Backbone.js *on* Rails

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

Contents

Chapter 1

Introduction

Welcome to the Backbone.js on Rails eBook sample. This is published directly from the book, so that you can get a sense for the content, style, and delivery of the product. We've included three sample sections. Two are specific to Rails integration: file organization, and a high-level overview of connecting a Backbone app inside your Rails app. The last is Backbone.js specific, and covers filtering and sorting your Backbone collections.

If you enjoy the sample, you can get access to the entire book and sample application at:

https://learn.thoughtbot.com/products/1-backbone-js-on-rails

Chapter 2

What can I expect from that?

Glad you asked!

The eBook covers intermediate to advanced topics on using Backbone.js, including content specific to integrating with Rails applications.

In addition to the book (in HTML, PDF, EPUB, and Kindle formats), you also get a complete example application, and the ability to get your questions about Backbone.js and Rails answered by the thoughtbot team.

The book is written using Markdown and pandoc, and hosted on GitHub. You get access to all this. You can also use the GitHub comment and issue features to give us feedback about what we've written and what you'd like to see. Give us your toughest Backbone.js questions, and we'll see what we can do. Last but not least, also included is a complete sample Backbone.js and Rails application. What the book describes and explains, the example app demonstrates with real, working code. Fully up to date for Rails 3.2.

Chapter 3

Contact us

If you have any questions, or just want to get in touch, drop us a line at learn@thoughtbot.com.

Chapter 4

Rails Integration

Organizing your Backbone code in a Rails app

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

Rails 3.0 and prior

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

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

```
      todos/
        todos_index.js
```

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

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

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

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

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

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

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

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

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

Jammit and a JST naming gotcha

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

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

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

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

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

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

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

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

Rails 3.1 and above

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

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

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

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

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

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

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

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

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

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

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

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

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

An overview of the stack: connecting Rails and Backbone

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

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

Setting up models

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Setting up Rails controllers

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

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

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

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

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

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

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

Validations and your HTTP API

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

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

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

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

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

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

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

```
class TasksController < ApplicationController
  respond_to :json

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

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

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

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

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

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

Setting Up Views

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

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

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

This application object will look like the following:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Converting an existing page/view area to use Backbone

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

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

Consider the following Rails view for a tasks index:

```
<h1>Tasks</h1>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

proxies to Underscore.js to provide these iteration functions as methods on `Backbone.Collection`.

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

```
<h1>Tasks</h1>

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

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

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

Breaking out the TaskView

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

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

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

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

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

```
<h1>Tasks</h1>

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

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

</table>
```

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

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

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

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

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

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

    return this;
  }
});
```

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

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

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

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

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

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

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

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

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

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

  initialize: function() {
  },

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

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

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

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

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

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

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

We've now covered the basic building blocks of converting Rails views to Backbone and getting a functional system. The majority of Backbone programming you will do will likely be in the views and templates, and there is a lot more to

them: event binding, different templating strategies, helpers, event unbinding, and more. Those topics are covered in the "Routers, Views, and Templates" chapter.

Chapter 5

Models and collections

Filters and sorting

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

Filters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Propagating collection changes

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

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

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

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

```
    applyFilter();

    return filteredCollection;
  }
};
```

Sorting

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

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

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

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

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

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

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

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

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

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

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

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

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

. . . And then into another reusable mixin:

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

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

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

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

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

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

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

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

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

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

    applySort();

    return sortedCollection;
  }
};
```

Chapter 6

Closing

Thanks for checking out the sample of our Backbone.js on Rails eBook. If you'd like to get access to the full content, the example application, ongoing updates, and the ability to get your questions about Bakcbone.js and Rails answered by us, you can pick it up on our website:

https://learn.thoughtbot.com/products/1-backbone-js-on-rails