The Rails View

Creating a Beautiful and Maintainable User Experience

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The Rails View

Create a Beautiful and Maintainable User Experience

John Athayde
and Bruce Williams

Edited by Brian P. Hogan

The Facets of Ruby Series

The Rails View

Creating a Beautiful and Maintainable User Experience

John Athayde
Bruce Williams

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Brian Hogan (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93435-687-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2012

5.1 Using Semantic Form Tags

When we build forms to create or update records in Rails, we use the form_for() helper. This helper does a lot of work for us, including automatically determining which HTML action and method to use for the record based on Rails and REST conventions.

We start the form for our ArtFlow creation using form_for() in app/views/creations/_form.html.erb:

```
<%= form_for @creation do |f| %>
<% end %>
```

That's as basic as a form gets, though this is quite a bit less than what's actually useful! What we can see, though, is that our form_for() helper accepts a block (everything between the do and the end) and yields a form builder instance (which we call f).[1] We'll use methods on this form builder instance to generate our fields themselves. We start with name and description, plus a submit button for good measure:

```
<%= form_for @creation do |f| %>
  <%= f.label :name %>
  <%= f.text_field :name %>

  <%= f.label :description %>
  <%= f.text_area :description, cols: 40, rows: 4 %>

  <%= f.submit %>
<% end %>
```

We added labels to our fields with label(); this form builder method generates <label> HTML tags, which tell our users the name of the field and provide an easy way to focus the related form control (by letting users click the labels). We associated our labels to the name and description form controls by passing their names as arguments to label().

The form controls themselves are pretty easy to understand too. The name form control uses text_field(), which produces an <input> element with type="text" suitable for single-line entry. Our description is a more substantial form control, a <textarea>, generated with the text_area() method.

We can't forget our file_field, can we? This would be a pretty poor creation form if we couldn't actually upload a creation!

1. We'll go into detail about form builders in Section 5.2, *Building Custom Form Builders*, on page ?.

Joe asks:

What About form_tag Instead of form_for?

form_tag() is the generic form tag helper and can be used to create ad hoc forms. Because form_tag() doesn't have the model-related smarts of form_for(), we don't use it when the form's subject is a model: we'd rather let Rails do the extra work for us. Why figure out which URL to send the form results to and what the current value of every field is if we don't have to?

One case where it makes sense to use form_tag() is when we'd just like a parameter or two sent to an action: for instance, a search form that submits a parameter named q entered in a text_field_tag().[a]

When your form is backed by a model, use form_for(). When it's not, use form_tag().

a. For an example of a search form that has been implemented with form_tag(), see http://guides.rubyonrails.org/form_helpers.html#a-generic-search-form.

```
➤ <%= form_for @creation, html: {multipart: true} do |f| %>
    <%= f.label :name %>
    <%= f.text_field :name %>
    <%= f.label :description %>
    <%= f.text_area :description, cols: 40, rows: 4 %>

➤   <%= f.label :file %>
➤   <%= f.file_field :file %>

    <%= f.submit %>
  <% end %>
```

Notice we modified our form_for() invocation to pass multipart: true. This option tells Rails to generate the <form> tag with an enctype attribute that will force the request to be encoded as multipart/form-data. If we forgot this, we'd only be sending the filename, so we're glad we caught it!

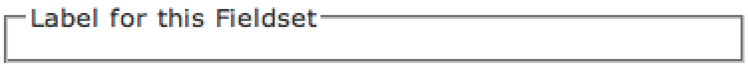
Right now the form is just a loose bag of mixed labels and form controls. We add some structure to more clearly define how these elements are related so it makes a little more sense when we read it (and so we can style it more easily). Let's start by adding a <fieldset>.

The Case for Fieldsets

There is an HTML element ignored by Rails scaffolding (and unknown to many developers) called <fieldset>. It is, as the name would imply, a group of fields that are somehow associated. In the wilds of static HTML, you might see an example like this:

```
<fieldset>
  <legend>Label for this Fieldset</legend>
  <!-- inputs, etc -->
</fieldset>
```

The <legend> here is, in effect, the label for the <fieldset>. A standard browser rendering of this looks something like this:



Using a <fieldset> is great for things like address blocks and credit card blocks in e-commerce systems, not to mention grouping a series of checkboxes or radio buttons into a single input. We use <fieldset> to group related fields in our forms and rely on it for styling. Even when there's one group of fields in a form, it's a good habit to get into, so we'll add it to our creation form, skipping the optional <legend>:

```
<%= form_for @creation do |f| %>
  <fieldset>
    <%= f.label :name %>
    <%= f.text_field :name %>

    <%= f.label :description %>
    <%= f.text_area :description, cols: 40, rows: 4 %>
  </fieldset>
  <%= f.submit %>
<% end %>
```

Let's get these fieldsets lined up.

Laying Out Fields

The layout of the form isn't quite right yet; we have our labels and their related form controls in a <fieldset>, but since they're all inline elements, everything is placed in one long horizontal line!

We'd like a vertical form with each label above its form control, which turns out to be more readable than side-by-side pairings[2]—and definitely more usable than the mess we have now!

We could add some CSS to force each element to display: block, but that won't give us the level of control we'd like for styling, plus it feels a little like a dirty hack. Since it's pretty safe to say that a <fieldset> consists of a *list* of fields,

2. See the extensive usability testing by Luke Wroblewski at http://www.lukew.com/resources/articles/web_forms.html.

let's use a list to break this up; we'd like a tag around each field that groups the label and form control for styling anyway.

```
<%= form_for @creation, html: {multipart: true} do |f| %>
  <fieldset>
    <ol>
      <li>
        <%= f.label :name %>
        <%= f.text_field :name %>
      </li>
      <li>
        <%= f.label :description %>
        <%= f.text_area :description, cols: 40, rows: 4 %>
      </li>
      <li>
        <%= f.label :file %>
        <%= f.file_field :file %>
      </li>
    </ol>
  </fieldset>
  <%= f.submit %>
<% end %>
```

Now we add a little CSS to give each label its own line by default; we want that vertical form! Let's add a style to app/assets/stylesheets/forms.css.scss:

artflow/forms/app/assets/stylesheets/forms.css.scss

```
fieldset label {
  display: block;
}
```

We can't forget to add a require directive for this style sheet to our application.css manifest so that it's included:

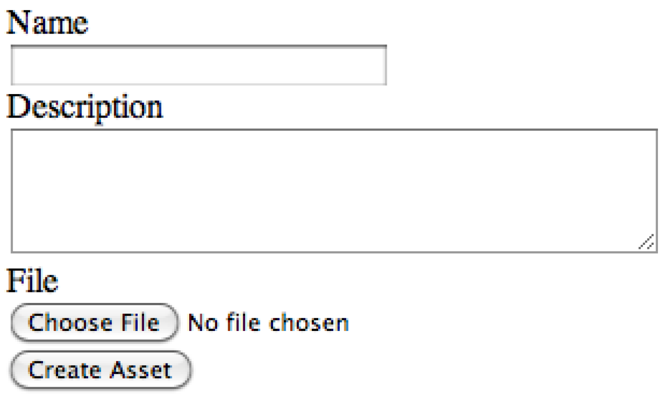
```
/*
 *= require_self
 *= require reset
 *= require layout
 *= require sidebar
➤ *= require forms
 */
```

Things are really coming together; as we can see in the browser, our labels and form controls are perfectly stacked against the left side of the form:

Wrapping Fields in Tags

There's a lot of conflicting opinions about which tag (or tags) is most suitable for grouping labels with their form controls. Persuasive arguments can be made for using `<li>` inside ordered and unordered lists, going the ultra-semantic route with description (formerly "definition") lists (whose `<dt>` and `<dd>` pairs sadly don't provide an easily stylable grouping) and treating the fieldset as a normal flow of paragraph tags. Some might even claim that, as a purely display-related issue, generic `<div>` tags are the best fit.

We picked an ordered list (`<ol>`) because it most resembles how we see the natural structure and purpose of a form: a list of fields our users fill in one by one. The fact that we could come up with a reasonable semantic choice ruled out using a `<div>` from the very beginning.



This is a good start on our form, but what happens when things get more complex?

Side-by-Side Fields

If only our form could stay this simple, but in reality sometimes we need to support side-by-side labels and form controls.

For our creation form we've been tasked to add a secondary fieldset with some printing-related metadata for our creations (since our designers have the good fortune of being tasked with print design, too):

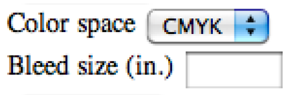
```
<fieldset id='creation-print' class='inline'>
  <ol>
    <li>
      <%= f.label :color_space %>
      <%= f.select :color_space, %w(CMYK RGB Other) %>
    </li>
    <li>
```

```
      <%= f.label :bleed, "Bleed size (in.)" %>
      <%= f.text_field :bleed, size: 5 %>
    </li>
  </ol>
</fieldset>
```

This new fieldset needs to look a bit different, with a smaller font size and side-by-side labels and form controls. Easy enough—with a bit of CSS we can make sure our labels sit to the left of the form controls and line up correctly:

artflow/forms/app/assets/stylesheets/forms.css.scss
```
fieldset#creation-print {
   font-size: 0.9em;
}
fieldset.inline label {
   display: inline;
   width: 240px;
}
```

We normally don't want to use a class named "inline," as it confuses style with markup. For this purpose, we'll show it, but we would want to replace that name with something more meaningful—for example, the contents of the fieldset or some other designator. With this in place, a horizontal layout pops into view for our print-related fields:



Supporting inline fields on a case-by-case basis isn't difficult either. For instance, in our creation form's main fieldset we need to include a form control designers can toggle to indicate if the creation is visible to the client. It's a checkbox, and it looks a bit weird placed below its label instead of next to it. We'll fix that by switching the order of the label and the field, then adding a more focused CSS selector for the <li>:

artflow/forms/app/assets/stylesheets/forms.css.scss
```
fieldset.inline label,
fieldset li.inline label {
  display: inline;
  width: 240px;
}
```

We can even switch the order of the label and the checkbox (so the checkbox is to the left) without having to edit the CSS, since we're just using inline and don't need to deal with floats:

```
<li class='inline'>
  <%= f.check_box :visible %>
  <%= f.label :visible, 'Visible to client?' %>
</li>
```

Our use of .inline for both <fieldset> and <li> brings up an important point: CSS specificity.

The Important of Being Explicit

Imagine if we had multiple developers/designers working on this form (and others in our application), and they all had ideas on different tags that could be used to define content that should be shown inline. What would happen if, instead of using specific CSS selectors, they decided to use a more generic selector like .inline? What if someone else modified it later to match how his/her tag should look? They'd step all over each other, especially in cases where a tag itself wasn't inline but contained elements that should be (like our <fieldset> above). In CSS, context (and specificity) is everything; to prevent unexpected styling elsewhere, we need to make our selectors a little more explicit.

Generic classes can cause nightmares for developers; CSS is hard to document clearly, and tracking down unexpected display behavior (someone else's idea of how everyone else's inline class should look) can be time-consuming and frustrating.

While providing more specific selectors may seem verbose and can create longer style sheets, it can also reduce nightmares of views breaking left and right when development is moving at a good clip. Why slam on the design brakes?

We don't *start* with generalized CSS classes; we work *toward* them as we notice commonalities appear in various places throughout our application. It's an interactive refactoring process that developers will be familiar with elsewhere.

Getting specific is easy to accomplish when using CSS selectors. There is a variety of selectors in the CSS2 and CSS3 references at the W3C.[3]

Now that we've tackled the basics of single-column form layout and have some solid CSS selector practices to follow, we'll look at how usability can become more complicated when our form grows another column.

Tabbing Order

Time passes and requirements change. Now we've been told our little fieldset of print-related metadata, which had been languishing at the bottom of our

3. http://www.w3.org/TR/CSS2/selector.html and http://www.w3.org/TR/css3-selectors/, respectively.

Joe asks:

What About CSS Frameworks?

There are a bevy of CSS frameworks out there, and many have extensive definitions loaded up from the get-go (some of them very useful in laying out forms). While this can help jump-start your work in certain instances, it can also be a chain around your neck as your forms become more complex. We looked at some of these in Chapter 3, *Adding Cascading Style Sheets*, on page ?.

form, needs to be placed at the top right of our form so print designers see it immediately.

Up to the top it goes:

```
  <%= form_for @creation, html: {multipart: true} do |f| %>
➤   <fieldset id='creation-print' class='inline'>
➤     <legend>Print Details</legend>
➤     <ol>
➤       <li>
➤         <%= f.label :color_space %>
➤         <%= f.select :color_space, %w(CMYK RGB Other) %>
➤       </li>
➤       <li>
➤         <%= f.label :bleed, "Bleed size" %>
➤         <%= f.text_field :bleed, size: 5, placeholder: "Inches" %>
➤       </li>
➤     </ol>
➤   </fieldset>
    <fieldset>
      <ol>
        <li>
          <%= f.label :name %>
          <%= f.text_field :name %>
        </li>
        <li>
          <%= f.label :description %>
          <%= f.text_area :description, cols: 80, rows: 4 %>
        </li>
        <li>
          <%= f.label :file %>
          <%= f.file_field :file %>
        </li>
      </ol>
    </fieldset>
    <%= f.submit %>
  <% end %>
```

We've also added a <legend > tag to describe these fields as "print details," and we've set the size and placeholder HTML attributes of bleed to give our users a hint as to the value that's expected (we might as well do things right). Now we float the fieldset to the right with a little CSS:

artflow/forms/app/assets/stylesheets/forms.css.scss
```
fieldset#creation-print {
   float: right;
}
```

Wait, now we've screwed up our tab ordering! Our users spend all day in the application, adding and updating creations, and they rely on tabbing to quickly fill out the form. While we need the print details at the top of the form, we don't want them replacing more important, general-purpose fields in the tab order.

Thankfully there's a way to manually indicate how tabbing between form controls works: while by default tabs will cycle through any <a>, <area>, <button>, <input>, <object>, <select>, or <textarea> element in the page, we can use tabindex to change this behavior.

First off, we need to think like a user. What are the first things our user would be likely to edit in our form? Probably the creation name and *not* the color space or bleed settings (nor the upsell link in the sidebar the guy in marketing added for clients to see). The tab ordering should directly relate to what's important for our users.

These should not be our navigation items. A good case in point from elsewhere in our application is the login form. tabindex="1" should probably be the user or email field, and the next item (2) should probably be the password field. And finally? You got it—the Submit button. We should take the same amount of care with all the forms in our application, especially a form as important as the creation form.

We can see this in our revised form in app/views/creations/_form.html.erb:

```
  <%= form_for @creation, html: {multipart: true} do |f| %>
➤   <fieldset id='creation-print' class='inline'>
➤     <legend>Print Details</legend>
➤     <ol>
➤       <li>
➤         <%= f.label :color_space %>
➤         <%= f.select :color_space, %w(CMYK RGB Other),
➤                      tabindex: '4'  %>
➤       </li>
➤       <li>
➤         <%= f.label :bleed, "Bleed size" %>
```

```
        <%= f.text_field :bleed, size: 5,
                                 placeholder: "Inches",
                                 tabindex: 5  %>
      </li>
    </ol>
  </fieldset>
  <fieldset>
    <ol>
      <li>
        <%= f.label :name %>
        <%= f.text_field :name, tabindex: '1' %>
      </li>
      <li>
        <%= f.label :description %>
        <%= f.text_area :description, cols: 80,
                                      rows: 4,
                                      tabindex: 2  %>
      </li>
      <li>
        <%= f.label :file %>
        <%= f.file_field :file, tabindex: 3  %>
      </li>
    </ol>
  </fieldset>
  <%= f.submit %>
<% end %>
```

Grouping Options

The <option> element is a child element of a form <select>. In our creation form, for instance, we'd like our designers to categorize the type and dimensions of a creation used in advertising across both print and Web.

We could mix these together, but as there's a pretty clear separation conceptually, why not group the options in our <select> as well? This will make it easier for users to find the option they're looking for when filling out the form. We can do this with <optgroup>.

The select() method on form builders doesn't support generating <optgroup> tags, so we'll need to create our <select> using a combination of the more general purpose select_tag() and grouped_options_for_select() helpers. Here's what we put in our form:

```
<%= ad_dimensions_tag(f) %>
```

In our CreationsHelper, we define ad_dimensions_tag():

artflow/forms/app/helpers/creations_helper.rb

```
def ad_dimensions_tag(builder)
  options = grouped_options_for_select(ad_dimensions_options,
```

Automatically Focusing Form Elements

A nice touch to making web apps more usable is to automatically move the focus to the first form field in our form. It's just one less tab that users have to hit—potentially one of many, depending on their settings.

In HTML5, we can simply add autofocus to the input that we want to have this jump to, like so:

```
<%= f.text_field :name, autofocus: true %>
```

Since this only works in browsers that recognize HTML5 attributes on elements, we need to go ahead and provide a JavaScript fallback. We want to detect this so that it doesn't run on our newer browsers. We add a new CoffeeScript file at app/assets/javascripts/forms.js.coffee to handle this.

artflow/forms/app/assets/javascripts/forms.js.coffee
```
$(document).ready ->
  if not Modernizr.input.autofocus
    $('#creation_name').trigger 'focus'
```

Remember Modernizr? We added it to our layout in *Turning on HTML5 for Internet Explorer and Older Browsers*, on page ?. Here we use it to see if the browser supports autofocus. If it doesn't, we autofocus our creation name field manually with a bit of jQuery.

Let's make sure we add a require directive for our form's JavaScript to our application.js manifest:

artflow/forms/app/assets/javascripts/application.js
```
  //= require modernizr-1.7.custom
  //= require jquery
  //= require jquery_ujs
  //= require comments
➤ //= require forms
```

Now we'll add some more design metadata to our creations form.

```
                                             builder.object.ad_dimensions)
  select_tag('creation[ad_dimensions]', options)
end

def ad_dimensions_options
  [
   ['Print',
     ['legal', 'letter', 'half letter', 'half legal', 'other print']],
   ['Web',
     ['full banner', 'half banner', 'vertical banner', 'button']]
  ]
end
```

Our ad_dimensions_tag() helper calls ad_dimension_options() for the possible values of our tag. For the moment we're using a limited set of options—there are a lot of standard advertisement dimensions. Later we may want to pull these from another source or even create a model for them (with an association to Creation).

To determine the current value of the field, ad_dimensions_tag() calls ad_dimensions() on builder.object, which is our creation instance.

The result speaks for itself (Figure 13, *Selection options for ad dimensions*, on page 17).

This provides a much clearer delineation and grouping of select options and allows for a user to find the ad dimensions faster; instead of hunting through a long list of mixed dimensions, a designer working on a legal-sized print ad can find it quicker under Print.

Now we'll see what we tell our users when they submit a form with bad or missing data.

Displaying Errors

Nothing's more frustrating to users than a form that they can't successfully submit for reasons they can't figure out. Providing users with useful feedback when they fail to provide a Rails action with the information it needs (to pass model validations, for instance) is something we need to take very seriously.

In our creation form, a number of the fields we've defined are required for validations. We've taken care to make this apparent by adding a required CSS class to the markup for those fields. For example, for name we use this:

```
<li class='required'>
  <%= f.label :name %>
  <%= f.text_field :name, autofocus: true %>
</li>
```

Our CSS for this is simple; we simply bold the <label>:

artflow/forms/app/assets/stylesheets/forms.css.scss

```
fieldset li.required label {
   font-weight: bold;
}
```

If a user submits our form and a validation for name fails, Rails will wrap both our <label> and <input> (generated by text_field()) in a <div> with a .field_with_errors CSS class when the form is re-rendered.

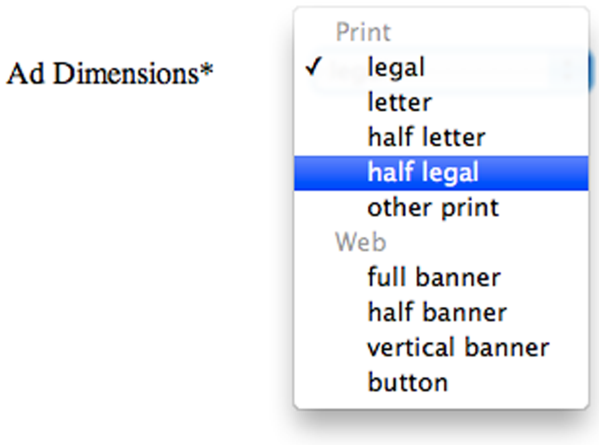


Figure 13—Selection options for ad dimensions

We can highlight our <input> tags and change the text color of our <label> easily enough to make the problem more obvious:

artflow/forms/app/assets/stylesheets/forms.css.scss
```
fieldset div.field_with_errors {
  label {
    color: #f00; /* red */
  }
  input {
    background: #ffc; /* light yellow */
  }
}
```

If we want to show the specific error messages, there are a number of options. We could use the error_messages() form builder method that's been extracted to a separate gem,[4] which would let us add the following to our form:

```
<%= f.error_messages %>
```

This is a pattern that was commonly used before Rails 3, and it provides a block of configurable errors that can be styled to look good, as Yahoo has done in its design pattern library.[5]

Another option is to check and display the errors beside the fields themselves. This is the preferred option, as it puts the cause of the problem directly next to the field where a user can fix it. Instead of adding this nice (but verbose)

4. https://github.com/joelmoss/dynamic_form
5. http://developer.yahoo.com/ypatterns/ and http://developer.yahoo.com/ypatterns/about/stencils/.

feature ourselves, we'll look at a prepackaged solution in *Formtastic*, on page ?.

We've put together a pretty solid beginning on our creation form, starting with a simple form_for() and adding <fieldset> tags, handling different form layouts, and tackling usability issues like grouped options, tabindex, and autofocus.

Next we'll look at how we can extend the form builders form_for() to do more of the heavy lifting, simplifying some of the semantic boilerplate we've been using.