Backbone.js on Rails

Table of Contents

1. Introduction

Modern web applications are 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 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.

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

A third common feature in rich client-side apps is presenting 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.

Often this 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.

Perhaps you've seen code like this:

```
TODO: Contact app example, $.ajax nested a few layers deep, updating hidden DOM
elements or a global object e.g. "window.contactsJSON" as persistence, then
cascading update to several views
```

What if it could look like this instead:

```
TODO: Backbone refactoring of above example.
```

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 your application can manage its complexity while its feature set grows.

It's important to note that Backbone is a library, not a framework. Though the distinction may seem subtle, it's largely one of intent and purpose. 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 - conventions for rich client-side applications aren’t as well set-down and individual use cases vary more widely. Instead of trying to serve as "the one way" etc (TODO: elaborate) Backbone provides a set of structures that help you organize your application by building your own framework with its own set of conventions.

1.1. Why use Backbone.js

Web applications are pushing an increasing amount of behavior to the client. The user experience can be quite a pleasure, but deeply nesting callbacks and relying on the DOM for app state aren’t. There is a host of new JavaScript client-side frameworks blossoming, and you have no shortage of choice.

From "least similar to Backbone" to "most similar to Backbone", here are a few of the options:

Are you building a desktop-like application? Would you benefit from a rich library of existing UI controls? Check out Cappuccino or SproutCore.

Are you very comfortable with the model-view-view model (MVVM) pattern, perhaps from Microsoft WCF or Silverlight? Take a look at Knockout.js, which has very robust object graph dependency tracking and declarative bindings between markup and view models.

Do you want a soup-to-nuts client-side framework, with a jQuery feel (and dependency), with generators, dependency management, builds, testing, and more? JavaScriptMVC provides all of this, with an MVC core that supports observables and data transports like JSON over REST. You can pick and choose a subset of functionality.

Server synchronization and data validation play a central role in structuring your application, and an opinion on it is one of the central design choices of Spine.js. Does the client generally take precedence, handling all its own validations, immediately returning to the user, and updating the server asynchronously? Or do you have significant server-side processing and validation? Spine.js strongly favors a client-centric approach, with a decoupled server. There are a few other API differences, but in other respects Spine is very similar to Backbone.

Backbone favors a pared-down and flexible approach. There is very little in the way of inheritance or class library, and the code you write ends up feeling very much like JavaScript. It does not prescribe much in the way of favoring a client over server, or a particular server syncronization approach. Although this means that you may need to write some of your own conventions, Backbone is built with that in mind: the source is small, very well annotated, and modularly designed so that it is easy to change. It is small and flexible enough to make it pleasant to introduce into an existing application, but provides enough convention and structure to help you organize your JavaScript.

1.2. The Example Application

Rails 3.1.0.rc5

Ruby 1.9.2

Backbone.js and Underscore.js are the non-minified versions. This is for informational purposes, but also because the Rails 3.1 asset pipeline will compress and minify them.

While Rails 3.1 defaults to CoffeeScript, we have decided to make all of the example code normal Javascript as we believe that will be the most understandable to the current readers.

2. Getting up to speed

2.1. Backbone.js online resources

This book is not an introduction, and assumes you have some knowledge of Javascript and of Backbone.js. Luckily, 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.js:

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

2.2. JavaScript online resources and books

I cannot recommend *JavaScript: The Good Parts* by Douglas Crockford highly enough. It’s concise, readable, and will make you a better JavaScript programmer.

http://www.amazon.com/exec/obidos/ASIN/0596517742/

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

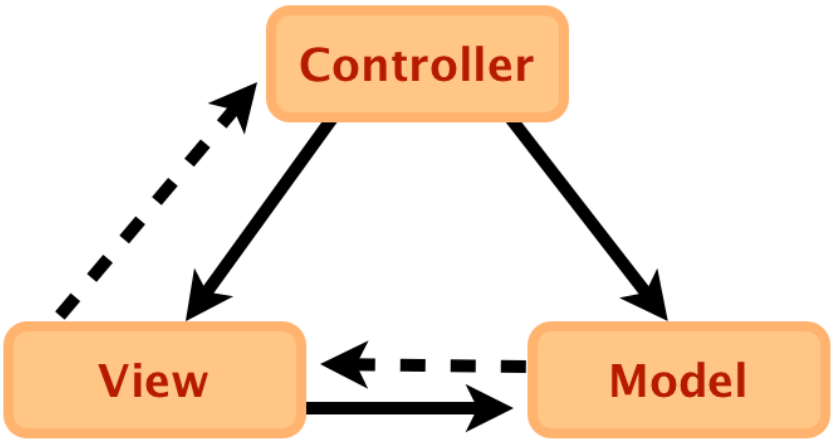
http://tddjs.com/

3. Organization

3.1. Backbone.js 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 1. Model-view-controller concept



In the above diagram a solid line represents a direct association and a dashed line represents an indirect association (for example, via 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 give you. However, Rails itself is not doing "traditional" MVC. A traditional MVC is event-based. This means that the views trigger events which the controller figures out what to do with. It can be argued that the requests generated by the browser are the "events" in Rails; however, due to the single-threaded, request-response nature of the web, the control flow between the different levels of MVC is much more straightforward.

Given that Javascript has events, and that much of the interactions between the different components of Backbone.js in the browser are not limited to request/response, programming with Backbone.js is in a lot of ways more like working with a traditional MVC architecture.

That said, technically speaking, Backbone.js is *not* MVC, and the creators of Backbone.js acknowledged this when they renamed Controllers to Routers in version 0.5.0.

What is Backbone.js then, if not MVC? Technically speaking, it's just the Models and the Views with a Router to handle flow between them. In Backbone.js the views and routers could handle many of the aspects that controllers would typically handle, such as actually figuring out what to do next and what to render.

On the other hand, it is possible to implement a Controller on top of the components provided by Backbone in order to listen to and handle events the same way traditional MVC frameworks work. One pragmatic approach is to realize the great organization that Backbone.js gives you is useful in and on itself, but keep in mind that complex applications may benefit from the clearer separation provided by a controller.

3.2. What Goes Where

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

The models in Backbone.js and Rails are analogous. Backbone.js collections are just ordered sets of models. Because it lacks controllers, Backbone.js 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. In Backbone.js, however, you have a separation between the view and templates.

Once you introduce Backbone.js into your stack, 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 everything going on in your application straight. Ultimately, the additional organization and functionality of Backbone.js outweighs the costs, so let's break it down.

Rails

- Model
- Controller
- View

Backbone.js

- Model and Collection
- Router
- View
- Template

In a typical Rails and Backbone.js 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.js initialization**, usually this is populating some **Backbone collections** as sets of **Backbone models** with JSON data provided by the **Rails view**
- The **Backbone.js router** determines which of its methods should handle the display based on the URL
- The **Backbone.js router** calls that method, some initial processing may be performed, and one or more **Backbone.js views** are rendered
- The **Backbone.js view** reads **templates** and uses **Backbone.js** models to render itself onto the page

At this point, the user will see a nice page 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.js model**, **Backbone.js views**, **Backbone.js router**, or requests to the remote server.

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

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

Which of the above remote server interactions you use will depend upon the desired result, and the type of user interface. This book should help you understand which interaction you'll want to choose for each portion of your application.

3.3. Namespacing your application

You will want to create an object in Javascript for your Backbone.js application to reside. This variable will serve as a namespace for your Backbone.js 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 didn't namespace your Task model then this would conflict.

This variable includes a place to hold Models, Collections, Views, and Routes, and an init function which will be called to initialize the application. It's very common to create a new Router in the init function, and Backbone.history.start() must be called in order to route the initial URL. This app variable will look like the following.

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

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

3.4. Mixins

Backbone provides a basic mechanism for inheritance. Often 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 [http://en.wikipedia.org/wiki/Mixin].

Backbone includes Backbone.Events [http://documentcloud.github.com/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's `_.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);
```

4. Rails Integration

4.1. Organizing your Backbone.js code in a Rails app

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

4.2. Rails 3.0 and prior

With Rails 3.0 and prior, store your Backbone.js 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 })
```

Note

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.

When using Jammit, there is a slightly sticky issue as an app grows from one template subdirectory to multiple template subdirectories.

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 coliding 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, 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

4.3. Rails 3.1

Rails 3.1 introduces the asset pipeline:

http://guides.rubyonrails.org/asset_pipeline.html

which uses the Sprockets library for preprocessing and packaging assets:

http://getsprockets.org/

To take advantage of the built-in asset pipeline, organize your Backbone.js 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, 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.

Note

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 .
```

Additionally, load order for Backbone.js and your Backbone.js app is very important. jQuery and Underscore.js must be loaded before Backbone.js, 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 underscore
//= require backbone
//= require backbone.authtokenadapter
//= 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`.

4.4. An Overview of the Stack: Connecting Rails and Backbone.js

By default Backbone.js communicates with your Rails application via JSON GET , POST and PUT requests. If you've ever made a JSON API for your Rails app, then for the most part this will be very similar.

If you've never made a JSON API for your Rails application before, luckily you, it's pretty straightforward.

4.4.1. Setting Up Rails Models

One important aspect to keep in mind as you plan out how your Backbone.js interface will behave, and how it will use your Rails back-end, is that there is no need to have a one-to-one mapping between your Rails models and your Backbone.js models.

The smaller an application is, the more likely that there will be a one-to-one mapping between both Backbone.js and Rails models and controllers.

However, if you have a sufficiently complex application, it's more likely that you *won't* have a one-to-one mapping due to the differences in the tools Backbone.js gives you and the fact that you're building a user-interface, not a back-end. Some of the reasons why you won't have a one to one mapping include:

- Because you're building a user interface, not a back-end, it's likely that some of your backbone models will aggregate information from multiple Rails models into one Backbone.js model.
- This Backbone.js model may or may not be named the same as one of your Rails models.
- Backbone.js gives you a new type of object not present in Rails: Collections.

- Backbone.js doesn't have the concept of relationships out of the box.

With that said, lets take the simple case first and look at how you might make a Backbone.js version of a Rails model.

In our example application, we have a Task model. The simplest Backbone.js representation of this model would be as shown below.

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

The urlRoot property above indicates to Backbone.js that the server url for instances of this model will be found at `/tasks/:id`.

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

The simplest Backbone.js 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 models 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. Like ActiveRecord, Backbone.js models get their attributes from the schema and data given to them. In the case of Backbone.js, this schema and data are the JSON from the server.

The default JSON representation of an ActiveRecord model is a Hash 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 related models 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. You customize the output of `to_json` by overriding the `as_json` method in your model. We'll touch on this more later in the section "Customizing your Rails-generated JSON."

4.4.2. Setting Up Rails Controllers

The Backbone models and collections will talk to your Rails controllers. While your models may not have a one-to-one mapping with their Rails counterparts, it is likely that you'll have at least one controller corresponding to every Backbone.js model.

Fortunately for us, Backbone.js models will 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.js does not make any use of the new action.

Therefore, it's just up to us to write a *normal* restful controller.

There are a few different ways you can write your controllers for interacting with your Backbone.js models and collections. However, the newest and cleanest way is to use the respond_with method introduced in Rails 3.0.

When using respond_with, in your controller you specify what formats are supported with the method respond_to. In your individual actions, you then specify the resource or resources to be delivered using respond_with, as shown in the example Tasks controller and index action below.

```
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 both the HTML and JSON formats. Then, in the index action, the respond_with call will perform the appropriate action for the requested format.

The above controller is equivalent to the following one, using the older respond_to method.

```
class TasksController < ApplicationController::Base
  def index
    @tasks = Task.all
    respond_to do |format|
      format.html
      format.json { render :json => @tasks }
    end
  end
end
```

Using respond_with you can create succinct controllers that respond with a normal web page, but also expose a JSON API that Backbone.js will use.

4.4.2.1. Validations and your HTTP API

If a Backbone.js model has a `validate` method defined, it will be validated 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 can 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.

In practice, much of the validation logic for your models will continue to be handled on the server, as fully implementing validations on the client side would often require duplicating a lot of server-side

business logic. Furthermore, you always want the server to be the definitive arbitrer for valid data, as this API can be abused outside of the backbone.js interface.

TODO: Is it possible to smoothly integrate Backbone.js and the client_side_validations gem?

Instead, your Backbone.js applications will likely rely on server-side validation logic. How to handle a failure scenario is passed in to Backbone.js model save call as a callback, as shown below.

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

The error callback will be triggered if your server returns a non-200 response. Therefore, you'll want your controller to return a non-200 HTTP response code if validations fail.

A controller that does this would be 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
```

Thankfully, the default Rails responders will respond with an unprocessable entity (422) status code when there are validation errors, so the above controller action can be rewritten more succintly like so:

```
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. For more information about handling and displaying errors on the client side, see the Form helpers section of the Views and Templates chapter.

4.4.3. Setting Up Views

Most Backbone.js applications will be a "single-page app". This means that your Rails application will render a single-page which properly sets up Backbone.js and the data it will use. From there, ongoing interaction with your Rails application occurs via the JSON APIs.

The most common page for this single-page application will be the index action of a controller, as in our example application and the tasks controller.

You will want to create an object in Javascript for your Backbone.js application to reside. For more information on this namespacing see the "Namespacing your application" section of the Organization chapter.

This namespace variable holds your Backbone.js application's Models, Collections, Views, and Routes, and has an init method which will be called to initialize the application.

This namespace variable will look like the following.

```
var ExampleApp = {
  Models: {},
  Collections: {},
  Views: {},
  Routers: {},
  init: function() {
    new ExampleApp.Routers.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.js 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. This will appear as follows.

```
<%= content_for :javascript do -%>
  <%= javascript_tag do %>
    ExampleApp.init();
  <% end %>
<% end -%>
```

For performance reasons, you will almost always "bootstrap" and give Backbone.js its initial data within the HTML view for this page. In our example, the tasks have already been provided to the view in a `@tasks` instance variable, and that can be used to bootstrap, as shown below.

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

The above example uses Erb to pass the JSON for the tasks to the init method.

Once you make this change, the ExampleApp.init method then becomes:

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

Finally, you must have a Router in place which knows what to do. We'll cover routers in more detail in the Routers, Views and Templates chapter. For a more in-depth presentation on writing and using routers please go there. However, routers are an important part of the infrastructure you need to start using Backbone.js and we can't make our example here work without them.

Backbone.js routers provide methods for routing application flow based on client-side URL fragments (#fragment).

```
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.js 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.
  }
});
```

A 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.

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

4.5. Customizing your Rails-generated JSON

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

First, it's likely that you'll want to switch from including all attributes (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, to switch to everything being explicitly included in 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 probably suspect, you can then customize the JSON for the comments by overriding the `as_json` method on the `Comment` model.

While these are the most common `as_json` options you'll use when working with Backbone.js, 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

4.5.1. ActiveRecord::Base.include_root_in_json

Depending on the versions, Backbone.js 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. In 3.1, it defaults to false. This reversal was made to simplify the JSON returned by default in Rails application, but it is 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 and you already have a published API; you may need to expose a new version of your API.

From the Backbone.js 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 the 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",

  // ...
});
```

4.6. Converting an existing page/view area to use Backbone.js

We'll cover Backbone.js Views and Templates in more detail in the Routers, Views, and Templates chapter, but this section is meant to get you started understanding how Backbone.js views work by illustrating the conversion of a Rails view to a Backbone.js view.

Its important to note that a Rails view is not directly analogous to a Backbone.js view. A Rails view is more like a Backbone.js template, and Backbone.js views are more like Rails controllers. This can cause confusion with developers just started with Backbone.js.

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>
```

Assuming we have the Backbone.js Task model and collection and the Rails Task model and controller discussed above, and we're priming the pump with all the tasks, before we can convert the template we must create a Backbone.js view which will render the Backbone.js template.

A Backbone.js 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. For more detailed coverage of Backbone.js views, see the Routers, Views, and Templates chapter.

The most rudimentary view we could introduce at this point would be one that merely renders the above page markup, looping over each task in the Tasks collection. While this would be insufficient for most actual applications, in order to illustrate the building blocks of a Backbone.js view, such a view would be like the one shown below.

```
ExampleApp.Views.TasksIndex = Backbone.View.extend({
  initialize: function() {
  },

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

The Backbone.js view above has an initialize method which will be called when the view is instantiated.

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

Each Backbone.js 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.

Finally, the Router must be changed to instantiate this view, passing in the collection for it to render, render the view, and insert its markup 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.js view in place that renders the template, and its being called by the router, we can focus on converting the above Rails view to a Backbone.js template.

Backbone.js depends on Underscore.js which provides templating. Fortunately, the delimiter and basic concepts used for both Underscore.js and Erb are the same, making conversion relatively painless. For this reason, we recommend using Underscore.js templates when converting a larger, existing Rails application to Backbone.js.

The tasks index 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. For example, each, map, and reject. Fortunately, Backbone.js also proxies to Underscore.js to provide 26 iteration functions on Backbone.Collection. This means that its possible to call the Underscore.js methods directly on Backbone.js collections.

So we'll use the each method to iterate through the Tasks collection that was passed to the view, as shown in the converted Rails template, which is now an 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>
```

As you can see above in the above example, the same delimiter, and the use of the each method make the conversion of the Rails view to an Underscore.js template straightforward.

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

4.6.1. Breaking out the TaskView

As mentioned above, this simple conversion of the index which merely loops over each of the tasks is not one you'd likely see in a real Backbone.js application.

Backbone.js views should represent the logic pieces of your web page. In the above example, we have an index view, which is a logic piece, but then it is made up of the display of individual tasks. Each of those individual tasks should be represented by a new Backbone.js view, named TaskView.

The benefit of this logical separation is covered in more detail in the Views section, but know that one of the major features of Backbone.js is event binding. With each of the Task models represented by an individual task view, when that individual model changes the view can be re-rendered automatically (by triggering events) and the entire page doesn't need to be re-rendered.

Continuing our task index example from above, a TaskView will be responsible for rendering just the individual table row for a Task, therefore, its template will appear as follows.

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

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

```
<h1>Tasks</h1>

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

</table>
```

As you can see above in the index template, the individual tasks are no longer iterated over and rendered inside the table. This will now happen in the TasksIndex and TaskView view, which is shown below.

```
ExampleApp.Views.TaskView = Backbone.View.extend({
  initialize: function() {
  },

  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 `TaskView`s into a list is left to the parent view object:

```
ExampleApp.Views.TasksIndex = Backbone.View.extend({
  initialize: function() {
  },

  render: function () {
    this.$el.html(JST['tasks/index']({ tasks: this.collection }));

    var self = this;
    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 parent element.

If you take a 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, and that is 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 produced invalid markup.

Fortunately, Backbone.js 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 be 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>
```

That is the basic building blocks of converting Rails views to Backbone.js and getting a functional system. The majority of Backbone.js 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.

4.7. Automatically using the Rails authentication token

When using Backbone.js in a Rails app, you will run into a conflict with the Rails built in Cross Site Scripting (XSS) protection.

When Rails XSS is enabled, each POST or PUT request to Rails should include a special token which is verified to ensure that the request originated from a user which is actually using the Rails app. In recent versions of Rails, Backbone.js Ajax requests are no exception.

To get around this, you have two options. Disable Rails XSS protection (not recommended), or make Backbone.js play nicely with Rails XSS.

To make Backbone.js play nicely with Rails XSS you can monkeypatch Backbone.js to include the Rails XSS token in any requests it makes.

The following is one such script:

```
//
// With additions by Maciej Adwent http://github.com/Maciek416
// If token name and value are not supplied, this code Requires jQuery
//
// Adapted from:
// http://www.ngauthier.com/2011/02/backbone-and-rails-forgery-protection.html
// Nick Gauthier @ngauthier
//

var BackboneRailsAuthTokenAdapter = {

  //
  // Given an instance of Backbone, route its sync() function so that
  // it executes through this one first, which mixes in the CSRF
  // authenticity token that Rails 3 needs to protect requests from
  // forgery. Optionally, the token's name and value can be supplied
  // by the caller.
  //
  fixSync: function(Backbone, paramName /*optional*/, paramValue /*optional*/){

    if(typeof(paramName)=='string' && typeof(paramValue)=='string'){
      // Use paramName and paramValue as supplied
    } else {
      // Assume we've rendered meta tags with erb
      paramName = $("meta[name='csrf-param']").attr('content');
      paramValue = $("meta[name='csrf-token']").attr('content');
    }

    // alias away the sync method
    Backbone._sync = Backbone.sync;

    // define a new sync method
    Backbone.sync = function(method, model, success, error) {

      // only need a token for non-get requests
      if (method == 'create' || method == 'update' || method == 'delete') {

        // grab the token from the meta tag rails embeds
        var auth_options = {};
        auth_options[paramName] = paramValue;

        // set it as a model attribute without triggering events
        model.set(auth_options, {silent: true});
      }

      // proxy the call to the old sync method
      return Backbone._sync(method, model, success, error);
    };
  },


  // change Backbone's sync function back to the original one
  restoreSync: function(Backbone){
    Backbone.sync = Backbone._sync;
  }
};

BackboneRailsAuthTokenAdapter.fixSync(Backbone);
```

The above patch depends on jQuery, and should be included after jQuery and Backbone.js are loaded. Using Jammit, you'd list it below the backbone.js file.

In Rails 3.1, you'll place this file in lib/assets/javascripts. In the example app, you can find this this in example_app/lib/assets/javascripts/backbone.authtokenadapter.js.

5. Routers, Views, and Templates

5.1. View explanation

A Backbone.js 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.

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

A basic Backbone.js 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
  }
};
```

5.1.1. Initialization

Backbone.js 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 to 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 with `new ExampleView({ model: Task })` then inside of the view the model you passed in as an option will be available in `this.model`.

5.1.2. The View's Element

Each Backbone.js 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, inserting 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.$el`. This is useful so you don't need to repeatedly call `$(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);
```

5.1.3. 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 time. Since `id` is likely to be individual to each model, its most likely to pass that in as an option rather than declaring it 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"` on the view will result in view's element with that additional class like `<div class="example">`.

5.1.4. 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, any other actions to modify the view may be taken.

Typical functionality in `render` in addition to rendering a template would be to add additional classes or attributes to `this.el` or fire or bind other events.

Backbone.js, 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.

5.1.5. 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.

Events bound automatically with the `events` hash, the DOM events are bound with the `$.delegate()` function. Backbone.js 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.

5.2. 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> Charp chedar </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 templates, let's start there.

An underscore template may look like so:

```
<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 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 on earlier chapters, it's 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's 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 difference with underscore 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 alows you to write what they call 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 so 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 javascript context is the task itself, so you access it's 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 writing logic in pure javascript using for example `if`, Mustache provides a set of tags that take on different meanings. They can render values or not render anything at all.

Both handlebars and mustache HTML escape rendered values by default.

You can learn more about handlebars at the project's home on the web [http://www.handlebarsjs.com/], and mustache at the project's man page [http://mustache.github.com/mustache.5.html] and javascript implementation [https://github.com/janl/mustache.js:]

5.3. Choosing a strategy

Like any technology choice, there are tradeoffs to evaluate and external forces to consider when choosing a templating approach.

The scenarios we've encountered usually involve weighing these questions: do I already have server-side templates written that I'd like to "Backbone-ify," or am I writing new Backbone functionality from scratch?

TODO: flesh this out a bit, or cut the "Choosing a strategy" section entirely

5.3.1. When you are 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 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 chose the templating language that your entire team is most comfortable with. Try to keep the cost of rewriting templates as low as possible, if that's the case. Also make sure that the entire team is comfortable with the chosen approach, including designers who will be working in that area of the app as well.

5.3.2. When you are 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').text(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. In turn, a major change in the markup may break the rendering because the selector's used to replace parts of the DOM may no longer find the same elements, or may not find any elements at all.

5.4. Routers

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

> **Note**
>
> Backbone.js now includes support for pushState, which can use real, full URLs instead of url fragments for routing.

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

pushState support is current limited to the latest versions of Firefox, Chrome, and 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.js, 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 URL used are real URLs, your server must know how to render each of the URLs. For example, if your Backbone.js 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.js 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 all the reasons and complications above, the examples in this book all currently use URL fragments and not pushState.

A typical Backbone.js router will appear as shown below.

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

  index: function() {
    // Render the index view
  }

  show: function(id) {
    // Render the show view
  }
});
```

5.4.1. 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.js 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 match through route, Backbone.js 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 clicking on a link, entering a URL into the browser's URL bar, or clicking the back button. In all of those cases, Backbone.js will look to see if the new URL matches an existing route. If it does, the specified function will be called with any parameters.

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

5.4.2. 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 Routes constructor will be passed to this `initialize` function.

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

5.5. 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.js you accomplish this with events.

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

With Backbone.js, 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, they notify the game, which in turn invokes any event handlers that are bound to that event.

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

gameEngine.bind("user_registered", function(user) {
  user.points += 10
});

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

`Backbone.Events` is included on `Backbone.Views`, `Backbone.Model` and `Backbone.Collection`. Backbone itself will trigger events for you at certain points. The flow of a user interface will usually react to certain well known events, and therefore they are so common that it just makes sense for it to do so. For example, when a model's attributes are changed, Backbone models conveniently trigger the `change` event. It is 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 extendes `Backbone.Events`. Additionally, if an event handler should always trigger regardless of which event got 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
- Backbone events triggered by the view's model or collection
- Custom view events

TODO: This three-point breakdown is the wrong way to slice this. Instead of "DOM, model/collection, custom" it should be "DOM, events I observe, events I publish". Events that your view observes need to be cleaned up upon disposing the view, regardless of where those events are triggered (models, collections, or other views, or other arbitrary objects). Events that your view publishes need to be handled in a different way.

TODO: Consider promoting events and binding/unbinding to its own top-level section; this isn't view-specific, although the view layer is where you'll be doing most of your binding.

5.5.1. 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() [http://documentcloud.github.com/backbone/#View-delegateEvents] function. It binds DOM events with `$.delegate()`, whether you're using the jQuery [http://api.jquery.com/delegate/] or Zepto [https://github.com/madrobby/zepto/blob/v0.7/src/event.js#L96-108] `.delegate()` function.

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

5.5.2. Binding to events triggered by `this.model` or `this.collection`

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 TasksIndex = 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.

5.5.3. Binding to custom events

With sufficiently complex views, you may encounter a situation where you want one view to change in response to another.

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

```
GenderFilter = 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.genderView = new GenderFilter();
    this.genderView.bind("changed", this.filterByGender);
    this.filteredCollection = this.collection;
    this.render();
  },

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

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

In the above snippet, the `GenderFilter` view is responsible for the filter control. When a 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 `UserTable` view which renders 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 step back and think through whether you should instead be binding to events on the underlying components.

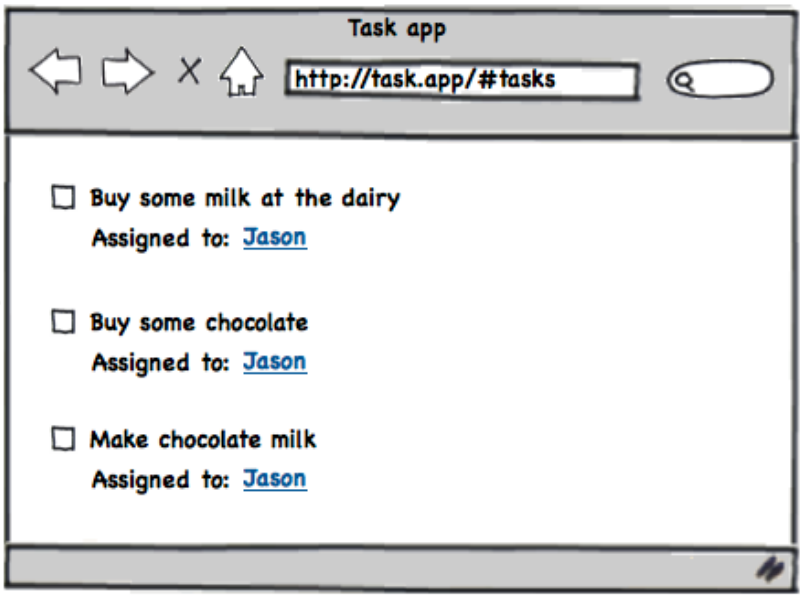
5.6. Cleaning Up: Unbinding

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.

5.6.1. Why do I have to unbind events?

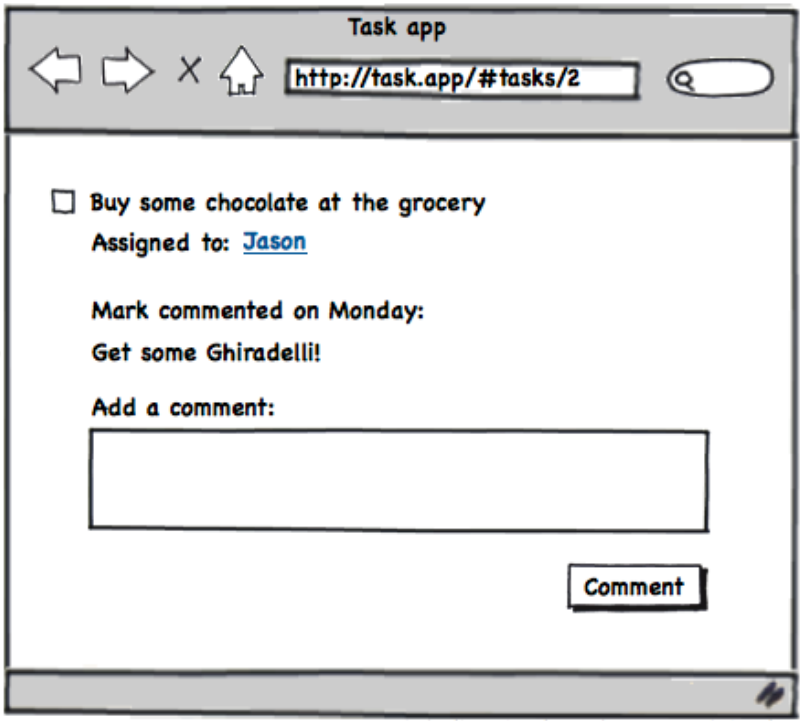
Consider two views in a Todo app: an index view which contains all the tasks that need to be done:

Figure 2. Tasks index view



and a detail view that shows detail on one task:

Figure 3. Tasks detail view



The interface switches between the two views.

Here’s the source for the aggregate index view:

```
var TasksIndex = 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}));
  }
});
```

and the source for the individual task detail view:

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

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

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

    this.model.bind("change", this.render);
    this.model.comments.bind("change", this.render);
    this.model.comments.bind("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.

Let's take a look at how to unbind various kinds of events.

5.6.2. Unbinding DOM events

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()`.

5.6.3. Unbinding model and collection events

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

When should we 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 doesn't sound like a very cohesive approach, so let's include the cleanup responsibility on this view.

TODO: Consider just overriding `Backbone.View.prototype.remove()` instead of making a new function, since `remove()` is very simple. What are the pros/cons?

Let's write a `leave()` function on our view that wraps `remove()` and handles any additional event unbinding we need to do. As a convention, when we use this view elsewhere, we'll call `leave()` instead of `remove()` when we'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...
});
```

5.6.4. 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 then 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()`.

TODO: Is it viable to use Function.caller inside Backbone.Events so this functionality is provided by Backbone.Events? https://gist.github.com/158a4172aea28876d0fc

TODO: Wrap `bindTo()` and `unbindFromAll()` into `Observer` which gets mixed into `CompositeView`.

5.6.5. Unbinding custom 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 observer, and are responsible for cleaning up the event binding when they are disposed.

However, additionally, 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...
});
```

5.6.6. 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 whereever 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`.

5.7. Swapping router

When switching from one view to another, we should clean up the previous view. We discussed previously 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.View` and `Backbone.Events` 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 who the current view is, so that next time `swap()` is invoked, it can be properly cleaned up as well.

5.7.1. 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 in 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 chose to just mix in a `swap` method, and called it from a direct descendant of `Backbone.Router`, an unaware (and unlucky) programmer now needs to go on a deep source dive in an attempt to figure out where that's coming from. At least with a subclass, the hunt should 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. First off, 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's `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 in the properties in the `Backbone.Router` prototype along with our new custom object that includes the `swap` function.

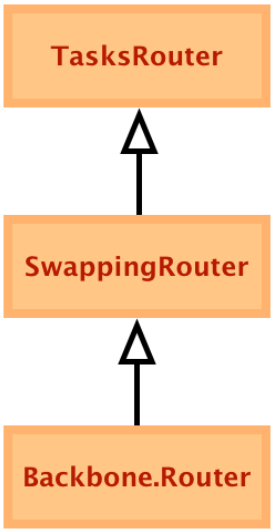
Finally, the subclass cake is topped off with some Backbone frosting: setting `extend`, which is a self-propagating function that all Backbone public classes use. Let's take a quick look at this function, as of Backbone 0.5.3:

```
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!
}
```

So, it's 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:

Figure 4. Router class inheritance



Phew! Hopefully this adventure into Backbone and JavaScript internals has taught you that although it's more code, it's hopefully going to 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 of the actions in that Router use `SwappingRouter.swap()` to invoke rendering of views, freeing itself from the complexities of cleaning them up.

5.8. 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.

5.8.1. Refactoring from a large view

One of the first refactorings you 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({
  template: JST['tasks/tasks_detail'],
  tagName: 'section',
  id: 'task',

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

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

    this.model.bind("change", this.render);
    this.model.comments.bind("change", this.render);
    this.model.comments.bind("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: 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;
```

TODO: Re-link to swapping-internals anchor once https://github.com/schacon/git-scribe/issues/33 is fixed

Similar to the `SwappingRouter`, the `CompositeView` base class solves common housekeeping problems by establishing a convention. See the Swapping Router 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 childs 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.

5.8.2. 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 app could in fact make the user's browser to 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 knows keeps track of the current view so that when we swap it with anther view, it can do the work of cleaning the old one up.

- A **Composite View** that knows how to keep track of any child views it has rendered, so that when it is asked to clean up, it knows to clean up its own children.

The piece that ties both of them together is the `leave()` function on the `CompositeView`. It takes on the task of completely cleaning up itself by removing itself from the DOM via jQuery's `remove()` function, as well as removing all events via a call to `Backbone.Events.off()`.

TODO: Mix `Observer` into `CompositeView`.

5.9. Forms

Who likes writing form code by hand? Rails' form builder API greatly helps reduce application code. 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.

5.9.1. 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` then 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 3.1 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. Yet, this approach still only generates markup; it doesn't serialize forms into data hashes or Backbone models.

5.9.2. Serializing forms

The second requirement is to serialize forms 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.

5.9.3. 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 implementation in progress is [`backbone-forms` by Charles Davison](https://github.com/powmedia/backbone-forms). It provides markup construction and serialization, as well as a method for declaring your schema (data types) to support both of those facilities.

5.9.4. Display error messages

We are assuming, with a hybrid Rails/Backbone application, that at least some of your business logic resides on the server.

5.10. 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

Rails internationalization (i18n) API [http://guides.rubyonrails.org/i18n.html], provided via the i18n gem [https://rubygems.org/gems/i18n], 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 `i18n-js` [https://github.com/fnando/i18n-js], that provides access to your i18n content as a JavaScript object, similar to how the JST object provides access to your templates.

From the documentation, you can link the 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 [https://github.com/fnando/i18n-js] for further usage information.

6. Models and collections

6.1. Model associations

Backbone.js doesn't prescribe a way to define associations between models, so we need to get creative and use the power of JavaScript to set up associations in such a way that its usage is natural.

6.1.1. Belongs to associations

Setting up a `belongs_to` association in Backbone is a two step process. Let's discuss setting up the association that may occur between a task and a user. The end result of the approach is a `Task` instance having a property called `user` where we store the associated `User` object.

To set this up, let's start by telling Rails to augment the task's JSON representation to also send over the associated user attributes:

```
class Task < ActiveRecord::Base
  belongs_to :user

  def as_json(options = nil)
    super((options || {}).merge(include: { user: { only: [:name, :email] } }))
  end
end
```

This means that when Backbone calls `fetch()` for a `Task` model, it will include the name and email of the associated user nested within the task JSON representation. Something like this:

```
{
  "title": "Buy more Cheeseburgers",
  "due_date": "2011-03-04",
  "user": {
    "name": "Robert McGraffalon",
    "email": "bobby@themcgraffalons.com"
  }
}
```

Now that we receive user data with the task's JSON representation, let's tell our Backbone User model to store the User object. We do that on the task's initializer. Here's a first cut at that:

```
var Task = Backbone.Model.extend({
  initialize: function() {
    this.user = new User(this.get('user'));
  }
});
```

We can make a couple of improvements to the above. First, you'll soon realize that you might be setting the user outside of the initialize as well. Second, the initializer should check whether there is user data in the first place. To address the first concern, let's create a setter for the object. Backbone provides a handy function called `has` that returns true or false depending on whether the provided attribute is set for the object:

```
var Task = Backbone.Model.extend({
  initialize: function() {
    if (this.has('user')) {
      this.setUser(new User(this.get('user')));
    }
  },

  setUser: function(user) {
    this.user = user;
  }
});
```

The final setup allows for a nice clean interface to a task's user, by accessing the task property of the user instance.

```
var task = Task.fetch(1);
console.log(task.get('title') + ' is being worked on by ' + task.user.get('name'));
```

6.1.2. Has many associations

You can take a similar approach to set up a `has_many` association on the client side models. This time, however, the object's property will be a Backbone collection.

Following the example, say we need access to a user's tasks. Let's set up the JSON representation on the Rails side first:

```
class User < ActiveRecord::Base
  has_many :tasks

  def as_json(options = nil)
    super((options || {}).merge(include: { tasks: { only: [:body, :due_date] } }))
  end
end
```

Now, on the Backbone `User` model's initializer, let's call the `setTasks` function:

```
var User = Backbone.Model.extend({
  initialize: function() {
    var tasks = new Tasks.reset(this.get('tasks'));
    this.setTasks(tasks);
  },

  setTasks: function(tasks) {
    this.tasks = tasks;
  }
});
```

Note that we are setting the relation to an instance of the `Tasks` collection.

TODO: Let's exapnd upon this, as it isn't the most flexible solution. (It is a good start.) We are setting the JSON representation of the Rails models to suit the Backbone.js concerns. Additionally, the `Task#as_json` method at the top is concerned with the User JSON representation. It should at least delegate to User#as_json. Going further, the JSON presentation for consumption by Backbone.js should be completely extracted into the JSON API endpoint controller action, or even a separate presenter class.

TODO: Some of this is repeated in the model_relationships section, unify.

6.2. 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.

6.2.1. Filters

To filter a `Backbone.Collection`, like with Rails named scopes, define functions on your collections that filter by your criteria, using the `select` function from Underscore.js, and 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 other 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);
```

6.2.2. 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;
  }
};
```

6.2.3. 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;
  }
};
```

6.3. Validations

The server is the authoritative place for verifying whether data that being stored is valid. Even though backbone.js exposes an API [http://documentcloud.github.com/backbone/#Model-validate] 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 up all the way to the interface for user feedback - and having it all be 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 it's therefore easier to extend and reuse.

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

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

_.extend(FormAttributes.prototype, {
  attributes: function() {
    var attributes = {};
    _.each($('input, select', 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 are 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 applciations:

```
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 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"] }
```

So 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 both for 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, so there's your chance for some other view to bind to that event and rerender 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 off 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`, who's 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 that will do it's fine job in 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 generic enough solutions to be even 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 [https://github.com/powmedia/backbone-forms].

6.4. 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.

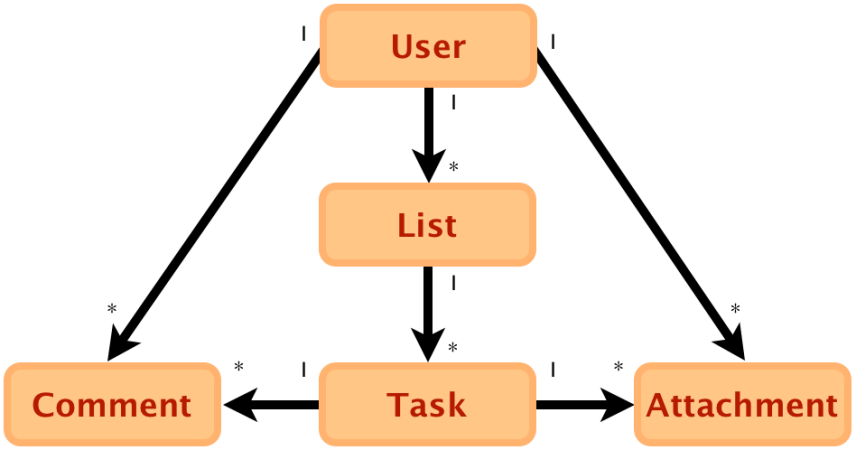
These model relationships don't have to be persisted by a relational database. In a chatroom application whose data is persisted in a key-value store, the data could still be modeled as a Room which has many Messages, as well as Memberships that link the Room to Users. A content management application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

6.4.1. Relations in the Task App

In the example application, there are Users which have many Tasks through Lists. Each Task has many Comments and Attachments.



6.4.2. 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, step back and 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 lists and the tasks on each list, but not the comments or attachments.
- When a user views the details of an individual task, the comments and attachments for that task are displayed.

This leads us to see that the Lists and Tasks for a user are used immediately upon navigating to the dasboard, but the Comment and Attachment data for a Task are not needed upon initial page load, and are possibly never needed at all.

Let’s say that we are also planning for the user to have continuous network access, but not to necessarily have a high speed connection. Also, users tend to view their lists of tasks frequently, but rarely view the comments and attachments.

Based on this, we will bootstrap the collections of Lists and Tasks inside the dashboard, and defer loading of associated Comments and 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 faster initial page load, at the cost of a longer time to actual interactivity: although the byte size of the page plus data is roughly the same, the overhead of additional HTTP requests incurs the extra load time.

- Preload all the information, including Comments and Attachments. This would work well if we expected users to frequently access the comments and attachments of many tasks.

- Use localStorage as the primary storage engine, and sync to the Rails server in the background. This would be advantageous if we expected network access to be intermittent, although it incurs the additional complexity of having to resolve conflicts on the server if two clients submit conflicting updates.

6.4.3. Designing the HTTP JSON API

Now that we know we'll bootstrap the Lists and Tasks and defer the Comments and Associations, we should decide how to deliver the deferred content. We have two options here. Our goal is to fetch to comments and attachments for an individual task.

One way we could approach this is the issue separate API calls for each nested resource:

```
$ curl http://tasksapp.local:3000/tasks/78/comments.json | ppjson
[
  {
    "id": 208,
    "user_id": 3,
    "body": "What do you think of this mock? (See attachment)"
  },
  {
    "id": 209,
    "user_id": 1,
    "body": "Looks great!  I'll implement that."
  }
]

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

> **Note**
>
> We will authenticate API requests with cookies, just like normal user login, so the actual curl request would need to include a cookie from a logged in user.

This approach has the advantage of adhering more to convention, and requiring less code in both the server-side JSON presentation and the client-side JSON parsing. Its disadvantage is performance: to fetch a task's associated data, we need to send 2 HTTP requests. When more kinds of associated resources are added in the future, the number of requests will increase.

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",
  "comments": [
    {
      "id": 208,
      "user_id": 3,
      "body": "What do you think of this mock? (See attachment)"
    },
    {
      "id": 209,
      "user_id": 1,
      "body": "Looks great!  I'll implement that."
    }
  ],
  "attachments": [
    {
      "id": "32",
      "upload_url": "https://s3.amazonaws.com/tasksapp/uploads/32/mock.png"
    }
  ]
}
```

This approach involves additional code in both producing the JSON on the server side and parsing the JSON on the client side. We'll take this approach for the example application, both because it requires fewer HTTP requests and because it's a more interesting example and illustrates the technique of parsing nested models in Backbone.js.

Now that we know we'll bootstrap the Lists and Tasks and defer the Comments and Attachments, we know that our HTTP JSON API should support at least the following Rails routes:

```
resources :lists, :only => [:create, :update, :delete]
resources :tasks, :only => [:show, :create, :update, :delete]
```

> **Tip**
>
> In some applications, you choose to expose a user-facing API. It's often 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.

6.4.4. Implementing the API: presenting the JSON

For building 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 necessarily the model's concern. Furthermore, the `as_json` API starts to fall appart when representing complex hierarchies. Other approaches such as creating a separate presenter object, or writing a builder-like view are all better approaches because additionally we don't pollute our models with presentational logic.

The RABL rubygem [https://github.com/nesquena/rabl] is a good generalization of the problem and can help with this particular aspect of your API implementation.

RABL allows you to create templates where you can easily specify the JSON representation of your models. If you've worked with the great `builder` library to generate arbitrary 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 and views may look like this:

```
class TasksController < ApplicationController
  respond_to :json
  def show
    @task = Task.find(params[:id])
    respond_with @task
  end
end
```

Rails responders 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 it, 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) }
```

Now it is much easier to extend and tweak the JSON generated on the server, while still keeping the model free of presentational behavior. Do look at the project's readme [https://github.com/nesquena/rabl#readme] for all the bells and whistles.

6.4.5. Parsing the JSON and instantiating client-side models

TODO: Expand outline

Outline: Discuss overriding Backbone Model parse() function. Talk about how parsing fits into the fetch/new object lifecycle. Point out inconsistencies (parse not invoked during reset, only fetch/set etc) Discuss https://github.com/PaulUithol/Backbone-relational

TODO: If a Backbone Task doesn't always have its associations filled (e.g. when rendering the TasksIndex Backbone view, whose JSON is built by bootstrapping, in `Tasks#index`), 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 associations. Whose concern is that? Presumably the TaskShow view. You could discuss lazily populating this by making the task associations functions instead of properties (compare task.attachments.each to task.attachments().each; in the latter, you could lazily fetch and populate, but then you run into the issue that fetch is async.)

6.5. 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 tradeoff here, between duplication and performance. It's desirable to have one and only one implementation of a particular concern in your domain, but it's also desirable for your application to perform responsively.

6.5.1. 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) {
    new ErrorView(errorResponse, this.$('form')).render();
  }
});
```

This uses the `ErrorView` class which is able to parse the error hash returned from Rails, which was discussed on the Validations section.

> **Note**
>
> This is also the first time you probably see `_.bindAll()`, so let's diverge 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. An user fills out a form for a creating a new Member in your application, submits the form, and, if they forgot

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. TODO: Link up that reference.

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 ErrorView = function(responseOrErrors, form) {
  this.form = $(form);

  if (responseOrErrors && responseOrErrors.responseText) {
    this.errors = JSON.parse(responseOrErrors.responseText);
  } else {
    this.errors = responseOrErrors;
  }
};
```

Now, 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 ErrorView able to handle either client-side or server-side error messages, we have a server-side guarantee of data correctness, [1] and a responsive UI that can validate the Member email presence without round-tripping to the server.

The tradeoff we've made is that of duplication; the concern of "what constituted 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 strategies for reducing duplication.

6.5.2. Kinds of logic you duplicate

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

[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.

- 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 such.

- 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 either are 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.

6.5.3. 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 automatable, 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 zipcode 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.

TODO: This is actually what the client_side_validations gem [https://github.com/bcardarella/client_side_validations] does…

TODO: The csv model branch is a wip for Backbone compliance, pretty neat: https://github.com/bcardarella/client_side_validations/tree/model

6.5.4. Querying

TODO: Expand on outline.

Outline: I think it's possible to establish conventions here, similar to validations, so that server-side scopes can be converted to client-side collection filtering. However, is this valuable? Do you actually often duplicate the same querying (sorting/filter) concerns across client and server?

Also, since this whole discussion is about perf, consider tradeoff of paginating anyways, that's interesting, so can you reduce duplication and generate code with that too?

6.5.5. Callbacks

TODO: Expand on outline.

Outline: These often depend on server-side persistence, so would you even want them on the client side? Perhaps, e.g. same lifecycle events for the analagous client-side models, but it's actually likely that your client-side models will differ sufficiently (since they're in the presentation tier) from server-side models that these concerns won't be duplicates, so it's less of a worry.

6.5.6. Algorithms

TODO: Expand on outline.

Outline: General algorithms are often the trickiest. It's possibly to write the logic in JS and then make that available to Ruby, if you have a REALLY large piece of logic, but weigh the cost of that overhead against the cost of duplicating the logic. At some point it probably makes sense, though. Also consider JS server-side and wrapping that as a webservice for Rails access… would that be easier? Need specific examples to motivate this well.

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

TODO: The ErrorList/ErrorView implementation here isn't quite consistent with those in the prior validations chapter. Refactor for consistency or, if that's inappropriate, do a better job explaining the changes.

6.6. Synchronizing between clients

A big 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 realtime. We can deliver a more seamless experience by propagating users' changes to one another as they take place: when we 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?

6.6.1. 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 [http://www.w3.org/TR/websockets/] 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 re-drawing themselves entirely. Backbone.js helps out in this department, as your client-side application app 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.

6.6.2. 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. 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 [http://faye.jcoglan.com] as the messaging backend, which has Ruby and JavaScript implementations for clients and server. Faye implements the Bayeux protocol [http://svn.cometd.com/trunk/bayeux/bayeux.html], prefers WebSocket for transport (thought 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.

TODO: System-partitioned sequence diagram

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.

6.6.3. 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, I like to keep 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
```

6.6.4. 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, 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",
```

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.

6.6.5. 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!

6.6.6. 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 on its messages and rerender 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 ditcate 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 there is a chapter dedicated to testing Backbone applications, this 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                  |
      | Purchase Cheeseburgers |
    And I am using session "Alice"
    And I sign in as "alice@example.com"
    Then I should see "Purchase Cheeseburgers"
    When I switch to session "Olivia"
    And I sign in as "olivia@example.com"
    And I edit the "Purchase Cheeseburgers" task and rename it to "Purchase Giant Cheesel
    And I switch to session "Alice"
    Then I should see "Purchase Giant 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 [https://github.com/thougthbot/factory_girl], 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 so:

```
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, in the case 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 succesfully, so it is the approach we recommend. For more information about using jasmine, refer to the chapter on testing.

6.6.7. More reading

> **Note**
>
> Faye implements a messaging protocol called Bayeux: http://svn.cometd.com/trunk/bayeux/bayeux.html

Note

Read up on idempotent messages. Check out this solid, readable article The Importance of Idempotence [http://devhawk.net/2007/11/09/the-importance-of-idempotence/].

6.7. Uploading attachments

While Ruby gems such as 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.

6.7.1. How to attach files to Backbone models

If you upload to a backbone model, you can't do it in a typical async request. Meaning, `model.save()` can't just send a file to the server like other attributes. Instead, we save the attachment in a separate request, and then just swap in an attachment id on the model. This does mean that you can have unclaimed attachments if the end user leaves the page before saving the parent model, but those can be periodically cleaned out if the 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. It's generally more straightforward to break the attachment out into its own model, because this can greatly simplify grabbing the attachment reference.

There are quite a few approaches to uploading files asynchronously, and browser support varies. There are features like multiple file upload and drag-and-drop to consider, too.

We'll use the HTML5 File API because it's a straightforward approach that is supported by modern browsers. The API is small and the wrapper code that we start with:

https://github.com/mockenoff/HTML5-AJAX-File-Uploader

is easy to read. This approach requires XHR2 and FormData:

- https://developer.mozilla.org/en/XMLHttpRequest/FormData

If you would like to provide fallback support for older browsers, using Flash or iframes, you can do so with plugins like (TODO: recommend plugins).

6.7.2. Example: Attaching images to Tasks

In our example task management app, we'd like for the owner of a task to attach several images to each task. We want uploads to happen in the task detail view, and for the uploads 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 to the TaskShow view to 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>
```

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 unit tests for the uploading:

```
//= require application

describe("ExampleApp.Views.TaskShow uploading", function() {
  var task, view, $el;

  beforeEach(function() {
    this.xhr = sinon.useFakeXMLHttpRequest();
    var requests = this.requests = []

    this.xhr.onCreate = function(xhr) {
      requests.push(xhr);
    };

    this.xhr.prototype.upload = {
      addEventListener: function() {}
    };

    task = new ExampleApp.Models.Task({
      id: 1,
      title: "Wake up"
    });

    view = new ExampleApp.Views.TaskShow({ model: task });
  });

  afterEach(function() {
    this.xhr.restore();
  });

  it("uploads the file when the upload button is clicked", function() {
    view.uploadInput = function() {
      return { files: ["uploaded file contents"], }
    };

    $el = $(view.render().el);
    view.upload();

    expect(this.requests.length).toEqual(1);
    expect(this.requests[0].url).toEqual("/tasks/1/attachments.json");
  });
});
```

TODO: Finish outline:

- NB: You can't overwrite input.files (a FileList instance), so you'll have to provide a point of fake injection; in our case, TaskShow#uploadInput().
- Make it pass by adding uploader.js and adding uploader logic to TaskShow view
- Now we are uploading, but the server isn't accepting/persisting
- Test-drive persistence on server side:
  - Add paperclip gem
  - Create Attachment model, route, controller. Test-drive the units.
  - NB on integration point: XHR requests from BB to Rails needs CSRF tokens, so inject as uploader.prefilter, analagous to $.ajaxPrefilter http://api.jquery.com/extending-ajax/
- Next, display existing attachments to the user.

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 attachments property that is instantiates with an AttachmentsCollection instance.

> **Note**
>
> This is written assuming that the model_relationships.asc chapter came first and discusses how to structure the JSON, which is bundling the comments and attachments associations under the Task's JSON representation. It should introduce and discuss using Rabl, too. Depending on how in-depth that section is, we may need to write more here to contextualize.

We're providing a JSON represenatation using Rabl, rooted at the Task:

```
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
```

Note that you have to have to tell Rabl to suppress the root JSON node, just like we suppress the root JSON node in ActiveRecord with `ActiveRecord::Base.include_root_in_json = false`:

```
# 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_spec.js:

```
//= require application

describe("ExampleApp.Views.TaskShow for a task with attachments", function() {
  var task, view, $el;

  beforeEach(function() {
    task = new ExampleApp.Models.Task({
      id: 1,
      title: "Buy pies",
      attachments: [
        {
          upload_file_name: "blueberries.jpg",
          upload_url: "http://www.realblueberries.com/images/Blueberry-Cluster-1.jpg"
        },
        {
          upload_file_name: "strawberries.jpg",
          upload_url: "http://strawberriesweb.com/three-strawberries.jpg"
        }
      ]
    });

    view = new ExampleApp.Views.TaskShow({ model: task });
    $el = $(view.render().el);
  });

  it("displays attachments", function() {
    expect($el).toContain(".attachments img[src='http://www.realblueberries.com/images/B
    expect($el).toContain(".attachments img[src='http://strawberriesweb.com/three-strawb
  });

  it("displays attachment filenames", function() {
    expect($el.find(".attachments p").first()).toHaveText('Attached: blueberries.jpg');
    expect($el.find(".attachments p").last()).toHaveText('Attached: strawberries.jpg');
  });
});
```

This depends on parsing the JSON from the client side, so test drive that for the ExampleApp.Models.Tasks Jasmine spec:

- Include spec/javascripts/models/task_spec.js
- Implement in task.js
- TDD Attachments collection and Attachment model
- Implement Attachments collection and Attachment model
- Green?

7. Testing

7.1. 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.

7.1.1. 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.

7.1.2. Capybara

Though there is a host of tools available to you for writing automated integration tests, we recommend [capybara](https://github.com/jnicklas/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).

7.1.3. 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
```

7.1.4. Drivers

Capybara supports multiple drivers through a common API, each with benefits and drawbacks. We prefer to use either [capybara-webkit](https://github.com/thoughtbot/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
```

And then tag a Cucumber scenario as @javascript. If you need to fall back to using Selenium, tag that scenario with @selenium.

7.2. 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; 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.

7.2.1. 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);
  });
});
```

7.2.2. 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 componenets 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.

7.2.3. Helpful Tools

- Spy/stub/mock, even your HTTP, with [sinon.js](http://sinonjs.org/)
- If you're looking for factory_girl.js, it's called [Rosie](https://github.com/bkeepers/rosie)
- [guard-jasmine](https://github.com/netzpirat/guard-jasmine) autotest your Jasmine with headless webkit ([phantomjs](http://www.phantomjs.org/))
- Write in CoffeeScript and use the 3.1 asset pipeline with [jasminerice](https://github.com/bradphelan/jasminerice)
- See other examples on James Newbery's blog: [testing Backbone with Jasmine](http://tinnedfruit.com/2011/03/03/testing-backbone-apps-with-jasmine-sinon.html) and check out his [examples on GitHub](https://github.com/froots/backbone-jasmine-examples)

7.3. Example: Test-driving a Task application

7.3.1. 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 fastest 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 already.

The other dependencies you can install by adding them to your Gemfile. The gems you'll need for testing are jasminerice, jasmine, 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', '3.1.0'
gem 'sqlite3'

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', "  ~> 3.1.0"
  gem 'coffee-rails', "~> 3.1.0"
  gem 'uglifier'
end

group :development, :test do
  gem "rspec-rails", "~> 2.6.1"
  gem "ruby-debug19"
  gem 'jasmine', "= 1.1.0.rc4"
  gem 'jasminerice'
  gem 'cucumber-rails', "~> 1.0.2"
end

group :test do
  gem 'turn', :require => false
  gem "capybara", "~> 1.1.1"
  gem 'selenium-webdriver', '~> 2.18.0'
  gem 'cucumber-rails', "~> 1.0.2"
  gem "factory_girl_rails"
  gem "bourne"
  gem "database_cleaner"
  gem "nokogiri"
  gem "shoulda"
  gem "launchy"
  gem "guard-spork"
  gem "spork", "~> 0.9.0.rc"
end
```

If you haven't already, you can bootstrap your application for Cucumber and Capybara:

```
rails generate cucumber:install
```

Next, bootstrap the application for Jasmine:

```
rails generate jasmine:install
```

You'll want to set up Jasminerice to load all your helpers and specs:

```
include::../../example_app/
```

Finally, you need to mount the Jasminerice engine so that you can run your Jasmine specs. Add the following routes to config/routes.rb:

```
if ["development", "test"].include? Rails.env
  mount Jasminerice::Engine => "/jasmine"
end
```

With this configuration, you can run cucumber scenarios with the cucumber command and you can run Jasmine tests by accessing http://localhost:3000/jasmine in your browser.

7.3.2. Step by step

We'll go outside in: cucumber first, then rspec or jasmine as needed.

TODO: This writing is terse. Come back and improve flow.

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](https://github.com/thoughtbot/factory_girl/blob/v2.1.0/GETTING_STARTED.md):

```
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 # features/step_definitions/web
  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
```

> **Note**
>
> A common gotcha when testing Backbone.js Rails apps 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 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 do
    resources :attachments, :only => [:create, :show]
  end

  root :to => 'tasks#index'

  if ["development", "test"].include? Rails.env
    mount Jasminerice::Engine => "/jasmine"
  end
end
```

Note

You can also see the route for the [jasminerice gem](http://rubygems.org/gems/jasminerice), which makes the Rails 3.1 asset pipeline (and all of our app assets) available to the Jasmine specs.

```
class TasksController < ApplicationController
  before_filter :authorize
  respond_to :html, :json

  def index
    respond_with(@tasks = current_user.tasks)
  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
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](https://github.com/velesin/jasmine-jquery) library (provided by jasminerice) to provide DOM matchers for Jasmine like `toContain()`.

To run the Jasmine spec, run the Rails server and visit http://localhost:3000/jasmine

To make this test pass, we'll add a small template and make the `TasksIndex` view render it:

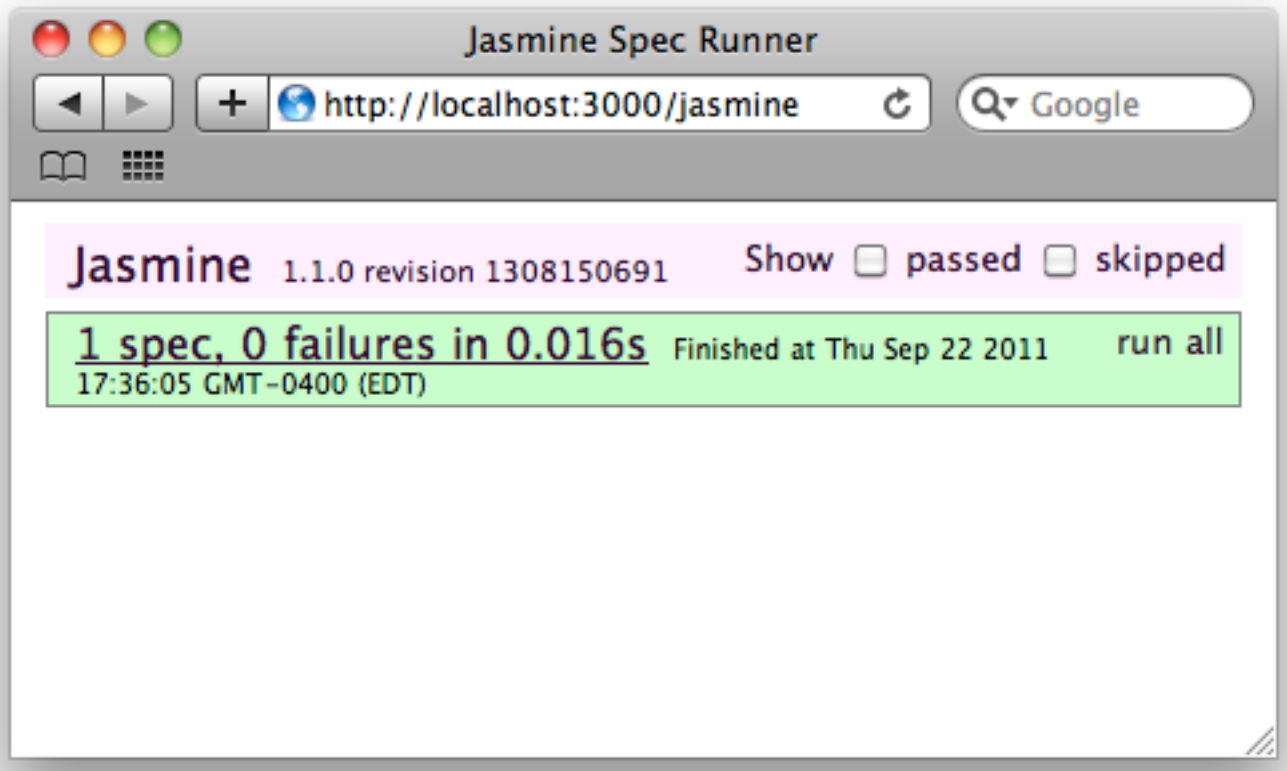
```
ExampleApp.Views.TasksIndex = Backbone.View.extend({
  tagName: 'div',
  id: 'tasks',

  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 5. 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                                          # features/view_tasks.featu
  Given the following tasks exist:                            # factory_girl-2.1.0/lib/fa
    | Title                                |
    | Purchase the backbone on rails ebook |
    | Master backbone                      |
  And I am on the home page                                   # features/step_definitions
  Then I should see "Master backbone" within the tasks list   # features/step_definitions
    expected there to be content "Master backbone" in "Title Completed" (RSpec::Expectati
    ./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 tas
```

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 0.008sFinished 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:3000/assets/views/tasks_index_spec.
```

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.bind("change:attachments", this.parseAttachments);
    this.parseAttachments();
  },

  parseAttachments: function() {
    this.attachments = new ExampleApp.Collections.Attachments(this.get('attachments'));
  },

  schema: {
    title: { type: "Text" }
  },

  urlRoot: '/tasks',

  isComplete: function() {
    return this.get('complete');
  }
});
```

and test-drive 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");
  });
});
```

```
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> </
```

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:

app/assets/javascripts/views/tasks_index.js:

```
ExampleApp.Views.TasksIndex = Support.CompositeView.extend({
  initialize: function() {
    _.bindAll(this, "render");
    this.collection.bind("add", this.render);
  },

  render: function () {
    this.renderTemplate();
    this.renderTasks();
    return this;
  },

  renderTemplate: function() {
    console.log(this.el);
    console.log(this.$el);
    this.$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);
    });
  }
});
```

app/assets/javascripts/templates/tasks/index.jst.ejs:

```
<table id="tasks-list">
  <thead>
    <tr>
      <th>Title</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: this is because the Jasmine spec is an isolation spec, and verifies that the TasksIndex view works in isolation.

```
Then I should see "Master backbone" within the tasks list # features/step_definitions/we
Unable to find css "#tasks table" (Capybara::ElementNotFound)
```

However, there is additional code we need to write to integrate the data present in the Rails test database with 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("init()", function() {
    it("accepts task JSON and instantiates a collection from it", function() {
      var tasksJSON = {"tasks": [{"title":"thing to do"}, {"title":"another thing"}]};
      ExampleApp.init(tasksJSON);

      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");
    });

    it("instantiates a Tasks router", function() {
      ExampleApp.Routers.Tasks = sinon.spy();
      ExampleApp.init({});
      expect(ExampleApp.Routers.Tasks).toHaveBeenCalled();
    });

    it("starts Backbone.history", function() {
      Backbone.history = { start: sinon.spy() };
      ExampleApp.init({});
      expect(Backbone.history.start).toHaveBeenCalled();
    });
  });
});
```

Get it to green:

```
var ExampleApp = {
  Models: {},
  Collections: {},
  Views: {},
  Routers: {},
  init: function(data) {
    this.tasks = new ExampleApp.Collections.Tasks(data.tasks);

    new ExampleApp.Routers.Tasks({ collection: this.tasks });
    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 %> }
</script>

<%= content_for :javascript do -%>
  <script type="text/javascript">
    $(function () {
      var json_div       = document.createElement('div');
      json_div.innerHTML = $('#bootstrap').text();
      var data           = JSON.parse(json_div.innerHTML);
      ExampleApp.init(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)
```

TODO: Refactoring step. Extract a TaskView class and loop & iterate. Note specs passing, cukes passing.

TODO: Possible, bind events on the child views to motivate making TasksIndex a CompositeView to avoid leaking refs.

TODO: Optionally TDD through the new/create cycle, too.

8. Security (stub)

8.1. Encoding data when bootstrapping JSON data

As it turns out, bootstrapping JSON data in your erb templates introduces 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 exmple, 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 bootstraped 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 json_div       = document.createElement('div');
    json_div.innerHTML = $('#bootstrap').text();
    var data           = JSON.parse(json_div.innerHTML);
    ExampleApp.init(data);
  });
</script>
```

A reliable way to unencode the HTML-encoded JSON string is to use the browser's native functionality, by retreiving a element's `innerHTML`. So in the above script, we create a `json_div` element, assign its `innerHTML` to the bootstrap script's text, and retreive back out, unencoded. The final result is the `data` variable containing proper JSON that can be parsed and passed along to your app's init function.

This approach can be seen on the example app on the `app/views/tasks/index.html.erb` template

TODO: Discuss using `json2.js`: