Ajax Support in Rails

Sang Shin
Michèle Garoche
www.javapassion.com
“Learn with Passion!”

Topics

- Ajax/JavaScript libraries in Rails
- PrototypeHelper
- ScriptaculousHelper
- JavaScriptMacrosHelper
- JavaScript related utility methods
- Ruby JavaScript template

Ajax/JavaScript Libraries in Rails

Ajax/JavaScript Libraries in Rails

- Prototype - used mainly for Ajax
  - *ActionView::Helpers::PrototypeHelper*
- Scriptaculous - used mainly for visual effects, drag and drop, auto-completion
  - *ActionView::Helpers::ScriptaculousHelper*
- JavaScript related utility methods
  - *ActionView::Helpers::JavaScriptHelper*
- Others can be easily added

How to Include Ajax?JavaScript Libraries

- Option 1: Use *<%= javascript_include_tag :defaults %>* in the HEAD section of your page (recommended):
  - Return references to the JavaScript files in your *public/javascripts* directory.
  - Recommended as the browser can then cache the libraries instead of fetching all the functions anew on every request.
- Option 2: Use *<%= javascript_include_tag 'prototype' %>*
  - Will only include the Prototype core library

How to Include Ajax?JavaScript Libraries

- Option 3: Use *<%= define_javascript_functions %>*
  - Will copy all the JavaScript support functions within a single script block.
  - Not recommended.

PrototypeHelper

What is Prototype?

- Prototype is a JavaScript library that provides
  - DOM manipulation
  - Ajax functionality
  - Object-oriented facilities for JavaScript

Ajax Helpers (from Prototype)

- *link_to_remote (vs. link_to)*
  - Uses XmlHttpRequest
  - GET is default
- *form_remote_tag (vs. form_tag)*
  - Uses XmlHttpRequest to send a form
  - POST is default
- *observe_field*
- *observe_form*

PrototypeHelper: link_to_remote

link_to_remote(...)

- *link_to_remote(name, options = {}, html_options = nil)*
- Returns a link to a remote action defined by *options[:url]* (using the *url_for* format) that's called in the background using XMLHttpRequest.
- The result of that request can then be inserted into a DOM object whose id can be specified with *options[:update]*.
- Usually, the result would be a partial prepared by the controller with *render :partial*.

link_to_remote(...) callback functions

- *:loading*: Called when the remote document is being loaded with data by the browser.
- *:success*: Called when the XMLHttpRequest is completed, and the HTTP status code is in the 2XX range.
- *:failure*: Called when the XMLHttpRequest is completed, and the HTTP status code is not in the 2XX range.
- *:complete*: Called when the XMLHttpRequest is complete (fires after success/failure if they are present).

link_to_remote

- *:url* - where to send request
- *:update* - element to update

```
<%= link_to_remote "Ajax call with indicator",
                   :update   => "date",
                   :loading  => "$('indicator').show()",
                   :complete => "$('indicator').hide()",
                   :url      => { :controller => "ajax",
                                  :action     => "current_date" } %>

<div id="date">
  <p>This text will be replaced.</p>
</div>
<%= image_tag "indicator.gif", :id => 'indicator', :style =>
    'display:none' %>
```

link_to_remote(...) Examples

```
# Generates: <a href="#" onclick="new Ajax.Updater('posts',
#                '/blog/destroy/3', {asynchronous:true, evalScripts:true});
#                return false;">Delete this post</a>
link_to_remote "Delete this post",
  :update => "posts",      # The result is inserted into “posts” DOM object
  :url => { :action => "destroy", :id => post.id }

# Generates: <a href="#" onclick="new Ajax.Updater('emails',
#           '/mail/list_emails', {asynchronous:true, evalScripts:true});
#           return false;"><img alt="Refresh" src="/images/refresh.png?" /></a>
link_to_remote(image_tag("refresh"),
  :update => "emails",   # The result is inserted into “emails” DOM object
  :url => { :action => "list_emails" })
```

link_to_remote(...) Examples

```
# You can override the generated HTML options by specifying a hash in
# options[:html].
link_to_remote "Delete this post", :update => "posts",
  :url  => post_url(@post), :method => :delete,
  :html => { :class  => "destructive" }

# You can also specify a hash for options[:update] to allow for easy redirection
# of output to an other DOM element if a server-side error occurs:
# Generates: <a href="#" onclick="new
# Ajax.Updater({success:'posts',failure:'error'}, '/blog/destroy/5',
#      {asynchronous:true, evalScripts:true}); return false;">Delete this post</a>
link_to_remote "Delete this post",
  :url => { :action => "destroy", :id => post.id },
  :update => { :success => "posts", :failure => "error" }
```

PrototypeHelper: form_remote_tag

form_remote_tag(...)

- *form_remote_tag(options = {}, &block)*
- Returns a form tag that will submit using *XMLHttpRequest* in the background instead of the regular reloading POST arrangement.
- Even though it‘s using JavaScript to serialize the form elements, the form submission will work just like a regular submission as viewed by the receiving side (all elements available in *params*)
- The options for specifying the target with :url and defining callbacks is the same as link_to_remote

form_remote_tag

```
<% form_remote_tag(:url => { :controller => 'ajax', :action =>
"save_event" }, :loading  => "$('indicator_form').show()") do %>
  <p>Title<br/>
  <%= text_field :event, :title %></p>

  <p>Location<br/>
  <%= text_field :event, :location %></p>

  <p>
    <%= submit_tag 'Save' %>
    <%= image_tag 'indicator.gif', :id => 'indicator_form', :style
    => 'display:none' %>
  </p>
<% end %>

<div id="count">
  <%= render :partial => 'events_count' %>
</div>
```

PrototypeHelper: observe_field, observe_form

observe_field(...)

- *observe_field(field_id, options = {})*
- Observes the field with the DOM ID specified by *field_id* and calls a callback when its contents have changed.
- The default callback is an Ajax call.
- By default the value of the observed field is sent as a parameter with the Ajax call.
- Required options are either of
  - *:url* - url_for-style options for the action to call when the field has changed.
  - :function - Instead of making a remote call to a URL, you can specify javascript code to be called instead.

observe_field(...)

```
# Generates: new Form.Element.Observer('suggest', 0.25,
# function(element, value) {new Ajax.Updater('suggest',
#     '/testing/find_suggestion', {asynchronous:true, evalScripts:true,
#         parameters:'q=' + value})})
<%= observe_field :suggest,
    :url => { :action => :find_suggestion },
    :frequency => 0.25,
    :update => :suggest,
    :with => 'q'
    %>
```

observe_field(...)

```
# Sends params: {:title => 'Title of the book'} when the
# book_title input field is changed.
observe_field 'book_title',
  :url => 'http://example.com/books/edit/1',
  :with => 'title'

# Sends params: {:book_title => 'Title of the book'} when the focus
# leaves the input field.
observe_field 'book_title',
  :url => 'http://example.com/books/edit/1',
  :on => 'blur'
```

observe_form(...)

- *observe_form(form_id, options = {})*
- Observes the form with the DOM ID specified by *form_id* and calls a callback when its contents have changed. The default callback is an Ajax call.
- By default all fields of the observed field are sent as parameters with the Ajax call.
- The options for *observe_form* are the same as the options for *observe_field*.
- The JavaScript variable value available to the *:with* option is set to the serialized form by default.

PrototypeHelper: periodically_call_remote

periodically_call_remote(...)

- *periodically_call_remote(options = {})*
- Periodically calls the specified url (options[:url]) every options[:frequency] seconds (default is 10).
- Usually used to update a specified div (options[:update]) with the results of the remote call. The options for specifying the target with :url and defining callbacks is the same as link_to_remote.

periodically_call_remote(...)

```
# Call get_averages and put its results in 'avg' every 10 seconds
# Generates:
#      new PeriodicalExecuter(function() {new Ajax.Updater('avg',
#       '/grades/get_averages',
#      {asynchronous:true, evalScripts:true})}, 10)
periodically_call_remote(:url => { :action => 'get_averages' }, :update =>
   'avg')

# Call invoice every 10 seconds with the id of the customer
# If it succeeds, update the invoice DIV; if it fails, update the error DIV
# Generates:
#      new PeriodicalExecuter(function() {new
#            Ajax.Updater({success:'invoice',failure:'error'},
#      '/testing/invoice/16', {asynchronous:true, evalScripts:true})}, 10)
periodically_call_remote(:url => { :action => 'invoice', :id => customer.id },
   :update => { :success => "invoice", :failure => "error" }
```

ScriptaculousHelper

What is ScriptaculousHelper?

- Provides a set of helpers for calling Scriptaculous JavaScript functions, including those which create Ajax controls and visual effects.
- To be able to use these helpers, you must include the Prototype JavaScript framework and the Scriptaculous JavaScript library in your pages.
- The Scriptaculous helpers' behavior can be tweaked with various options.

ScriptaculousHelper Methods

- *draggable_element*
- *drop_receiving_element*
- *sortable_element*
- *visual_effect*

ScriptaculousHelper: draggable_element & drop_receiving_element

draggable_element(...)

- *draggable_element(element_id, options = {})*
- Makes the element with the DOM ID specified by element_id draggable.

```
<%= draggable_element("my_image", :revert
  => true)
```

drop_receiving_element(...)

- *drop_receiving_element(element_id, options = {})*
- Makes the element with the DOM ID specified by element_id receive dropped draggable elements (created by draggable_element) and make an AJAX call.
  - By default, the action called gets the DOM ID of the element as parameter.

```
<%= drop_receiving_element("my_cart", :url =>
  { :controller => "cart", :action => "add" }) %>
```

ScriptaculousHelper: draggable_element & drop_receiving_element

sortable_element(...)

- *sortable_element(element_id, options = {})*
- Makes the element with the DOM ID specified by element_id sortable by drag-and-drop and make an Ajax call whenever the sort order has changed.
- By default, the action called gets the serialized sortable element as parameters.

```
<%= sortable_element("my_list", :url => {
 :action => "order" }) %>
```

In the example, the action gets a "my_list" array parameter containing the values of the ids of elements the sortable consists of, in the current order.

ScriptaculousHelper: visual_effect

visual_effect(...)

- *visual_effect(name, element_id = false, js_options = {})*
- Returns a JavaScript snippet to be used on the Ajax callbacks for starting visual effects.

```
<%= link_to_remote "Reload", :update => "posts",
    :url => { :action => "reload" },
    :complete => visual_effect(:highlight, "posts",
                               :duration => 0.5)
```

JavaScriptMacroHelper

What is JavaScriptMacrosHelper?

- Provides a set of helpers for creating JavaScript macros that rely on and often bundle methods from JavaScriptHelper into larger units.
- These macros also rely on counterparts in the controller that provide them with their backing.
  - The in-place editing relies on ActionController::Base.in_place_edit_for and the autocompletion relies on ActionController::Base.auto_complete_for.

JavaScriptMacrosHelper:
auto_complete_for

auto_complete_for(..)

- *auto_complete_field(field_id, options = {})*
- Adds AJAX autocomplete functionality to the text input field with the DOM ID specified by field_id.
- This function expects that the called action returns an HTML <ul> list, or nothing if no entries should be displayed for autocompletion.

```
# Controller
class BlogController < ApplicationController
  auto_complete_for :post, :title
end

# View
<%= text_field_with_auto_complete :post, title %>
```

JavaScriptMacrosHelper: in_place_editor

in_place_editor(..)

- *in_place_editor(field_id, options = {})*
- Makes an HTML element specified by the DOM ID field_id become an in-place editor of a property.
- A form is automatically created and displayed when the user clicks the element, something like this:

```
<form id="myElement-in-place-edit-form"
 target="specified url">
 <input name="value" text="The content of
 myElement"/>
 <input type="submit" value="ok"/>
 <a onclick="javascript to cancel the editing">cancel</a>
</form>
```

JavaScript Related Utility Methods

button_to_function(name, *args, &block)

- Returns a button that'll trigger a JavaScript function using the onclick handler
- Examples

```
button_to_function "Greeting", "alert('Hello world!')"
button_to_function "Delete", "if (confirm('Really?'))
  do_delete()"
button_to_function "Details" do |page|
  page[:details].visual_effect :toggle_slide
end
button_to_function "Details", :class =>
  "details_button" do |page|
  page[:details].visual_effect :toggle_slide
end
```

link_to_function(name, *args, &block)

- Returns a link that will trigger a JavaScript function using the onclick handler and return false after the fact.
- Examples

```
link_to_function "Greeting", "alert('Hello world!')"
Produces:
<a onclick="alert('Hello world!'); return false;"
   href="#">Greeting</a>

link_to_function(image_tag("delete"), "if (confirm('Really?'))
   do_delete()")
Produces:
<a onclick="if (confirm('Really?')) do_delete(); return false;"
   href="#">
       <img src="/images/delete.png?" alt="Delete"/>
</a>
```

javascript_tag(..., html_options = {}, &block)

- Returns a JavaScript tag with the content inside.
- Examples

javascript_tag "alert('All is good')"

Produces:

```
<script type="text/javascript">
//<![CDATA[
alert('All is good')
//]]>
</script>
```

Ruby JavaScript Template

Ruby JavaScript Template

- Allows you to express the response to user interactions in pure Ruby code, which is then transformed into JavaScript before being sent to the browser
- Easier to debug and maintain
  - Extract Ajax responses from your ERb templates to a separate file
- You can call any Rails methods and assign them to JavaScript variables

Ruby JavaScript Template

- The templates end in .rjs.
- Unlike conventional templates which are used to render the results of an action, these templates generate instructions on how to modify an already rendered page.
  - This makes it easy to modify multiple elements on your page in one declarative Ajax response.
  - Actions with these templates are called in the background with Ajax and make updates to the page where the request originated from.

page Object

- An instance of the *JavaScriptGenerator* object named *page* is automatically made available to your template, which is implicitly wrapped in an *ActionView::Helpers::PrototypeHelper#update_page* block.

JavaScript Template Example

- When an *.rjs* action is called with *link_to_remote*, the generated JavaScript is automatically evaluated.
- Example:

```
link_to_remote :url => {:action => 'delete'}
```

  - The subsequently rendered *delete.rjs* might look like:

```
page.replace_html  'sidebar', :partial => 'sidebar'
page.remove        "person-#{@person.id}"
page.visual_effect :highlight, 'user-list'
```

  - This refreshes the sidebar, removes a person element and highlights the user list.

JavaScript Template Examples

```
# Replace the count div
page.replace_html 'count', :partial => 'events_count'
page.visual_effect :highlight, 'count'

# Clean up the UI
page['event_title'].value = ''
page['event_location'].value = ''
page['event_title'].focus

# Hide the indicator image
page['indicator_form'].hide
```

<< (javascript)

```
# Writes raw JavaScript to the page.
page << "alert('JavaScript with Prototype.');"
```

[] (id)

Returns a element reference by finding it through id in the DOM. This element can then be used for further method calls.

```
page['blank_slate']                  # => $('blank_slate');
page['blank_slate'].show             # => $('blank_slate').show();
page['blank_slate'].show('first').up # => $('blank_slate').show('first').
```

insert_html (position, id, *options_for_render)

Inserts HTML at the specified position relative to the DOM element identified by the given id.

:top - HTML is inserted inside the element, before the element's existing content.

:bottom - HTML is inserted inside the element, after the element's existing content.

:before - HTML is inserted immediately preceding the element.

:after - HTML is inserted immediately following the element.

insert_html (position, id, *options_for_render)

options_for_render may be either a string of HTML to insert, or a hash of options to be passed to ActionView::Base#render. For example:

```
# Insert the rendered 'navigation' partial just before the DOM
# element with ID 'content'.
# Generates: new Insertion.Before("content",
#                                 "-- Contents of 'navigation' partial --");
insert_html :before, 'content', :partial => 'navigation'

# Add a list item to the bottom of the <ul> with ID 'list'.
# Generates: new Insertion.Bottom("list", "<li>Last item</li>");
insert_html :bottom, 'list', '<li>Last item</li>'
```

alert (message)

```
# Display an alert message
page.alert('This message is from Rails!')
```

assign (variable, value)

Assigns the JavaScript variable the given value.

```
# Generates: my_string = "This is mine!";
page.assign 'my_string', 'This is mine!'

# Generates: record_count = 33;
page.assign 'record_count', 33

# Generates: tabulated_total = 47
page.assign 'tabulated_total', @total_from_cart
```

call (function, *arguments, &block)

Calls the JavaScript function, optionally with the given arguments.

If a block is given, the block will be passed to a new JavaScriptGenerator; the resulting JavaScript code will then be wrapped inside function() { ... } and passed as the called function‘s final argument.

 # Generates: Element.replace(my_element, "Goodbye")
page.call '*Element.replace*', '*my_element*', "*Goodbye*"

 # Generates: alert('My message!')
page.call '*alert*', '*My message!*'

Designing Rails Actions for Ajax: Turn Off Layout

A Few Things To Remember

- Turn off layout for actions that receive Ajax call
  - Whatever your action would normally return to the browser, it will return to the Ajax call. As such, you typically don‘t want to render with a layout.

How can I Turn off layouts?

- Turn off layout for a set of actions

```
class SiteController < ActionController::Base
  layout "standard", :except => [:ajax_method,
                                 :more_ajax,
                                 :another_ajax]
end
```

- Optionally, you could do this in the method you wish to lack a layout

```
render :layout => false
```

How can I Tell a Request is a Ajax call?

- *request.xhr?* returns true if the request's "X-Requested-With" header contains "XMLHttpRequest". (The Prototype Javascript library sends this header with every Ajax request.)

```
def name
  # Is this an XmlHttpRequest request?
  if (request.xhr?)
    render :text => @name.to_s
  else
    # No?  Then render an action.
    render :action => 'view_attribute', :attr => @name
  end
end
```

How can I Turn Off Layout only for Ajax calls?

- Dropping the following code fragment in your *ApplicationController* turns the layout off for every request that is an "xhr" request.

```
layout proc{ |c| c.request.xhr? ? false : "application" }
```

Thank you!

We do Instructor-led Codecamps!
http://www.javapassion.com/codecamps