11
Mar

9 Essential Rails Tips

posted by gchatz 12 comments rails

In this article:

Rails is a framework about conventions. It generates a basic structure which you mold into your dream application.
Over the course of the years, we have gathered some basic, rails specific, hints and tips you might want to check before going live. They are split into sections to make it easier to scan them, and pick the ones you haven’ t indulged (yet) in. Read on, have fun, and comment a lot…

ActiveRecord

ActiveRecord’ s (AR) magic can be somewhat exciting for the newcomer. But there are some pitfalls to consider:

Don’ t forget to :select

AR is RoR’ s backbone. If you for example wanted to render all users whose username begins with the letter ‘A’ you would do:

```
1. @users = User.find(:all,
2.              :conditions => ["username like ?", "A%"])
```

In fact most of the time you don’ t actually need what the above command returns. It executes a `select *` on the user table returning a carload of data, which leads to memory and cpu bloat.

which leads to memory and cpu bloat.
What you usually want to show is a small set of the User model attributes.

Using @ :select => [list of attribute names for your view] @ you' ll reduce database traffic, keep AR happy and slim, and those mongrels to a minimum of your available system memory:

```
1. @users = User.find(:all,
2. :select => "username, email, registered_on",
3. :conditions => ["username like ?", "A%"])
```

h3. Eager Loading? Yes, but…

One of the basic performance tips in every rails book/class/blog is to use `:include => AssociationName`.
By not using it, the following code:

```
1. <% for post in @posts %>
2.     <!-- ........ -->
3.     <%=h post.user.username %> <!-- user association requires one query -->
4. <% end %>
```

will result in extra queries one for each loop execution.
Using eager loading, you can avoid excessive queries:

```
1. def index
2.     @posts = Post.find(:all, :order => "posted_on desc", :limit => 15, :include =>
          [:user])
3. end
```

Now the user table is joined and we are all happy.

Rails is a convention framework and a pretty smart one too. Rails 2.0 added a caching mechanism to cache queries for the same action in case you (of course by devilish mistake) execute the same query more then once. But that' s just a countermeasure, you should make sure that your queries are properly joined.

?here was a but here, wasn' t it? I can hear you screaming that AR is now including all of the users data, even columns you don' t need.
And you probably think that you' ll just have to use `:select` to avoid it.

ActiveRecord completely ignores the select directive if you use include. Ouch!

There are two ways to work around this issue

- use select_with_include, a plugin that lets you `:select` together with `:include`
- use `:join` , instead of include

select_with_include

First you have to install the gem

```
> gem install select_with_include
```

Then you must require the gem in your environment.rb

```
require 'select_with_include'
```

Now, you can use the :select option, remembering to use the table name even if a column name is not ambigious.

```
def index
    @posts = Post.find(:all,
     :select => "posts.title, posts.created_on, posts.comments_count,
        users.username",
     :order => "posted_on desc",
     :limit => 15,
     :include => [:user])
end
```

Select_with_include doesn't support table selects (e.g. users.*) and calculated fields. If you want to include all columns of a table, you'll have to type each and every one of them.

I don't know if select_with_include has any implications on performance. It's a syntax parser so that should add some overhead but haven't found any sources that actually prove it.

Using :joins, instead of :include

AR's find method provides an option to manually specify join tables, using :joins. In that case AR will not try to create objects for you, it will just append the join directive on the sql output and the :select option will not be ignored. The following code snippets will show you what happens in each case:

```
@posts = Post.find(:all,
:select => "posts.id, posts.title, posts.subject, users.username",  :include =>
   :user)
```

Will generate the following sql command:

```
SELECT  `posts`.`id` AS t0_r0,
          `posts`.`title` AS t0_r1,
          `posts`.`subject` AS t0_r2,
          `posts`.`user_id` AS t0_r3,
           ...
           `users`.`username` AS t1_r1, `users`.
           ....
 LEFT OUTER JOIN `users` ON `users`.id = `posts`.user_id
```

As you can see, the :select option was ignored by AR. We can use :joins instead:

```
@posts = Post.find(:all,
:select => "posts.id, posts.title, posts.subject, users.username",
:joins => "left outer join users on users.id = posts.user_id")
```

that results to an sql query you can actually read:

```
SELECT
```

```
    posts.id,
    posts.title,
    posts.subject,
    users.username
FROM `posts`
LEFT OUTER JOIN users on users.id = posts.user_id
```

Note that you have to access the user's attribute from within the post object and not from the association. That means to access the user name you should use @posts.first.username and not @posts.first.user.username. The latter will execute a new query to fetch the result.

In 99% of the cases that kind of hacking is unnecessary. You should identify parts in your application that need optimizing and apply whatever can make your app go faster.

Save without validations

A typical update or create action will end up with the model being saved by calling AR's `object.save`. Depending on the validation chain results, `save` will return true or false and we'll take the appropriate actions to handle that. Something like this:

```
def some_action
    ....
    if @post.save
end
```

When calling `save` any validations defined in the model are validated in the background (some validations can also mean extra database queries). But what if we are updating the views_count of a blog post? We don't want to add extra load to the database for no reason.

You can call `object.save_with_validation(false)` to skip validations when doing trivial updates that require no validations. You can also use AR's `update` class if you don't have the object in scope.

One more thing to consider. In a perfect world, your model instances should always validate. But there are cases where that will not happen. You added a new validation, someone changed something directly in the database, shit happens.

If you are updating a very important attribute (like someone's purchased credits) avoid calling `object.save`. You don't want a user complaining about missing credits he paid for just because his email didn't have the right format.

Going live

There are some nice little tools in the Rails world which make you wonder how you ever deployed an application without them.

Exception Notifier

First off this nifty little plugin. If you are not using it by now, start using it. It mails you a stack trace for every exception that occurs.

Set it up wisely, use `ExceptionNotifier.email_prefix` with a unique prefix for every application, and filter those mails in your email client.

Asset packager

Rails as most of the other Web 2.0 frameworks comes bundled with a plethora of js files, most of us really never trim down to the very necessary. In order to minimize traffic for the js files you can use the asset packager.

Asset packager has to rebundle the js files each time you make changes. Make sure you set up a nice rake task to automate this.

Rendering the cool way

Partials and layouts in Rails can make your life easy. Nevertheless, some of the defaults that we are used to when coding the rails way, can cause problems if someone doesn't pay any attention to them.

Move js to the bottom

It is important to have the process in mind a browser goes through when rendering your page.

First of all the HTML file is downloaded. Then the browser beginns top to bottom, to parse the HTML and execute if necessary extra calls as they are encountered.

We usually include our javascript files at the top of the layout. That means that browsers will try to download the javascript files before even having the DOM in place. As a result the actual content of the application will come delayed just because the browser was occupied opening connections and downloading files that we are not going to use only until after the markup code is fetched.

Moving your javascripts to the bottom will save you some loading time, especially for the newcomer that will not have the files cached.

DNS lookups for js files should be avoided. If the file doesn't change download it once and serve it from your server.

If you are using rails javascript helpers like autocomplete, which are called before everything is dowloaded, you can still make them work using a simple trick. We've blogged about it before, so you can read everyting about it here

Watch your form's attributes

So you code your form using the form helpers, and then pass the parameters to the newly created or to be updated object and save. Let's say we have this user object with username, email, and purchased_credits and we're designing a form for the user to change his email.

```
<% form_for :user, @user, :url => {:controller => "users", :action =>
   "update_email", :id => @user.id} do |f| -%>
  <p>
    <label for="user_email">Email: </label> <%= f.text_field :email %>
  </p>
  <p>
    <%= submit_tag "Change" %>
</p>
<% end %>
```

Then inside the controller we update the user's mail

```
def update_email
  #all access and db logic handled with before filters
  if @user.update_attributes(params[:user])
    flash[:notice] = "We did it"
  else
    flash[:alert] = "Nope. We didn't make it"
  end
end
```

When you are doing bulk updates like `User.new(params[:user])` or `user.update_attributes(params[:user])` you open the door to anyone updating important attributes that you only have (or should) have access to. In the above example someone can append `user[:purchased_credits] = 1000` and make himself a gift.

What's most important in this case is to always have this pitfall in mind when you are coding forms. The solution may vary depending on your business logic, but there is an elegant way of securing your "for your eyes only" attributes with `attr_accessible` or `attr_protected`.

Always secure your model's important attributes from bulk updates using `attr_accessible` or @attr_protected* in the model declaration. And write tests about it. Always!

```
class User << ActiveRecord::Base

  attr_accessible :username, :email
  #or
  attr_protected :purchased_credits
  #check rails documentation for more info
.......
end
```

There are some more tips, most of them have to do with "application testing" and it wouldn't be fair to squeeze them in one post. Stay tuned for part 2.

Comments (12)

 ek said on Mar 12, 01:12 PM: