Extracted from:

Advanced Rails Recipes

Handle Multiple Models in One Form

By Ryan Bates (http://railscasts.com/)
Ryan has been involved in web development since 1998. In 2005 he started working professionally with Ruby on Rails and is now best known for his work on Railscasts, the free Ruby on Rails screencast series.

Problem

Most of the form code you see handles one model at a time. That's not always practical. Sometimes you need to create and/or update two (or more) models in a single form, where there is a one-to-many association between them.

Solution

Let's say we're keeping track of tasks we need to do on projects. When we create or update a project, we'd like to add, remove, and update its tasks in a single form. Here's what we're aiming for:

Let's start by creating a has_many relationship between Project and Task. To keep things simple, we'll give each model a required attribute called name.

```
class Project < ActiveRecord::Base
  has_many :tasks, :dependent => :destroy
  validates_presence_of :name
end
```

```
class Task < ActiveRecord::Base
  belongs_to :project
  validates_presence_of :name
end
```

We'll be using the Prototype JavaScript library, so before we go any further, let's make sure it's loaded in our layout file:

Download MultiModelForm/app/views/layouts/application.html.erb

```
<%= javascript_include_tag :defaults %>
```

Now we turn our attention to the form for creating a project along with its associated, multiple tasks. When dealing with multiple models in one form, it's helpful to make one model the primary focus and build the other models through the association.

In this case, we'll make Project the primary model and build its tasks through the has_many association. So in the new action of our ProjectsController, we create a Project object like normal. However, we also initialize a new Task (in memory) that's associated with the Project so that our form has something to work with:

Download MultiModelForm/app/controllers/projects_controller.rb

```
def new
  @project = Project.new
  @project.tasks.build
end
```

The form template is a bit tricky since we need to handle fields for the Project model and each of its Task models. So, let's break the problem down a bit by using a partial to render the Task fields and an add_task_link helper to create the link that adds a new task:

Download MultiModelForm/app/views/projects/_form.html.erb

```
<%= error_messages_for :project %>

<% form_for @project do |f| -%>
  <p>
    Name: <%= f.text_field :name %>
  </p>
  <div id="tasks">
    <%= render :partial => 'task', :collection => @project.tasks %>
  </div>
  <p>
    <%= add_task_link "Add a task" %>
  </p>
  <p>
    <%= f.submit "Submit" %>
  </p>
```

```
<% end -%>
```

The new and edit templates simply render this form partial so that we have a consistent form for creating and updating a project. The form partial turns around and renders a task partial for each of the project's tasks. Before we get into the contents of the task partial, let's take a look at that add_task_link helper method:

Download MultiModelForm/app/helpers/projects_helper.rb

```
def add_task_link(name)
  link_to_function name do |page|
    page.insert_html :bottom, :tasks, :partial => 'task', :object => Task.new
  end
end
```

When we click the "Add a task" link, we want a new set of task fields to appear at the bottom of the existing task fields in the form. Rather than bother the server with this, we can use JavaScript to add the fields dynamically. The link_to_function method accepts a block of RJS code. We usually associate RJS code with asynchronous calls back to the server. But in this case the RJS code generates JavaScript that gets executed in the browser immediately when the user clicks the link. The upshot is rendering the fields for adding a new task does not require a trip back to the server, which leads to faster response times.

Looking back to the form partial, we're using form_for to dedicate the form to the @project model. How then do we add fields for each of the project's tasks? The task partial holds the answer:

Download MultiModelForm/app/views/projects/_task.html.erb

```
<div class="task">
<% new_or_existing = task.new_record? ? 'new' : 'existing' %>
<% prefix = "project[#{new_or_existing}_task_attributes][]" %>

<% fields_for prefix, task do |task_form| -%>
  <p>
    Task: <%= task_form.text_field :name %>
    <%= link_to_function "remove", "$(this).up('.task').remove()" %>
  </p>
<% end -%>
</div>
```

The key ingredient here is the fields_for method. It behaves much like form_for but does not render the surrounding form HTML tag. This lets us switch the context to a different model in the middle of a form—as if we're embedding one form within another.

The first parameter to fields_for is very important. This string will be used as the prefix for the name of each task form field. Because we'll be using this partial to also render existing tasks—and we want to keep them separate when the form is submitted—in the prefix we include an indication of whether the task is new or existing. (Ideally we'd create the prefix string in a helper, but we've inlined it here to avoid further indirection.)

The generated HTML for a new task name input looks like this:

```
<input name="project[new_task_attributes][][name]" size="30" type="text"/>
```

If this were an existing task, Rails would automatically place the task ID between the square brackets, like this:

```
<input name="project[existing_task_attributes][7][name]" size="30" type="text"/>
```

Now when the form is submitted, Rails will decode the input element's name to impose some structure in the params hash. Square brackets that are filled in become keys in a nested hash. Square brackets that are empty become an array. For example, if we submit the form with two new tasks, the params hash looks like this:

```
"project" => {
  "name" => "Yard Work",
  "new_task_attributes" => [
    { "name" => "rake the leaves" },
    { "name" => "paint the fence" }
  ]
}
```

Notice that the attributes for the project *and* each task are nestled inside the project hash. This is convenient because it means the create action back in our controller can simply pass all the project attributes through to the Project model without worrying about what's inside:

Download MultiModelForm/app/controllers/projects_controller.rb

```
def create
  @project = Project.new(params[:project])
  if @project.save
    flash[:notice] = "Successfully created project and tasks."
    redirect_to projects_path
  else
    render :action => 'new'
  end
end
```

This looks like a standard create action for a single-model form. But there's something subtle happening here. When we call Project.new(params[:project]),

Active Record assumes that our Project model has a corresponding attribute called new_task_attributes because it sees a key called new_task_attributes in the params(:project) hash. That is, Active Record will try to mass assign all the data in the params(:project) hash to corresponding attributes in the Project model. But we don't have a new_task_attributes attribute in our Project model.

One convenient way to keep all this transparent from the controller's perspective is to use a virtual attribute. To do that, we just create a setter method in our Project model called new_task_attributes=, which takes an array and builds a task for each element:

Download MultiModelForm/app/models/project.rb

```
def new_task_attributes=(task_attributes)
  task_attributes.each do |attributes|
    tasks.build(attributes)
  end
end
```

It may not look like these tasks are being saved anywhere. In fact, Rails will do that automatically when the project is saved because both the project and its associated tasks are new records.

That's it for creating a project; now let's move on to updating one.

Just like before, we need to be able to add and remove tasks dynamically, but this time if a task already exists, it should be updated instead. The controller actions need to be concerned only about the project, so they're fairly conventional. As before, the updating of the tasks will be handled in the Project model:

Download MultiModelForm/app/controllers/projects_controller.rb

```
def edit
  @project = Project.find(params[:id])
end

def update
  params[:project][:existing_task_attributes] ||= {}

  @project = Project.find(params[:id])
  if @project.update_attributes(params[:project])
    flash[:notice] = "Successfully updated project and tasks."
    redirect_to project_path(@project)
  else
    render :action => 'edit'
  end
end
```

One important note: The first line of the update action sets the existing_task_attributes parameter to an empty hash if it's not set already. Without this line, there would be no way to delete the last task from a project. If there are no task fields on the form (because we removed them all with JavaScript), then existing_task_attributes() won't be assigned by the form, which means our Project#existing_task_attributes= method won't be invoked. By assigning an empty hash here if existing_task_attributes() is empty, we ensure the Project#existing_task_attributes= method is called to delete the last task.

The form partial can stay the same. However, when we submit the form with existing tasks, the params(:project) hash will include a key called existing_task_attributes. That is, when we update the project, the POST parameters will look like this:

```
"project" => {
  "name" => "Yard Work",
  "existing_task_attributes" => [
    {
      "1" => {"name" => "rake the leaves"},
      "2" => {"name" => "paint the fence"},
    }
  ]
  "new_task_attributes" => [
    { "name" => "clean the gutters" }
  ]
}
```

To handle that, we need to add an existing_task_attributes= method to our Project model, which will take each existing task and either update it or destroy it depending on whether the attributes are passed:

Download MultiModelForm/app/models/project.rb

```
after_update :save_tasks

def existing_task_attributes=(task_attributes)
  tasks.reject(&:new_record?).each do |task|
    attributes = task_attributes[task.id.to_s]
    if attributes
      task.attributes = attributes
    else
      tasks.delete(task)
    end
  end
end

def save_tasks
  tasks.each do |task|
```

```
    task.save(false)
  end
end
```

Notice that we're saving the tasks in an after_update callback. This is important because, unlike before, the existing tasks will not automatically be saved when the project is updated.[18] And since callbacks are wrapped in a transaction, it will properly roll back the save if an unexpected problem occurs.

Passing false to the task.save method bypasses validation. Instead, to ensure that all the tasks get validated when the project is validated, we just add this line to the Project model:

```
validates_associated :tasks
```

This ensures everything is valid before saving. And if validation fails, then the use of error_messages_for :project in the form template includes the validation errors for the project and any of its tasks.

So now we can create and edit projects and their tasks in one fell swoop. And by using virtual attributes, we kept the controller code happily ignorant that we were handling multiple models from a single form.

Discussion

Once you start putting more than one model in a form, you'll likely want to create a custom error message helper to do things such as ignore certain errors and clarify others. See Snack Recipe 17, *Customize Error Messages*, on page 91 for how to write a custom error_messages_for method.

Date fields cause problems because, for some reason, Rails removes the [] from the name of the field. This can be fixed by manually specifying the :index option and setting it to an empty string if the task is new:

```
<%= task_form.date_select :completed_at,
                          :index => (task.new_record? ? '' : nil) %>
```

Unfortunately, checkboxes won't work in this recipe because their value is not passed by the browser when the box is unchecked. Therefore, you cannot tell which task a given checkbox belongs to when a new project

18. This behavior can vary depending on the type of association and whether the records are new. It's a good idea to thoroughly test each combination to ensure every model is validated and saved properly.

is created. To get around this problem, you can use a select menu for boolean fields:

```
<%= task_form.select :completed, [['No', false], ['Yes', true]] %>
```