At the Forge

RJS Templates

Reuven M. Lerner

Abstract

The power of Ajax to fetch and run JavaScript generated by your server-side language.

The past few months, I've written a number of articles in this space about JavaScript. This language, built in to nearly every modern Web browser, has now come into its own and is at the heart of a paradigm for modern Web development known as Ajax. Whereas knowledge of JavaScript was long an optional skill for Web developers, it has become a must-have skill, along with SQL, HTML, HTTP and CSS.

One of the reasons for JavaScript's renaissance is the emergence of cross-platform libraries, which hide the incompatibilities that long plagued the language. For quite some time, programs written in JavaScript had to contain many if/then statements that looked at possible cross-platform incompatibilities.

Today, we can avoid having such if/then statements in our code by using libraries that take care of these low-level tasks for us. Prototype and Dojo, two of the JavaScript libraries I profiled in previous columns, have become popular precisely because they hide many of these details. They make JavaScript a truly cross-platform language, where “platform” means the Web browser as much as the operating system.

Some clever programmers, in an effort to make JavaScript standardization even more complete and effortless, have gone one step further. Why not use your server-side programming language to generate the JavaScript for you? That is, if you are using Ruby on Rails, perhaps you could write commands in Ruby and have them translated into JavaScript. Doing so would allow you to use roughly the same code in all of your templates, without having to switch syntax in different parts of the template.

This might sound like a strange idea, but the more I think about it, the more I like it. RJS (short for Ruby JavaScript) templates are one incarnation of this. If you prefer to create your JavaScript in Java, you might want to look at the Google Web Toolkit, which is now available under an open-source license and has gained many fans in the Java world.

This month, we look at RJS and how it makes life much easier for Web developers. Although I don't think that JavaScript will ever disappear, or that Web developers will be able to ignore it completely, technologies such as RJS mean that it might become something like machine code today—available and sitting at the bottom of the pyramid, but generally ignored by high-level programmers.

Ajax and Rails
To create a new Rails project called ajaxdemo, type:

```
rails ajaxdemo
```

Now, let's create a simple controller called showme:

```
script/generate controller showme
```

We're not going to have any model in this system, but you still might have to define one or more lines in config/database.yml. Instead, let's create a new view within our showme controller, stored as app/views/showme/index.rhtml, shown in Listing 1. *Garrick, small font in Listing 1.*

**Listing 1. index.rhtml**

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title id="title">Sample HTML page</title>
    <%= javascript_include_tag 'prototype' %>
    <script type="text/javascript">
    function updateHeadline()
    {
      var headline = $('headline');
      var new_headline_text = $F('future-headline');
      Element.update(headline, new_headline_text) ;
    }
    </script>
  </head>
  <body>
    <h1 id="headline">Headline</h1>
    <p><input type="text" id="future-headline" /></p>
    <p><input type="button" onclick="updateHeadline()" value="Update headline" /></p>
  </body>
</html>
```

As you can see, the index.rhtml page is a relatively standard page of HTML, with some JavaScript code that uses the Prototype library. The page consists of a headline, a text field and a button. Pressing the button invokes the function updateHeadline. This function takes the current value of the future-headline text field and changes the headline to reflect its contents.

### Using a Remote Call

So far, we haven't done anything special. Now, however, we're going to do something a bit more sophisticated: send the contents of our text field to the server in an Ajax call. The server's response will be our headline, translated into Pig Latin.

Making this change requires doing two things. First, we need to write a method in our application controller that takes the headline, turns it into Pig Latin, and then returns that text. Second, we need to modify our template so that it gets the updated text from the server, rather than from a local JavaScript
function.

Our updated template is shown in Listing 2. We have made a number of changes, starting with the fact that our form now has an id attribute associated with it, named theForm. The form contains a single element, a text field whose name is future_headline. Note that we need to use the name attribute instead of the id attribute, so that the form element will be submitted with our Ajax call. Also notice that we have changed the name to a Ruby-friendly future_headline (with an underscore), rather than the CSS-friendly future-headline (with a hyphen). **Garrick, small font in Listing 2.**

### Listing 2. index.rhtml (Ajax Version)9673l2.qrk

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title id="title">Sample HTML page</title>
  <%= javascript_include_tag 'prototype' %>
</head>
<body>
  <h1 id="headline">Headline</h1>
  <form id="theForm">
    <p><input type="text" name="future_headline" /></p>
  </form>
  <p><%= submit_to_remote "submit-button", "Pig Latin it!",
    :url => { :action => "piglatin_sentence" },
    :submit => "fakeForm",
    :update => "headline" %></p>
</body>
</html>
```

We also have replaced our button with a call to the submit_to_remote helper:

```rhtml
<p><%= submit_to_remote "submit-button", "Pig Latin it!",
     :url => { :action => "piglatin_sentence" },
     :submit => "fakeForm",
     :update => "headline" %></p>
```

The above code does quite a few things:

- It creates a button, whose DOM ID is submit-button.
- The button has a label of “Pig Latin it!”.
- When the button is clicked, it uses Ajax to invoke the piglatin_sentence action on the server, within the current controller.
- The contents of the form with the ID fakeForm are submitted.
• The value returned from the Ajax invocation is used to update the contents of the HTML element with the ID headline.

All that's left for us to examine is our controller, shown in Listing 3. The controller doesn't necessarily know that it is being invoked by a background Ajax process or that its contents will be used to update the headline element. Rather, it simply is invoked like any method, turning the words into Pig Latin. The translated sentence is returned to the user's browser as a plain-text file.

Listing 3. showme_controller.rb

class ShowmeController < ApplicationController
  def piglatin_sentence
    # Get the headline
    sentence = params[:future_headline]
    words = sentence.split
    sentence = ""
    # Go through each word, applying the secret
    # Pig Latin algorithm
    words.each do |word|
      if word =~ /^[aeiou]$/i
        word << "way"
      else
        first_letter = word.slice(0,1)
        rest = word.slice(1..-1)
        word = "#{rest}#{first_letter}ay"
      end
    end
    sentence << word
    sentence << " 
    end
    render :text => sentence
  end
end

Returning JavaScript

Now, the real magic begins. As we just saw, our controller (piglatin_sentence) returns a plain-text document to its caller. Of course, we're free to return data in whatever format we please. One possible format might be XML. Indeed, the term Ajax is supposed to stand for Asynchronous JavaScript and XML, so it should come as no surprise that XML is a common format for return values. Another format that is growing in popularity is JSON (JavaScript Object Notation), a textual version of JavaScript objects that makes it fast and easy to exchange data.

But, in the world of Ruby on Rails, there is another type of data that the controller might return to the user's browser, namely JavaScript. This might not sound all that clever, but consider what Prototype does with it. If a controller is invoked with link_to_remote or submit_to_remote, and if the HTTP response has a content-type of xml+javascript, the JavaScript is evaluated.
This could potentially save time—instead of returning the text that should be used for the headline and then using JavaScript to insert it. (True, we were able to say this tersely by using an :update parameter to the call to submit_to_remote. But the code still exists.) Rather, we simply could return JavaScript code that uses the DOM to modify the document. This would simplify our code quite a bit. *Garrick, small font in Listing 4 and 5.*

**Listing 4. index.rhtml (Updated for JavaScript in the HTTP Response)**

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title id="title">Sample HTML page</title>
    <%= javascript_include_tag 'prototype' %>
  </head>
  <body>
    <form id="fakeForm">
      <h1 id="headline">Headline</h1>
      <p><input type="text" name="future_headline" /></p>
    </form>
    <p><%= submit_to_remote "submit-button", "Pig Latin it!", :url => { :action => "piglatin_sentence" }, :submit => "fakeForm" %></p>
  </body>
</html>
```

**Listing 5. showme_controller.rb (Updated to Return JavaScript)**

```ruby
class ShowmeController < ApplicationController

  def piglatin_sentence
    # Get the headline
    sentence = params[:future_headline]
    words = sentence.split

    sentence = ""

    # Go through each word, applying the # secret Pig Latin algorithm
    words.each do |word|
      if word =~ /^[aeiou]/i
        word << "way"
      else
        first_letter = word.slice(0,1)
        rest = word.slice(1..-1)
        word = "#{rest}#{first_letter}ay"
      end
    end
    sentence << word
    sentence << " "
  end
  output = "Element.update($('headline'), '#{sentence}');"
  render :text => output, :content_type => "text/javascript"
end
```
To see this in action, look at Listings 4 and 5. Listing 4 is an updated version of our template, index.rhtml, and it is basically unchanged from the previous version, except that we now are able to remove the :update parameter from the call to submit_to_remote:

```ruby
<p><%= submit_to_remote "submit-button",
   "Pig Latin it!",
   :url => { :action => "piglatin_sentence" },
   :submit => "fakeForm" %></p>
```

Instead of indicating what should be changed on the client side, we instead do that on the server side: 

*Garrick, small font below.*

```ruby
output = "Element.update($('headline'), '#{sentence}');"
render :text => output, :content_type => "text/javascript"
```

In other words, we tell our controller to produce a response of type text/javascript, knowing that whatever we send will be evaluated by the user's browser. We then send a response that uses Element.update to change the headline to our translated sentence. Sure enough, as soon as we install this new version of our software, the headline continues to be changed.

The power that's associated with this is tremendous. For example, we could update the headline conditionally, checking it against a dictionary of forbidden words in our server's database. We could keep track of what words are most commonly used. We could restrict users to a certain number of headline updates per day.

*Even better, because we're returning a JavaScript program, rather than the contents of an individual HTML element, we can modify multiple parts of the page and even throw in some Scriptaculous effects for good measure.* Returning JavaScript is a seemingly simple feature that Prototype provides, but it is one that opens the door to tremendous possibilities.

**RJS**

And now, the moment you've been waiting for—you might be thinking that although the notion of evaluated JavaScript responses is powerful, it's annoying to have to create and maintain JavaScript code in Ruby controllers. It's bad enough that we need to have SQL in there; three languages in a single file seems like overkill.

And, that's where RJS templates come in. The basic idea is that we assume a response will be in the form of JavaScript, and that it will modify one or more elements on the current page. RJS provides us with a compact syntax for making those changes, so we can create very small files that do a great deal.

The changes we need to make are minor. First, we modify our piglatin_sentence method such that it modifies not sentence (a local variable) but @sentence (an instance variable). We also remove the call to render, because we won't be rendering anything directly.

We then create a file called piglatin_sentence.rjs. This is a view, just like an .rhtml file, and thus goes
alongside it in the views directory. But, it consists of a single line:

```ruby
page[:headline].replace_html @sentence
```

In other words, we should take the current page, find the element with an ID of headline and replace its HTML content with the value of @sentence, which we got from the method.

**Listing 6. showme_controller.rb (Updated to Use RJS)**

```ruby
class ShowmeController < ApplicationController
  def piglatin_sentence
    # Get the headline
    sentence = params[:future_headline]
    words = sentence.split

    @sentence = ""

    # Go through each word, applying the
    # secret Pig Latin algorithm
    words.each do |word|
      if word =~ /^([aeiou])/i
        word << "way"
      else
        first_letter = word.slice(0,1)
        rest = word.slice(1..-1)

        word = "#{rest}#{first_letter}ay"
      end
    end

    @sentence << word
    @sentence << " "
  end
end
```

Sure enough, this works well. With a tiny bit of code, we've managed to do quite a bit. And, as before, we can add Scriptaculous calls, update multiple elements on the page, show and hide HTML elements—basically, anything we might want to do.

**Conclusion**

When I first read about RJS templates, I thought it was one of the weirdest ideas I had heard of. That's because the notion of writing JavaScript in Ruby seemed both strange and unnecessary. I now understand the power and cleverness of this type of template and look forward to using it on many of my Ajax-powered sites. With a bit of study and some changes in how you think about Web development, you're likely to make the same discovery.

**Resources**

There are many good printed and on-line resources for learning about Ajax. One of my favorite sites is [http://ajaxian.com](http://ajaxian.com), in which the authors discuss and review Ajax-related tools.
Two good books might come in handy. The Pragmatic Programmers have published a second edition of the award-winning *Agile Development in Rails* by Dave Thomas and David Heinemeier-Hansson, and O'Reilly has released *Ajax on Rails* by Scott Raymond. Between these two books, you should expect to get a full understanding of not only how Ruby on Rails works, but also how to use JavaScript and RJS to create interesting and dynamic Web applications.