Backbone.js on Rails

Table of Contents

1. Preface (section unstarted)

2. Getting up to speed (section unstarted)

2.1. Backbone.js online resources

2.2. JavaScript online resources and books

3. Introduction (section unstarted)

3.1. Why use Backbone.js

3.2. When not to use Backbone.js

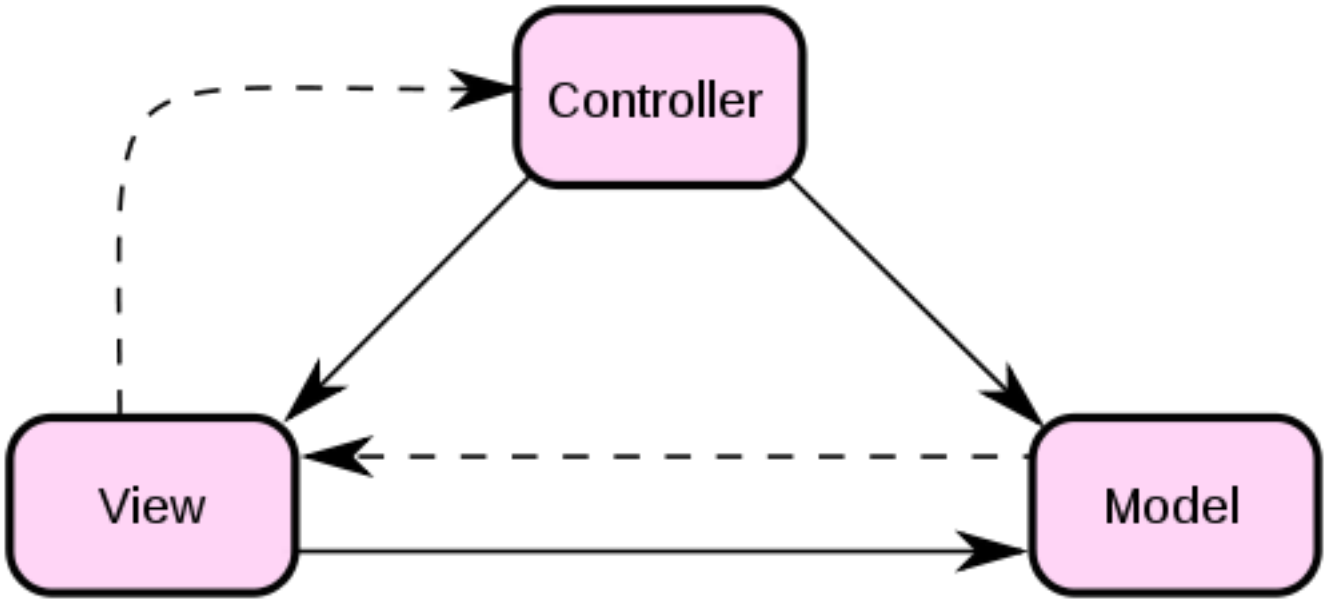
3.3. Why not SproutCore, Cappuccino, Knockout.js, Spine, etc.

4. Organization

4.1. Backbone.js and MVC

Model–View–Controller (MVC) is an architectural pattern used in many applications to isolate "domain 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, a dashed line 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 straight-forward.

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

So 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 will handle many of the aspects that controllers would typically handle, such as actually figuring out what to do next and what to render.

While you could do it, the benefit of actually introducing a Controller in your application would be limited, and the more pragmatic approach is to realize the great organization that Backbone.js gives you is much better than what you had before. The fact that it doesn't have a nice name, or strict adherence to a pattern, isn't worth worrying about.

4.2. 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, Views, Templates, and Routers.

The models in Backbone.js and Rails are analogous. 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
- 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 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** method 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** 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.

4.3. Namespacing your application (chapter unstarted)

5. Rails Integration

5.1. 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.

5.2. 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
    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 })
```

5.2.1. A note on 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, all JST references are prefixed with their parent directory name so they are unambiguous:

```
JST['tasks/form']
JST['tasks/show']
JST['tasks/index']
JST['users/new']
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. One issue with Jammit we've encountered and worked around is that the JST template path can change when adding new templates. We have an open issue and workaround:

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

5.3. Rails 3.1

Rails 3.1 introduces the asset pipeline:

http://edgeguides.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 the asset pipeline. Classes go in `app/assets/javascripts/`, and templates go alongside, in `app/assets/templates/`:

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

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.

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 .
```

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

5.4. Converting your Rails models to Backbone.js-friendly JSON (chapter unstarted)

5.5. Converting an existing page/view area to use Backbone.js (chapter unstarted)

5.6. 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;
  }
};
```

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

6. Views and Templates

6.1. View explanation (chapter unstarted)

6.2. Templating strategy (chapter unstarted)

6.3. View helpers (chapter unstarted)

6.4. Form helpers (chapter unstarted)

6.5. Event binding (chapter unstarted)

6.6. Cleaning up: understanding binding and unbinding (in progress)

Imagine you're writing a todo app. Consider two views: an index view which contains all the tasks, and a detail view that shows detail on one task. The interface switches between the two views, and both views can modify existing tasks (say, to indicate that the task is complete or incomplete).

Figure 2. Tasks index view

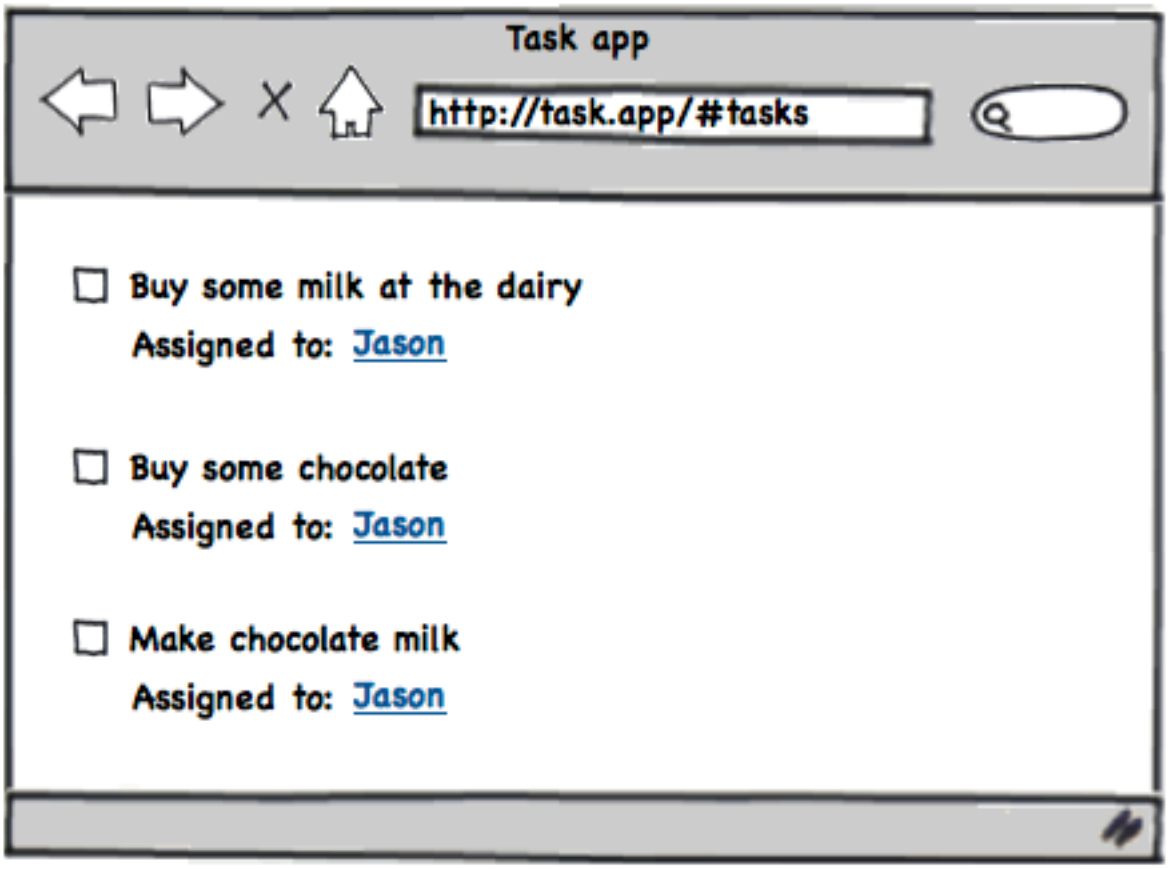
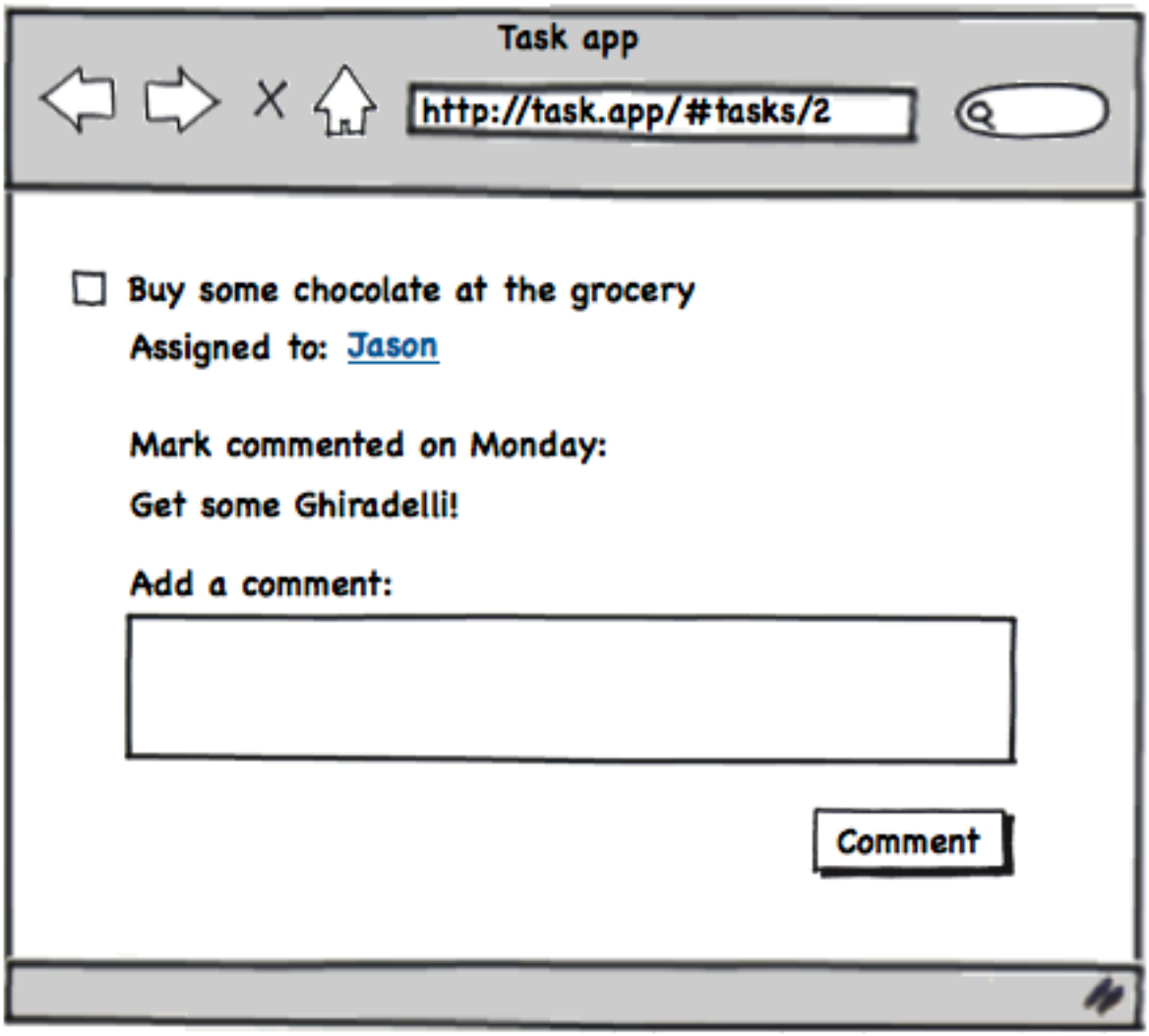


Figure 3. Tasks detail view



The view classes look something like this:

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

  initialize: function() {
    this.tasks = this.options.tasks;

    _.bindAll(this, "render");
    TaskApp.tasks.bind("change", this.render);
    TaskApp.tasks.bind("add", this.render);
  },

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

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, but navigating back and forth between the views without disposing of the previous view will keep creating and binding more views.

The solution is to make sure you unbind and remove views when you leave them. Our approach to this is to use a convention in `Router` instances, and reuse this as a `Router` subclass, `SwappingRouter`.

6.7. Swapping router (in progress)

When switching from one view to another, we should clean up the previous view. Let's augument our view to include the ability to clean itself up:

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

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

  // ...
});
```

The `unbind()` and `remove()` functions are provided by `Backbone.Events` and `Backbone.View`. `unbind()` will remove all callbacks registered on the view, and `remove()` will remove the view's element from the DOM.

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`:

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

_.extend(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);
  }
});
```

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

6.8. Composite views (in progress)

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:

along with the template for that view:

```
<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:

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

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

  removeChild: function(view) {
    var index = this.children.indexOf(view);
    this.children.splice(index, 1);
  },

  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).html('').append(view.el);
  },

  _leaveChildren: function() {
    var clonedChildren = this.children.slice(0);
    _.each(clonedChildren, function(view) {
      if (view.leave) {
        view.leave();
      }
    });
  },
```

```
  _removeFromParent: function() {
    if (this.parent) {
      this.parent.removeChild(this);
    }
  }
});

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

Similar to the `SwappingRouter`, the `CompositeView` base class solves common housekeeping problems by establishing a convention. In this case, a parent view 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 = Backbone.View.extend({
  tagName: 'section',
  id: 'task',

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

  render: function() {
    this.renderLayout();
    this.renderDetail();
    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 = this.options.model;
    this.model.comments.bind("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:

- Split up like this, 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.

6.9. How to use multiple views on the same model/ collection (chapter unstarted)

6.10. Internationalization (chapter unstarted)

7. Models and collections

7.1. Naming conventions (chapter unstarted)

7.2. Nested resources (chapter unstarted)

7.3. 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 a way that it’s usage is natural.

7.3.1. 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 = {})
    super(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'));
  }
});
```

A couple of improvements to the above: You'll soon realize that you will might be setting the user outside of the initialize as well. Also, the initializer should check whether there 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 whether the provided option is set for the object:

```
var User = Backbone.Model.extend({
  initialize: function() {
    if (this.has('user')) {
      this.user = 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 fo the user instance.

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

7.3.2. 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 = {})
    super(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.

7.4. Scopes and filters

To filter a `Backbone.Collection`, like with Rails named scopes, define functions on your collections that filter by your criteria 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);
  }
});
```

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, you can 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(this.select(function(task) {
      return task.isComplete();
    }));
  },

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

7.5. Sorting

The simplest way to sort a `Backbone.Collection` is to define a `comparator` function:

```
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 resuable 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.createdAt;
    });
  },

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

7.6. Client/Server duplicated business logic (chapter unstarted)

7.7. Validations (chapter unstarted)

7.8. Synchronizing between clients (chapter unstarted)

8. Binding models and views (section unstarted)

9. Testing (section unstarted)

9.1. Full-stack integration testing

9.2. Isolated unit testing

10. The JavaScript language (section unstarted)

10.1. Model attribute types and serialization

10.2. Context binding (JS `this`)

10.3. CoffeeScript with Backbone.js