Backbone.js on Rails

Jason Morrison <<jmorrison@thoughtbot.com>>

Table of Contents

1. Preface

2. Getting up to speed

2.1. Backbone.js online resources

2.2. JavaScript online resources and books

3. Introduction

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

4.2. What goes where in MVC

4.3. Namespacing your application

5. Rails integration

5.1. ActiveRecord and Backbone.js

5.2. Converting your Rails models to Backbone.js-friendly JSON

5.3. Organizing your Backbone.js and Rails code side-by-side

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

5.5. Automatically using the Rails authentication token

6. Views and Templates

6.1. View explanation

6.2. Templating strategy

6.3. View helpers

6.4. Form helpers

6.5. Event binding

6.6. How to use multiple views on the same model/collection

6.7. Composed views

6.8. Cleaning up: understanding binding and unbinding

6.9. Internationalization

7. Models and collections

7.1. Naming conventions

7.2. Nested resources

7.3. Relationships

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

7.7. Validations

7.8. Synchronizing between clients

8. Testing

8.1. Full-stack integration testing

8.2. Isolated unit testing

9. The JavaScript language

9.1. Model attribute types and serialization

9.2. Context binding (JS `this`)

9.3. CoffeeScript with Backbone.js