the { buckblogs :here }

assorted_ramblings.by Buck

Skinny Controller, Fat Model

Posted by Jamis on October 18, 2006 @ 07:50 AM

When first getting started with Rails, it is tempting to shove lots of logic in the view. I'll admit that I was guilty of writing more than one template like the following during my Rails novitiate:

```
<!-- app/views/people/index.rhtml -->
<% people = Person.find(
      :conditions => ["added_at > ? and deleted = ?", Time.now
      :order => "last_name, first_name") %>
<% people.reject { |p| p.address.nil? }.each do |person| %>
  <div id="person-<%= person.new_record? ? "new" : person.id %
    <span class="name">
      <%= person.last_name %>, <%= person.first_name %>
    </span>
    <span class="age">
      <%= (Date.today - person.birthdate) / 365 %>
    </span>
  </div>
<% end %>
```

Not only is the above difficult to read (just you *try* and find the HTML elements in it), it also completely bypasses the "C" in "MVC". Consider the controller and model implementations that support that view:

```
# app/controllers/people_controller.rb
class PeopleController < ActionController::Base
end

# app/models/person.rb
class Person < ActiveRecord::Base
  has_one :address
end
```

Just look at that! Is it really any wonder that it is so tempting for novices to take this approach? They've got all their code in one place, and they don't have to go switching between files to follow the logic of their program. Also, they can pretend that they haven't actually written any Ruby code; I mean, look, it's just the template, right?

For various reasons, though, this is a very, very bad idea. MVC has been successful for many reasons, and some of those reasons are "readability", "maintainability", "modularity", and "separation of concerns". You'd like your code to have those properties, right?

A better way is to move as much of the logic as possible into the controller. Seriously, isn't that what the controller is for? It is supposed to mediate between the view and the model. Let's make it earn its right to occupy a position in our source tree:

```
<!-- app/views/people/index.rhtml -->
<% @people.each do |person| %>
  <div id="person-<%= person.new_record? ? "new" : person.id %
    <span class="name">
      <%= person.last_name %>, <%= person.first_name %>
    </span>
    <span class="age">
      <%= (Date.today - person.birthdate) / 365 %>
    </span>
  </div>
<% end %>
```

```
# app/controllers/people_controller.rb
class PeopleController < ActionController::Base
  def index
    @people = Person.find(
      :conditions => ["added_at > ? and deleted = ?", Time.now
      :order => "last_name, first_name")
    @people = @people.reject { |p| p.address.nil? }
  end
end
```

Better! Definitely better. We dropped that big noisy chunk at the top of the template, and it's more immediately obvious what the structure of the HTML file is. Also, you can see by reading the controller code roughly what kind of data is going to be displayed.

However, we can do better. There's still a lot of noise in the view, mostly related to conditions and computations on the model objects. Let's pull some of that into the model:

```
# app/models/person.rb
class Person < ActiveRecord::Base
  # ...

  def name
    "#{last_name}, #{first_name}"
  end

  def age
    (Date.today - person.birthdate) / 365
  end

  def pseudo_id
    new_record? ? "new" : id
  end
end
```

```
<!-- app/views/people/index.rhtml -->
<% @people.each do |person| %>
  <div id="person-<%= person.pseudo_id %>">
    <span class="name"><%= person.name %></span>
    <span class="age"><%= person.age %></span>
  </div>
<% end %>
```

Wow. Stunning, isn't it? The template is reduced to almost pure HTML, with only a loop and some simple insertions sprinkled about. Note, though, that this is not just a cosmetic refactoring: by moving `name`, `age` and `pseudo_id` into the model, we've made it much easier to be consistent between our

views, since any time we need to display a person's name or age we can simply call those methods and have them computed identically every time. Even better, if we should change our minds and decide that (e.g.) `age` needs to be computed differently, there is now only one place in our code that needs to change.

However, there's still a fair bit of noise in the controller. I mean, look at that `index` action. If you were new to the application, coming in to add a new feature or fix a bug, that's a lot of line noise to parse just to figure out what is going on. If we abstract that code into the model, we can not only slim the controller down, but we can effectively document the operation we're doing by naming the method in the model appropriately. Behold:

```
 1 # app/models/person.rb
 2 class Person < ActiveRecord::Base
 3   def self.find_recent
 4     people = find(
 5       :conditions => ["added_at > ? and deleted = ?", Time.now
 6       :order => "last_name, first_name")
 7     people.reject { |p| p.address.nil? }
 8   end
 9
10   # ...
11 end
12
13 # app/controllers/people_controller.rb
14 class PeopleController < ActionController::Base
15   def index
16     @people = Person.find_recent
17   end
18 end
```

Voila! Looking at `PeopleController#index`, you can now see immediately what is going on. Furthermore, in the model, that query is now self-documenting, because we gave the method a descriptive name, `find_recent`. (If you wanted, you could even take this a step further and override the `find` method itself, as I described in Helping ActiveRecord finders help you. Then you could do something like `Person.find(:recent)` instead of `Person.find_recent`. There's not a big advantage in that approach in this example, so it mostly depends on what you prefer, esthetically.)

Be aggressive! Try to keep your controller actions and views as slim as possible. A one-line action is a thing of wonder, as is a template that is mostly HTML. It is also *much* more maintainable than a view that is full of assignment statements and chained method calls.

Another (lesser) nice side-effect of lean controllers: it allows `respond_to` to stand out that much more, making it simple to see at a glace what the possible output types are:

```
 1 # app/controllers/people_controller.rb
 2 class PeopleController < ActionController::Base
 3   def index
 4     @people = Person.find_recent
 5
 6     respond_to do |format|
 7       format.html
 8       format.xml { render :xml => @people.to_xml(:root => "pec
 9       format.rss { render :action => "index.rxml" }
10     end
11   end
12 end
```

Give all this a try in your next project. Like adopting RESTful practices, it may take some time to wrap your mind around the refactoring process, especially if you're still accustomed to throwing lots of logic in the view. Just be careful not to go too far; don't go putting actual view logic in your model. If you find your model rendering templates or returning HTML or Javascript, you've refactored further than you should. In that case, you should make use of the helper modules that `script/generate` so kindly stubs out for you in `app/helpers`. Alternatively, you could look into using a presenter object.

Posted in **Essays and Rants**

Comments

Have something to add? Click here to leave a comment.

1. Christoffer Sawicki said...

Good write-up! I'd argue that pseudo_id makes more sense like a helper method, just like dom_id in simply_helpful.

2. Geir said...

Good stuff. "Where should I put this code" is a question I quite often find myself asking when writing Rails code. Actually, it should get more attention in the Agile Rails book IMHO.

3. Nicolas Sanguinetti said...

Excellent! :P

I was also guilty of writing ugly code in my views before, as (almost) everyone has been, I'm sure. And I agree with Geir: the question of "Where does this go?" comes up often enough, and is one of the things that **must** be helped from early on.

One thing though, why would you filter by `address.nil?` in a `find(:recent)`? (I mean, what do addresses have to do with **recent**?) I'd leave the controller action to something like:

```
def index
  @people = People.find(:recent).reject { |p| p.address.nil? }
end
```

4. zerohalo said...

Excellent write-up in helping beginners understand the principles of Rails. Keep'm coming!

As Christoffer said, pseudo_id could be a helper, I think it fits fine in the model. I'm sometimes unsure whether to put little functions like that as part of the model or as a helper. Personally, I think helpers are better for things that are used beyond one particular model. However, I do find myself wondering whether I should put a helper that applies to a particular controller only in the controller file itself or in the helper file tied to the controller. I sometimes wonder whether the controller-specific helper files aren't redundant, and that those helpers which are controller-specific should go in the controller, with those that are global in the application_helper file. Then you don't have to look in both the controller and its helper file for stuff. (Same for models.)

5. Rich said...

Wow! Thanks a ton. There's a lot I can learn from this single post! Being new to Rails I often wonder if there is a better way to