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 (section unstarted)

4.1. Backbone.js and MVC

4.2. What goes where in MVC

4.3. Namespacing your application

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 primary 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

5.2.1. A note on JSTs and Jammit

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 to the `window.JST` function. 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

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 (chapter unstarted)

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 task management application. 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 1. Tasks index view

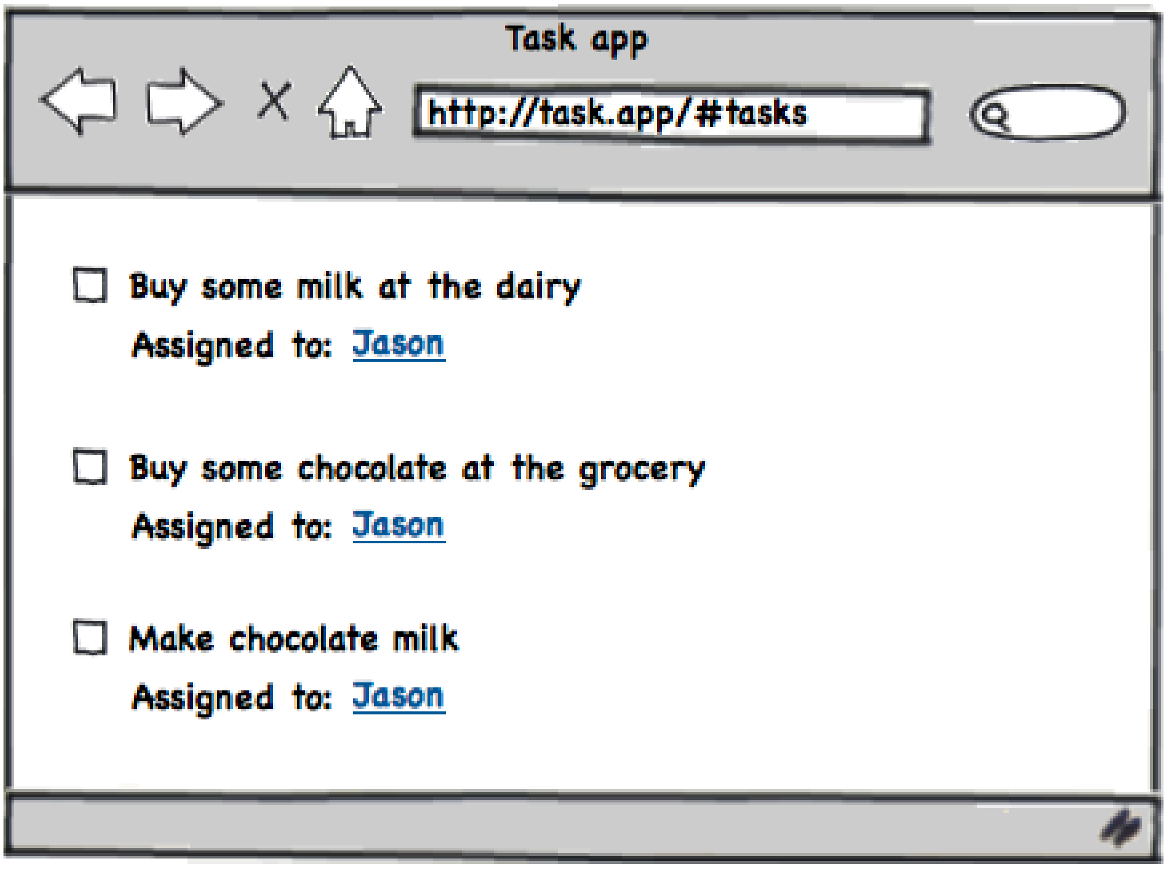
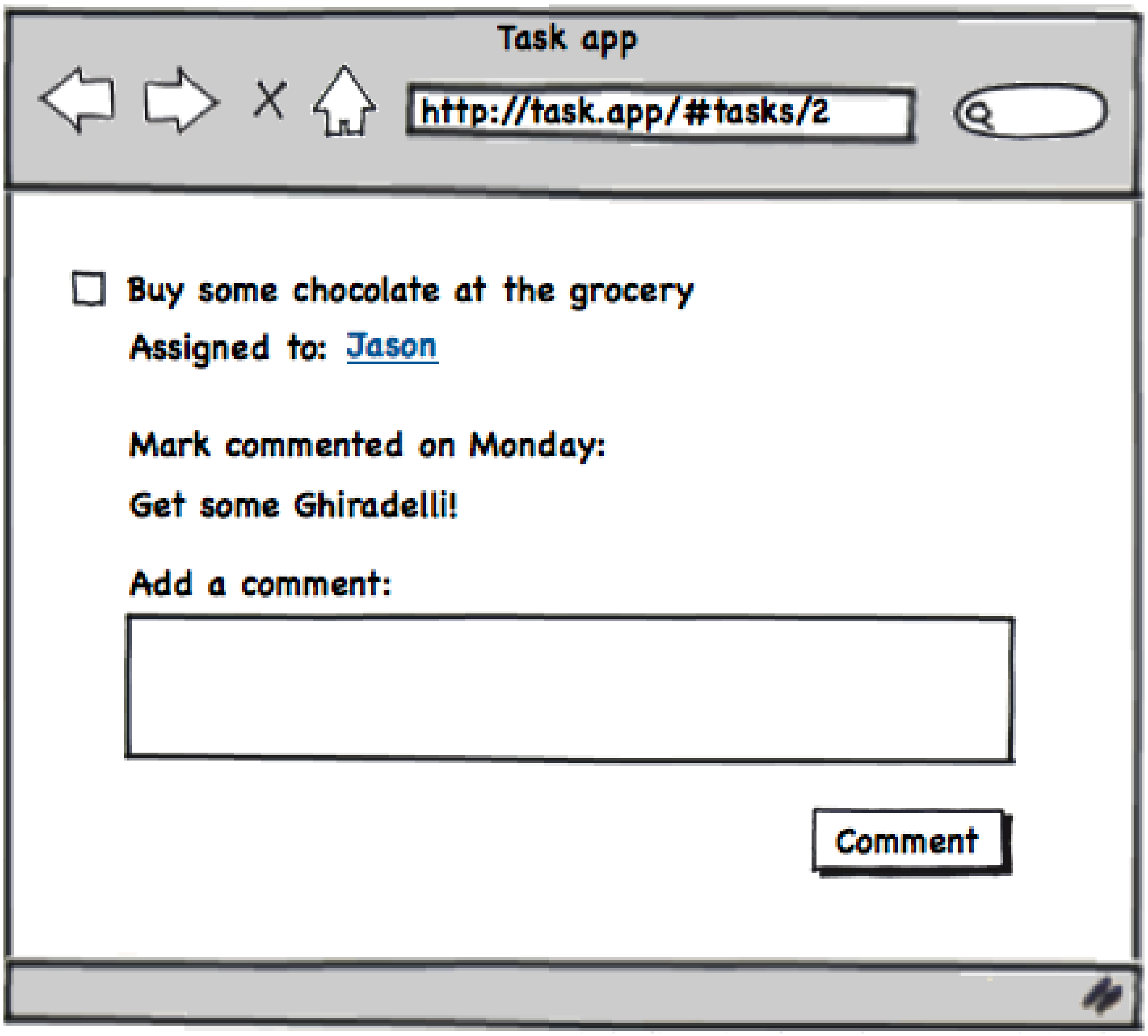


Figure 2. Tasks detail view



The view classes look something like this:

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

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

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

var TasksDetail = Backbone.View.extend({
  tagName: 'section',
  id: 'tasks',
  initialize: function() {
    _.bindAll(this, "render", "renderCompletedTasks", "renderOverdueTasks");
    TaskApp.tasks.bind("change", this.render);
    TaskApp.tasks.bind("add", this.render);
  },

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

TODO: Bind to change event on collection in index and detail. Link to detail from index page. If you don’t unbind index when leaving it, and go to the detail view, and change the model (e.g. check "Completed"), then the index view class re-renders itself. This is undesirable, and can cause visible bugs. Briefly introduce convention of SwappingController. Then, next section is SwappingController.

6.7. Swapping controllers (in progress)

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

_.extend(SwappingController.prototype, Backbone.Controller.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);
  }
});

SwappingController.extend = Backbone.Controller.extend;
```

6.8. Composite views (in progress)

TODO: Refactor the TaskIndex view class in tasks_index_and_detail_view_classes.js to be composite. Discuss other common places you’ll find composite views. Then, motivate the CompositeView superclass by referencing the cleanup issue from before, and noting that even if parent view classes unbind, child view classes may not.

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

_.extend(CompositeView.prototype, Backbone.View.prototype, {
  leave: function() {
    this.unbind();
    this.remove();
    _(this.children).invoke("leave");
  },

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

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

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

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. Relationships (chapter unstarted)

7.4. Scopes and filters

To filter a `Backbone.Collection`, like with Rails named scopes, define functions on your collections that return new collection instances, filtered by your criteria. 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 `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 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. Testing (section unstarted)

8.1. Full-stack integration testing

8.2. Isolated unit testing

9. The JavaScript language (section unstarted)

9.1. Model attribute types and serialization

9.2. Context binding (JS `this`)

9.3. CoffeeScript with Backbone.js