Working with JavaScript in Rails

Ruby on Rails

Working with JavaScript in Rails

Working with JavaScript in Rails

This guide covers the built-in Ajax/JavaScript functionality of Rails (and more); it will enable you to create rich and dynamic Ajax applications with ease! After reading this guide, you will know:

- The basics of Ajax.
- Unobtrusive JavaScript.
- How Rails' built-in helpers assist you.
- How to handle Ajax on the server side.
- The Turbolinks gem.

1 An Introduction to Ajax

In order to understand Ajax, you must first understand what a web browser does normally.

When you type `http://localhost:3000` into your browser's address bar and hit 'Go,' the browser (your 'client') makes a request to the server. It parses the response, then fetches all associated assets, like JavaScript files, stylesheets and images. It then assembles the page. If you click a link, it does the same process: fetch the page, fetch the assets, put it all together, show you the results. This is called the 'request response cycle.'

JavaScript can also make requests to the server, and parse the response. It also has the ability to update information on the page. Combining these two powers, a JavaScript writer can make a web page that can update just parts of itself, without needing to get the full page data from the server. This is a powerful technique that we call Ajax.

Rails ships with CoffeeScript by default, and so the rest of the examples in this guide will be in CoffeeScript. All of these lessons, of course, apply to vanilla JavaScript as well.

As an example, here's some CoffeeScript code that makes an Ajax request using the jQuery library:

```
$.ajax(url: "/test").done (html) ->
  $("#results").append html
```

This code fetches data from "/test", and then appends the result to the `div` with an id of `results`.

Rails provides quite a bit of built-in support for building web pages with this technique. You rarely have to write this code yourself. The rest of this guide will show you how Rails can help you write websites in this way, but it's all built on top of this fairly simple technique.

2 Unobtrusive JavaScript

Rails uses a technique called "Unobtrusive JavaScript" to handle attaching JavaScript to the DOM. This is generally considered to be a best-practice within the frontend community, but you may occasionally read tutorials that demonstrate other ways.

Here's the simplest way to write JavaScript. You may see it referred to as 'inline JavaScript':

```
<a href="#" onclick="this.style.backgroundColor='#990000'">Paint it red</a>
```

When clicked, the link background will become red. Here's the problem: what happens when we have lots of JavaScript we want to execute on a click?

```
<a href="#" onclick="this.style.backgroundColor='#009900';this.style.color='#FFFI
```

Awkward, right? We could pull the function definition out of the click handler, and turn it into CoffeeScript:

```
paintIt = (element, backgroundColor, textColor) ->
  element.style.backgroundColor = backgroundColor
  if textColor?
    element.style.color = textColor
```

And then on our page:

```
<a href="#" onclick="paintIt(this, '#990000')">Paint it red</a>
```

That's a little bit better, but what about multiple links that have the same effect?

```
<a href="#" onclick="paintIt(this, '#990000')">Paint it red</a>
<a href="#" onclick="paintIt(this, '#009900', '#FFFFFF')">Paint it green</a>
<a href="#" onclick="paintIt(this, '#000099', '#FFFFFF')">Paint it blue</a>
```

Not very DRY, eh? We can fix this by using events instead. We'll add a `data-*` attribute to our link, and then bind a handler to the click event of every link that has that attribute:

```
paintIt = (element, backgroundColor, textColor) ->
  element.style.backgroundColor = backgroundColor
  if textColor?
    element.style.color = textColor

$ ->
  $("a[data-background-color]").click (e) ->
    e.preventDefault()

    backgroundColor = $(this).data("background-color")
    textColor = $(this).data("text-color")
    paintIt(this, backgroundColor, textColor)
```

```
<a href="#" data-background-color="#990000">Paint it red</a>
<a href="#" data-background-color="#009900" data-text-color="#FFFFFF">Paint it g
<a href="#" data-background-color="#000099" data-text-color="#FFFFFF">Paint it b
```

We call this 'unobtrusive' JavaScript because we're no longer mixing our JavaScript into our HTML. We've properly separated our concerns, making future change easy. We can easily add behavior to any link by adding the data attribute. We can run all of our JavaScript through a minimizer and concatenator. We can serve our entire JavaScript bundle on every page, which means that it'll get downloaded on the first page load and then be cached on every page after that. Lots of little benefits really add up.

The Rails team strongly encourages you to write your CoffeeScript (and JavaScript) in this style, and you can expect that many libraries will also follow this pattern.

3 Built-in Helpers

Rails provides a bunch of view helper methods written in Ruby to assist you in generating HTML. Sometimes, you want to add a little Ajax to those elements, and Rails has got your back in those cases.

Because of Unobtrusive JavaScript, the Rails "Ajax helpers" are actually in two parts: the JavaScript half and the Ruby half.

rails.js provides the JavaScript half, and the regular Ruby view helpers add appropriate tags to your DOM. The CoffeeScript in rails.js then listens for these attributes, and attaches appropriate handlers.

3.1 form_for

form_for is a helper that assists with writing forms. `form_for` takes a `:remote` option. It works like this:

```
<%= form_for(@article, remote: true) do |f| %>
  ...
<% end %>
```

This will generate the following HTML:

```
<form accept-charset="UTF-8" action="/articles" class="new_article" data-remote='
  ...
</form>
```

Note the `data-remote="true"`. Now, the form will be submitted by Ajax rather than by the browser's normal submit mechanism.

You probably don't want to just sit there with a filled out `<form>`, though. You probably want to do something upon a successful submission. To do that, bind to the `ajax:success` event. On failure, use `ajax:error`. Check it out:

```
$(document).ready ->
  $("#new_article").on("ajax:success", (e, data, status, xhr) ->
    $("#new_article").append xhr.responseText
  ).on "ajax:error", (e, xhr, status, error) ->
    $("#new_article").append "<p>ERROR</p>"
```

Obviously, you'll want to be a bit more sophisticated than that, but it's a start. You can see more about the events in the jquery-ujs wiki.

3.2 form_tag

form_tag is very similar to `form_for`. It has a `:remote` option that you can use like this:

```
<%= form_tag('/articles', remote: true) do %>
  ...
<% end %>
```

This will generate the following HTML:

```
<form accept-charset="UTF-8" action="/articles" data-remote="true" method="post">
  ...
</form>
```

Everything else is the same as `form_for`. See its documentation for full details.

3.3 link_to

link_to is a helper that assists with generating links. It has a `:remote` option you can use like this:

```
<%= link_to "an article", @article, remote: true %>
```

which generates

```
<a href="/articles/1" data-remote="true">an article</a>
```

You can bind to the same Ajax events as `form_for`. Here's an example. Let's assume that we have a list of articles that can be deleted with just one click. We would generate some HTML like this:

```
<%= link_to "Delete article", @article, remote: true, method: :delete %>
```

and write some CoffeeScript like this:

```
$ ->
  $("a[data-remote]").on "ajax:success", (e, data, status, xhr) ->
    alert "The article was deleted."
```

3.4 button_to

button_to is a helper that helps you create buttons. It has a `:remote` option that you can call like this:

```
<%= button_to "An article", @article, remote: true %>
```

this generates

```
<form action="/articles/1" class="button_to" data-remote="true" method="post">
  <div><input type="submit" value="An article"></div>
</form>
```

Since it's just a `<form>`, all of the information on `form_for` also applies.

4 Server-Side Concerns

Ajax isn't just client-side, you also need to do some work on the server side to support it. Often, people like their Ajax requests to return JSON rather than HTML. Let's discuss what it takes to make that happen.

4.1 A Simple Example

Imagine you have a series of users that you would like to display and provide a form on that same page to create a new user. The index action of your controller looks like this:

```
class UsersController < ApplicationController
  def index
    @users = User.all
    @user = User.new
  end
  # ...
```

The index view (`app/views/users/index.html.erb`) contains:

```
<b>Users</b>

<ul id="users">
<%= render @users %>
</ul>

<br>

<%= form_for(@user, remote: true) do |f| %>
  <%= f.label :name %><br>
  <%= f.text_field :name %>
  <%= f.submit %>
<% end %>
```

The `app/views/users/_user.html.erb` partial contains the following:

```
<li><%= user.name %></li>
```

The top portion of the index page displays the users. The bottom portion provides a form to create a new user.

The bottom form will call the `create` action on the `UsersController`. Because the form's remote option is set to true, the request will be posted to the `UsersController` as an Ajax request, looking for JavaScript. In order to serve that request, the `create` action of your controller would look like this:

```
# app/controllers/users_controller.rb
  # ......
```

```
  def create
    @user = User.new(params[:user])

    respond_to do |format|
      if @user.save
        format.html { redirect_to @user, notice: 'User was successfully created.
        format.js   {}
        format.json { render json: @user, status: :created, location: @user }
      else
        format.html { render action: "new" }
        format.json { render json: @user.errors, status: :unprocessable_entity }
      end
    end
  end
```

Notice the format.js in the `respond_to` block; that allows the controller to respond to your Ajax request. You then have a corresponding `app/views/users/create.js.erb` view file that generates the actual JavaScript code that will be sent and executed on the client side.

```
$("<%= escape_javascript(render @user) %>").appendTo("#users");
```

5 Turbolinks

Rails 4 ships with the Turbolinks gem. This gem uses Ajax to speed up page rendering in most applications.

5.1 How Turbolinks Works

Turbolinks attaches a click handler to all `<a>` on the page. If your browser supports [PushState] (https://developer.mozilla.org/en-US/docs/DOM/Manipulating_the_browser_history#The_pushState().C2.A0method), Turbolinks will make an Ajax request for the page, parse the response, and replace the entire `<body>` of the page with the `<body>` of the response. It will then use PushState to change the URL to the correct one, preserving refresh semantics and giving you pretty URLs.

The only thing you have to do to enable Turbolinks is have it in your Gemfile, and put `//= require turbolinks` in your CoffeeScript manifest, which is usually `app/assets/javascripts/application.js`.

If you want to disable Turbolinks for certain links, add a `data-no-turbolink` attribute to the tag:

```
<a href="..." data-no-turbolink>No turbolinks here</a>.
```

5.2 Page Change Events

When writing CoffeeScript, you'll often want to do some sort of processing upon page load. With jQuery, you'd write something like this:

```
$(document).ready ->
  alert "page has loaded!"
```

However, because Turbolinks overrides the normal page loading process, the event that this relies on will not be fired. If you have code that looks like this, you must change your code to do this instead:

```
$(document).on "page:change", ->
  alert "page has loaded!"
```

For more details, including other events you can bind to, check out the Turbolinks README.

6 Other Resources

Here are some helpful links to help you learn even more:

- jquery-ujs wiki
- jquery-ujs list of external articles
- Rails 3 Remote Links and Forms: A Definitive Guide
- Railscasts: Unobtrusive JavaScript
- Railscasts: Turbolinks

Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the master branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.