RJS Templates for Rails

By Cody Fauser

....
Publisher: O'Reilly
Pub Date: June 01, 2006
ISBN: 0-596-52809-4
Pages: 56

Table of Contents

Overview

RJS templates are an exciting and powerful new type of template added to Rails 1.1. Unlike conventional Rails templates that generate HTML or XML, RJS templates generate JavaScript code that is executed when it is returned to the browser. This JavaScript generation allows you to perform multiple page updates in-place without a page reload using Ajax. All the JavaScript you need is generated from simple templates written in Ruby. This document helps you get acquainted with how RJS templates fit into the Rails framework and gets you started with a few easy-to-follow examples.

PREV

RJS Templates for Rails

By Cody Fauser

....
Publisher: O'Reilly
Pub Date: June 01, 2006
ISBN: 0-596-52809-4
Pages: 56

Table of Contents

PREV

PREV

Chapter 1. Introduction

PREV

What Is RJS?

Ajax is a technology that allows a more desktop-like, interactive user experience to happen in the web browser. Basically, Ajax allows the browser to make remote requests in the background. These requests update the current page without reloading. Ruby on Rails has excellent support for Ajax baked right into the framework. Remote JavaScript (RJS) templates build upon the Ajax support offered by Rails 1.0, but go one step further, by allowing you to easily update multiple page elements.

RJS templates are a powerful new addition to Rails 1.1. Unlike other Rails templates, which are rendered and sent to the browser, RJS templates are used to update pages that have already been rendered.

RJS was the brainchild of Rails core developer Sam Stephenson. Sam is also the author of the Prototype library that is bundled with Rails.

Rails, prior to 1.1, already had support for updating a single page element with the result of a remot Ajax call, however, things started to get tricky if you wanted to update multiple page elements. RJS allows you to update multiple page elements using Ruby code with a single Ajax request. You can us any number of Scriptaculous visual effects from a single template, and you don't have to write any JavaScript at all. For most cases, there is no longer a need to switch mental contexts and program in JavaScript. All the JavaScript is generated for you by the Rails JavaScript Generator, and the Ajax response is automatically evaluated by the prototype library.

As you can see, RJS is going to make it a lot easier to update multiple page elements and create complex visual effects from a single Ajax request.

PREV

Who Should Read this Document?

This document is for anybody with an interest in using Ajax with the Ruby on Rails framework. This document assumes that you have some familiarity with Ruby on Rails, Ruby, and the common conventions that are used in each. It is also good to have a basic understanding of JavaScript, what Ajax is and why you'd want to include it in your application. If you find at any time that you are struggling to understand any of the material, the best reference available is *Agile Web Development with Rails* by Dave Thomas et al. (Pragmatic). This book gives a thorough overview of all aspects of Rails and provides a great background to the material I present here.

PREV

Rails Version

RJS templates require Rails 1.1 or greater. If you are running anything less than Rails 1.1 then you'll need to update your installed version of Rails to the latest version. If you are updating an existing project to use RJS templates, you must update your project's JavaScript libraries. This can be accomplished using a Rake task included with Rails:

```
cody> rake rails:update
```

This Rake task will update your project's configs, scripts, and JavaScripts with the latest versions from Rails.

PREV

Acknowledgments

First, I would like to thank everybody who put up with my absence during the writing of this document. This includes jaded Pixel, Michael Goodman, all of my friends, and especially Maria, my wonderful new wife. I would also like to thank Tobias Lütke and Daniel Weinand for sanity checking my work and providing me with countless ideas. Thanks to Rick Olson and Marcel Molina Jr. for doing great technical reviews. Also, I would like to say thank you to the Bridgehead on Elgin St. in Ottawa, where most of this work took place. The Bridgehead staff never once commented about my loitering and they tried to cut off my supply of Americanos only once. Last but not least, I would like to thank the Rails core team and the countless other contributors for making Rails the amazing framework an community that it is.

PREV

Chapter 2. Getting Started with a Simple Application

To get our feet wet, let's start with a simple introductory example. I am going to call the application "Thought Log." Thought Log will simply take what is entered in a text field and log it into the current pa without reloading. First, create the new Rails project.

```
cody> rails thought_log
```

Now, let's generate a new controller, `ThoughtsController` , that will hold our actions.

```
cody> ruby script/generate controller Thoughts
          exists  app/controllers/
          exists  app/helpers/
          create  app/views/thoughts
          create  test/functional/
          create  app/controllers/thoughts_controller.rb
          create  test/functional/thoughts_controller_test.rb
          create  app/helpers/thoughts_helper.rb
```

The generator has created the controller, helper, and a folder for the controller's views. Now we'll add tv actions. The first action, `index()` , displays the initial empty list of thoughts. The second action, `log()` , is called in the background using Ajax and the response adds the new thought to the already rendered ind page.

```
class ThoughtsController < ApplicationController
  def index
  end

  def log
    @thought = params[:thought]
  end
end
```

The `log()` action simply assigns the value of `params[:thought]` to the instance variable `@thought` and the renders the view template `app/views/thoughts/log.rjs` . We are following the Rails convention that the controller by default renders a view template with the same name as the action that is executing. Since templates are just another template, Rails looks in the controller's views folder for a template named `log` . The only drawback to this is that *.rhtml* and *.rxml* templates will be found and rendered before *.rjs* templates when multiple templates with the same name exist in the view folder. The following are equiv to Rails:

```
def log
end

def log
  render :action => 'log'
end
```

It is also worth noting that you don't need to declare the index() action, since Rails calls the index() acti
and renders the view template index.rhtml in the controller's view folder by default.

NOTE

RJS templates are dependent on the prototype.js JavaScript library. If you want to use all of the Scriptaculous visual effects and controls, you'll also need effects.js , controls.js and dragdrop.js . You can include them all in your project by adding javascript_include_tag :defaults to your view. This will also include your application.js file if it exists.

Next we'll create the initial view. This is the page that creates the remote Ajax request. Normally you wo
create a layout for your application, but since this example is a one-shot deal, we'll just put everything i
the view template. Create index.rhtml in the view folder app/views/thoughts .

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <title>Thought Log</title>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    <h1>My thoughts</h1>

    <%= form_remote_tag :url => { :action => 'log' }, :html => { :id => 'thought-form' }
    <%= text_field_tag 'thought', nil, :size => 40 %>
    <%= submit_tag 'Log thought' %>
    <%= end_form_tag %>

    <div id="thoughts"></div>
  </body>
</html>
```

Including the Rails JavaScript libraries is very simple when you use the helper method javascript_include_tag . Passing the symbol :defaults as the parameter instructs the helper to include
of the Rails JavaScript libraries, as well as your custom application.js , if it exists.

After including the required JavaScript files, we create a simple form with a single text field. Instead of u
the traditional form_tag() helper, we use the form_remote_tag() helper. The difference is that
form_remote_tag() serializes the form and submits the data in the background using Ajax, instead of jus

posting and reloading the page like form_tag() . The controller action that receives the form data is spec with the :url option. In this case, we're submitting the data to the log() action of the current controller, which is ThoughtsController . We've also given the form an id using the :html option. This allows us to r the form from the RJS template upon the completion of the form submission.

Finally, the view has an empty *<div>* tag with id of thoughts . Once again, the id provides a way for us t reference the element from the RJS template. The thoughts *<div>* is a container for appended thoughts logged by the form. Next, we'll create the actual RJS template and you'll see how everything fits togethe

Now that the main view template is ready to go, we need a small partial template for rendering the logg thoughts. The code placed in the partial could be placed right within the RJS template, but the RJS temp stays cleaner and simpler when a partial is used. Create the partial app/views/thoughts/_thought.rhtml a the following view code.

```
<p>
  <span style="font-size: 0.8em;">[<%= Time.now.to_s(:db) %>]</span>
  <%=h thought %>
</p>
```

The partial is very simple. It just displays the logged thought with the current time within a paragraph element. It is good practice to escape text input by users with the h() method to prevent unwanted scrip from executing in your pages.

NOTE

Never use the :update option when making remote calls to actions that render RJS templates. The :update option instructs Rails to generate an Ajax.Updater Prototype object instead of an Ajax.Request object. The Ajax.Updater updates a single DOM element with the contents of the HTML response. RJS templates return JavaScript to the browser, which must be evaluated to produce the desired effects.

Finally, to create the RJS template, we follow the Rails convention of giving the template the same nam the controller action. This way you don't have to explicitly call render() in the controller. Create app/views/thoughts/log.rjs and add the following code to it.

```
page.insert_html :bottom, 'thoughts', :partial => 'thought'
page.visual_effect :highlight, 'thoughts'
page.form.reset 'thought-form'
```

First of all, where did this page object come from? The page object is actually an instance of the Rails JavaScriptGenerator , which generates all of the JavaScript that is returned to the browser. All RJS met calls are made on the page object. We'll take a closer look at how RJS templates are processed by Rails i the next section.

Now that we have a little bit of background on the page object, we can look at the individual calls made i template. First, the partial template app/views/thoughts/_thought.rhtml is rendered, and the resulting content is inserted into the bottom of the thoughts *<div>* . The first parameter is the position of the new content. There are four positions to choose from, including :before , :after , :bottom , and :top . The content is inserted in relation to the id of the DOM element passed in as the second parameter. The thir

parameter can either be a string literal or parameters to a call to `ActionView#render()` passed in as a ha
Passing in parameters to `render` allows you to modify DOM elements with the output of a rendered parti
template by using the `:partial` option, or by using the output of an inline template using the `:inline` opt
Each successive thought logged appears after all the other previously logged thoughts.

Next, a Highlight visual effect is applied to the entire `thoughts` *`<div>`* (see Figure 1). This effect causes th
entire `thoughts` *`<div>`* to be highlighted using the "Yellow Fade Technique" invented by 37 Signals. The v
effect to apply to the element is chosen by passing in the underscored name of the visual effect as eithe
symbol or a string. The regular options to the `visual_effect()` method can also be passed in as the thir
parameter within a hash. Finally, an RJS class proxy is used to reset the form by proxying a call to the
Prototype static form helper method `Form.reset()` and passing in forms the `id` of the form. We'll discuss
proxies further in the next section.

Figure -1. View with a remote form tag. Form data is submitted in the background with Ajax and the page is updated without reloading.

It sure didn't take very much code to create the Thought Log application. The nice thing is that the
`JavaScriptGenerator` did all of the hard work for us. Even nicer is that we get to code the RJS template i
Ruby and the generator takes care of the JavaScript generation.

Figure 2 shows the interaction between the browser and Rails.

Figure -2. The overall request flow of the Thought Log application.

In the next section we'll take a closer look at how RJS fits into the rest of the Rails framework. After tha
we'll start looking at some of the other ways RJS can help our projects.

Chapter 3. RJS and Rails

Like everything else in Rails, RJS is well integrated into the framework. In this section we'll take a look at how RJS templates fit in with the rest of the Rails.

PREV

Debugging

What would software be without bugs? Fortunately, Rails provides a few built in mechanisms to help you locate the source of your problems and get your bugs resolved quickly.

Development Mode Debugging

By default, in development mode all RJS calls are wrapped in JavaScript`try`/`catch` blocks. The `TRy`/`catch` blocks catch all exceptions that occur during the execution of the RJS JavaScript response. When exceptions do occur, details about what went wrong are presented on a series of two alert boxes. The exception itself is detailed on the first alert box. The second alert box shows the code tha generated the exception.

Debugging is controlled by the configuration parameter`config.action_view.debug_rjs` in `environments/development.rb`. If you'd like to disable debugging in your development environment, set this parameter to `false` and restart your development web server.

Monitoring the Logfile

If Rails raises an unhandled exception while processing an Ajax request, the response will be an HTM error page instead of the generated JavaScript code that you really wanted. The easiest way to debug these problems is to monitor the logfiles. An easy way to monitor your logs is to use the Unix or Linux `tail` command. Execute the following from your project's root directory to have`tail` monitor the logfile.

```
cody> tail -f log/development.log
```

The `-f` flag tells `tail` to output appended data as the file grows.

PREV

ActionController

RJS templates fit seamlessly into the Rails framework, just like RHTML and Builder templates. This means that RJS templates are rendered just like the other type of template. The same conventions are followed and most of the same `render` options can be used in your controller actions. The following is an overview of how RJS works with `ActionController`.

Default View

By default, the controller searches for a template with same name as the action that is executing. One caveat is that the controller will find any `.rhtml` or `.rxml` templates before finding the `.rjs` template. If you have a template named `product.rhtml` and another template named `product.rjs`, `ActionController` will render `product.rhtml` and send it as the response with a `Content-Type` header of `text/html`. This won't create the results you expected and may be tricky to debug, since you won't receive any errors in the log. If you do have have both a `product.rhtml` and a `product.rjs` template you can specify which type of template you want to render by including the file extension when specifying the action.

```
def product
 # skip product.rhtml or product.rxml if either exists
 render :action => "product.rjs"
end
```

As you can see from the code, we've explicitly specified that we want to render an RJS template by tacking on the extension .rjs. Another way to configure the type of response you'd like to return is with the new `respond_to()` block. The `respond_to()` block will return the appropriate response depending on the HTTP Accept header. The remote Ajax request made by the Prototype library specifies the Accept header as `text/javascript`, `text/html`, `application/xml`, `text/xml`, `*/*`. This tells Rails that the Ajax request prefers a JavaScript response, but will accept the other types in the list in decreasing preference. The following code sample will help to illustrate this concept more clearly.

```
def product
  respond_to do |format|
    format.html # all html requests
    format.js   # all ajax requests
    format.xml  # all XML requests
  end
end
```

The `respond_to()` block above returns the correct content based on the Accept header and also illustrates that we have three types of content available: HTML (`product.html`), JavaScript (`product.rjs`) and XML (`product.xml`). You can also further customize the `respond_to()` block by specifying more details in a code block following each type.

```
def product
  respond_to do |format|
    format.html { flash["notice"] = 'here is a product' }
    format.js { render :action => "product_rjs.rjs" }
    format.xml { render :xml => @product.to_xml }
  end
end
```

The customizations to the responses in the code above are as follows: Flash content is added to the HTML response, the RJS template to render is specified as product_rjs.rjs in the JavaScript response, and the `to_xml()` feature built into `ActiveRecord` is utilized to generate the XML response instead of using a Builder template in the XML response. As you can see, the combination of the HTTP Accept header with the `respond_to()` block allows you to consolidate a lot of otherwise unwieldy controller code into a single code block.

Layouts

Rails follows the principle of least surprise, so it isn't surprising that`ActionController` is smart enough to skip layouts whenever you render an RJS template. You don't need to worry about passing `:layout => false` to any action that renders an RJS template.

Rendering

There are a lot of options to the `render` call within a controller, but you don't have to worry about most of them with RJS. Table 1 summarizes the options to the`render()` call in `ActionController`, and shows whether or not each particular option is useful in the context of RJS.

Table 3-1. Options available when rendering RJS templates

Option	Works with RJS?	Returns Content-Type = text/javascript?
`:action`	Works as expected	Yes
`:template`	Works as expected	Yes
`:file`	Works as expected	Yes
`:inline`	Not useful for RJS	No
`:partial`	Not useful for RJS	Yes
`:text`	Not useful for RJS	No

Inline Rendering

Rails also has support for inline RJS template rendering to the controller. This saves having to create an entire `.rjs` template file for those really simple one- or two-line tasks.

Inline RJS rendering is performed by passing the `:update` parameter to `render()` from within the controller. What is normally defined in an RJS template is declared in a code block associated with th `render()` call. The code block is passed an instance of the `JavaScriptGenerator`, just like in normal RJS templates. The following controller action update replaces the `innerHTML` of the DOM element with `id header` with the rendered partial header.

```
def update
  render :update do |page|
    page.replace_html, 'header', :partial => 'header'
  end
end
```

Performing the render inline means that you don't have to go searching through your project for a `.rjs` template to see what the action does. Inline rendering does bring view code into the controller. If the inline RJS code is more than one or two short lines, it is advisable to create an explicit RJS template for that functionality.

Browser Redirection

If you need to redirect the browser during an Ajax request, you have to look outside the standard Rails `redirect_to()` method. The Prototype library doesn't respond to HTTP status codes like a browser does, and doesn't follow the 3xx redirect status codes. Fortunately, the `JavaScriptGenerator` also offers a `redirect_to()` method, which generates the JavaScript necessary to redirect the browser.

This new `redirect_to()` method is called on the page object in your RJS template, or an update bloc in your controller. It is probably more common that you will be calling `redirect_to()` using inline RJS in your controller than in your templates, since the `redirect_to()` method is a task more suited to the controller. This new method is used just like the standard `redirect_to()` method that you are used to, except it is called from of the `page` object. Let's take a look at a few examples to show how redirecting with RJS works in practice.

```
render :update do |page|
  page.redirect_to :controller => 'employees', :action => 'list'
end
```

Here we are redirecting the browser to the list action of the `EmployeesController`. Nothing new or surprising, just redirect like you normally would, but make the redirect method call on the `page` object, so that the `JavaScriptGenerator` can generate the appropriate JavaScript. Of course, you have to put the code within a controller action, or without the `render :update` block in an RJS template.

You can also redirect to an absolute URL by passing in a string containing the URL.

```
render :update do |page|
  page.redirect_to 'http://www.shopify.com'
end
```

PREV

ActionView

RJS templates are rendered in an implicit `update_page()` block that is passed an instance of the Rails `JavaScriptGenerator` named `page`.

Our RJS template from the "Thought Log" example gets transformed by `ActionView` to the following before it is executed.

```
update_page do |page|
  page.insert_html :bottom, 'thoughts', :partial => 'thought'
  page.visual_effect :highlight, 'thoughts'
  page.form.reset 'thought-form'
end
```

`ActionView` saves us two lines of code in every template by wrapping the RJS code in the `update_page()` block for us.

Many RJS methods take a variable length parameter list, `options_for_render`. If this parameter is a `Hash`, the parameters are passed through to a call to `ActionView#render()`. This allows you to update DOM elements with a `String` or a rendered template.

When passing in the `:partial` option in an RJS call, you can also pass in the other parameters `:object` and `:locals` that you're used to using with RHTML templates. The following partial could be used to render information about a user.

```
<div id="user">
  <p>Name: <%= name %></p>
  <p>Title: <%= title %></p>
</div>
```

This partial requires the local variables `name` and `title`, so you pass them in using the `:locals` hash right in the RJS method call.

```
page.replace_html 'user', :partial => 'user',
                          :locals => { :name => 'Cody Fauser',
                                       :title => 'El Presidente' }
```

Element, Class, and Collection Proxies

When RJS was first introduced, the only way of manipulating the page's DOM objects was to call methods on the page object and pass in the `id` of the object to perform an action on. Element and collection proxies introduce a new way of interacting with the DOM objects using the Proxy design pattern. The proxy provides access to a DOM element or elements from within an RJS template.

Element Proxies

The element proxy adds the [] method to the JavaScriptGenerator. The proxy is similar to calling methods like visual_effect() on the page object, except that the methods are proxied to an actual DOM object in the generated JavaScript. The proxy objects are accessible by their DOMid from the page object. The following code hides the DOM element withid header when called from an RJS template:

```
page['header'].hide
```

The elements are also accessible by Symbol, so the following would also work:

```
page[:header].hide
```

The proxy also supports method chaining, so you can chain together multiple methods:

```
page[:header].hide.show
```

Class Proxies

The RJS class proxy provides access to JavaScript class methods. The classes may be defined in Prototype, such as the class Form, or defined in your own custom scripts. Instead of selecting a DOM object by its id, as with element proxies, you use the class proxy by chaining methods directly to the page object.

A practical use for class proxies is the manipulation of forms using Prototype'sForm class. The Form class contains support for form disabling, enabling, resetting, and many other things:

```
page.form.reset('employee-form') # => Form.reset("employee-form");
```

However, the class proxies aren't only valuable when working with Prototype. You can use them to call your own static JavaScript methods. Let's say you have the following JavaScript class defined in your public/javascripts/application.js file.

```
var Alerter = {
  displayMessage: function(text) {
    alert(text);
  }
}
```

With the Alerter class defined, you can use the RJS class proxying to call the staticdisplayMessage() function from an RJS template.

```
page.alerter.display_message('Welcome') # => Alerter.displayMessage("Welcome");
```

The class proxies provide a nice bridge between complex JavaScript that you've encapsulated in your JavaScript libraries and ease of use with RJS.

Collection Proxies

The `select()` method returns an array of DOM objects. `select()` takes as an argument a CSS-based selector and returns an array of DOM objects matching that selector. So to get all paragraphs within a *<div>* with `id content`, we would use:

```
page.select('#content p')
```

Now combine this with the enumerable methods available through the collection proxy and you can do things such as:

```
page.select('#content p').each do |element|
  element.hide
end
```

All *<p>* elements within the DOM element with `id content` will be hidden.

Support for attribute-based selectors was also recently added, which allows you to select elements based on their CSS attributes.

```
page.select('#form input[type=text]').each do |element|
  element.hide
end
```

This code hides all of the input elements of type text that have an element with`id form` as an ancestor.

The attribute selectors support `=`, `~=`, `|=`, `existence`, and `!=`. You can use multiple selectors at the same time, such as `input[class=link][href="#"]`. The only downside to the selectors support is that they will not work in Internet Explorer.

Making Ajax Calls with Rails

RJS wouldn't be very useful if you couldn't make Ajax calls from your pages. Rails offers many ways to perform Ajax calls and each is particularly suited to a particular situation. See the Ruby on Rails documentation for more information on the following methods.

link_to_remote(name, options = {}, html_options = {})

This is most common way to make an Ajax call with Rails. `link_to_remote()` is a helper that generates a hyperlink that makes an Ajax request when clicked. Rails generates either an `Ajax.Request` or an `Ajax.Updater` in the `onclick()` event of the `<a>` tag, depending on whether or not the `:update` option was passed to `link_to_remote()`. For the purposes of RJS, the `:update` option should not be used because the Prototype `Ajax.Updater` object expects an HTML response and RJS returns a JavaScript response. If you are having any weird problems with parts of your RJS response appearing on your page, then you're probably using the `:update` option with an RJS template.

link_to_function(name, function, html_options = {})

Generates a hyperlink that executes a JavaScript function or code when clicked. This doesn't actually create an Ajax request, but it can be used to execute custom JavaScript functions that do. Use this method to call your custom JavaScript libraries that use `Ajax.Request` to make the Ajax calls.

observe_field(field_id, options = {})

Observes a field and makes an Ajax request when the content has changed or the event specified with `:on` has occurred.

remote_function(options)

Generates the JavaScript to make a background Ajax request to a remote controller action. This method is useful for making remote Ajax calls from the event handlers of DOM objects, such as the `onchange()` event of a `<select>` tag. Takes the same options as `link_to_remote()`.

observe_form(form_id, options = {})

Works just like `observe_field()`, but observes an entire form.

form_remote_tag(options = {})

Creates a form tag that submits the contents of the form using a background Ajax request. This is another very common way to make Ajax requests.

form_remote_for(object_name, object, options = {}, &proc)

Just like `form_remote_tag()`, but uses the form_for semantics introduced in Rails 1.1.

submit_to_remote(name, value, options = {})

Creates a button that will submit the contents of the parent form to the remote controller action. The options are the same as for `form_remote_tag()`.

in_place_editor_field(object, method, tag_options = {}, in_place_editor_options = {})

Makes an Ajax request when changes to the field are committed. To use this method with RJS you have to pass in the option `:script => true` to the `in_place_editor_options` hash.

in_place_editor(field_id, options = {})

This is the method that the `in_place_editor_field()` wraps. To use this method with RJS you need to pass in the `:script => true` option to the `options` hash.

drop_receiving_element(element_id, options = {})

Makes an Ajax request when droppable elements are dropped onto the element.

sortable_element(element_id, options = {})

An Ajax request is made whenever this element is sorted using drag and drop.

Ajax.Request(url, options)

All Rails helpers use this Prototype object to make the actual Ajax requests. You can also use this object to make remote Ajax requests from your JavaScript libraries. This is a JavaScript object and not a Ruby object.

PREV

Helpers

Rails helpers can be used within RJS templates to group multiple page operations into one method. The only difference between traditional helpers and RJS helpers is that RJS helpers must make use o the implicitly available `page` object.

```
def insert_item(list_id, item)
  page.insert_html :bottom, list_id, '<li>#{item.title}</li>'
  page.visual_effect :highlight, 'item_#{item.id}', :duration => 0.5
end
```

Now, instead of always calling `insert_html()` and `visual_effect()` in every template, we can just call the method `insert_item()`, which will make for simpler, more readable templates. The new helper method can be called from an RJS template as follows, where `@item` is an instance variable available to the RJS template and `my_list` is the unordered list we want to append the item to:

```
page.insert_item 'my_list', @item
```

Chapter 4. RJS in Practice: The Expense Tracker

So far we have completed the "Thought Log" application and taken a look at how RJS fits into the Rails framework. Now it is time to examine an example that is a bit more realistic and solves some o the problems that you might actually run into in your own projects. My expenses have been getting out of hand lately, so let's build a simple application to help track them.

PREV

Creating the Models

First, we'll run the Rails model generator to create the models used throughout this project. The Rails model generator automatically creates stub files for the models and database migrations. Then we'll edit the generated files to add our own functionality.

```
expenses> ruby script/generate model Project
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/project.rb
      create  test/unit/project_test.rb
      create  test/fixtures/projects.yml
      create  db/migrate
      create  db/migrate/001_create_projects.rb

expenses> ruby script/generate model Expense
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/expense.rb
      create  test/unit/expense_test.rb
      create  test/fixtures/expenses.yml
      exists  db/migrate
      create  db/migrate/002_create_expenses.rb
```

The generator creates the Project model in `app/models/project.rb` and the Expense model in `app/models/expense.rb`, along with unit test stubs and test fixtures. The generator also created two migrations for us: `db/migrate/001_create_projects.rb` and `db/migrate/002_create_expenses.rb`.

Now that the model generator has created these two new migrations, we need to add the column definitions that will be used by the models as attributes. For now we'll only track the name of each project. Open up `db/migrate/001_create_projects.rb` and edit it to look like this:

```
class CreateProjects < ActiveRecord::Migration
  def self.up
    create_table :projects do |t|
      t.column :name, :string
    end
  end

  def self.down
    drop_table :projects
  end
end
```

We only added a single line, `t.column :name, :string`, to the migration. This line adds the column `name` of type `String` to the database table `projects`. Next, define the columns for the `expenses` table. Same routine: open up `db/migrate/002_create_expenses.rb` and add the columns `project_id`, `description` and `amount`.

```
class CreateExpenses < ActiveRecord::Migration
  def self.up
    create_table :expenses do |t|
      t.column :project_id, :integer
      t.column :description, :string
      t.column :amount, :float
    end
  end

  def self.down
    drop_table :expenses
  end
end
```

Assuming that the database has already been created and the database connection has been configured, we can run the migrations. This will add the tables and columns defined in the two migration files to the development database configured in `config/database.yml`.

```
expenses> rake migrate
```

Now that the database contains the schema for the Expense Tracker we can define the relationships between the models. A `Project` has many `Expense` objects, so add the `has_many()` relationship to the `Project` model in the file `app/models/project.rb`.

```
class Project < ActiveRecord::Base
  has_many :expenses, :dependent => :delete_all
end
```

We added the `:dependent => :delete_all` option to the `has_many()` call because we don't want any orphaned expenses lingering around in our database without a Project. Now define the `belongs_to()` relationship in the `Expense` model. An Expense object `belongs_to()` a Project because the Expense contains the foreign key. Open up `app/models/expense.rb`.

```
class Expense < ActiveRecord::Base
  belongs_to :project
end
```

Now that the models are defined and the database is ready to go we can move on to the next step generating and defining the controllers.

PREV

Defining the Controllers

Let's generate two controllers. The first controller is for projects and the second is for expenses.

```
expenses> ruby script/generate controller Projects
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/projects
      exists  test/functional/
      create  app/controllers/projects_controller.rb
      create  test/functional/projects_controller_test.rb
      create  app/helpers/projects_helper.rb

expenses> ruby script/generate controller Expenses
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/expenses
      exists  test/functional/
      create  app/controllers/expenses_controller.rb
      create  test/functional/expenses_controller_test.rb
      create  app/helpers/expenses_helper.rb
```

Once again, using the generator simplified the process of adding new functionality to the project. The controller generator not only automatically creates the controller file requested, but also creates the view folder and functional test stub for the controller.

The `show()` action simply finds the Project object from the value associated with the key `:id` in the `params Hash`. Edit `app/controllers/projects_controller.rb` and add the following code to the skeleton that the generator created for us.

```
class ProjectsController < ApplicationController
  def show
    @project = Project.find(params[:id])
  end
end
```

Next, edit `app/controllers/expenses_controller.rb` and add the code to create a new `Expense` object.

```
class ExpensesController < ApplicationController
  before_filter :find_project

  def new
    @expense = @project.expenses.create(params[:expense])
  end
  private
  def find_project
    @project = Project.find(params[:project])
  end
end
```

The ExpensesController is a bit more complex than the ProjectsController. Since every Expense object belongs to a Project we can save a lot of effort and duplicate code by using a before_filter. The before_filter executes before each controller action. The filter we've defined automatically finds the Project based on the value associated with the :project key in the params Hash and stores it in the instance variable @project.

The new action, as the name indicates, adds a new Expense object to a Project. Rails follows the same convention for RJS templates as for RHTML and RXML templates. Rails looks for a template witl the same name as the controller action with the corresponding file extension. So in the case of the new action, the controller automatically discovers the view app/views/expenses/new.rjs.

PREV

PREV

Setting Up a Route

One more step is needed to wire up the `before_filter` and create nice-looking URLs. The controller looks up the project using the value stored in `params[:project]`. We need to add a simple route to catch this and produce nice URL paths like `/projects/1/expenses/new`.

Open up `config/routes.rb` and add the following line above the default route at the bottom of the file:

```
map.expenses 'projects/:project/expenses/:action/:id', :controller => 'expenses'
```

The change to the routing should be picked up right away if you're running your application in development mode. Moving right along, we're going to create a layout for all of our application templates.

PREV

Creating an Application Layout

We need a layout to put our content in. ActionController::Base descendants automatically look for a layout based on the name of the controller's class name, with Controller removed. This means that ApplicationController will automatically use a layout named application.rhtml. Our controllers are all descendants of ApplicationController, so the layout will inherit as well, unless overridden. Create app/views/layouts/application.rhtml and add the following code with your favorite text editor:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <title>Expense Tracker</title>
    <%= stylesheet_link_tag "screen.css" %>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    <div id="content">
      <%= yield %>
    </div>
  </body>
</html>
```

Notice that we are also using the Rails helper stylesheet_link_tag() to include the stylesheet screen.css, which is found in public/stylesheets/screen.css. The stylesheet I'm using is very basic and looks like the following:

```
th { text-align: left; }
#content { margin: 10px; }
#content p { margin-bottom: 10px; }
#expenses,#summary { border: none; border-collapse: collapse; width: 600px;}
.amount { width: 40%; }
.amount, .total { text-align: right; }
#new-expense { margin-top: 2em; background-color: #eeede5; padding: 1em; }
#new-expense h3 { margin-top: 0.5em; }
#total { border: none; border-collapse: collapse; width: 600px;}
#total-amount, .total { font-weight: bold; background-color: #eeede5; }
#total-amount { border-top: 2px solid black; }
.total { width: 90%; padding-right: 10px; }
```

Just because my stylesheet is basic and boring doesn't mean that yours has to be. Dress up your Expense Tracker to your heart's content.

PREV

PREV

Entering Some Data

We need some data to display on our page, so I'll enter a few expenses that I incurred while writing this document. Start up the console.

```
expenses> ruby script/console
>> rjs_book = Project.create(:name => 'RJS Templates for Rails')
=> #<Project:0xb72e444c8 ...>
>> rjs_book.expenses.create(:description => 'Americano at Bridgehead', :amount => 1.93)
=> #<Expense:0xb72cb84c ...>
>> rjs_book.expenses.create(:description => 'Sandwich at La Bottega', :amount => 4.27)
=> #<Expense:0xb72c3b24 ...>
>> quit
```

I don't think I'll get away with writing off expenses like those, but at least I'll be able to get a better ide about where my money is being spent. Now we need some views to present this sample data.

PREV

Creating the Views

There isn't yet a way for us to view our expenses for this`Project`. We need to create a view that shows the `Expense` objects for the sample `Project` that was just created. The views are separated intc one template and two partial templates. We specifically separate out the`_expense.rhtml` partial so that we can render an individual row when updating the table with RJS.

Create the view `app/views/projects/show.rhtml` and add the following:

```
<h1><%= @project.name %></h1>

<h2>Expenses</h2>
<table id="expenses">
  <tr><th>Description</th><th class="amount">Amount</th></tr>
  <%= render :partial => 'expenses/expense', :collection => @project.expenses %>
</table>

<%= render :partial => 'expenses/new' %>
```

This partial gives the table the `id` `expenses` so that we can refer to it when updating the page. We've also given a relative path to the partial, because we're rendering`show.rhtml` from `ProjectsController`, but we're keeping the expense partials in the`app/views/expenses` view folder.

Now we create partial `app/views/expenses/_expense.rhtml`. This partial renders the actual `Expense` within the `<table>`.

```
<tr id="expense-<%= expense.id %>">
  <td><%=h expense.description %></td>
  <td class="amount"><%=h number_with_precision(expense.amount, 2) %></td>
</tr>
```

Notice how each row is given an `id` based on the `Expense` object's `id` attribute. We've done this for the same reason that we gave the `<table>` an `id`: it allows us to refer to the row in the future. The method `number_with_precision()` is just a built-in Rails helper method that displays the amount with the specified number of decimal places. We also used the`h()` method, which escapes the HTML rendered on the page. Escaping the HTML prevents a malicious user from adding JavaScript scripts tc the project's title.

Last, but not least, we'll add the partial for the form. Place the following code into `app/views/expenses/_new.rhtml`:

```
<div id="new-expense">
  <h3>Add an expense</h3>
  <% form_remote_for :expense,
                     Expense.new,
                     :url => hash_for_expenses_url(:project => @project,
                                                   :action => 'new'
                                                  ),
                     :html => { :id => 'expense-form' } do |f| %>
    <label for="expense_description">Description:</label><br />
    <%= f.text_field 'description', :size => 60 %><br />

    <label for="expense_amount">Amount:</label><br />
    <%= f.text_field 'amount', :size => 10 %><br /><br />

    <%= submit_tag 'Add Expense' %>
  <% end %>
</div>
```

This isn't a regular form. We've used the `form_remote_for()` method, which posts the data from the form to our controller action in the background with an Ajax request. The first parameter passed is the name of the object; it is the key under which the form data will be located in the `params Hash`. The second parameter is the object that provides the form's initial values. Then we pass in the `:url` option, which tells the form where to post the form data. Notice we call the routing helper method `hash_for_expenses_url()`, which is generated based on our named route `expenses` in `config/routes.rb`. Calling this method saves us from the hassle of specifying the controller name in the `:url Hash`. We also gave the form an `id` of `expense-form` so that it can be referenced from our RJS templates and other JavaScript code.

Finally, let's create the RJS template that updates our project's expense page when we add a new expense. Create `app/views/expenses/new.rjs` and add the following code to it:

```
page.insert_html :bottom, 'expenses', :partial => 'expense'
page.visual_effect :highlight, "expense-#{@expense.id}"
page.form.reset 'expense-form'
```

The first line inserts the HTML rendered by the partial, `_expense.html`, into the DOM element with `id expenses`. The option `:bottom` specifies that the HTML from the partial template will be inserted inside the element, but after the element's existing content. Since there is an instance variable `@expense` and the partial template is also named expense, `@expense` automatically becomes available within the partial as the local variable `expense`, as though we passed in the option `:object => @expense`.

The second line applies the Scriptaculous visual effect Highlight to the new `Expense` object. Finally, the third line resets the expense form using an RJS class proxy. Next, we'll give the Expense Tracker a test drive and see what happens.

PREV

What We Have So Far

Now that everything is ready, let's try out what we have so far. Go ahead and add a new expense to the project. I added the new keyboard I bought this morning and captured the output (see Figure 3).

Figure -1. Updating and highlighting without a page refresh.

Wow, that was so easy. All of that functionality and we didn't even have to write a single line of JavaScript. The exciting part is that this is just beginning! The new expense form is pretty cool, but i would be great to improve the form to show the total expenses or some other interesting summary information. The form could also provide some sort of feedback when a remote request is being processed. It would also be nice to disable the form so that the user doesn't accidentally click the Ad Expense button more than once.

So far we've done a lot of standard Rails tasks and just added a sprinkle of RJS in for flavor. Next, we're going to take a look at FireBug. FireBug is an invaluable tool in any Ajax developer's toolkit. Getting familiar with it will allow you to see exactly what is going in every Ajax request made throughout the rest of the document.

PREV

Chapter 5. FireBug: Awesome JavaScript Utility

The one problem with Ajax-enabled applications is that it is difficult to know what is going behind the scenes. It is even more difficult to know what happened when something goes wrong. Your activity indicator image keeps spinning, but the nothing seems to happen on the page. FireBug will help you figure exactly what went wrong and where.

PREV

PREV

What Is FireBug?

FireBug is an extension for Firefox 1.5 that was created by Joe Hewitt and it helps with the debuggin of JavaScript and Ajax. FireBug includes a DOM inspector, a JavaScript console and a command-line JavaScript interpreter, all in one. We're mostly interested in its ability to log Ajax requests. The current release of FireBug is 0.3.2. The 0.4 version is currently in alpha and includes a JavaScript debugger. The addition of the debugger will make FireBug even more useful and harder to live without.

PREV

PREV

Installation

You need Firefox 1.5 to use FireBug. Even if your primary browser isn't Firefox, it is worthwhile to have Firefox with the FireBug inspector installed on your machine just for the purpose of debugging. Installation is very simple. Simply go to http://www.joehewitt.com/software/firebug and click the large "Install FireBug" button and follow the directions. After installation you'll need to restart Firefox for the extension to complete installation.

PREV

A Brief Tour

After reopening your browser, you can change FireBug's visibility by using Firefox's view menu bar at the top of the screen, or by using F12.

Let's add an expense and see what is happening behind the scenes. Ensure that FireBug is enabled and that the monitoring of XMLHttpRequests is enabled. You can enable the monitoring of XMLHttpRequests from FireBug's options menu. Once the monitoring of XMLHttpRequests is enabled, an entry will appear in the FireBug console for every Ajax request that is made.

Now, let's take a brief look at the response that was returned when I added my new keyboard expense (see Figure 4).

Figure -1. Viewing an Ajax response in FireBug.

Clicking the triangle to the left of the response displays three extra tabs:Post, Response, and Headers. The Response tab shows the JavaScript code that Rails returned. Taking a look at the JavaScript responses is a great way to learn the correlation between RJS and the JavaScript it generates. They are also good to review when you don't get the results you expect from an Ajax call. The Post tab is useful when using form_remote_tag and form_remote_for, since you can see the data that was posted from your form to your Rails action. Lastly, the Headers tab shows all of the information about the headers returned with your response. You can double check the Headers tab if your RJS calls don't seem to be executing to make sure that the Content-Type header returned is text/javascript. Prototype will not evaluate the remote response if the Content-Type header isn't set correctly.

In addition to monitoring Ajax requests, FireBug's JavaScript interpreter is also a very powerful tool. The interpreter allows you to execute JavaScript commands in the context of the current page. This means that you can quickly try out features and effects from the Prototype and Scriptaculous libraries without having to make a new HTML document each time. Let's give the command interpreter a try by toggling the visibility of the first expense we added. Type the following into the command interpreter at the bottom of Firebug and hit enter:

```
>>> Element.toggle("expense-1");
```

You should notice the "Americano at Bridgehead" expense appearing and disappearing each time you execute the command. You can even take this a step further and apply visual effects to the current page. Have fun with it; FireBug is a great way to learn about JavaScript, Prototype, and Scriptaculous.

FireBug also allows you to inspect DOM elements. First, click the inspect button and then click an element on the rendered page. This will switch you to the`Inspector` tab, which shows the entire DOM tree. From there you can use the `source`, `style`, `layout` and `events` tabs to view the properties of the chosen element. You can find out pretty much anything that you might want to know about the selected DOM object.

FireBug has support for logging to the console (FireBug version 0.4 includes enhanced support for logging to the console). In versions of FireBug < 0.4 you must define the`printfire()` method as is shown in the following sample. You can define the code in`public/javascripts/applications.js`.

```
function printfire() {
   if (document.createEvent)
   {
      printfire.args =  arguments;
      var ev = document.createEvent("Events");
      ev.initEvent("printfire", false, true );
      dispatchEvent(ev);
   }
}

var Logger = {}

Logger = {
  log: printfire
}
```

Notice that I've created a simple wrapper to the `printfire()` method. This is simply to improve the aesthetics of using the logger. Using the RJS class proxy is a great and simple way to log to the console from your RJS templates.

```
page.logger.log 'Executing the RJS template'
```

NOTE

This was written as version 0.4 was in alpha and therefore doesn't take advantage of the more powerful logging capabilities of the new version. You'll have to use the new `console.log()` and other more enhanced functions instead of `printfire()`.

FireBug is an invaluable tool; the sooner you figure it out, the faster you'll be able to solve Ajax and JavaScript problems in your pages. You should now use FireBug to inspect the JavaScript responses returned by all of the Ajax examples throughout the rest of the book.

PREV

Chapter 6. Enhancing the Expense Tracker

The Expense Tracker currently uses an Ajax call in the background to add`Expense` objects to a `Project`. Although the form works and successfully adds expenses to a project, it could really use some enhancements. Next we're going to add an activity indicator to the page and later on we'll add a summary section that shows some statistics about the project.

PREV

PREV

Ajax Activity Indicator

One problem with Ajax is that it break a user's assumptions about how his web browser works. The user used to having the entire page reload after performing an action that interacts with the server. With no indication that the page is busy, the user is left wondering what is going on. The user may also think tha nothing is happening and repeatedly click the link or button, causing undesired effects.

One solution to this problem is to place some kind of indicator on the page that lets the user know that remote call is in progress. In this example we'll use an animated GIF, but some descriptive text is also used. We can also disable the form while the request is in progress to prevent the user from accidentally clicking the submit button more than once.

We can do all of this by hooking into the JavaScript callbacks offered by the Ajax request. Rails lets you hook into the callbacks by passing in options to the remote call. The available callbacks are: `:uninitialized` , `:loading` , `:loaded` , `:interactive` , `:complete` , `:failure` , and `:success` . See the Rails documentation for more information regarding these callbacks.

Create the file `public/javascripts/application.js` if it doesn't already exist and add the following code it:

```
var ExpenseTracker = {}

ExpenseTracker = {
  disableExpenseForm: function() {
    Element.show('form-indicator');
    Form.disable('expense-form');
  },

  enableExpenseForm: function(form) {
    Element.hide('form-indicator');
    Form.enable('expense-form');
  }
}
```

We've created a new JavaScript object `ExpenseTracker` that will be available from our page. We then added a simple method, `disableExpenseForm()` , that shows a spinning indicator and disables the form while the Ajax request is loading. We also added`enableExpenseForm()` , which hides the indicator and enables the form when the request is complete. We could have called these methods directly without th `ExpenseTracker` object, but we'll be adding more functionality to each method in the future and it is nice keep the functionality encapsulated in one place. I also like the fact that the code that manages the forr won't be cluttering up the RJS templates.

Since we used `javascript_include_tag :defaults` in our layout, Rails is smart enough to include `public/javascripts/application.js` along with the Rails JavaScript libraries. Now that our simple JavaScript functions are ready to go, we can hook them into the callbacks offered by the remote reques Open up `app/views/expenses/_new.rhtml` and modify it to include the callbacks. The `form_remote_for()`

method call should look like this when you've finished:

```
<% form_remote_for :expense,
                   Expense.new,
                   :url => hash_for_expenses_url(:project => @project, :action => 'new')
                   :loading => 'ExpenseTracker.disableExpenseForm()',
                   :complete => 'ExpenseTracker.enableExpenseForm()',
                   :html => { :id => 'expense-form' } do |f| %>
```

The form is given the `id` `expense-form` so that we can refer to it within the JavaScript function. We could achieve the same results that we get from the `enableExpenseForm()` method by calling the equivalent methods in the RJS template, but it is nice to keep all of the callback code in the `ExpenseTracker` object. This also keeps the mechanics of managing the form from cluttering up the RJS templates. We could als reset the form in the `enableExpenseForm()` method, but this would always reset the form. We want to be able control when the form is reset so that the user doesn't have to re-enter the data in the case of faili validations or other problems.

Next, add the indicator image after the `submit_tag()` call in `app/views/expenses/_new.rhtml` . Set the initial style to `display:none` so that the indicator isn't visible when the page is first loaded. My indicator i just a simple animated GIF that mimics Mozilla Firefox's spinning indicator. I placed the image in `public/images` folder so that it is available to our project.

```
<%= image_tag 'indicator.gif', :id => 'form-indicator', :style => 'display:none;' %>
```

Now when submitting the form, the browser displays the spinning indicator image and disables the form (see Figure 5). This technique offers a visual cue that an Ajax request is in progress and prevents the us from submitting the new expense multiple times. The indicator is hidden and the form re-enabled when request has completed. Since the entire process occurs very quickly in the normal case, it can be very hard to see that the form is being disabled and the indicator shown. You can slow things down a bit by putting a call to `sleep()` in the `new()` action of the `ExpenseController` . Obviously you'd only want to do this in development to test that the process is actually working. The controller code with a call to `sleep()` is shown in the following sample:

```
class ExpensesController < ApplicationController
  before_filter :find_project

  def new
    @expense = @project.expenses.create(params[:expense])
    # Sleep for 3 seconds
    sleep 3
  end

  private
  def find_project
    @project = Project.find(params[:project])
  end
end
```

Figure -1. The expense form is disabled and an indicator is shown while the Ajax request is in progress.

PREV

Ajax Global Responders

The :loading and :complete callbacks of the form_remote_tag() worked very well for showing and hidin the form indicator. The only problem is that if you have a lot of Ajax functionality built into your page, it can be tedious to add the image and wire up the :loading and :complete callbacks for every remote operation. This is where the Ajax global responders come in handy.

The Prototype Global Ajax Responders are a great place to register JavaScript functions that you would l to have executed on every Ajax request. The Ajax global responders are provided by the Prototype libra they allow you to hook functions into the various callbacks of all Ajax requests. Let's move code that shows and hides the indicator image out of the :loading and :complete callbacks and instead use the Aja global responders.

Instead of placing an indicator image beside every form or element that makes Ajax calls, the Ajax Glob Responders allow you to set up a single indicator on the page that will be shown during any Ajax reques made from the page.

Setting up the Global Responders is really simple. You can add the code to wire up a global response in public/javascripts/application.js :

```
Ajax.Responders.register({
 onCreate: function() {
   if (Ajax.activeRequestCount > 0)
     Element.show('form-indicator');
 },
 onComplete: function() {
   if (Ajax.activeRequestCount == 0)
     Element.hide('form-indicator');
 }
 });
```

The code is straightforward. Ajax.Responders.register() takes an anonymous JavaScript object, where the property name is the name of the Ajax callback and the value is a JavaScript function. We are wiring up a function that will be executed on every onCreate() callback and another function that will be execut on every onComplete() callback. The first function shows the DOM element with id form-indicator when there is one or more active Ajax Request. The second function hides the indicator when there are no act Ajax requests processing.

We can now either remove the lines that show and hide the form indicator image from the ExpenseTracke object in public/javascripts/application.js or we can remove the ExpenseTracker JavaScript code altogether and just write the code inline in the callback. This is what the form_remote_for call in app/views/expenses/_new.rhtml would look like if we eliminated the ExpenseTracker JavaScript object an simply wrote the code inline:

```
<% form_remote_for :expense,
                   Expense.new,
                   :url => hash_for_expenses_url(:project => @project, :action => 'new')
                   :loading => 'Form.disable("expense-form")',
                   :complete => 'Form.enable("expense-form")',
                   :html => { :id => 'expense-form' } do |f| %>
```

Now the callbacks only enable and disable the form. The code to show and hide the indicator image is being executed by the Ajax Global Responders. The indicator appears whenever there is any Ajax activit and is hidden when all requests have completed. When we add more features, it is a good idea to move the indicator image to another part of the screen, or use an animated LightBox image. For now, we'll jus leave the image in the same location.

Ajax Global Responders offer a great way to perform actions during the life cycle of every Ajax request. This not only cuts down on code duplication, which makes our templates more easily understandable, bu also saves a whole lot of typing.

PREV

PREV

Model Validations

In its current state, the Expense Tracker will accept any input and try to create`Expense` objects. The problem is that the application chokes on invalid input. The most likely case is`ActiveRecord` tHRowing an exception that isn't caught by our code. Our newly-added Ajax indicator will just keep spinning away and the user won't know what happened. Lucky for us, Rails has wonderful support model validations. We can validate the new`Expense` objects and return a nice alert box to the user showing any problems. Let's add some validations to the`Expense` model in `app/models/expense.rb` :

```
class Expense < ActiveRecord::Base
  belongs_to :project

  validates_presence_of :description
  validates_numericality_of :amount

  protected
  def validate
    errors.add(:amount, "must be greater than 0") unless amount.nil? || amount >= 0.01
  end
end
```

This validation code will ensure that the`description` is not blank and that the amount of the`Expense` is a number greater than 0. Now we just have to modify our RJS template slightly to display the errors. Open up `app/views/expenses/new.rjs` and modify the template to look like the following:

```
if @expense.new_record?
  page.alert "The Expense could not be added for the following reasons:\n" +
             @expense.errors.full_messages.join("\n")
else
  page.insert_html :bottom, 'expenses', :partial => 'expense'
  page.visual_effect :highlight, "expense-#{@expense.id}"
  page.form.reset 'expense-form'
end
```

The code checks to see if the`Expense` object is still a new record. If it is still a new object, then there must have been a problem saving it and the errors are shown. Otherwise, the normal action of inserting and highlighting the new`Expense` is performed. Notice that the form is only reset when the operation was successful. This way the user doesn't have to retype the`description` and `amount` when there are errors.

In this case we just used a simple JavaScript alert box to show the errors. This is the simplest method of displaying the errors with RJS. Another solution would be to replace the entire form and insert the rendered output of `error_messages_for()` into the page. This would take advantage the built-in Rails helpers, but also has more overhead in the RJS templates, as you would have to remove or hide the rendered error section after the`Expense` object was successfully added.

PREV

Adding Some Calculations

Since the Expense form is disabled when a request is being processed, it is a lot more usable. But there's a long way to go. I still don't know what my total expenses for the project are. Also, while we're at it, let's add some code that displays other interesting data, such as the minimum expense, maximum expense, and the average expense of the project. We'll have to make sure that all this additional information gets updated as we add expenses to the project.

First, we need to add these calculation methods to our Project model. Open up `app/models/project.rb` and add the calculation methods. Your model should look something like the this:

```
class Project < ActiveRecord::Base
  has_many :expenses, :dependent => :delete_all

  def total_expenses
    expenses.sum(:amount)
  end

  def min_expense
    expenses.minimum(:amount)
  end

  def max_expense
    expenses.maximum(:amount)
  end

  def avg_expense
    expenses.average(:amount)
  end
end
```

These methods are all ridiculously simple. We use the power of the new Active Record Calculations (added in Rails 1.1) to do all of the dirty work. Notice that the calculation methods are being called from the `expenses` collection. Calling each calculation from the collection instead of from the`Expense` class causes the calculation to be scoped to the current`Project`, which is what we want in this case. We pass in the `Symbol :amount` to each calculation because that is the `Expense` attribute on which we want to perform the calculation.

We might as well display all of this information on the page that shows the project's expenses. We can show the total expenses using a partial that we'll render directly under the list of expenses. Create `app/views/expenses/_total.rhtml`, which will look like this:

```
<table id="total">
  <tr>
    <td></td>
    <td class="total">Total</td>
    <td id="total-amount" class="amount"><%= number_to_currency(total) %></td>
  </tr>
</table>
```

number_to_currency() is another Rails numerical helper method. It formats the amount passed to it with two decimal places and places a dollar sign before the number. What about the other calculations? We can place them in a table above the list of expenses. Create another partial named app/views/expenses/_summary.rhtml, and add the summary table.

```
<table id="summary">
  <tr>
    <td>Min expense</td><td class="amount"><%= number_to_currency(min) %></td>
  </tr>
  <tr>
    <td>Max expense</td><td class="amount"><%= number_to_currency(max) %></td>
  </tr>
  <tr>
    <td>Ave expense</td><td class="amount"><%= number_to_currency(average) %></td>
  </tr>
</table>
```

Now we need to render the partials we just created. Open upapp/views/projects/show.rhtml and add the lines that render the new partials. The view should look like this:

```
<h1><%= @project.name %></h1>

<h2>Summary</h2>
<%= render :partial => 'expenses/summary',
           :locals => { :min => @project.min_expense,
                        :max => @project.max_expense,
                        :average => @project.avg_expense } %>

<h2>Expenses</h2>
<table id="expenses">
  <tr><th>Description</th><th>Amount</th></tr>
  <%= render :partial => 'expenses/expense', :collection => @project.expenses %>
</table>
<%= render :partial => 'expenses/total', :object => @project.total_expenses %>
<%= render :partial => 'expenses/new' %>
```

We added an *<h2>* header tag for the summary and rendered the summary partial underneath it. We pass the minimum, maximum, and average expense values into the partial as local variables in the locals Hash. Then, after the list of expenses, we render the total table. Notice that we pass @project.total_expenses into the _total.rhtml partial as the value of the :object key. This makes the value @project.total_expenses available in the partial as the local variable total, which is also the name of the partial template.

At this point, we just need to add the code that updates the total expenses and the new summary table. Let's edit the RJS template app/views/expenses/new.rjs again and add the following code:

```
if @expense.new_record?
  page.alert "The Expense could not be added for the following reasons:\n" +
             @expense.errors.full_messages.join("\n")
else
  page.replace 'summary', :partial => 'summary',
                          :locals => { :min => @project.min_expense,
                          :max => @project.max_expense,
                          :average => @project.avg_expense }
  page.insert_html :bottom, 'expenses', :partial => 'expense'
  page.visual_effect :highlight, "expense-#{@expense.id}"
  page.replace_html 'total-amount', number_to_currency(@project.total_expenses)
  page.form.reset 'expense-form'
end
```

First we are replacing the summary table with the rendered _summary.rhtml partial. Since we're using replace, the entire summary element is replaced and not just its contents. Again, like inshow.rhtml, we pass in the :locals Hash that contains the calculations. Then we apply a Highlight effect to the summary. Next, we replace the contents of the total-amount element with the updated total amount. This time we call replace_html, which replaces the innerHTML of the of the DOM element whose id is specified as the first parameter.

Now we have much more functionality in our page. It is great to have summary calculations and tota expenses update automatically. It is amazing that all of this can be done with so little code (see Figure 6).

Figure -2. Instant gratification   updating multiple page elements with new data.

Refactoring with RJS Helpers

Now that the new expenses are being correctly inserted and the total is properly updated, we can go ahead and refactor the RJS template using the RJS helper methods. The separate tasks are displaying the errors to the user, updating the summary, inserting and highlighting a new expense, and updating the total amount. Let's create helpers for these. Open up `app/helpers/expenses_helper.rb` and add the following code:

```
module ExpensesHelper
  def display_errors(expense)
    page.alert "The Expense could not be added for the following reasons:\n" +
               expense.errors.full_messages.join("\n")
  end

  def update_summary(project)
    page.replace 'summary', :partial => 'summary',
                            :locals => { :min => project.min_expense,
                                         :max => project.max_expense,
                                         :average => project.avg_expense }
  end

  def insert_expense(expense)
    page.insert_html :bottom, 'expenses', :partial => 'expense', :object => expense
    page.visual_effect :highlight, "expense-#{expense.id}"
  end

  def update_total(amount)
    page.replace_html 'total-amount', amount
  end
end
```

Now edit `app/views/expenses/new.rjs` to look like this:

```
if @expense.new_record?
  page.display_errors @expense
else
  page.update_summary @project
  page.insert_expense @expense
  page.update_total number_to_currency(@project.total_expenses)
  page.form.reset 'expense-form'
end
```

That certainly makes the code a lot cleaner. I placed the call to`number_to_currency()` outside the helper because the `ActionView::Helpers` modules are not included in the context in which the RJS helper executes. To move the `number_to_currency` call into the helper, you have to include

ActionView::Helpers::NumberHelper in the helper module.

```
module ExpensesHelper
  include ActionView::Helpers::NumberHelper

  def display_errors(expense)
    page.alert "The Expense could not be added for the following reasons:\n" +
               expense.errors.full_messages.join("\n")
  end

  def update_summary(project)
    page.replace 'summary', :partial => 'summary',
                            :locals => { :min => project.min_expense,
                                         :max => project.max_expense,
                                         :average => project.avg_expense }
  end

  def insert_expense(expense)
    page.insert_html :bottom, 'expenses', :partial => 'expense', :object => expense
    page.visual_effect :highlight, 'expense-#{expense.id}'
  end

  def update_total(amount)
    page.replace_html 'total-amount', number_to_currency(amount)
  end
end
```

The RJS helpers really help make the code in our RJS template a lot simpler. Obviously, there isn't a lot of value in extracting the method calls into helpers if the code is only called in one place. Howeve the helpers are great when the code in your RJS templates is being duplicated in several different templates. The helpers methods are also available to inline RJS calls from your controllers when you are using `render :update`. If you are duplicating code in your RJS templates and inline RJS calls, you can probably benefit from extracting that functionality into helpers.

PREV

A Look Ahead

As you can see from the Expense Tracker, it is almost too easy to implement complex Ajax features with Rails and RJS templates. The Expense Tracker just barely scratches the surface of what is possible. The RJS Reference section has examples of usage so you can master all aspects of RJS for your own projects. Have fun!

PREV

PREV

Chapter 7. RJS Reference

This reference section covers all of the methods supported by the`JavaScriptGenerator` as well as the `JavaScriptElementProxy` and the `JavaScriptCollectionProxy` classes. At the end of the section, there is a list of Scriptaculous visual effects shipping with Rails 1.1.

PREV

PREV

JavaScriptGenerator

The following is a list of all of the methods public methods offered by the`JavaScriptGenerator`. These methods are called on the `page` object in your RJS templates.

Since RJS is all about generating JavaScript, it is nice to know what is going on behind the scenes. Knowing about the JavaScript that is generated makes it much easier to debug problems and create more complex applications. At some point, your RJS code may become too complex or there may be a task that you can't perform elegantly with RJS. If you understand how RJS generates JavaScript, you can easily port your code into a JavaScript library and use RJS to access your new JavaScript objects and methods.

Therefore, for all of the following definitions, I have placed the JavaScript that the method generates after the Ruby code. The Ruby code is marked with the comment`# Ruby code`, and the JavaScript is marked with `// Generated JavaScript`.

<<(javascript)

Writes raw JavaScript to the page.

[](id)

Returns a `JavaScriptElementProxy` for the DOM element with the specified `id`. Methods can be called on the returned element. Multiple method calls can also be chained together.

```
# Ruby code
page['header'].show

// Generated JavaScript
$("header").show();

# Ruby code
page['header'].first.second

// Generated JavaScript
$("header").first().second();
```

assign(variable, value)

Assigns a value to the JavaScript variable specified. Ruby objects are automatically converted to JavaScript objects by calling the object's`to_json` method if it has one, or `inspect` if it doesn't.

```
# Ruby code
page.assign 'name', { :first => "Cody", :last => "Fauser" }
```

```
// Generated JavaScript
name = { "first": "Cody", "last": "Fauser" };
```

alert(message)

Displays a JavaScript alert dialog box with the provided message.

```
# Ruby code
page.alert 'An error occurred while processing your request'

// Generated JavaScript
alert("An error occurred while processing your request");
```

call(function, arg, ...)

Calls a JavaScript function and passes in zero or more arguments.

```
# Ruby code
page.call 'displayError', 'An error occurred', 'Critical'

// Generated JavaScript
displayError("An error occurred", "Critical");
```

You can call methods on custom objects that you've added to your page by specifying the variable name and the method call.

```
# Ruby code
page.call 'inventory.showTotal'

// Generated JavaScript
inventory.showTotal();
```

delay(seconds = 1)

Executes the code within the block after delaying for the specified number of seconds.

```
# Ruby code
page.delay(5) do
  page.visual_effect :highlight, 'navigation'
end

// Generated JavaScript
setTimeout(function() {
;
new Effect.Highlight("navigation", {});
}, 5000);
```

draggable(id, options = {})

Makes the DOM element specified by the `id` draggable.

```
# Ruby code
page.draggable('photo', :revert => true)

// Generated JavaScript
new Draggable('photo', {revert: true});
```

drop_receiving(id, options = {})

Makes the DOM element specified by the `id` receive dropped draggable elements. Draggable elements are created using the RJS `draggable` method or by using `draggable_element()` Scriptaculous helper.

```
# Ruby code
page.drop_receiving('photo', :url => { :action => 'add' })

// Generated JavaScript
Droppables.add("photo", {onDrop:function(element){new
Ajax.Request('/hello_world/add', {asynchronous:true, evalScripts:true,
parameters:'id=' + encodeURIComponent(element.id)})}});
```

hide(id, ...)

Hides one or more DOM elements. Specify the elements to hide by their DOM `id`s.

```
# Ruby code
page.hide('first', 'second')

// Generated JavaScript
Element.hide("first", "second");
```

*insert_html(position, id, *options_for_render)*

Inserts the HTML into the specified position in relation to the element.

The available positions are:

:before

The content is inserted into the page before the element.

:after

The content is inserted into the page after the element.

:top

The content is inserted into the element before the element's existing content.

:bottom

The content is inserted into the element after the element's existing content.

```
# Ruby code
page.insert_html(:bottom, 'products', '<li>Refrigerator</li>')

// Generated JavaScript
new Insertion.Bottom("products", "<li>Refrigerator</li>");
```

redirect_to(location)

Redirect the browser to the location specified. `redirect_to()` passes the location to `url_for()`, so any of the arguments you normally use with `url_for()` can also be used with `redirect_to()`.

```
# Ruby code
page.redirect_to('http://www.google.com')

// Generated JavaScript
window.location.href = "http://www.google.com";

# Ruby code
page.redirect_to(:controller => 'inventory', :action => 'list')

// Generated JavaScript
window.location.href = "http://localhost:3000/inventory/list";
```

remove(id, ...)

Removes one or more DOM elements from the page.

```
# Ruby code
page.remove('first', 'second')

// Generated JavaScript
["first", "second"].each(Element.remove);
```

*replace(id, *options_for_render)*

Replaces the entire element specified or `outerHTML`. `replace` is useful with partial templates so that the entire partial, which includes a container element, can be rendered and used to replac content during an RJS call. An example is an unordered list. A partial containing a `<li>` tag can be rendered instead of having to replace the `innerHTML` of the `<li>`. Replacing just the `innerHTML` would require the `<li>` tag to be moved out of the partial.

```
# Ruby code
product.replace 'banner', '<id="banner">Welcome back Cody</div>'

// Generated JavaScript
Element.replace("banner", "<id=\"banner\">Welcome back Cody</div>");
```

*replace_html(id, *options_for_render)*

Replaces the content or `innerHTML` of the DOM element with the `id` with either a `String` or the output of a rendered partial template.

```
# Ruby code
page.replace_html 'timestamp', Time.now

// Generated JavaScript
Element.update("timestamp", "Sat Apr 29 15:14:24 EDT 2006");
```

select(pattern)

Selects DOM elements using CSS-based selectors. Returns a `JavaScriptElementCollectionProxy` that can then receive proxied enumerable methods. See the `JavaScriptCollectionProxy` methods in the next section.

```
# Ruby code
page.select "#content p"

// Generated JavaScript
# => $$("#content p");
```

sortable(id, options = {})

Makes the element with the DOM ID `id` sortable using drag and drop. The options are the same as the options for `ScriptaculousHelper#sortable_element()`.

```
# Ruby code
# Assuming the current controller is ProjectsController
page.sortable 'project-65', :url => { :action => 'sort' }

// Generated JavaScript
Sortable.create("project-65", {onUpdate:function(){
  new Ajax.Request('/projects/sort', {
    asynchronous:true, evalScripts:true, parameters:Sortable.serialize("project-65")
    }
  )}
});
```

show(id, ...)

Makes one or more hidden DOM elements visible. As with`hide()`, the elements are specified by their DOM `id`s.

```
# Ruby code
page.show 'element-20', 'element-30'

// Generated JavaScript
Element.show("element-20", "element-30");
```

toggle(id, ...)

Toggles the visibility of one or more DOM objects.

```
# Ruby code
page.toggle 'product-1', 'product-2', 'product-3'

// Generated JavaScript
Element.toggle("product-1", "product-2", "product-3");
```

visual_effect(name, id = nil, options = {})

Creates a new Scriptaculous visual effect to the element with the DOM`id`. See the following section on visual effects for a list of the visual effects shipping with Rails 1.1.

```
# Ruby code
page.visual_effect :highlight, 'list-item-69', :duration => 5

// Generated JavaScript
new Effect.Highlight("list-item-69",{duration:5});
```

JavaScriptElementProxy

The `JavaScriptElementProxy` is a proxy to a real DOM object. You can proxy method calls to the any of the DOM object's JavaScript methods. A few additional methods have been added to the proxy objects to make them more useful. This reference covers those enhanced non-JavaScript methods.

*replace_html(*options_for_render)*

Replaces the `innerHTML` of the DOM object. See the `JavaScriptGenerator#replace_html()` method for more information.

*replace(*options_for_render)*

Replaces the `outerHTML` for the DOM object. See `JavaScriptGenerator#replace()` method for more information.

reload

Reloads the content for the element by re-rendering a partial with the same name as the DOM element's `id`.

```
page['header'].reload

# Equivalent to:

page['header'].replace :partial => 'header'
```

PREV

JavaScriptCollectionProxy

The JavaScriptCollectionProxy is the most difficult part of RJS to understand and work with. This section will first cover some of the trickier aspects of the enumerable support and the actual reference will begin. This will help you avoid trouble and maintain your productivity.

Block Variables and the Enumerable methods

JavaScriptGenerator#select() returns an instance of the JavaScriptElementCollectionProxy class. JavaScriptElementCollectionProxy inherits from JavaScriptCollectionProxy, which provides all of the functionality. The trickiest part about the collection proxy enumerable methods is the code that goes within the block. The code within the block is translated into a JavaScript iterator function. Another tricky aspect of the enumerable block is that you can use any variable names you like for the block parameters, but in the generated JavaScript iterator function the variable names used are always value and index. This rule applies to any expression you use within the block that isn't a simple method called on the proxy object.

The first block parameter is always the element and the second is always the index, except with inject() and pluck(). If you don't access the value or index within the block, then you don't have to pass them into the block. In the following example, the block parameter is named element.

```
# Ruby code
page.select('#elements span').all('allVisible') do |element, index|
  element.visible
end

// Generated JavaScript
var allVisible = $$("#elements span").all(function(value, index) {
return value.visible();
});
```

As you can see from the generated JavaScript, the variable value was used and not element, as it was named in the Ruby code. This works well, as long as the code within the block is a simple proxie method call, as it was in the preceding code. More complex expressions need to be written as a strin and passed to the << method of the page object.

When using a direct element proxy rather than a string passed to <<, the generator assumes that the proxied call is a method call and automatically adds (). This means that you can't directly proxy an element's property, such as element.innerHTML.

```
page.select('#elements span').any('allVisible') do |value, index|
  page << '!value.visible()'
end
```

The last expression in the block is the statement appended to the return statement in the generated JavaScript code, so it is possible to safely add more method calls to the block.

Inspecting the Results of the Enumerations

The one problem with the enumerable functions is that it is difficult to inspect the return JavaScript variables assigned by the function. The FireBug console is unable to display the assigned variable when the variable is assigned during the execution of`eval()` after an Ajax call. I used a small `Logger` class that uses the `printfire()` method discussed in the FireBug chapter to log to the JavaScript console. I then created a simple class to inspect the JavaScript variables assigned by the enumerable functions. You can define the class in `public/javascripts/application.js`.

```
var Inspector = {}

Inpsector = {
  inspect: function(val) {
    if (val.innerHTML) {
      Logger.log(val.innerHTML);
    } else {
      Logger.log(val);
    }
  },

  inspectArray: function(array) {
    array.each(this.inspect);
  }
}
```

Following is an example of the usage of the Logger class for inspecting the variable assigned by the return value of sort_by(). In this case `sort_by()` sorts all the paragraphs in the page by the length of their content and stores the sorted objects in an `Array` named `sortedParagraphs`.

```
page.select('p').sort_by('sortedParagraphs') do |value, index|
  page << 'value.innerHTML.length;'
end

page << 'Logger.inspectArray(sortedParagraphs);'
```

This code simply iterates through the sorted paragraphs and displays the HTML content of each object to the FireBug console. Feel free to expand and improve on the code to meet your own needs.

Method Reference

All of the following enumerable methods are simply proxies to the Prototype enumerable method of the same name. For each method (except for `pluck`), there is a corresponding Ruby enumerable method with the same name. The Ruby enumerable method may have an added`?` (e.g., `all?`). The

Prototype methods were meant to mimic Ruby's enumerable methods. Therefore, you can get a pretty good idea of the overall intent of each method by examining the Ruby version.

all(variable) {|value,index| block }

Each element is passed to the block. The block is converted to a JavaScript iterator function. The JavaScript `variable` is set to `TRue` if the iterator function never returns `false` or `null`.

```
# Ruby code
page.select('#line-items li').all('allVisible') do |value, index|
  value.visible
end

// Generated JavaScript
var allVisible = $$("#line-items li").all(function(value, index) {
return value.visible();
});
```

any(variable){|value, index| block }

Each element is passed to the block. The block is converted to a JavaScript iterator function. The JavaScript `variable` is set to `TRue` if the iterator function never returns `false` or `null`.

```
# Ruby code
page.select('#line-items li').any('anyVisible') do |value, index|
  value.visible
end

// Generated JavaScript
var anyVisible = $$("#line-items li").any(function(value, index) {
return value.visible();
});
```

collect(variable){|value, index| block }

The block is converted to a JavaScript iterator. This method assigns a new JavaScript`Array` to the JavaScript `variable` containing the results of executing the iterator function once for each element.

```
# Ruby code
page.select('#orders tr').collect('heights') do |value, index|
  value.get_height
end

// Generated JavaScript
var heights = $$("#orders tr").collect(function(value, index) {
return value.getHeight();
});
```

detect(variable){|value, index| block }

Each element is passed to the given block. The block is converted to a JavaScript iterator function. The JavaScript `variable` is assigned to the first element for which the iterator functior does not return `false`.

```
# Ruby code
page.select('#content p').detect('first_visible') do |value, index|
  value.visible
end

// Generated JavaScript
var first_visible = $$("#content p").detect(function(value, index) {
return value.visible();
});
```

each{|value, index| block }

Passes each element to the block, which is converted to a JavaScript iterator function. The iterator function is called once for each element.

```
# Ruby code
page.select('#content p').each do |value, index|
  page << 'alert(value.innerHTML);'
end

// Generated JavaScript
$$("p").each(function(value, index) {
alert(value.innerHTML);
});
```

find(variable){|value, index| block }

A synonym for `JavaScriptCollectionProxy#detect()`.

find_all(variable){|value, index| block }

Each element is passed to the given block. The block is converted to a JavaScript iterator function. The JavaScript `variable` is assigned to an `Array` of all elements for which the iterator function does not return `false`.

```
# Ruby code
page.select('#content p').find_all('empty_paragraphs') do |value, index|
  value.empty
end

// Generated JavaScript
var empty_paragraphs = $$("#content p").collect(function(value, index) {
return value.empty();
});
```

grep(variable, pattern){|value, index| block }

Each DOM element with a `toString()` matching the `Regexp` pattern is passed to the given block. The block is converted to a JavaScript iterator function. The JavaScript variable is assigned to an `Array` containing the results of executing the iterator function once for each element.

This method isn't entirely useful in the context of RJS. Its only real use is matching DOM elements using the pattern based on their type, which is output by the`toString()` method. The `toString()` method outputs `[object HTMLDivElement]` for a *<div>* element. Use the FireBug console to view the `toString()` output for other elements.

```
# Ruby code
page.select('#content p').grep('hidden', /Div|Paragraph/) do |value, index|
  value.hidden
end

// Generated JavaScript
var hidden = $$("#content p").grep(/Div|Paragraph/, function(value, index) {
return value.hidden();
});
```

inject(variable, memo){|memo, value, index| block }

Each element and the accumulator value `memo` is passed to the given block. The block is converted to a JavaScript iterator function. The iterator function is called once for each elemen and the result is stored in `memo`. The result returned to the JavaScript `variable` is the final value of `memo`. The initial value of `memo` is set in one of two ways. The value of the parameter `memo` is used as the initial value if a non `nil` value is passed in to the method. Otherwise, the first element of the collection is used as the initial value.

```
# Ruby code
page.select('#content .columns').inject('totalWidth', 0) do |memo, value, index|
   page << '(memo + value.offsetWidth)'
end

// Generated JavaScript
var totalWidth = $$(".columns").inject(0, function(memo, value, index) {
return(memo + value.offsetWidth);
});
```

map(variable){|value, index| block }

Synonym for `JavaScriptCollectionProxy#collect()`.

max(variable){|value, index| block }

Each element is passed to the given block. The block is converted to a JavaScript iterator function. The JavaScript `variable` is assigned to the largest value returned by the iterator function.

```
# Ruby code
page.select('p').max('longestParagraphLength') do |value, index|
  page << 'value.innerHTML.length'
end

// Generated JavaScript
var longestParagraphLength = $$("p").max(function(value, index) {
return value.innerHTML.length;
});
```

min(variable) { |value, index| block }

Each element is passed to the given block. The block is converted to a JavaScript iterator function. The JavaScript `variable` is assigned to the smallest value returned by the iterator function.

```
# Ruby code
page.select('p').min('shortestParagraphLength') do |value, index|
  page << 'value.innerHTML.length'
end

// Generated JavaScript
var shortestParagraphLength = $$("p").min(function(value, index) {
return value.innerHTML.length;
});
```

partition(variable) { |value, index| block }

Each element is passed to the given block. The block is converted to a JavaScript iterator function and is executed once for each element. The JavaScript`variable` is set to an `Array` containing to `Array`s. The first `Array` contains all of the elements for which the iterator function evaluated to true. The second `Array` contains all of the remaining elements.

```
# Ruby code
page.select('.products').partition('productsByVisibility') do |value, index|
  value.visible
end

// Generated JavaScript
var productsByVisibility = $$(".products").partition(function(value, index) {
return value.visible();
});
```

pluck(variable, property)

Iterates through the collection creating a new`Array` from the value of the given property of each object without a function call. The resulting`Array` is assigned to the JavaScript `variable`.

```
# Ruby code
page.select('.amounts').pluck('amounts', 'innerHTML')

// Generated JavaScript
var amounts = $$(".amounts").pluck("innerHTML");
```

reject(variable){|value, index| block }

Each element is passed to the given block. The block is converted to a JavaScript iterator function. The JavaScript `variable` is assigned to an `Array` of all elements for which the iterator function returns `false`.

```
# Ruby code
page.select('p').reject('paragraphsWithContent') do |value, index|
  value.empty
end

// Generated JavaScript
var paragraphsWithContent = $$("p").reject(function(value, index) {
return value.empty();
});
```

select(variable){|value, index| block }

A synonym for `JavaScriptCollectionProxy#find_all()`.

sort_by(variable){|value, index| block }

Each element is passed to the given block. The block is converted to a JavaScript iterator function. Assigns the JavaScript `variable` to an `Array` of the elements sorted using the result of the iterator function as keys for the comparisons when sorting.

```
# Ruby code
page.select('p').sort_by('sortedParagraphs') do |value, index|
  page << 'value.innerHTML.length;'
end

// Generated JavaScript
var sortedParagraphs = $$("p").sortBy(function(value, index) {
return value.innerHTML.length;
});
```

zip(variable, arg1, ...){|value, index| block }

Merges elements with the arrays passed as parameters. Elements in the corresponding index o each array form a new array. The resulting array of arrays is assigned to the JavaScript `variable`. If a block is provided it is converted to an iterator function and is applied to each resulting `Array`. The name of the parameter to the iterator function is `array`.

For this example, assume that the page has the following paragraph elements:

```
<p id="first">First Paragraph</p>
<p id="second">Second Paragraph</p>
```

First, applying `zip` without a block:

```
# Ruby code
page.select('p').zip('zipped', [1, 2], [3,4])

// Generated JavaScript
var zipped = $$('p').zip([1,2], [3,4]);
```

The resulting array of arrays, where the paragraphs (represented by their`id`s) are DOM objects:

```
[[< id="first">, 1, 3], [<p id="second" >, 2, 4]]
```

Applying `zip` with a block:

```
# Ruby code
page.select('p').zip('zipped', [1, 2], [3,4]) do |array|
  page.call 'array.reverse'
end

// Generated JavaScript
var zipped = $$("p").zip([1,2], [3,4], function(array) {
return array.reverse();
});
```

The resulting `Array` of arrays, where the paragraphs (represented by their `id`s) are DOM objects:

```
[[3, 1, <p id="first">], [4, 2, <p id="second" >]]
```

Visual Effects

For the purposes of RJS templates, all visual effects are accessed using the name of the effect as a symbol, e.g., `:highlight` for the `Highlight` effect. The chosen effect is passed to `JavaScriptGenerator#visual_effect()`. The effect name is converted to camel-case. So for example, so if a new effect was added to Scriptaculous named `FizzleOut`, you would access it from RJS using its underscored name `fizzle_out`. Following is a list of the current effects shipping with the Scriptaculous library as of Rails 1.1:

appear

Element gradually appears.

blind_down

Causes the `div` element specified to slowly slide down into visibility like a blind being pulled down.

blind_up

The opposite of `blind_down`. Causes the element to slowly slide up and out of view like a blind being drawn up.

drop_out

Causes the element to drop down out of view.

fade

The opposite of appear. The element gradually fades from view.

fold

First blinds up the element about 90% of the way and then squishes it over to the left side of the region it originally occupied.

grow

Element grows up and into view from the bottom of the area occupied by the element.

highlight

Perform the "Yellow Fade Technique" on the element.

puff

The element expands and becomes transparent until it disappears.

pulsate

Make the element flash on and off.

shake

Causes the element to shake back and forth horizontally. Great for drawing the user's attention.

shrink

The opposite of `grow`. The element shrinks away off of the screen toward the bottom of the area it occupies.

slide_down

Causes the entire element to slide down into view.

slide_up

The opposite of `slide_down`. Causes the element to slide up and out of view.

squish

Squishes the element out of view by quickly compressing it into the upper left hand corner of the space it occupies.

switch_off

Causes the element to flicker and the drops out of view, similar to`drop_out`.

toggle_appear

Toggles the element between being visible and hidden. Uses`appear` when making the element visible and `fade` when hiding the element.

toggle_blind

The same as `toggle_appear`, but uses `blind_down` and `blind_up` for showing and hiding the elements respectively.

toggle_slide

The same as `toggle_appear`, but uses `slide_down` and `slide_up` for showing and hiding the elements respectively.

PREV

PREV

Chapter 8. Appendix

PREV

PREV

Bibliography

[TH05]

Thomas, David et al. *Agile Web Development with Rails*. Raleigh and Dallas: Pragmatic. 2005.

PREV

PREV

Online Resources

Prototype

Sam Stephenson's Weblog ((http://sam.conio.net/))

The weblog of Sam Stephenson the creator of RJS and the Prototype library.

Encytemedia ((http://encytemedia.com/blog/))

Justin Palmer's weblog featuring many in-depth articles about Prototype.

Developer Notes for prototype.js ((http://www.sergiopereira.com/articles/prototype.js.html))

The first reference on the Prototype library. Very in-depth reference, but only covers up to Prototype version 1.4.0.

Prototype Dissected ((http://www.snook.ca/archives/000531.php))

A wonderful graphical overview of the Prototype library by Jonathan Snook.

Scriptaculous

Script.aculo.us ((http://script.aculo.us))

Scriptaculous developer Thomas Fuchs' weblog. Chock-full of wonderful effects examples with source code.

Effect.Queue ((http://www.railsdevelopment.com/2006/01/15/effectqueue/))

An excellent and in-depth article by Abdur-Rahman Advany that explains in detail how to use Scriptaculous effects queues and how the effects queues are used from within Rails.

RJS

Rails Weenie ((http://rails.techno-weenie.net/))

Rails core developer Rick Olson's site for Rails troubleshooting and tips. Contains a few great articles about RJS.

Test Driven RJS with ARTS ((http://glu.ttono.us/articles/2006/05/29/guide-test-driven-rjs-with-arts))

Kevin Clark's brand new (at the time of writing) plugin that allows you to functional test your RJS templates.

Ruby on Rails

Ruby on Rails Homepage ((http://www.rubyonrails.org))

The official Ruby on Rails homepage with links to the Rails documentation, the Rails weblog an
even Rails screencasts.