Extracted from:

Rails Recipes

Rails 3 Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

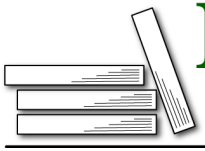
Rails Recipes

Rails 3 Edition

Chad Fowler

Edited by John Osborn

The Facets of Ruby Series



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

John Osborn (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93435-677-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2012

Problem

In a well-designed database, tables are partitioned according to their meanings in the application domain and the most efficient methods of access. As database programmers, we spend a lot of time and energy making sure our databases are well-designed. Unfortunately, this design doesn't always translate well into our user interfaces. What's good for a relational database management system may not be good for a human trying to do data entry. Go figure.

The mismatch between design and UI is a particular problem when we want to create a form for entering or editing data that belongs to two or more models. With the form_for() helper that ships with Rails, we can only create forms that wrap *one* ActiveModel object. So, how can we create a form we can use to interact with data from multiple, associated models?

Solution

The keys to creating multimodel forms in Rails are Active Record's accepts_nested_attributes_for() method and Action View's fields_for() method.

Imagine we have a Recipe model with a has_many() association to ingredients. The model code might look like this:

rr2/nested_forms/app/models/recipe.rb

```
class Recipe < ActiveRecord::Base
  has_many :ingredients
end
```

rr2/nested_forms/app/models/ingredient.rb

```
class Ingredient < ActiveRecord::Base
  belongs_to :recipe
end
```

A recipe has a name and a long text field of instructions. An ingredient belongs to a recipe and has a name and a quantity. Recipes can have many ingredients.

When users create a recipe entry in the site's user interface, they aren't going to think of each ingredient as a separate record, even though that's how we've chosen to model them in the database. They're going to want to create a recipe and its ingredients on a single form. We can enable this using the built-in view helper fields_for().

When we create a form with form_for(), it yields an ActionView::Helpers::FormBuilder to the block in our view. Rails developers usually call this local variable f. The FormBuilder is responsible for wrapping the object for which the form is being built, binding to any existing data in that object, and forming the necessary

parameter names to match the Rails parameter conventions also used by our controllers.

If we call the fields_for() helper on the FormBuilder, Rails constructs a new FormBuilder instance for us, but this time it wraps an *associated* record or set of records rather than the primary subject of the form. Here's an example that wraps our Recipe and Ingredient models:

rr2/nested_forms/app/views/recipes/new.html.erb

```
<h1>Add a Recipe</h1>
<%= form_for @recipe do |f| %>
  <p>
  <%= f.label :name %>
  <%= f.text_field :name %>
  </p>

  <p>
  <%= f.label :instructions %>
  <%= f.text_area :instructions %>
  </p>
  <h2>Ingredients</h2>
  <p>
  <%= f.fields_for(:ingredients) do |ingredients_form| %>
    <%= ingredients_form.label :name %>
    <%= ingredients_form.text_field :name %>
    <%= ingredients_form.label :quantity %>
    <%= ingredients_form.text_field :quantity %>
  <% end %>
  </p>
  <%= f.submit %>
<% end %>
```

To signal to the model that we're going to be pulling in all of these associated attributes from the form, we'll add the following declaration to our Recipe model:

rr2/nested_forms/app/models/recipe.rb

```
accepts_nested_attributes_for :ingredients
```

Combined, these examples create a single form that posts, by convention, to the RecipesController's create() action. However, in addition to the usual fields for the @recipe object, it also wraps fields for a new Ingredient. This seems great so far, but if we were to load this page in our browser, we'd be greeted with an empty list of Ingredients and no way to add one. This is because the fields_for() method generates fields for an *existing* object. If we want to add new Ingredients, we need to first create empty Ingredient objects and associate them with the Recipe.

One way to do that would be to add a new Ingredient to the @recipe when we instantiate it in the new() action in the controller. That might look something like this:

rr2/nested_forms/app/controllers/recipes_controller.rb

```
def new
  @recipe = Recipe.new(:ingredients => [Ingredient.new])
end
```

With this in place, we should see one empty slot for an Ingredient when we view the new Recipe form. Let's look at the generated HTML for the part of the form that wraps associated Ingredients:

```
<h2>Ingredients</h2>
  <p>

    <label for="recipe_ingredients_attributes_0_name">
      Name
    </label>
    <input id="recipe_ingredients_attributes_0_name"
           name="recipe[ingredients_attributes][0][name]"
           size="30"
           type="text" />
    <label for="recipe_ingredients_attributes_0_quantity">
      Quantity
    </label>
    <input id="recipe_ingredients_attributes_0_quantity"
           name="recipe[ingredients_attributes][0][quantity]"
           size="30"
           type="text" />
  </p>
```

From this generated HTML source we can start to get a feeling for how Rails will parse and process this form for our controller. If we submit this form with no values, we'll see the following param structure on the server:

```
{"utf8"=>"✓",
 "authenticity_token"=>"dUdoPRMb9EFdX0oCF6wJ0yhK7R2PAUQ9Dkz3epC0EdM=",
 "recipe"=>{
   "name"=>"",
   "instructions"=>"",
   "ingredients_attributes"=>{"0"=>{"name"=>"", "quantity"=>""}}},
   "commit"=>"Create Recipe"
}
```

Notice that the ingredients_attributes key is nested in the main recipe Hash, which means as per Rails convention, the method ingredients_attributes=() will be invoked when a new Recipe is instantiated with this data. Guess what the accepts_nest-

ed_attributes_for() macro does? That's right! It metaprograms a method onto Recipe, which defines ingredients_attributes=().

If all we need to do is add one ingredient to a Recipe when we create it, we're done. But this still leaves a little to be desired. For example, every recipe is likely to need more than one ingredient, so providing for only a single addition isn't so great. Also, when we're editing an existing recipe, it might be nice to be able to delete associated ingredients. Let's tackle those two problems.

There are many ways to allow users to add ingredients. The simplest way is to simply preallocate a number of empty ingredients whenever the form is loaded. Rather than hard-code this allocation into the controller as we saw in the previous example, let's make a nice model-level method to do it for us. We'll add a new instance method to the Recipe class:

rr2/nested_forms/app/models/recipe.rb

```
def with_blank_ingredients(n = 5)
  n.times do
    ingredients.build
  end
  self
end
```

Now in our call to form_for(), we can add a reference to this method. Because with_blank_ingredients() returns self, its return value can be passed directly into form_for():

rr2/nested_forms/app/views/recipes/new_prealloc.html.erb

```
<%= form_for @recipe.with_blank_ingredients do |f| %>
```

Now five blank ingredients will appear on the form. If we fill out the form, those ingredients will be saved. If we were to use this form for our edit() action, the existing ingredients would appear as well as five blank ingredient fields. Preallocating a set number of blank form elements is a little ugly, but it works. The one major problem with this implementation is that when we save the form, the blank ingredients that we did *not* fill in will also be saved. We can fix that by adding an option to our accepts_nested_attributes_for() call, as shown in the following code:

```
accepts_nested_attributes_for :ingredients,
  :reject_if => lambda { |attrs|
                  attrs.all? { |key, value| value.blank? }
                }
```

This tells the ingredients_attributes=() not to save any Ingredient records whose passed form values are blank.

Finally, let's look at how to remove existing child records in a nested form. One option is, of course, to simply create a button next to each Ingredient row on the form that calls the destroy() action in the IngredientsController. But, our goal here is to allow our users to do as much as possible on this one form, and sending them on round-trips with page refreshes defeats the purpose. So, instead, we can take advantage of yet another Rails convention.

If in our nested form fields we create an attribute called _destroy, we can use it to ask accepts_nested_attributes_for() to automatically destroy nested records for us. Here's what we would have to add to our view:

```
<%= unless ingredients_form.object.new_record?
              ingredients_form.check_box('_destroy') +
                ingredients_form.label('_destroy', 'Remove')
end %>
```

So, if we're working with an Ingredient that has yet to be saved, it doesn't make sense to ask to destroy it. If the record has been saved, we generate a checkbox with the special attribute name _destroy. All that's left to do now is to tell accepts_nested_attributes_for() that it's OK to destroy records. We do that with the :allow_destroy option:

```
accepts_nested_attributes_for :ingredients,
  :reject_if => lambda { |attrs|
                  attrs.all? { |key, value| value.blank? }
                },
  :allow_destroy => true
```

And, now, if we pass down a value for the _destroy attribute associated with an Ingredient, Active Record will destroy that record for us!

Rather than preallocate an arbitrary number of new records for a nested form, it's common practice to use JavaScript to generate those rows. Using your favorite JavaScript library, it can be trivial to templatize and dynamically add elements to the browser's document object model. The trick is in understanding the structure necessary for those new elements. If we look at the generated HTML for one of the Ingredient elements in our solution, we'll see something like this:

```
<input id="recipe_ingredients_attributes_0_name"
       name="recipe[ingredients_attributes][0][name]"
       size="30"
       type="text" />
```

The secret here is that the literal “0” doesn’t have to be a number! It just has to be unique in the set of values we pass from the browser. So, when using JavaScript to dynamically generate nested form elements, you can use any trick for generating a per-form unique value. A good choice, for example, might be to use the current timestamp.

An important point to note about nested forms is that although Rails makes it *relatively* painless to implement them, big forms can clutter the view and make life harder for your users. Before turning to a complex nested form on your next application, ask yourself whether it would be better for the user to break the form into multiple steps.