` function. It expects a field with an id containing a certain format. Therefore, in order for this to work, the form's HTML must contain a matching element. In our case, it was the list item with an id of `task_title_input`.

When a Backbone view's `el` is already on the DOM, we need to pass it into the view's constructor. In the case of the `ErrorView` class, we want to operate on the view that contains the form that originated the errors.

To use these classes, we take the response from the server and pass that along to the `ErrorList` constructor, which we then pass to the `ErrorView`, which will do its fine job inserting the inline errors when we call `render()` on it. Putting it all together, our save call's callbacks now look like this:

```
var self = this;
var model = new YourApp.Models.Task(attributes);
model.save({
  error: function(model, response) {
```

```
    var errors = new ErrorList(response);
    var view   = new ErrorView( { el: self.el, errors: errors } );
    view.render();
  }
});
```

Here, we've shown how you can decouple different concerns into their own classes, creating a system that is easier to extend, and potentially arriving at solutions generic enough even to be shared across applications. Our simple `FormAttributes` class has a long way to go. It can grow up to handle many other cases, such as dates.

One example of a generic library that handles much of what we've done here, as well as helpers for rendering the forms, is Backbone.Form. In order to know how to render all attributes of a model, it requires you to specify a "schema" on the model class - and it will take it from there. The source for Backbone.Form can be found on github.

Model relationships

In any non-trivial application, you will have relationships in your domain model that are valuable to express on the client side. For example, consider a contact management application where each person in your contact list has many phone numbers, each of a different kind.

Or, consider a project planning application where there are Teams, Members, and Projects as resources (models and collections). There are relationships between each of these primary resources, and those relationships in turn may be exposed as first-class resources: a Membership to link a Team and a Member, or a Permission to link a Team with a Project. These relationships are often exposed as first-class models - so they can be created and destroyed the same way as other models, and so that additional domain information about the relationship, such as a duration, rate, or quantity, can be described.

These model relationships don't have to be persisted by a relational database. In a chatroom application whose data is persisted in a key-value store, the data could still be modeled as a Room which has many Messages, as well as Memberships that link the Room to Users. A content management application that stores its data in a document database still has the notion of hierarchy, where a Site contains many Pages, each of which constitutes zero or more Sections.

In a vanilla Rails application, the object model is described on the server side with ActiveRecord subclasses, and exposed to the Backbone client through a JSON HTTP API. You have a few choices to make when designing this API, largely focused on the inherent coupling of model relationships and data – when

you handle a request for one resource, which of its associated resources (if any) do you deliver, too?

Then, on the client side, you have a wide degree of choice in how to model the relationships, when to eagerly pre-fetch associations and when to lazily defer loading, and whether to employ a supporting library to help define your model relationships.

Backbone-relational plugin

If your use cases are supported by it, Paul Uithol's Backbone-relational is arguably the most popular and actively maintained library for this. It lets you declare one-to-one, one-to-many, and many-to-one relations on your Backbone models by extending a new base class, `Backbone.RelationalModel`. It's good to understand how this works under the hood, so we'll cover one way to implement a relational object model in Backbone below, but we encourage you to check out the `Backbone-relational` plugin as a way to work at a higher level of abstraction.

Relations in the task app

In the example application, there are users which have many tasks. Each task has many attachments and assignments. Tasks are assigned to users through assignments, so tasks have many assigned users as well.

Deciding how to deliver data to the client

Before you decide how to model your JSON API or how to declare your client-side model relationships, consider the user experience of your application. For `TaskApp`, we decided to have interactions as follows:

- A user signs up or logs in
- The user is directed to their dashboard
- The dashboard shows all tasks, including assigned users, but without attachments
- When a user views the details of an individual task, the attachments for that task are displayed

This leads us to see that a user's tasks and their assignees are used immediately upon navigating to the dashboard, but the attachment data for a task are not needed upon initial page load, and may well never be needed at all.

Let's say that we are also planning for the user to have continuous network access, but not necessarily with a high speed connection. We should also keep in mind that users tend to view their list of tasks frequently, but rarely view the attachments.

Based on these points, we will bootstrap the collections of tasks and assignees inside the dashboard, and defer loading of associated attachments until after the user clicks through to a task.

We could have selected from several other alternatives, including:

- Don't preload any information, and deliver only static assets (HTML, CSS, JS) on the dashboard request. Fetch all resources over separate XHR calls. This can provide for a shorter initial page load time, with the cost of a longer wait for actual interactivity. Although the byte size of the page plus data is roughly the same, the overhead of additional HTTP requests incurs extra load time.
- Preload all the information, including attachments. This would work well if we expect users to frequently access the attachments of many tasks, but incurs a longer initial page load.
- Use `localStorage` as the primary storage engine, and sync to the Rails server in the background. While this would be advantageous if we expected network access to be intermittent, it incurs the additional complexity of server-side conflict resolution if two clients submit conflicting updates.

Designing the HTTP JSON API

Now that we know we'll bootstrap the tasks with assignees and defer the Associations, we should decide how to deliver the deferred content. Our goal is to fetch attachments for an individual task. Let's discuss two options.

One way we could approach this is to issue an API call for the nested collection:

```
$ curl http://localhost:3000/tasks/78/attachments.json | ppjson
[
  {
    "id": "32",
    "file_url": "https://s3.amazonaws.com/tasksapp/uploads/32/mock.png"
  },
  {
```

```
    "id": "33",
    "file_url": "https://s3.amazonaws.com/tasksapp/uploads/33/users.jpg"
  }
]
```

Note that we will authenticate API requests with cookies, just like normal user logins, so the actual curl request would need to include a cookie from a logged-in user.

Another way we could approach this is to embed the comment and attachment data in the JSON representation of an individual task, and deliver this data from the `/tasks/:id` endpoint:

```
$ curl http://tasksapp.local:3000/tasks/78.json | ppjson
{
  /* some attributes left out for clarity */

  "id": 78,
  "user_id": 1,
  "title": "Clean up landing page",
  "attachments": [
    {
      "id": "32",
      "upload_url": "https://s3.amazonaws.com/tasksapp/uploads/32/mock.png"
    }
  ]
}
```

We'll take this approach for the example application, because it illustrates parsing nested models in Backbone.

At this point, we know that our HTTP JSON API should support at least the following Rails routes:

```
resources :tasks, :only => [:show, :create] do
  resources :attachments, :only => [:create]
end
```

As an aside: In some applications, you may choose to expose a user-facing API. It's valuable to dogfood this endpoint by making use of it from your own Backbone code. Often these APIs will be scoped under an "/api" namespace, possibly with an API version namespace as well like "/api/v1".

Implementing the API: presenting the JSON

To build the JSON presentation, we have a few options. Rails already comes with support for overriding the `Task#as_json` method, which is probably the easiest thing to do. However, logic regarding the JSON representation of a model is not the model's concern. An approach that separates presentation logic is preferable, such as creating a separate presenter object, or writing a builder-like view.

The RABL gem helps you concisely build a view of your models, and keeps this logic in the presentation tier.

RABL allows you to create templates where you can easily specify the JSON representation of your models. If you've worked with the `builder` library to generate XML such as an RSS feed, you'll feel right at home.

To use it, first include the `rabl` and `yajl-ruby` gems in your Gemfile. Then you can create a view ending with `.json.rabl` to handle any particular request. For example, a `tasks#show` action may look like this:

```
class TasksController < ApplicationController
  respond_to :json

  def show
    @task = Task.find(params[:id])
    respond_with @task
  end
end
```

Rails' responder will first look for a template matching the controller/action with the format in the file name, in this case `json`. If it doesn't find anything, it will invoke `to_json` on the `@task` model, but in this case we are providing one in `app/views/tasks/show.json.rabl`, so it will render that instead:

```
object @task
attributes(:id, :title, :complete)
child(:user) { attributes(:id, :email) }
child(:attachments) { attributes(:id, :email) }
```

Parsing the JSON and instantiating client-side models

Now that our API delivers the `Task` JSON to the client, including its nested `Attachments`, we need to correctly handle this nested data in the client-side

model. Instead of a nested hash of attributes on the `Task`, we want to instantiate a Backbone collection for the attachments that contains a set of Backbone `Attachment` models.

The JSON for the attachments is initially set on the Backbone `Task` model as a Backbone attribute which can be accessed with `get()` and `set()`. We are replacing it with an instance of a Backbone `Attachments` collection and placing that as an object property:

```
taskBeforeParsing.get('attachments')
// => [ { id: 1, upload_url: '...' }, { id: 2, upload_url: '...' } ]
taskBeforeParsing.attachments
// => undefined

/* parse attributes... */

taskAfterParsing.get('attachments')
// => undefined
taskAfterParsing.attachments
// => ExampleApp.Collection.Attachments(...)
```

One way to do this is to override the `parse` function on the `Task` model.

There are two `parse` functions in Backbone: one on `Collection` and another on `Model`. Backbone will invoke them whenever a model or collection is populated with data from the server; that is, during `Model#fetch`, `Model#save` (which updates model attributes based on the server's response to the HTTP PUT/POST request), and `Collection#fetch`. It's also invoked when a new model is initialized and `options.parse` is set to `true`.

It's important to note that `parse` is not called by `Collection#reset`, which should be called with an array of models as its first argument. Backbone does support calling `Collection#reset` with just an array of bare attribute hashes, but these will not be routed through `Model#parse`, which can be the source of some confusion.

Another way to intercept nested attributes and produce a full object graph is to bind to the `change` event for the association attribute - in this case, `task.attachments`:

```
ExampleApp.Models.Task = Backbone.Model.extend({
  initialize: function() {
    this.on("change:attachments", this.parseAttachments);
    this.parseAttachments();
  },

  parseAttachments: function() {
```

```
    this.attachments = new ExampleApp.Collections.Attachments(this.get('attachments'));
    delete this.attachments;
  },

  // ...
```

This ensures that our custom parsing is invoked whenever the `attachments` attribute is changed, and when new model instances are created.

When to fetch deferred data

Since a Backbone task doesn't always have its associations filled, when you move from `TasksIndex` to `TasksShow`, you need to invoke `task.fetch()` to pull all the task attributes from `GET /tasks/:id` and populate the `attachments` association. Whose concern is that? Let's talk it through.

You could lazily populate this association by making the `task.attachments` association a function instead of a property. Compare `task.attachments.each` to `task.attachments().each`; in the latter, the accessing function encapsulates the concern of laziness in fetching and populating, but then you run into the issue that fetch is asynchronous. Passing a callback into `attachments()` is kludgy; it exposes the deferred nature of the association everywhere you need to access it.

We'll instead prefer to treat the deferred nature explicitly in the `Routers.Tasks#show` route, a natural application seam to the `TaskShow` view. This frees `TaskShow` from having to know about the persistence details of the model:

```
ExampleApp.Routers.Tasks = Support.SwappingRouter.extend({
  // ...

  show: function(taskId) {
    var task = this.collection.get(taskId);
    var tasksRouter = this;
    task.fetch({
      success: function() {
        var view = new ExampleApp.Views.TaskShow({ model: task });
        tasksRouter.swap(view);
      }
    });
  }
});
```

Now, we have successfully deferred the `Task#attachments` association and kept the concern clear of the view.

Complex nested models

As your domain model grows more complex, you might find that you want to deliver information about more than one model together in a request; i.e., nested attributes. ActiveRecord provides `accepts_nested_attributes_for`, a facility for conveniently passing nested attributes through from requests to ActiveRecord and sorting out the relationships there.

With more interactive web applications, one relevant change is that pages often have several independently usable sections which update more frequently and fluidly compared to their synchronous full-page submitting counterparts. To support this more finely-grained interface, the client-side implementation and the HTTP JSON API are often more finely-grained to match, resulting in fewer bulk submissions with composite data structures.

A useful way to categorize these situations is by whether they are comprised of singular (one-to-one) relationships or plural relationships. It's worth discussing an alternative to `accepts_nested_attributes_for` that works for singular associations. Then, we'll dive into how to model bulk updates for plural associations from Backbone.

Composite models

Consider a signup form that allows a customer to quickly get started with a project management application.

They fill out information for their individual user account, as well as information about the team they represent (and will eventually invite others users from) and perhaps some information about an initial project. One way to model this is to present a `signup` resource that handles creating the correct user, team, and project records. The implementation would involve a vanilla `SignupsController` and a Ruby class `Signup` class that delegates its nested attributes to their respective models.

This composite class encodes the responsibility for translating between the flat data structure produced by the user interface and the cluster of objects that is produced. It's best suited for representing a handful of related records that each have singular relationships - `has_one`/`belongs_to`, rather than plural `has_many` relationships.

There are a few other benefits to these composite classes, too. They are handy for adding any conditional logic in the composition, such as a Signup creating a Billing entry for paid Plan levels. The composite class should be easier to isolation test, compared to testing the persistence outcomes of `accepts_nested_attributes_for`. It's also useful to note that the composite Signup class is not actually persisted; it simply represents a convenient abstraction in the domain model.

In this case, it's straightforward to provide an HTTP API endpoint that exposes the `signups` resource and to model this on the client side as a corresponding Backbone model. All of the attributes on the composite resource are at a single level (not nested), so this is a familiar client-side implementation.

This general pattern encapsulates the composite nature of the resource, leaving the fact that it is persisted across multiple tables as an implementation detail. This keeps the presentation tier simpler, unconcerned with the composite nature of the resource.

accepts_nested_attributes_for

A classic situation to encounter nested attributes is in `has_many :through` relationships. For example, consider a workflow in which you assign multiple people to perform a job. The three domain models are `Job`, `Worker`, and the join model `Assignment`.

```
class Job < ActiveRecord::Base
  has_many :assignments
  has_many :workers, :though => :assignments
end

class Assignment < ActiveRecord::Base
  belongs_to :job
  belongs_to :worker
end

class Worker < ActiveRecord::Base
  has_many :assignments
  has_many :jobs, :through => :assignments
end
```

Earlier, we discussed how Ajax-enabled web applications often provide more finely-grained user interfaces that allow the user to submit information in smaller chunks and allow the developer to model the persistence and HTTP API in finer pieces. Let's say that we have a user interface where we create a job and bulk assign several workers to the new job all in one form. It's possible to achieve a good, fast user experience while still creating the job and its child assignment records in separate requests.

However, it may still be preferable in some cases to perform these bulk submissions, creating a parent record along with several child records all in one HTTP request. We'll model this on the backend with Rails' `accepts_nested_attributes_for`:

```
class Job < ActiveRecord::Base
  has_many :assignments
  has_many :workers, :though => :assignments
  accepts_nested_attributes_for :assignments
end
```

As a quick refresher, this allows us in our Rails code to set `@job.assignments_attributes = [{}, {}, ...]` with an Array of Hashes, each containing attributes for a new `Assignment`, the join model. This behavior of Rails `accepts_nested_attributes_for` shapes our HTTP API: A simple API endpoint controller should be able to pass the request parameters straight through to ActiveRecord, so the JSON going over the HTTP request will look like this:

```
/* POST /api/v1/jobs */
{
  name: "Move cardboard boxes to new warehouse",
  description: "Move boxes from closet C3 to warehouse W2",
  assignmment_attributes: [
    { worker_id: 1 },
    { worker_id: 3 },
    { worker_id: 5 }
  ]
}
```

Shifting our focus to the client-side implementation, we can largely ignore the `Assignment` join model in Backbone, and just model this nested association directly. We'll use a `Job` Backbone model containing a `Workers` collection. This is a simplified perspective of the relationship, but it is all that the client needs to know.

```
MyApp = {};
MyApp.Models = {};
MyApp.Collections = {};

MyApp.Models.Worker = Backbone.Model.extend({
});

MyApp.Collections.Workers = Backbone.Collection.extend({
  model: ExampleApp.Models.Worker
});

MyApp.Models.Job = Backbone.Model.extend({
  urlRoot: '/api/v1/jobs',

  initialize: function() {
```

```
    this.workers = new MyApp.Collections.Workers();
  },

  toJSON: function() {
    var json = _.clone(this.attributes);

    json.assignment_attributes = this.workers.map(function(worker) {
      return { worker_id: worker.id };
    });

    return json;
  }
});
```

Now, you can add workers directly to the job:

```
var worker3 = new MyApp.Models.Worker({ id: 3 });
var worker5 = new MyApp.Models.Worker({ id: 5 });

var job = new MyApp.Models.Job();
job.set({ title: "Raise barn walls" });
job.workers.add(worker3);
job.workers.add(worker5);

JSON.stringify(job.toJSON()) // Results in:
                             //
                             // {
                             //   "title":  "Raise barn walls",
                             //   "assignment_attributes": [
                             //     {"worker_id":3},
                             //     {"worker_id":5}
                             //   ]
                             // }
```

. . . and saving the Backbone `Job` model will submit correctly structured JSON to the Rails server.

This, of course, only covers the creation of nested bulk models. Subsequently fetching a nested object graph from the server involves a handful of separate design decisions around producing JSON on the server and parsing it on the client. These concerns are discussed in the "Model relationships" chapter.

Example for `accepts_nested_attributes_for`

In the example application, a task may be assigned to zero or more users. The association is tracked through an `Assignment` join model, and you can create assignments and tasks at the same time. Users can see tasks they have created or tasks that others have created and assigned to them.

We use `accepts_nested_attributes_for` for persisting the task and its nested assignments. The `Task` Backbone model takes care of parsing the assignment JSON to nest an `Assignments` collection inside itself. It also provides correctly-formatted JSON so that Rails picks up the nested association.

The `TasksNew` view handles the expanding interface for adding more assignees, and is also responsible for finding the Backbone `User` models by email to associate them to the task while it is constructed.

Duplicating business logic across the client and server

When you're building a multi-tier application where business logic is spread across tiers, one big challenge you face is to avoid duplicating that logic across tiers. There is a trade-off here, between duplication and performance. It's desirable to have only one implementation of a particular concern in your domain, but it's also desirable for your application to perform responsively.

An example: model validations

For example, let's say that a user must have an email address.

At one end of the scale, there is no duplication: All business logic is defined in one tier, and other tiers access the logic by remote invocation. Your Rails `Member` model provides a validation:

```
class Member < ActiveRecord::Base
  validate :email, :presence => true
end
```

The Backbone view attempts to persist the member as usual, binding to its `error` event to handle the server-side error:

```
var MemberFormView = Backbone.View.extend({
  events: {
    "submit form": "submit"
```

```
  },

  initialize: function() {
    _.bindAll(this, "error");
    this.model.bind("error", this.error);
  },

  render: function() {
    // render form...
  },

  submit: function() {
    var attributes = new FormSerializer(this.$('form')).attributes();
    this.model.save(attributes);
  },

  error: function(model, errorResponse) {
    var errors = new ErrorList(errorResponse);
    new ErrorView({ el: self.el, errors: errors }).render();
  }
});
```

This uses the `ErrorView` class, which is able to parse the error hash returned from Rails, which was discussed in the "Validations" section of the "Models and Collections" chapter.

> This is probably the first time you'll see `_.bindAll()`, so let's pause briefly to introduce what it is doing.
>
> When an event is triggered, the code invoking the callback is able to set the JavaScript context. By calling `_.bindAll(this, "error")`, we are instead overriding whatever context it may have been, and setting it to `this`. This is necessary so that when we call `this.$(\'form\')` in the `error()` callback, we get the right object back.
>
> Always use `_.bindAll` when you need to force the JavaScript context (`this`) within a function's body.

In the case of no duplication, your Backbone `Member` model does not declare this validation. A user fills out a form for a creating a new member in your application, submits the form, and, if they forget to include an email address, a validation message is displayed. The application delegates the entire validation concern to the server, as we saw in the "Validations" section of the "Models and Collections" chapter.

However, round-tripping validation to the server can be too slow in some cases, and we'd like to provide feedback to the end user more quickly. To do this, we

have to implement the validation concern on the client side as well. Backbone provides a facility for validating models during their persistence, so we could write:

```
var Member = Backbone.Model.extend({
  validate: function() {
    var errors = {};
    if (_.isEmpty(this.get('email'))) {
      errors.email = ["can't be blank"];
    }
    return errors;
  }
});
```

Conveniently, we've structured the return value of the `validate()` function to mirror the structure of the Rails error JSON we saw returned above. Now, we *could* augment the `ErrorView` class's constructor function to handle either client-side or server-side errors:

```
var ErrorList = function(responseOrErrors) {
  if (responseOrErrors && responseOrErrors.responseText) {
    this.attributesWithErrors = JSON.parse(response.responseText);
  } else {
    this.attributesWithErrors = responseOrErrors;
  }
};
```

With Backbone, the `validate()` function is called for each invocation of `set()`, so as soon as we set the email address on the member, its presence is validated. For the user experience with the quickest response, we could observe changes on the email form field, updating the model's `email` attribute whenever it changes, and displaying the inline error message immediately.

With `ErrorList` able to handle either client-side or server-side error messages, we have a server-side guarantee of data correctness, footnote[1] and a responsive UI that can validate the member's email presence without round-tripping to the server.

The tradeoff we've made is that of duplication; the concern of "what constitutes a valid member" is written twice – in two different languages, no less. In some cases this is unavoidable. In others, there are mitigation strategies for reducing the duplication, or at least its impact on your code quality and maintainability.

Let's take a look at what kinds of logic you might find duplicated, and then at strategies for reducing duplication.

[1] At least, we have a guarantee at the application level; database integrity and the possibility of skew between Rails models and DB content is another discussion entirely.

Kinds of logic you duplicate

In Rails applications, our model layer can contain a variety of kinds of business logic:

- Validations: This is pretty straightforward, since there's a well-defined Rails API for validating ActiveModel classes.
- Querying: Sorting and filtering fall into this category. Implementations vary slightly, but are often built with `named_scope` or class methods returning `ActiveRecord::Relation` instances. Occasionally querying is delegated to class other than the ActiveRecord instance.
- Callbacks: Similar to validations, there's a well-defined API for callbacks (or "lifecycle events") on Rails models; `after_create` and so on.
- Algorithms: Everything else. Sometimes they're implemented on the ActiveRecord instances, but are often split out into other classes and used via composition. One example from commerce apps would be an `Order` summing the costs of its `LineItems`. Or consider an example from an agile project planning application, where a `ProjectPlan` recalculates a `Project`'s set of `UserStory` objects into weekly `Iteration` bucket objects.

There are often other methods on your Rails models, but they are either a mix of the above categories (a `state_machine` implementation could be considered a mix of validations and callback) and other methods that don't count as business logic - methods that are actually implementing presentation concerns are a frequent example.

It's worth considering each of these categories in turn, and how they can be distributed across client and server to provide a responsive experience.

Validations

Validations are probably the lowest-hanging fruit. Since the API for declaring validations is largely declarative and well-bounded, we can imagine providing an interface that introspects Rails models and builds a client-side implementation automatically.

Certainly, there are cases which aren't possible to automate, such as custom Ruby validation code or validations which depend on a very large dataset that would be impractical to deliver to the client (say, a ZIP code database). These cases would need to fall back to either an XHR call to the server-side implementation, or a custom-written client-side implementation - a duplicate implementation.

This is actually what the client_side_validations gem does, only it is not available for Backbone yet. However, it is on the roadmap, and the "model" branch is a work in progress of this functionality. We will be keeping an eye on this branch: https://github.com/bcardarella/client_side_validations/tree/model

Querying

Like validations, Rails the syntax and outcome of many common Rails query methods are relatively declarative. It may be possible to convert server-side scopes into client-side collection filtering. However, that is of questionable value in most Backbone applications we've encountered.

In most Backbone apps there ends up being little duplication between client and server sorting and filtering. Either the logic happens on the client and is therefore not needed on the server, or the search logic happens on the server and is not needed on the client.

If you find that your application has duplication here, consider whether there may be a better way to separate responsibilities.

Callbacks

We've found that model callbacks are rarely duplicated between the client and server sides. It's actually more likely that your client-side models will differ sufficiently from the server-side models, since they are in the presentation tier and the concerns are different.

As you continue to push more logic client-side - as we've found is the trend when using Backbone - you may find that some life-cycle events may move or be duplicated from the server to the client. The implementation and concern of these often varies significantly from what they were on the server. For example, a callback translated to Backbone will likely be implemented as an event being fired and listened to by another object.

Algorithms

General algorithms are often the trickiest things for which to resolve duplication between client and server. It's also common that important algorithms are, in fact, needed on both client and server.

The more complicated the algorithm, the more troubling this will become. Bugs may be introduced, and the client- and server-side algorithms might not actually produce the same results.

You could implement the actual logic of the algorithm in JavaScript and then make that available to Ruby, by using something like ExecJS to run the JavaScript code from Ruby. But you must weigh the cost of that additional complexity and overhead against the code of duplicating logic.

Also, you could consider JavaScript on the server side in something like Node.js, exposed via a webservice that Rails can access. However, it is debatable whether this is actually easier.

Finally, it may be possible to reduce duplication by splitting responsibility for the algorithm in pieces: half on the client and half on the server, and then use coordinated communication and caching to accomplish the algorithm and improve the performance, respectively.

More information about this technique can be found here:

- http://c2.com/cgi/wiki?HalfObjectPlusProtocol
- http://c2.com/cgi/wiki?HoppPatternLanguage

Synchronizing between clients

A driving force behind the move to rich client web apps is to improve the user experience. These applications are more responsive and can support more detailed and stateful interactions.

One such interaction involves multiple concurrent users interacting with the same resource in real time. We can deliver a more seamless experience by propagating users' changes to one another as they take place: When you and I edit the same document, I see your changes on my screen as you type them. If you've ever used Google Docs or Google Wave, you've seen this in action.

So, how can we build this functionality into our own applications?

The moving parts

There are a few different pieces that we'll put together for this. The basic parts are:

1. **Change events.** The fundamental unit of information that we broadcast through our system to keep clients in sync. Delivered as messages, these events contain enough information for any receiving client to update its own data without needing a full re-fetch from the server.
2. **An event source.** With trusted clients, changes can originate directly from the client. More often, however, we will want the server to arbitrate changes so that it can apply authorization, data filtering, and validations.

3. **A transport layer that supports pushing to clients.** The WebSocket API is such a transport, and is ideal for its low overhead and latency.

4. **Event-driven clients.** Clients should be able to react to incoming change events, ideally handling them with incremental UI updates rather than redrawing themselves entirely. Backbone helps in this department, as your client-side application is likely already set up to handle such events.

5. **A message bus.** Separating the concern of message delivery from our main application helps it stay smaller and helps us scale our messaging and application infrastructure separately. There are already several great off-the-shelf tools we can use for this.

Putting it together: A look at the life cycle of a change

Revisiting our todo application, we'd like to add the ability to collaborate on todo lists, so that different users will be able to work on the same todo list concurrently. Several users can look at the same list; adding, changing, and checking off items.

There are a few technical decisions mentioned previously. For this example, we will:

1. Use Rails on the server and Backbone on the client.

2. Use the server as the canonical event source so that clients do not have to trust one another. In particular, we'll employ an `ActiveRecord::Observer` that observes Rails model changes and dispatches a change event.

3. Use Faye as the messaging backend, which has Ruby and JavaScript implementations for clients and server. Faye implements the Bayeux protocol, prefers WebSocket for transport (though it gracefully degrades to long polling, CORS, or JSON-P), and supports a bunch of other goodies like clustering and extensions (inbound- and outbound- message filtering, like Rack middleware).

In our application, there are several connected clients viewing the same todo list, and one user, "Alice," makes a change to an item on the list.

Let's take a look at the lifecycle of one change event.

Setup:

1. An instance of JavaScript class `BackboneSync.FayeSubscriber` is instantiated on each client. It is configured with a channel to listen to, and a collection to update.

2. The Faye server is started.
3. The Rails server is started, and several clients are connected and viewing `#todo_lists/1`.

On Alice's machine, the client responsible for the change:

1. Alice clicks "Save" in her view of the list.
2. The "save" view event is triggered.
3. The event handler invokes `this.model.save(attributes)`.
4. `Backbone.Model.prototype.save` calls `Backbone.sync`.
5. `Backbone.sync` invokes `$.ajax` and issues an HTTP PUT request to the server.

On the server:

1. Rails handles the PUT request and calls `#update_attributes` on an ActiveRecord model instance.
2. An `ActiveRecord::Observer` observing this model gets its `#after_save` method invoked.
3. The observer dispatches a change event message to Faye.
4. Faye broadcasts the change event to all subscribers.

On all clients:

1. `FayeSubscriber` receives the change event message, likely over a WebSocket.
2. The subscriber parses the event message, picking out the event (`update`), the `id` of the model to update, and a new set of attributes to apply.
3. The `FayeSubscriber` fetches the model from the collection, and calls `set` on it to update its attributes.

Now all the clients have received the changeset that Alice made.

Implementation: Step 1, Faye server

We'll need to run Faye to relay messages from publishers to subscribers. For Rails apps that depend on Faye, We recommend keeping a `faye/` subdirectory under the app root that contains a `Gemfile` and `config.ru`, and maybe a shell script to start Faye:

```
$ cat faye/Gemfile

source 'http://rubygems.org'
gem 'faye'

$ cat faye/config.ru

require 'faye'
bayeux = Faye::RackAdapter.new(:mount => '/faye', :timeout => 25)
bayeux.listen(9292)

$ cat faye/run.sh

#!/usr/bin/env bash
BASEDIR=$(dirname $0)
BUNDLE_GEMFILE=$BASEDIR/Gemfile
bundle exec rackup $BASEDIR/config.ru -s thin -E production

$ ./faye/run.sh

>> Thin web server (v1.2.11 codename Bat-Shit Crazy)
>> Maximum connections set to 1024
>> Listening on 0.0.0.0:9292, CTRL`C to stop
```

Implementing it: Step 2, ActiveRecord observers

Now that the message bus is running, let's walk through the server code. The Rails app's responsibility is this: Whenever a todo model is created, updated, or deleted, it will publish a change event message.

This is implemented with an `ActiveRecord::Observer`. We provide the functionality in a module:

```
module BackboneSync
  module Rails
    module Faye
      mattr_accessor :root_address
      self.root_address = 'http://localhost:9292'
```

```
    module Observer
      def after_update(model)
        Event.new(model, :update).publish
      end

      def after_create(model)
        Event.new(model, :create).publish
      end

      def after_destroy(model)
        Event.new(model, :destroy).publish
      end
    end

    class Event
      def initialize(model, event)
        @model = model
        @event = event
      end

      def broadcast
        Net::HTTP.post_form(uri, :message => message)
      end

      private

      def uri
        URI.parse("#{BackboneSync::Rails::Faye.root_address}/faye")
      end

      def message
        { :channel => channel,
          :data => data          }.to_json
      end

      def channel
        "/sync/#{@model.class.table_name}"
      end

      def data
        { @event => { @model.id => @model.as_json } }
      end
    end
  end
end
```

```
end
```

. . . and then mix it into a concrete observer class in our application. In this case, we name it `TodoObserver`:

```
class TodoObserver < ActiveRecord::Observer
  include BackboneSync::Rails::Faye::Observer
end
```

This observer is triggered each time a Rails `Todo` model is created, updated, or destroyed. When one of these events happen, the observer sends along a message to our message bus, indicating the change.

Let's say that a `Todo` was just created:

```
>> Todo.create(title: "Buy some tasty kale juice")
=> #<Todo id: 17, title: "Buy some tasty kale juice", created_at: "2011-09-06 20:49:03", up
```

The message looks like this:

```
{
  "channel": "/sync/todos",
  "data": {
    "create": {
      "17": {
        "id": 17,
        "title": "Buy some tasty kale juice",
        "created_at": "2011-09-06T20:49:03Z",
        "updated_at": "2011-09-07T15:01:09Z"
      }
    }
  }
}
```

Received by Faye, the message is broadcast to all clients subscribing to the `/sync/todos` channel, including our browser-side `FayeSubscriber` objects.

Implementing it: Step 3, In-browser subscribers

In each browser, we want to connect to the Faye server, subscribe to events on channels that interest us, and update Backbone collections based on those messages.

Faye runs an HTTP server, and serves up its own client library, so that's easy to pull in:

```
<script type="text/javascript" src="http://localhost:9292/faye.js"></script>
```

To subscribe to Faye channels, instantiate a `Faye.Client` and call `subscribe` on it:

```
var client = new Faye.Client('http://localhost:9292/faye');
client.subscribe('/some/channel', function(message) {
  // handle message
});
```

When the browser receives messages from Faye, we want to update a Backbone collection. Let's wrap up those two concerns into a `FayeSubscriber`:

```
this.BackboneSync = this.BackboneSync || {};

BackboneSync.RailsFayeSubscriber = (function() {
  function RailsFayeSubscriber(collection, options) {
    this.collection = collection;
    this.client = new Faye.Client('<%= BackboneSync::Rails::Faye.root_address %>/faye');
    this.channel = options.channel;
    this.subscribe();
  }

  RailsFayeSubscriber.prototype.subscribe = function() {
    return this.client.subscribe("/sync/" ‘ this.channel, _.bind(this.receive, this));
  };

  RailsFayeSubscriber.prototype.receive = function(message) {
    var self = this;
    return $.each(message, function(event, eventArguments) {
      return self[event](eventArguments);
    });
  };

  RailsFayeSubscriber.prototype.update = function(params) {
    var self = this;
    return $.each(params, function(id, attributes) {
      var model = self.collection.get(id);
      return model.set(attributes);
    });
  };

  RailsFayeSubscriber.prototype.create = function(params) {
    var self = this;
    return $.each(params, function(id, attributes) {
```

```
      var model = new self.collection.model(attributes);
      return self.collection.add(model);
    });
  };

  RailsFayeSubscriber.prototype.destroy = function(params) {
    var self = this;
    return $.each(params, function(id, attributes) {
      var model = self.collection.get(id);
      return self.collection.remove(model);
    });
  };

  return RailsFayeSubscriber;
})();
```

Now, for each collection that we'd like to keep in sync, we instantiate a corresponding `FayeSubscriber`. Say, in your application bootstrap code:

```
MyApp.Routers.TodosRouter = Backbone.Router.extend({
  initialize: function(options) {
    this.todos = new Todos.Collections.TodosCollection();
    new BackboneSync.FayeSubscriber(this.todos, { channel: 'todos' });
    this.todos.reset(options.todos);
  },

  // ...
});
```

Now run the app, and watch browsers receive push updates!

Testing synchronization

Of course, this introduces a great deal of complexity into your app. There's a new daemon running on the server (Faye), and every client now has to correctly listen to its messages and re-render the appropriate views to show the new data. This gets even more complex when the resource being updated is currently being edited by another user. Your own requirements will dictate the correct behavior in cases like that, but what's most important is that you are able to reproduce such workflows in automated tests.

While this book includes a chapter dedicated to testing Backbone applications, this next section describes the tools and approach that will allow you to verify this behavior in tests.

Following an outside-in development approach, we start with an acceptance test and dive into the isolated testing examples when the acceptance tests drive us to them. There's nothing novel in regards to isolation testing of these components, so we will not touch on them here. Instead, we'll describe how to write an acceptance test for the above scenario.

The required pieces for the approach are:

- Ensure a faye server running on your testing environment
- Fire up a browser session using an browser acceptance testing framework
- Sign in as Alice
- Start a second browser session and sign in as Olivia
- Edit some data on Alice's session
- See the edited data reflected on Olivia's session

We will be using Cucumber with Capybara and RSpec for this example.

To ensure the Faye server is running, we merely try to make a connection to it when Cucumber boots, failing early if we can't connect. Here's a small snippet that you can drop in `features/support/faye.rb` to do just that:

```
begin
  Timeout.timeout(1) do
    uri = URI.parse(BackboneSync::Rails::Faye.root_address)
    TCPSocket.new(uri.host, uri.port).close
  end
rescue Errno::ECONNREFUSED, Errno::EHOSTUNREACH, Timeout::Error
  raise "Could not connect to Faye"
end
```

With that in place, we are now sure that Faye is running and we can move on to our Cucumber scenario. Create a `features/sync_task.feature` file and let's describe the desired functionality:

```
  @javascript
  Scenario: Viewing a task edited by another user
    Given the following users exist:
      | email               |
      | alice@example.com   |
      | olivia@example.com  |
    Given the following task exists:
      | title             |
```

```
    | Get Cheeseburgers |
  And I am using session "Alice"
  And I sign in as "alice@example.com"
  Then I should see "Get Cheeseburgers"
  When I switch to session "Olivia"
  And I sign in as "olivia@example.com"
  And I edit the "Get Cheeseburgers" task and rename it to "Buy Cheeseburgers"
  And I switch to session "Alice"
  Then I should see "Buy Cheeseburgers"
```

Thankfully, Capybara allows us to run acceptance tests with client-side behavior by specifying different drivers to run scenarios that require JavaScript vs. those which don't. The very first line above, `@javascript`, tells Capybara to use a JavaScript-enabled driver such as Selenium or capybara-webkit.

The following two steps that create some fixture data are provided by FactoryGirl, which looks into your factory definitions and builds step definitions based on their attributes and associations.

But then we get into the meat of the problem: switching sessions. Capybara introduced the ability to name and switch sessions in your scenarios via the `session_name` method. The definition for the `I am using session "Alice"` step looks like this:

```
When /^I (?:am using|switch to) session "([^"]‘)"$/ do |new_session_name|
  Capybara.session_name = new_session_name
end
```

This allows us to essentially open up different browsers, if you're using the Selenium driver, and it is the key to exercising background syncing code in Capybara acceptance testing.

With this in place, the rest is quite straightforward - we simply interact with the application as you would with any Cucumber scenario; visiting pages, filling in forms, and verifying results on the page, all the while specifying which session you're interacting with.

Additionally, the `BackboneSync.FayeSubscriber` JavaScript class should also be tested in isolation. We've used Jasmine for testing JavaScript behavior successfully, so it is the approach we recommend. For more information about using Jasmine, refer to the "Testing" chapter.

Further reading

For a solid, readable background on idempotent messages, check out *The Importance of Idempotence*.

Uploading attachments

While Ruby gems like paperclip make the API for attaching files to models very similar to the standard ActiveModel attribute persistence API, attaching files to Backbone models is not quite as straightforward. In this section, we'll take a look at the general approach for attaching files, and then examine the specific implementation used in the example application.

Saving files along with attributes

When you save a Backbone model, its attributes are sent to the server via Backbone.sync. It would be ideal to treat file uploads in the same fashion, storing the files as attributes on the client-side Backbone model and uploading them, along with all the other attributes, when then model is saved.

`Backbone.Model#save` delegates to `Backbone.sync` which, by default, transmits data using `$.ajax` with a `dataType` of `json`.

We *could* send files along here, too, using the HTML5 File API to read the file data and send it serialized inside the JSON payload. But this would require us to make server-side changes to parse the file from JSON, and there is no IE support for the File API as of IE9. http://caniuse.com/fileapi

A slightly more sophisticated approach would be to use the FormData API and XMLHttpRequest Level 2 to serialize attributes instead, transmitting them to the server as multipart/form-data, which already has a defined serialization for files. This would allow you to work without modifying your server, but still leaves IE completely unsupported.

To support the broadest set of browsers, but still deliver file uploads in the same request as attributes, you'll use a hidden iframe technique. Probably the most transparent approach is to take advantage of jQuery's AJAX Transport functionality with the jquery.iframe-transport.js plugin. There is a caveat with this approach too, however, as we cannot get at the response headers, breaking automatic content type detection and, more importantly, breaking the use of HTTP response codes to indicate server-side errors. This approach would deliver the smoothest user experience - at the cost of more integration code.

The Remotipart gem provides some conventions for delivering response information back to the client side, although the use-case is slightly different and the library uses the `jquery.form.js ajaxSubmit()` function to perform an iframe upload, instead of the smaller `jquery.iframe-transport.js` plugin.

Separating file upload and model persistence

The general approach we'll take here is to separate the file upload request from the model persistence requests. The server will respond to the upload with an

identifier, which we can use on the client side to populate an attribute on a Backbone model, whether it is a new model or an existing one.

This does mean that you can have unclaimed attachments if the end user leaves the page before saving the parent model, and these should be periodically swept if disk usage is an issue.

When modeling this from the Rails side, you can choose to persist the file upload identifier (e.g., the local path or S3 URL) on one of your models directly, or you can break the attachment out into its own ActiveRecord model and store a foreign key relation on your primary model. For our example we'll do the latter, adding an `Attachment` model and resource to the app.

We'll use the HTML5 File API because it's a straightforward approach to illustrate.

Example, Step 1: Upload interface

In our example task management app, we'd like the owner of a task to attach several images to each task. We want uploads to happen in the task detail view, and to appear in-page as soon as they are uploaded. We don't need to display uploads on the index view.

First, let's write an acceptance test to drive the functionality:

@javascript Feature: Attach a file to a task

As a user I want to attach files to a task So that I can include reference materials

Background: Given I am signed up as "email@example.com" When I sign in as "email@example.com" And I go to the tasks page And I create a task "Buy" And I create a task "Eat"

Scenario: Attach a file to a task When I attach "spec/fixtures/blueberries.jpg" to the "Buy" task Then I should see "blueberries.jpg" attached to the "Buy" task And I should see no attachments on the "Eat" task

Scenario: Attach multiple files to a task When I attach "spec/fixtures/blueberries.jpg" to the "Buy" task And I attach "spec/fixtures/strawberries.jpg" to the "Buy" task Then I should see "blueberries.jpg" attached to the "Buy" task And I should see "strawberries.jpg" attached to the "Buy" task

The first failures we get are from the lack of upload UI. We'll drop down to unit tests to drive this out:

```
//= require application

describe("ExampleApp.Views.TaskShow", function() {
  var task, view, $el;
```

```
  beforeEach(function() {
    task = new ExampleApp.Models.Task({
      id: 1,
      title: "Wake up"
    });

    view = new ExampleApp.Views.TaskShow({ model: task });
    $el = $(view.render().el);
  });

  it("renders the detail view for a task", function() {
    expect($el).toHaveText(/Wake up/);
  });

  it("renders a file upload area", function() {
    expect($el).toContain(".upload label:contains('Attach a file to upload')");
    expect($el).toContain(".upload button:contains('Upload attachment')");
    expect($el).toContain(".upload input[type=file]");
  });

  it("links the upload label and input", function() {
    var $label = $el.find('.upload label');
    var $input = $el.find('.upload input');
    expect($label.attr('for')).toEqual($input.attr('id'));
  });
});
```

Then, we'll add the upload form in the `tasks/show.jst.ejs` template, so the UI elements are in place:

```
<p>Task title</p>

<ul class="attachments">
</ul>

<div class="upload">
  <label for="input">Attach a file to upload</label>
  <input type="file" name="file" />
  <button>Upload attachment</button>
</div>

<a href="#">Back to task list</a>
```

Once our units pass, we run the acceptance tests again. The next failure we see is that nothing happens upon upload. We'll drop down to Jasmine here to write a spec for uploading that asserts the correct upload request is issued:

```
it("uploads the file when the upload method is called", function() {
  view.upload();
  expect(this.requests.length).toEqual(1);
  expect(this.requests[0].requestBody.constructor).toEqual(FormData);
});

it("uploads an attachment for the current task", function() {
  view.upload();
  expect(this.requests[0].url).toEqual("/tasks/1/attachments.json");
});
```

and implement using the `uploader.js` library:

```
render: function () {
  // ...
  this.attachUploader();
  return this;
},

// ...

attachUploader: function() {
  var uploadUrl = "/tasks/" ‘ this.model.get('id') ‘ '/attachments.json';

  this.uploader = new uploader(this.uploadInput(), {
    url:      uploadUrl,
    success:  this.uploadSuccess,
    prefix:   'upload'
  });
},
```

The acceptance tests still aren't passing, and a little digging will reveal that we need to manually set the CSRF token on the upload request. Normally, this would be set by `jquery_ujs.js` with a jQuery AJAX prefilter, but the upload code we are using manually constructs an `XMLHttpRequest` instead of using `$.ajax, so that it may bind to the`onprogress‘ event.

We write a spec:

```
it("sets the CSRF token for the upload request", function() {
  view.upload();
  var expectedCsrfToken = $('meta[name="csrf-token"]').attr('content');
  expect(this.requests[0].requestHeaders['X-CSRF-Token']).toEqual(expectedCsrfToken);
});
```

And add the CSRF token implementation at the end of `attachUploader`:

```
attachUploader: function() {
  // ...

  this.uploader.prefilter = function() {
    var token = $('meta[name="csrf-token"]').attr('content');
    if (token) this.xhr.setRequestHeader('X-CSRF-Token', token);
  };
},
```

And the spec is green.

Example, Step 2: Accept and persist uploads in Rails

At this point, we are sending the upload request from the client, but the server isn't responding, much less persisting the file. This portion is vanilla Rails and Paperclip. We create an `Attachment` model, a nested `:attachments` route under the `tasks` resource, and `AttachmentsController`. Then, we add the paperclip gem to the Gemfile, and include the `has_attached_file` directive in the `Attachment` model along with the corresponding `have_attached_file` example to the `Attachment` model spec.

Now that attachments are uploaded to the server, the final step is to display attachments to the user.

Example, Step 3: Display Uploaded Files

For structuring the attachments in Backbone, we want to be able to do something like the following:

```
<% this.task.attachments.each(function(attachment) { %>
  Attached: <img src="<%= attachment.get('upload_url')" /> %>
<% }); %>
```

So, the task model will have an attachments property that instantiates with an `AttachmentsCollection` instance.

We're providing a JSON representation rooted at the task model using Rabl, which we discussed previously in "Implementing the API: presenting the JSON."

```
object @task

attributes :id, :created_at, :updated_id, :title, :complete, :user_id

child :attachments do
  attributes :id, :created_at, :updated_id, :upload_file_name, :upload_url
end
```

We also tell Rabl to suppress the root JSON node, much like we did with `ActiveRecord::Base.include_root_in_json`:

```
# config/initializers/rabl_init.rb
Rabl.configure do |config|
  config.include_json_root = false
end
```

We can test drive the attachment display from Jasmine; see `task_show_with_attachments_spec.js`:

```
//= require application

describe("ExampleApp.Views.TaskShow for a task with attachments", function() {
  var task, view, $el, blueberryUrl, strawberryUrl;

  beforeEach(function() {
    blueberryUrl = "http://whatscookingamerica.net/Fruit/Blueberries4.jpg";
    strawberryUrl = "http://strawberriesweb.com/three-strawberries.jpg"
    task = new ExampleApp.Models.Task({
      id: 1,
      title: "Buy pies",
      attachments: [
        {
          upload_file_name: "blueberries.jpg",
          upload_url: blueberryUrl
        },
        {
          upload_file_name: "strawberries.jpg",
          upload_url: strawberryUrl
        }
      ]
    });

    view = new ExampleApp.Views.TaskShow({ model: task });
    $el = $(view.render().el);
  });

  it("displays attachments", function() {
    expect($el).toContain(".attachments img[src='" + blueberryUrl + "']")
    expect($el).toContain(".attachments img[src='" + strawberryUrl + "']")
  });

  it("displays attachment filenames", function() {
    var attachments = $el.find(".attachments p");
    expect(attachments.first()).toHaveText('Attached: blueberries.jpg');
```

```
    expect(attachments.last()).toHaveText('Attached: strawberries.jpg');
  });
});
```

We'll represent attachments as an associated collection on `Task`, so we'll need a Backbone model and collection for attachments, too. First, the task model should parse its JSON to populate the associated attachments. Test drive that in the `ExampleApp.Models.Tasks` Jasmine spec:

```
//= require application

describe("ExampleApp.Models.Tasks", function() {
  it("knows if it is complete", function() {
    var completeTask = new ExampleApp.Models.Task({ complete: true });
    expect(completeTask.isComplete()).toBe(true);
  });

  it("knows if it is not complete", function() {
    var incompleteTask = new ExampleApp.Models.Task({ complete: false });
    expect(incompleteTask.isComplete()).toBe(false);
  });
});

describe("ExampleApp.Models.Tasks#initialize", function() {
  var attributes, task;

  beforeEach(function() {
    attributes = {"id":1,
                  "title":"Sweet Task",
                  "attachments":[
                    {"upload_url":"/uploads/1.jpg"},
                    {"upload_url":"/uploads/2.jpg"}
                  ]};
    task = new ExampleApp.Models.Task(attributes);
  });

  it("creates collections for nested attachments", function() {
    var attachments = task.attachments;
    var typeCheck = attachments instanceof ExampleApp.Collections.Attachments;
    expect(typeCheck).toEqual(true);
    expect(attachments.size()).toEqual(2);
  });

  it("populates the collection with Attachment models", function() {
    var attachments = task.attachments;
```

```
    var typeCheck = attachments.first() instanceof ExampleApp.Models.Attachment;
    expect(typeCheck).toEqual(true);
    expect(attachments.first().get('upload_url')).toEqual('/uploads/1.jpg');

    typeCheck = attachments.last() instanceof ExampleApp.Models.Attachment;
    expect(typeCheck).toEqual(true);
    expect(attachments.last().get('upload_url')).toEqual('/uploads/2.jpg');
  });
});

describe("ExampleApp.Models.Task attachments:change", function() {
  it("re-parses the collection", function() {
    var task = new ExampleApp.Models.Task({"attachments":[
                                           {"upload_url":"1.jpg"},
                                           {"upload_url":"2.jpg"}]});
    expect(task.attachments.size()).toEqual(2);

    task.set({"attachments":[{"upload_url":"1.jpg"}]});
    expect(task.attachments.size()).toEqual(1);
  });
});
```

The first failures reference the Backbone attachment model and attachments collection, so we add those, driving the collection out with a spec.

Next, we can implement the task model's JSON parsing to populate its associated attachments:

```
ExampleApp.Models.Task = Backbone.Model.extend({
  initialize: function() {
    this.bind("change:attachments", this.parseAttachments);
    this.parseAttachments();
  },

  parseAttachments: function() {
    var attachmentsAttr = this.get('attachments');
    this.attachments = new ExampleApp.Collections.Attachments(attachmentsAttr);
  },

  // ...

});
```

At this point, we return back to the acceptance test, and it's fully passing.

Chapter 7

Testing

Full-stack integration testing

Your application is built from a collection of loosely coupled modules, spreading across several layers of the development stack. To ensure the application works correctly from the perspective of the end user, full-stack integration testing drives your application and verifies correct functionality from the user interface level. This is also referred to as "acceptance testing."

Introduction

Writing a full-stack integration test for a JavaScript-driven web application will always involve some kind of browser, and although writing an application with Backbone can make a world of difference to you, the tools involved are all the same as far as your browser is concerned. Because your browser can run Backbone applications just like any JavaScript application, you can write integration tests for them just like you would for any JavaScript application. Also, because of tools like Capybara that support various drivers, you can generally test a JavaScript-based application just like you'd test a web application where all the logic lives on the server. This means that having a powerful, rich-client user interface won't make your application any harder to test. If you're familiar with tools like Capybara, Cucumber, and RSpec, you can dive right in and start testing your Backbone application. If not, the following sections should give you a taste of the available tools for full-stack integration tests written in Ruby.

Capybara

Though there is a host of tools available to you for writing automated integration tests, we recommend Capybara. In a hybrid Rails application, where some portions are regular request/response and other portions are JavaScript, it's valuable to have a testing framework that abstracts the difference as much as possible.

Capybara is a high-level library that allows you to write tests from a user's perspective. Consider this example, which uses RSpec:

```
describe "the login process", :type => :request do
  it "accepts an email and password" do
    User.create(:email => 'alice@example.com', :password => 'password')
    visit '/'
    fill_in 'Email', :with => 'alice@example.com'
    fill_in 'Password', :with => 'password'
    click_button 'Log in'
    page.should have_content('You are logged in as alice@example.com')
  end
end
```

Notice that, as you read the spec, you're not concerned about whether the login interface is rendered with JavaScript, or whether the authentication request is over AJAX or not. A high-level library like Capybara keeps you from having to consider the back-end implementation, freeing you to focus on describing the application's behavior from an end-user's perspective. This perspective of writing specs is often called "behavior-driven development" (BDD).

Cucumber

You can take another step toward natural language tests, using Cucumber to define mappings. Cucumber is a test runner and a mapping layer. The specs you write in Cucumber are user stories, written in a constrained subset of English. The individual steps in these stories are mapped to a testing library - in our case, and probably most cases, to Capybara.

This additional layer of abstraction can be helpful for a few reasons. Some teams have nontechnical stakeholders writing integration specs as user stories. Cucumber sits at a level of abstraction that fits comfortably there: high-level enough for nontechnical stakeholders to write in, but precise enough to be translated into automated tests.

On other teams, the person writing the story is the same person who implements it. Still, it is valuable to use a tool that reinforces the distinction between the description phase and the implementation phase of the test. In the description phase, you are writing an English description of the software interaction:

```
Given there is a user account "alice@example.com" with the password "password"
When I go to the home page
And I fill in the login form with "alice@example.com" and "password"
And I click the login button
Then I should see "You are logged in as alice@example.com"
```

In the implementation phase of the test, you define what these steps do. In this case, they are defined to run Capybara methods:

```
Given /^there is a user account "(.*)" with the password "(.*)"$/ do \\
|email, password|
  User.create(:email => email, :password => password)
end

When "I go to the home page" do
  visit "/"
end

When /^I fill in the login form with "(.*)" and "(.*)"$/ do |email, password|
  fill_in 'Email', :with => email
  fill_in 'Password', :with => password
end

When "I click the login button" do
  click_button "Login"
end

Then /^I should see "(.*)"$/ do |text|
  page.should have_content(text)
end
```

Drivers

Capybara supports multiple drivers through a common API, each with benefits and drawbacks. We prefer to use either capybara-webkit or Selenium.

When possible, we use capybara-webkit. It's a fast, headless fake browser written using the WebKit browser engine. It's generally faster than Selenium and it's dependent on your system settings once compiled. This means that upgrading the browser you use every day won't ever affect your tests.

However, capybara-webkit is still young, and sometimes there's no substitute for having a real browser to run your tests through. In these situations, we fall back to using Selenium. Selenium will always support anything you can do in your actual browser, and supports multiple browsers, including Firefox, Chrome, Safari, and even Internet Explorer.

Capybara makes it easy to switch between drivers. Just set your default driver to capybara-webkit:

```
Capybara.javascript_driver = :webkit
```

Then, tag a Cucumber scenario as @javascript. If you need to fall back to using Selenium, tag that scenario with @selenium.

Isolated unit testing

Integration testing your application is great for ensuring that the product functions as intended, and works to mitigate against risk of regressions. There are additional benefits, though, to writing tests for individual units of your application in isolation, such as focused failures and decoupled code.

When an integration test fails, it can be difficult to pin down the exact reason why; particularly when a regression is introduced in a part of the application seemingly far away from where you're working. With the finer granularity of a unit test suite, failures are more targeted and help you get to the root of the problem more quickly.

Another benefit comes from unit testing when you test-drive code; i.e., when you write the tests before the implementation. Since you are starting with a piece of code which is client to your implementation modules, setup and dependency concerns are brought to your attention at the beginning of implementation, rather than much later during development when modules are integrated. Thinking about these concerns earlier helps you design modules which are more loosely coupled, have smaller interfaces, and are easier to set up. If code is hard to test, it will be hard to use. Writing the test first, you have a clear and concrete opportunity to make your implementation easier to use.

Finally, there are some behaviors that are difficult or impossible to test using a full-stack integration test. Here's a common example: you want to display a spinner graphic or disable a UI element while waiting for the server to respond to a request. You can't test this with an integration test because the time the server takes to respond is variable; by the time your test checks to look for the spinner graphic, the response will probably be finished. Even if it passes once, it may fail on the next run, or on the run after that. And if you decide to do an almost-full-stack test and fake out a slow response on the server, this will slow down your tests and introduce unnecessary indirection to an otherwise simple component. During isolation tests, it's easy to use techniques like dependency injection, stubbing, and mocking to test erratic behaviors and side effects that are difficult to observe during integration tests.

If you'd like to read more on test-driven development, check out Kent Beck's *Test Driven Development: By Example* and Gerard Meszaros' *xUnit Test Patterns: Refactoring Test Code*.

As there is plentiful content available for testing tools and strategies in Rails, we'll focus on isolation testing your Backbone code.

Isolation testing in JavaScript

There are many JavaScript testing frameworks available. Some run in-browser and provide facility for setting up DOM fixtures. Others are designed for standalone JavaScript code and can run on browserless JavaScript runtimes.

We'll use the Jasmine framework for writing our isolation specs. It integrates easily into a Rails application, and provides an RSpec-like syntax for writing specs:

```
describe("ExampleApp.Models.Tasks", function() {
  it("knows if it is complete", function() {
    var completeTask = new ExampleApp.Models.Task({ complete: true });
    expect(completeTask.isComplete()).toBe(true);
  });

  it("knows if it is not complete", function() {
    var incompleteTask = new ExampleApp.Models.Task({ complete: false });
    expect(incompleteTask.isComplete()).toBe(false);
  });
});
```

To run the Jasmine tests in the example application, simply run `bundle exec rake jasmine` and visit http://localhost:8888.

What to test?

We frequently found it difficult to test JavaScript components in isolation before we started using Backbone. Although jQuery really takes the pain out of working with the DOM and communicating with the server, it's not object-oriented and provides nothing to help split up your application. Because most of our HTML was in ERB-based templates, it was generally difficult to test the JavaScript that relied on that HTML without also loading the web application. This meant that almost all of our early JavaScript tests were full-stack integration tests.

Using Backbone, it's much easier to test components in isolation. View code is restricted to views, and templates contain only HTML or interpolation code that can be interpreted by the JavaScript view layer, such as jst or Mustache templates. Models and collections can be given data in their constructor, and simple dependency injection allows unit tests to fake out the remote server. We don't test routers in isolation as often because they're very light on logic,

but those are also easy to test by calling action methods directly or triggering events.

Since Backbone components are just as easy to test in isolation as they are to test full-stack, we generally use the same guidelines as we do for all Rails applications to decide what to test where.

Start with a top-down, full-stack Cucumber or RSpec scenario to describe the feature you're writing from a high-level perspective, and begin implementing behavior from the top as necessary. If you find that the feedback loop between a test failure and the code to pass it starts to feel too long, start writing isolated unit tests for the individual components you need to write to get closer to passing a higher-level assertion. As an example, an assertion from Capybara that fails because of a missing selector may need new models, controllers, views, and routes both on the server and in Backbone. Rather than writing several new components without seeing the failure message change, write a unit test for each piece as you progress down. If it's clear what component you need to add from the integration test failure, add that component without writing an isolated unit test. For example, a failure from a missing route or view file reveals an obvious next step, but missing text on a page, because a model method doesn't actually do anything, may motivate a unit test.

Many features will have edge cases or several logical branches. Anything that can't be described from a high-level, business value perspective should be tested from an isolated unit test. For example, when testing a form, it makes sense to write a scenario for the success path, where a user enters valid data that gets accepted and rendered by the application, and one extra scenario for the failure path, where a user enters invalid data that the system can't accept. However, when adding future validations or other reasons that a user's data can't be accepted, it makes sense to just write an extra isolated unit test, rather than adding a new scenario that largely duplicates the original failure scenario.

When writing isolation tests, the developer needs to decide exactly how much isolation to enforce. For example, when writing a unit test for a model, you'll likely decide not to involve an actual web server to provide data. However, when testing a view that composes other subviews, you'll likely allow the actual subview code to run. There are many cases when it will make sense to just write a unit test that involves a few components working together, rather than writing a full-stack scenario.

The overall goals when deciding how much to test via integration vs. isolation are to keep high-level business logic described in top-down tests, to keep details and edge cases described in unit tests, and to write tests that exercise the fewest number of components possible while remaining robust and descriptive without becoming brittle.

Helpful Tools

- Spy/stub/mock, even your HTTP, with Sinon.js
- If you're looking for factory_girl.js, it's called Rosie
- Use the Rails asset pipeline with the latest edge versions of the Jasmine gem
- See other examples on James Newbery's blog: testing Backbone with Jasmine and check out his examples on GitHub

Example: Test-driving a task application

Setup

In this example, we'll be using Cucumber, Capybara, RSpec, and Jasmine to test-drive a todo list.

The Selenium driver comes configured with Capybara and is the quickest driver to get running. By default, it runs your tests in a remote-controlled Firefox session, so you'll want to install Firefox if you don't have it.

If you'd like to test on a WebKit-based browser, you can set up the Selenium ChromeDriver to run integration tests against Chrome.

The other dependencies you can install by adding them to your Gemfile. The gems you'll need for testing are jasmine (currently tracking the edge version from GitHub), cucumber-rails, rspec-rails, and capybara. You'll want to add RSpec, Cucumber, and Jasmine to both the test and development groups so that you can run generators. With all our testing dependencies in place, the Gemfile in our sample application looks like this:

```
source 'http://rubygems.org'

gem 'rails', '3.2.6'
gem 'sqlite3', '~> 1.3.5'

gem 'rails-backbone', '~> 0.7.0'
gem 'jquery-rails'
gem 'ejs'
gem "flutie", "~> 1.3.2"
gem "clearance", "~> 0.13.0"
gem 'paperclip'
gem 'rabl'
gem 'backbone-support'
```

```
group :assets do
  gem "sass-rails", "~> 3.2.0"
  gem 'coffee-rails', "~> 3.2.0"
  gem 'uglifier'
end

group :development, :test do
  gem "rspec-rails", "~> 2.9.0"
  gem "ruby-debug19"
  gem 'jasmine', :git => "git://github.com/pivotal/jasmine-gem.git",
                 :ref => "34c1529c3f7"
  gem 'cucumber-rails', "~> 1.0.2", :require => false
end

group :test do
  gem 'turn', :require => false
  gem "capybara", "~> 1.1.1"
  gem 'selenium-webdriver', '~> 2.18.0'
  gem "factory_girl_rails", "~> 3.4.0"
  gem "bourne"
  gem "database_cleaner"
  gem "nokogiri"
  gem "shoulda-matchers"
  gem "launchy"
  gem "guard-spork"
  gem "spork", "~> 0.9.0.rc"
end
```

If you haven't already, bootstrap your application for Cucumber and Capybara:

```
rails generate cucumber:install
```

Next, bootstrap the application for Jasmine:

```
rails generate jasmine:install
```

With this configuration, you can run Cucumber scenarios with the Cucumber command and you can run Jasmine tests by running `bundle exec rake jasmine` and visiting http://localhost:8888, or by running `bundle exec rake jasmine:ci`, which uses Selenium to verify the Jasmine results.

One final helpful configuration change is to include the `jasmine:ci` task in the default rake task. This way, running `rake` will run all your specs, including Jasmine specs:

```
# Rakefile
# ...
task :default => ['spec', 'jasmine:ci', 'cucumber']
```

Step by step

We'll go outside in: Cucumber first, then RSpec or Jasmine as needed.

TIP: For an in-depth explanation of outside-in test-driven development, see *The RSpec Book*.

We'd like to be able to add items to a todo list. We know this will involve two parts: a list of existing tasks, and an interface for adding new items to the list. We'll start with the list of items, and create fixture data with Factory Girl Cucumber steps:

```
Feature: Viewing Tasks
  As a user
  So that I can see what I have to do
  I want to be able to see all my tasks

  Background:
    Given I am signed up as "email@example.com"
    When I sign in as "email@example.com"

  @javascript
  Scenario: View tasks
    Given the following tasks exist:
      | Title                              | user                      |
      | Purchase the Backbone on Rails ebook | email: email@example.com |
      | Master Backbone                    | email: email@example.com |
    And I am on the home page
    Then I should see "Master Backbone" within the tasks list
    And I should see "Purchase the Backbone on Rails ebook" within the tasks list
```

Running this, we see a failure:

```
Then I should see "Master Backbone" within the tasks list
  Unable to find css "#tasks table" (Capybara::ElementNotFound)
  (eval):2:in `find'
  ./features/step_definitions/web_steps.rb:29:in `with_scope'
  ./features/step_definitions/web_steps.rb:36:in `/^(.*) within (.*[^:])$/'
  features/view_tasks.feature:13:in `Then I should see "Master Backbone" \\
  within the tasks list'
```

A common mis-step when testing Rails apps with our structure is seeing false positives in bootstrapped data. Consider that, if we had just written the step `Then I should see "Master Backbone"` instead of scoping it with `within the tasks list`, then some test drivers would count the JSON that is used to bootstrap Backbone collections as visible text on the page, and the test would pass without us actually rendering the text to the page.

Since this we are doing outside-in development and testing for user interface, we will need to outline the UI first. To do this, first we'll need a page to host our code. Let's create and route a Rails `TasksController`. We'll bootstrap the Backbone app on `tasks#index`.

```
ExampleApp::Application.routes.draw do
  resources :tasks, :only => [:show, :create, :update, :index] do
    resources :attachments, :only => [:show, :create]
  end

  root :to => 'tasks#index'
end
```

```
class TasksController < ApplicationController
  before_filter :authorize
  respond_to :html, :json

  wrap_parameters :task, :include => [:assignments_attributes, :title, :complete]

  def index
    @tasks = tasks_visible_to_current_user
    @users = user_id_and_email_attributes
  end

  def show
    @task = current_user.tasks.find(params[:id])
  end

  def create
    respond_with(current_user.tasks.create(params[:task]))
  end

  def update
    task = current_user.tasks.find(params[:id])
    task.update_attributes(params[:task])
    respond_with(task)
  end

  private
```

```
  def user_id_and_email_attributes
    User.all.map { |user| { :id => user.id, :email => user.email } }
  end

  def tasks_visible_to_current_user
    (current_user.tasks + current_user.assigned_tasks).uniq
  end
end
```

To render our tasks, we'll want a TasksIndex Backbone view class. But before we write this class, we'll motivate it with a Jasmine isolation spec:

```
describe("ExampleApp.Views.TasksIndex", function() {
  it("renders a task table", function() {
    var view = new ExampleApp.Views.TasksIndex();
    view.render();

    expect(view.$el).toBe("#tasks");
    expect(view.$el).toContain("table");
  });
});
```

We use the jasmine-jquery library to provide DOM matchers for Jasmine like `toContain()`.

To run the Jasmine spec, run `bundle exec rake jasmine` and visit http://localhost:8888.

To make this test pass, we'll add a small template and make the `TasksIndex` view render it:

```
ExampleApp.Views.TasksIndex = Backbone.View.extend({
  tagName: 'div',
  id: 'tasks',

  initialize: function() {
  },

  render: function () {
    this.$el.html(JST['tasks/index']({}));
    return this;
  }
});
```

The `app/assets/templates/tasks/index.jst.ejs` template:

```
<table></table>
```

Now our Jasmine specs pass:

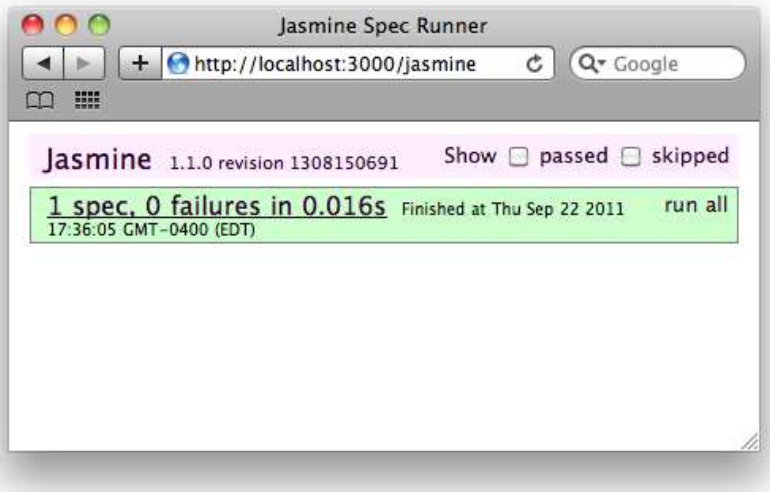

Figure 7.1: Passing Jasmine spec

Since the Jasmine specs pass, we'll pop back up a level and run the Cucumber story. Running it again, the failure is slightly different. The `"#tasks table"` element is present on the page, but doesn't contain the content we want:

```
@javascript
Scenario: View tasks
  Given the following tasks exist:
    | Title                                |
    | Purchase the Backbone on Rails ebook |
    | Master Backbone                      |
  And I am on the home page
  Then I should see "Master Backbone" within the tasks list
    expected there to be content "Master Backbone" in "Title Completed" \\
(RSpec::Expectations::ExpectationNotMetError)
    ./features/step_definitions/web_steps.rb:107:in \\
‘/^(?:|I )should see "([^"]*)"$/’
    features/view_tasks.feature:13:in ‘Then I should see "Master Backbone" \\
within the tasks list’
```

Drop back down to Jasmine and write a spec motivating the `TasksIndex` view to accept a collection and render it. We'll rewrite our existing spec, since we are changing the `TasksIndex` interface to require that a collection be passed in:

```
//= require application

describe("ExampleApp.Views.TasksIndex", function() {
  it("renders a collection of tasks", function() {
    var tasksCollection = new ExampleApp.Collections.Tasks();
    tasksCollection.reset([
      { title: "Wake up" },
      { title: "Brush your teeth" }
    ]);

    var view = new ExampleApp.Views.TasksIndex({collection: tasksCollection});
    var $el = $(view.render().el);

    expect($el).toHaveText(/Wake up/);
    expect($el).toHaveText(/Brush your teeth/);
  });
});
```

This spec fails:

```
1 spec, 1 failure in 0.008s
Finished at Thu Sep 22 2011 18:10:26 GMT-0400 (EDT)
ExampleApp.Views.TasksIndex
renders a collection of tasks
TypeError: undefined is not a function
TypeError: undefined is not a function
    at [object Object].<anonymous> \\
(http://localhost:8888/assets/views/tasks_index_spec.js?body=1:4:27)
```

It's failing because we haven't defined `ExampleApp.Collections.Tasks` yet. We need to define a task model and tasks collection. We'll define the model:

```
ExampleApp.Models.Task = Backbone.Model.extend({
  initialize: function() {
    this.on("change:attachments", this.parseAttachments);
    this.parseAttachments();

    this.on("change:assigned_users", this.parseAssignedUsers);
    this.parseAssignedUsers();
  },
```

```
  parseAttachments: function() {
    var attachmentsAttr = this.get('attachments');
    this.attachments = new ExampleApp.Collections.Attachments(attachmentsAttr);
  },

  parseAssignedUsers: function() {
    var usersAttr = this.get('assigned_users');
    this.assignedUsers = new ExampleApp.Collections.Users(usersAttr);
  },

  urlRoot: '/tasks',

  isComplete: function() {
    return this.get('complete');
  },

  toJSON: function() {
    var json = _.clone(this.attributes);
    json.assignments_attributes = this.assignedUsers.map(function(user) {
      return { user_id: user.id };
    });
    return json;
  }
});
```

. . . write a test to motivate the collection:

```
describe("ExampleApp.Collections.Tasks", function() {
  it("contains instances of ExampleApp.Models.Task", function() {
    var collection = new ExampleApp.Collections.Tasks();
    expect(collection.model).toEqual(ExampleApp.Models.Task);
  });

  it("is persisted at /tasks", function() {
    var collection = new ExampleApp.Collections.Tasks();
    expect(collection.url).toEqual("/tasks");
  });
});
```

. . . and pass the test by implementing the collection:

```
ExampleApp.Collections.Tasks = Backbone.Collection.extend({
  model: ExampleApp.Models.Task,
  url: '/tasks'
});
```

Running the Jasmine specs again, we're making progress. The `TasksIndex` view is accepting a collection of tasks, and now we have to render it:

```
Expected '<div id="tasks"><table> <tbody><tr> <th>Title</th> \\
<th>Completed</th> </tr> </tbody><div></div><div></div></table> </div>' to \\
have text 'Wake up'.
```

The simplest thing we can do to get the spec passing is to pass the `tasks` collection into the template, and iterate over it there:

```
ExampleApp.Views.TasksIndex = Support.CompositeView.extend({
  initialize: function() {
    _.bindAll(this, "render");
    this.collection.on("add", this.render);
  },

  render: function () {
    this.renderTemplate();
    this.renderTasks();
    return this;
  },

  renderTemplate: function() {
    this.$el.html(JST['tasks/index']({ tasks: this.collection }));
  },

  renderTasks: function() {
    var self = this;
    this.collection.each(function(task) {
      var row = new ExampleApp.Views.TaskItem({ model: task });
      self.renderChild(row);
      self.$('tbody').append(row.el);
    });
  }
});
```

```
<table id="tasks-list">
  <thead>
    <tr>
      <th>Title</th>
      <th>Assignees</th>
      <th>Completed</th>
    </tr>
  </thead>
  <tbody>
```

```
  </tbody>
</table>

<a class="create" href="#new">Add task</a>
```

Now, Jasmine passes, but the Cucumber story is still failing:

```
Then I should see "Master Backbone" within the tasks list
Unable to find css "#tasks table" (Capybara::ElementNotFound)
```

This is because the Jasmine spec is an isolation spec, and verifies that the `TasksIndex` view works in isolation. There is additional code we need to write to hook up the data in the Rails test database to the Backbone view. Adding this code to bootstrap the Backbone application should wrap up our exercise and get the tests passing.

We'll motivate writing a top-level Backbone application object with a spec. Note the use of a `sinon.spy` for verifying the router instantiation:

`spec/javascripts/example_app_spec.js` “‘ describe(“ExampleApp”, function(){ it(“has a namespace for Models”, function() { expect(ExampleApp.Models).toBeTruthy(); });

it(“has a namespace for Collections”, function() { expect(ExampleApp.Collections).toBeTruthy(); });

it(“has a namespace for Views”, function() { expect(ExampleApp.Views).toBeTruthy(); });

it(“has a namespace for Routers”, function() { expect(ExampleApp.Routers).toBeTruthy(); });

describe(“initialize()”, function() { it(“accepts data JSON and instantiates a collection from it”, function() { var data = { “tasks”: [{“title”:“thing to do”}, {“title”:“another thing”}], “users”: [{“id”:“1”,“email”:“alice@example.com”}] }; ExampleApp.initialize(data);

```
  expect(ExampleApp.tasks).not.toEqual(undefined);
  expect(ExampleApp.tasks.length).toEqual(2);
  expect(ExampleApp.tasks.models[0].get('title')).toEqual("thing to do");
  expect(ExampleApp.tasks.models[1].get('title')).toEqual("another thing");

  expect(ExampleApp.users.length).toEqual(1);
});

it("instantiates a Tasks router", function() {
  sinon.spy(ExampleApp.Routers, 'Tasks');
  ExampleApp.initialize({});
```

```
  expect(ExampleApp.Routers.Tasks).toHaveBeenCalled();
  ExampleApp.Routers.Tasks.restore();
});

it("starts Backbone.history", function() {
  Backbone.history.started = null;
  Backbone.history.stop();
  sinon.spy(Backbone.history, 'start');
  ExampleApp.initialize({});

  expect(Backbone.history.start).toHaveBeenCalled();

  Backbone.history.start.restore();
});
```

}); }); ""

Get it to green:

```
window.ExampleApp = {
  Models: {},
  Collections: {},
  Views: {},
  Routers: {},
  initialize: function(data) {
    this.tasks = new ExampleApp.Collections.Tasks(data.tasks);
    this.users = new ExampleApp.Collections.Users(data.users);

    new ExampleApp.Routers.Tasks({ collection: this.tasks, users: this.users });
    if (!Backbone.history.started) {
      Backbone.history.start();
      Backbone.history.started = true;
    }
  }
};
```

Then we bootstrap the app from the Rails view:

```
<h1>Tasks</h1>

<div id="tasks">
</div>

<script type="text/json" id="bootstrap">
  {
```

```
    "tasks": <%= @tasks.to_json(include:
                                { assigned_users: { only: [:id, :email] } }) %>,
    "users": <%= @users.to_json %>
  }
</script>

<%= content_for :javascript do -%>
  <script type="text/javascript">
    $(function () {
      var div = $('<div></div>');
      div.html($('#bootstrap').text());
      var data = JSON.parse(div.text());

      ExampleApp.initialize(data);
    });
  </script>
<% end %>
```

And the integration test passes!

```
Feature: Viewing Tasks
  As a user
  So that I can see what I have to do
  I want to be able to see all my tasks

  @javascript
  Scenario: View tasks
    Given the following tasks exist:
      | Title                              |
      | Purchase the Backbone on Rails ebook |
      | Master Backbone                    |
    And I am on the home page
    Then I should see "Master Backbone" within the tasks list
    And I should see "Purchase the Backbone on Rails ebook" within the tasks list

1 scenario (1 passed)
5 steps (5 passed)
```

Chapter 8

Security

Encoding data when bootstrapping JSON data

As it turns out, bootstrapping JSON data in your ERB templates can introduce a security vulnerability. Consider the case when a user enters a malicious `<script>` as the title of a task. When the `tasks#index` page is reloaded, and we naively bootstrap task data on the page, the browser will interpret and execute the script. Since it's possible for this script to run on another user's session, it can be quite damaging if it goes on to, for example, edit or destroy the user's data.

To protect against this, we make use of the fact that on HTML5 documents, script tags that do not have a type of `text/javascript` won't be automatically evaluated by the browser. Therefore, we can create an element with the HTML-encoded bootstrapped data enclosed in a script of type `text/json`, fetch it using a simple jquery selector, and parse it ourselves.

Here's an example:

```
<script type="text/json" id="bootstrap">
  { "tasks": <%= @tasks.to_json %> }
</script>

<script type="text/javascript">
  $(function () {
    var div, data;

    div = $('<div></div>');
    div.html($('#bootstrap').text());
```

```
    data = JSON.parse(div.text());

    ExampleApp.initialize(data);
  });
</script>
```

A reliable way to unencode the HTML-encoded JSON string is to use the browser's native functionality by setting an element's `innerHTML`. So in the above script, we create a `json_div` var, assign its `innerHTML` to the bootstrap script's text, and retrieve the innerText back out, unencoded. The final result is the `data` variable containing proper JSON with no HTML escaping that can be parsed and passed along to your app's `initialize` function.

This approach can be seen on the example app on the `app/views/tasks/index.html.erb` template.

Now, the application's models, populated by the bootstrap data structure, contain raw data that is not HTML-escaped. When you render this data into the DOM, make sure you escape the HTML at that point:

```
// From app/assets/javascripts/views/task_item.js:

this.$('label').html(this.model.escape('title')); // not model.get
```