

%w(Akita On Rails) * 2.0

Rolling with Rails 2.0 – The First Full Tutorial – Part 2

escrito por AkitaOnRails em December 12th, 2007 @ 01:18 PM

This is the continuation of [Part 1](#).

For the Screencast that I did, that inspired this tutorial, click [here](#)

I am looking for volunteers who are willing to translate this 2 part tutorial into Brazilian Portuguese and someone that can convert this into nice PDFs for everyone to download. I will do it myself eventually but maybe someone else have more time to spare than me right now.

Hope you enjoy the ride!

Other Niceties

So, we are basically done here: a full blog system, with authentication support for administrative tasks. The whole shebang! Now let's walk through a little bit and talk about other niceties in Rails 2.0, some of them 'invisible' to the user.

Query Cache

My favorite change in ActiveRecord is **Query Cache**. The idea is pretty simple: while processing a request, you can end up doing the same SQL query more than once. Sometimes you just do this:

```
@results ||= Posts.find(:all)
```

Meaning you're 'caching' manually. But sometimes you have pretty complex conditions and it may not be obvious on how to cache it. So that's what Query Cache does. We can see it like this:

```
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
  def index
    @posts = Post.find(:all)
    @posts2 = Post.find(:all)
    ...
  end
  ...
end
```

If we call `http://localhost:3000/posts` and then see the `log/development.log`, that's the snippet we're gonna see:

```
Parameters: {"action"=>"index", "controller"=>"posts"}
Post Load (0.000357)  SELECT * FROM `posts`
CACHE (0.000000)  SELECT * FROM `posts`
```

The first finder issued a normal query to the database. But the second one, being identical, will not hit the database again, it will get the results from the internal Cache! That's clever.

This is not a full featured database cache. It is not a replacement for solutions like Memcached. It is only a small addition to the Rails toolset that we would do manually anyway.

Assets

Speaking of performance, there's another thing that annoys web administrators. Let's pretend we are going to add some Ajax at our blog. The first thing to do is add the needed Scriptaculous and Prototype libraries:

```
<!-- app/views/layouts/application.html.erb -->
...
<%= javascript_include_tag :defaults %>
...
```

If we reload the browser and check the generated HTML source code, that's what we are going to get:

```
<script src="/javascripts/prototype.js?1197463288" type=
<script src="/javascripts/effects.js?1197463288" type="t
<script src="/javascripts/dragdrop.js?1197463288" type="
<script src="/javascripts/controls.js?1197463288" type="
<script src="/javascripts/application.js?1197463288" typ
```

Now, that's nasty. And it can get worse, because we will probably add more libraries depending on the complexity of the interface we are building. The problem is: each of these lines represents another HTTP request to the server. We are hitting the web server with at least 5

requests to build one single page.

Let's modify it a bit:

```
<!-- app/views/layouts/application.html.erb -->
...
<%= javascript_include_tag :defaults, :cache => true %>
...
```

Notice the `:cache` option in the `javascript_include_tag` method. To see it working, we have to restart our server in production mode. As a reminder we never created tables there, so first of all, we have to migrate everything we did so far:

```
rake db:migrate RAILS_ENV=production
./script/server -e production
```

Now, reload your browser and check out the HTML source code again:

```
<script src="/javascripts/all.js?1197470157" type="text/
```

Much nicer! All those 5 or more individual HTTP requests to load javascript were reduced to a single one. Rails 2.0 wraps all javascripts with the `:cache` option within a single one. Depending on the size of your website this could mean a faster load time for the client's browser. Now, there are other similar solutions that adds string compression and obfuscation to the mix, but this is an out-of-the-box nicety that's good to have.

Don't forget to quit the server that is production mode and start it over again in development mode.

Ajax Helper

Now, speaking of Ajax, there is some new helpers as well. One of them makes is easy for us to identify individual elements in the browser DOM. Like this:

```
<!-- app/views/posts/index.html.erb -->
...
<% for post in @posts %>
  <% div_for(post) do %>
    <tr>
      <td><%=h post.title %></td>
      <td><%=h post.body %></td>
      <td><%= link_to 'Show', post %></td>
    </tr>
  <% end %>
<% end %>
...

```

Pay attention to the `div_for` helper. We give it the `Post` instance and when we reload the browser at `http://localhost:3000/posts`, that's what we get in the HTML source code in the browser:

```
...
<div class="post" id="post_1">
<tr>
  <td>Hello Brazil!</td>
  <td>Yeah!</td>
  <td><a href="/posts/1">Show</a></td>
</tr>
</div>
...

```

Get it? You get a full blown `div` tag with `class` and `id` already set to a nice default. Now we can use Prototype to grab this single item like this:

```
item = $('post_1')
```

And so on and so forth. For Ajax, this is a blessing. One more DRY-ification.

DISCLAIMER: I am well aware that this is NOT WEB STANDARDS! Meaning: never, ever put a `<tr>` tag between `<div>` tags. First of all, avoid using tables where you don't need to. Reason I am doing this heresy here: just for being very fast and down to the point, because the scaffold already generated tables, I just didn't want to digress changing its layout. Inside inadequate tags: don't.

Atom Feeds

Now let's pretend that we want our web site to have Atom feeds! Of course, every blog has a feed. So how do we do that? Fortunately for us Rails already understands the `.atom` format. That's what we have to do:

```
class PostsController < ApplicationController
  def index
    @posts = Post.find(:all)

    respond_to do |format|
      ...
      format.atom # index.atom.builder
    end
  end
  ...
end
```

By just adding that `format.atom` call, without any blocks after it, will automatically render the `index.atom.builder`. So let's create it:

```
# app/views/posts/index.atom.builder
atom_feed do |feed|
  feed.title "My Great Blog!!"
  feed.updated((@posts.first.created_at))

  for post in @posts
    feed.entry(post) do |entry|
      entry.title(post.title)
      entry.content(post.body, :type => 'html')
      entry.author do |author|
        author.name("Fabio Akita")
      end
    end
  end
end
```

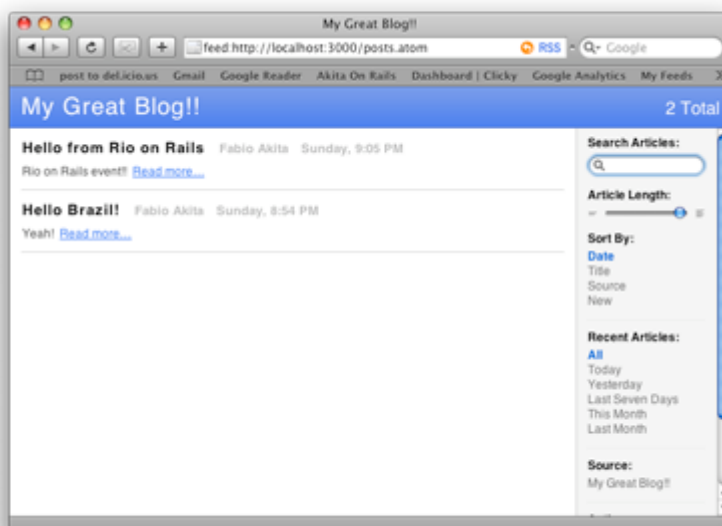
Here, we are using Atom Feed Helper, which is a new addition to Rails 2.0. It is a simplification of the good ol' XMLBuilder that we already had. That's a full blown DSL to build Atom feeds.

Now an explanation, you probably noticed that we don't have templates with the `'rhtml'` extension anymore. Scaffold generated `'html.erb'` extensions. That's because the new convention in Rails is:

```
engine]
```

So, `index.html.erb` means, the template for the `'index'` action, returning `'html'` content and rendered using the `'erb'` engine for templates. Then, `index.atom.builder` means, another template for the `'index'` action, returning Atom content and rendered using the `'builder'` engine. Rails 2.0 will still recognize the old file names for the time being, but you will want to use this new format as soon as possible.

So, with this all set, we can just call `http://localhost:3000/posts.atom` (notice the extension in the URL):



So, Safari understood it as an Atom feed and decorated it properly. Other web clients will behave differently but that's that the Atom Feed Helper generated:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom"
  <id>tag::posts</id>
  <link type="text/html" rel="alternate" href="http://:"
  <title>My Great Blog!!</title>
  <updated>2007-12-09T20:54:15-02:00</updated>
  <entry>
    <id>tag::Post1</id>
    <published>2007-12-09T20:54:15-02:00</published>
    <updated>2007-12-09T20:55:31-02:00</updated>
    <link type="text/html" rel="alternate" href="http://:"
    <title>Hello Brazil!</title>
    <content type="html">Yeah!</content>
    <author>
      <name>Fabio Akita</name>
    </author>
  </entry>
  ...
</feed>
```

Nice, so our blog also has a Feed. But we don't need to stop there.

Supporting the iPhone

Let's do something even fancier. What if I want to have a different homepage when iPhone users access our blog? We can create custom Mime-Types for that. First of all, let's configure the mime type:

```
# config/initializers/mime-types.rb
Mime::Type.register_alias "text/html", :iphone
```

As I explained before, this is a modular environment configuration just for mime types. The line above will register the custom :iphone type to be HTML. We have to restart the server now so this works properly.

Then, we need to change our Posts controller to identify when an iPhone browser hits it.

```
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
  before_filter :adjust_format_for_iphone

  def index
    @posts ||= Post.find(:all)

    respond_to do |format|
      ...
      format.iphone # index.iphone.erb
    end
  end

  ...
  def adjust_format_for_iphone
    if request.env['HTTP_USER_AGENT'] &&
      request.env['HTTP_USER_AGENT'][/iPhone/]
      request.format = :iphone
    end
  end
end
```

DISCLAIMER: It seems that Apple recommends looking for the "Mobile Safari" string instead of "iPhone" as I did in the above snippet of code. I haven't tested it but makes sense.

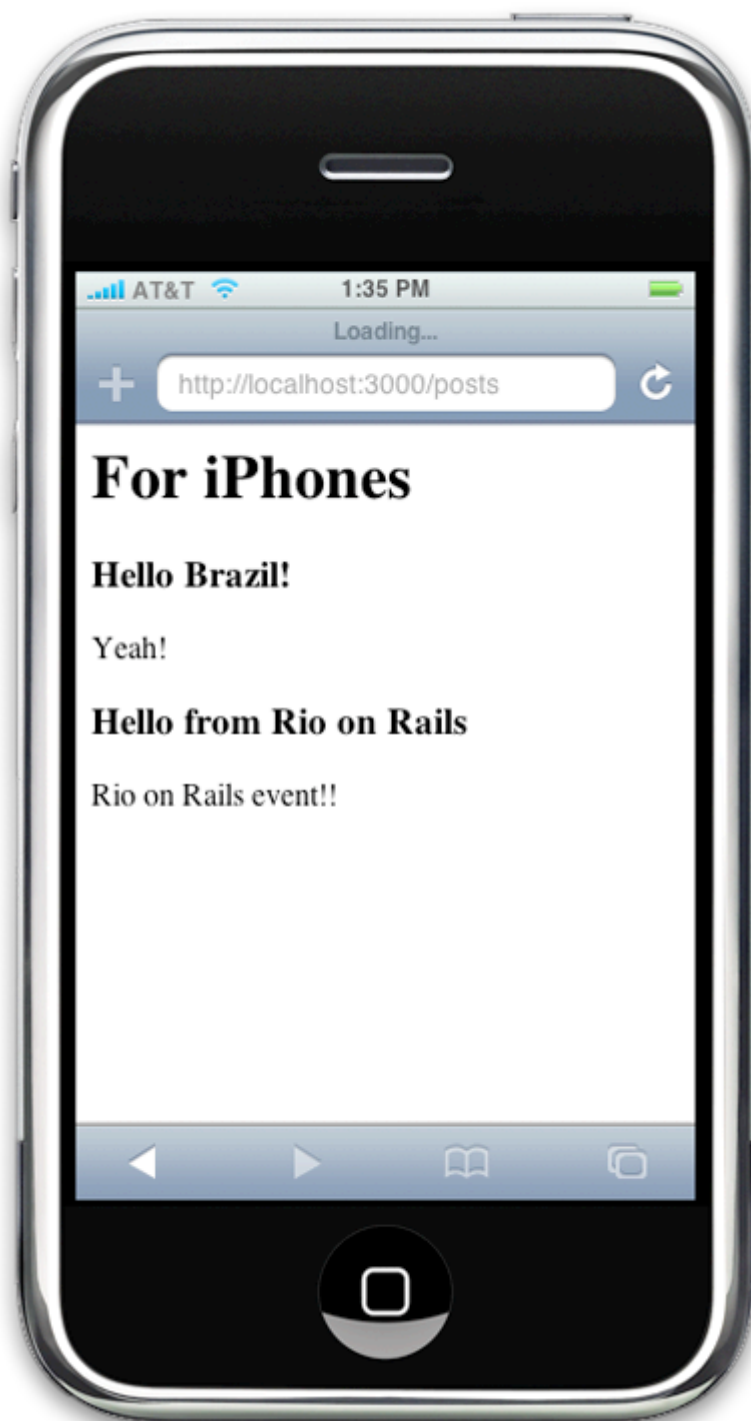
In this particular case we are analysing the HTTP header that was returned to the server from the browser. This data is available in the 'request' hash. So, if the User Agent mentions iPhone, we should act properly by changing the default request.format from :html to :iphone.

And in the 'index' action or any other action that we want to behave differently, it's now just a matter of responding to the correct format. As we explained before, this will render the index.iphone.erb template. Let's create a simplified template for it:

```
<!-- app/views/posts/index.iphone.erb -->
<h1>For iPhones</h1>

<% for post in @posts %>
<h3><%=h post.title %></h3>
<p><%=h post.body %></p>
<% end %>
```

Much simpler. To test it we can use a number of plugins for many browsers that will just send in the correct user agent. We can use the real deal or, if you're in a Mac, download the **iPhoney** simulator. That's what we are going to have:



Great, our Blog is becoming more and more capable by the minute. Not only does it have Atom feeds but a full support for iPhone devices as well. And that's the way we can handle many other devices or file formats as PDFs, CSVs and so forth.

Debugging

The old Rails have something called the Breakpointer. But no one used it seriously because it was not well implemented. A few months ago another player came into the party and that's **ruby-debug**. It is so good that Rails now incorporates hooks for it. First of all, you have to install its gem:

```
sudo gem install ruby-debug
```

Now, we need to restart the server with a new options:

```
./script/server -u
```

Finally, we can just add the 'debugger' command anywhere we want. For instance, let's do it in this place:

```
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
  # GET /posts
  # GET /posts.xml
  def index
    @posts = Post.find(:all)
    debugger
    ...
  end
  ...
end
```

If we reload the browser now at <http://localhost:3000/posts> you will notice that the browser hangs, and if we peek at the terminal, that's what we will have:

```
/Users/akitaonrails/tmp/blog_demo/app/controllers/posts_controller.rb:7 re
(rdb:5) @posts
[#, #]

(rdb:5) @posts << Post.new(:title => "Post from Console")
[#, #, #]

(rdb:5) list
[2, 11] in /Users/akitaonrails/tmp/blog_demo/app/controllers/posts_contro:
 2 # GET /posts
 3 # GET /posts.xml
 4 def index
 5   @posts = Post.find(:all)
 6   debugger
=> 7   respond_to do |format|
 8     format.html # index.html.erb
 9     format.xml { render :xml => @posts }
10   end
11   end

(rdb:5) c
```

So, we have a mix of the script/console shell and the ruby-debug shell together. So we can manipulate all of our Rails objects in real-time, modify them. We can step-through code using the 's'(tep) command or just go on processing using the 'c'(ontinue). The 'list' command is also helpful to show us where we are in the source code. Ruby-debug has dozens of options and commands that's worth mastering.

Nothing replaces a good test suite, though! Do not replace tests for debugging. Both were made to work together. And speaking of tests, Fixtures got some care as well. Rails 2.0 incorporates what's called **Foxy Fixtures**.

Foxy Fixtures

One of the most painful things to do was keeping track of primary keys and foreign-keys within fixtures as everything was hard coded. If you have simple fixtures, that's no big deal. But when you have dozens of fixtures, each with dozens of rows and several kinds of model associations going on, specially many-to-many, then it becomes a burden to maintain everything working.

Now, take a look on how Fixtures look like in Rails 2.0:

```
# test/fixtures/posts.yml
DEFAULTS: &DEFAULTS
  created_at: <%= Time.now %>
  updated_at: <%= Time.now %>

post_one:
  title: MyString
  body: MyText
  <<: *DEFAULTS

post_two:
  title: MyString
  body: MyText
  <<: *DEFAULTS
```

```
# test/fixtures/comments.yml
DEFAULTS: &DEFAULTS
  created_at: <%= Time.now %>
  updated_at: <%= Time.now %>

comment_one:
  post: post_one
  body: MyText
  <<: *DEFAULTS

comment_two:
  post: post_two
  body: MyText
  <<: *DEFAULTS
```

David once said that *naming a fixture row is an art* and it is even more important now. Give fixture rows a meaningful name, one that makes it easy to know which tests requires it.

The new thing here is that we can suppress manually setting the primary key (id field). The other thing is that we can make associations by name, instead of ids. Take a look how we make a comment associated to a post: by the post's name.

And when we have several rows with the same values in the same columns, we can dettach them and reuse in every row we need. In the above example I am dettaching the datetime columns that every row uses. This sounds like yet another eye candy, but it isn't: this will make

tests go to a whole new level. Developers usually don't like to write tests, so making them easier can only help motivate them to do so.

Cookie Store

The default way for Web Apps to control state between requests is to save a session and send its id back and forth to the user's browser through Cookies. This works but leaves us with a problem: where to store those sessions? Up until 1.2 Rails stores all sessions as individual files in the 'tmp' folder. You can also switch on to SQL Session Store (storing sessions in a database table).

The first option has the advantage of being initially faster, but you end up having problems because you're not 'Shared Nothing' anymore: the files are in one server and if you have many boxes you have to deal with nasty stuff like NFS. The database is easier to manage but you pay the price of hitting the database for every user request. Files are also no good if you forget to clean up the tmp folder from time to time. Filesystems can choke easily when you start having hundreds of thousands of small files.

So, Rails 2.0 comes up with a very nice and clean solution called 'Cookie Session Store'. That's the default options. It all starts in the config/environment.rb file where you will find something like this:

```
config.action_controller.session = {  
  :session_key => '_blog_demo_session',  
  :secret      => '2f60d1e...3ebe3b7'  
}
```

It will use strong cryptography to generate a secret key (initial beta versions were insecure, but the released version is very strong).

If you analyze the HTTP packages being transferred you will find this header:

```
Set-Cookie: _blog_demo_session=BAh7BiI...R7AA%253D%253D--f92a00...2dc27c;
```

I shortened the strings here: they are rather large. The concept is: do not store sessions in the server, give it back to the browser. This is the kind of data that the user doesn't care about. The transferred data is just a Base64 blob protected by a Hash to avoid it being tainted.

This follows the **DO NOT STORE BIG OBJECTS IN THE SESSION** and the other mantra **DO NOT STORE SECRET OR CRITICAL DATA IN THE SESSION**. If you're storing big collections of highly complex data structures in the cache, you are probably doing something very wrong. A typical Rails session wouldn't have more than the User ID and maybe a Flash message. So it is very small. Compared to other assets in the web page, it almost weights nothing.

The advantage is that you get rid of server-side maintenance of session files and you don't suffer a performance hit from the database. This is

an excellent option and should be used unless you have some edge condition. If you simply start using Rails 2.0 and never care to look through, you're automatically using the Cookie Store. It's better than using the file-based PStore and forget to clean up the sessions folder just to be bitten by it one day.

Deprecations

So, many things were added. There are much more that I am not talking about in this tutorial. If you are running Rails 1.2.6, take a look at your logs: they will show many things you are using that don't exist anymore at Rails 2.0.

The main things that you are probably still using:

- In the controller, don't use @params, @session, @flash, @request or @env. You now have equivalent methods names, respectively, params, session, flash, request and env. Use them instead.
- In the models, don't use find_one or find_all, use find(:first) and find(:all)

Some main components that were ripped off:

- acts_as_list
- acts_as_tree
- acts_as_nested_set
- All database adapters, except MySQL, SQLite, PostgreSQL
- 'Classic' Pagination
- ActionWebService

The acts_as plugins are now optional and you can still have them. They are all here: <http://svn.rubyonrails.org/rails/plugins/>

The old pagination is still maintained by Err the Blog, so just do:

```
./script/plugin install \  
svn://errtheblog.com/svn/plugins/classic_pagination
```

But of course, don't use it anymore, instead use **will_paginate**.

The database adapters are now separated gems. This is actually a good thing to do because now their individual evolution are not tied to Rails releases. They can release more often and you can choose whether you need them or not. The default way to install them is:

```
sudo gem install activerecord-[database]-adapter
```

The other good thing about all those things being separated is that the Rails Core can remain slimmer, without loading up stuff that most people will never use.

SOAP vs REST

The last deprecation that I like is taking off ActionWebService in favor of ActiveResource. Creating SOAP APIs were not difficult in Rails but not particularly fun as well. Let me show you what you can do with ActiveResource.

Keep the Blog we just create up and running. From another terminal shell, let's create another rails project:

```
rails blog_remote
cd blog_remote
./script/console
```

That's it, a bare bones rails app. For this example the Rails console alone will do. Inside it type the following:

```
class Post < ActiveResource::Base
  self.site = 'http://akita:akita@localhost:3000/admin'
end
```

Now, be amazed! From within the same console, type this now:

```
Post.find(:all)
```

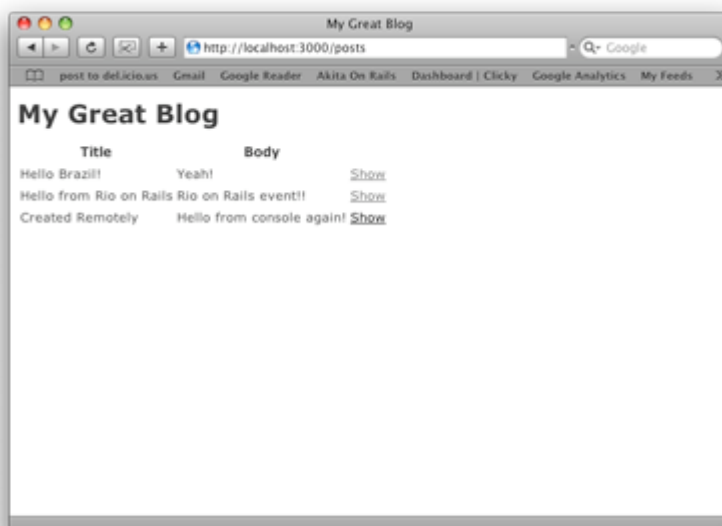
That's right! Remember: in this new Rails project we never configured the database nor created any models. That's one example result from the call we just did:

```
from (irb):4>> Post.find(:all)
=> [#<Post:0x1254ea4 @prefix_options={}, @attributes={
  "updated_at"=>Sun Dec 09 22:55:31 UTC 2007, "body"=>'
  "title"=>"Hello Brazil!", "id"=>1, "created_at"=>Sun D
#<Post:0x1254e90 @prefix_options={}, @attributes={
  "updated_at"=>Sun Dec 09 23:05:32 UTC 2007, "body"=>'
  "title"=>"Hello from Rio on Rails", "id"=>2,
  "created_at"=>Sun Dec 09 23:05:32 UTC 2007}>]
```

We can even create a new post, like this:

```
>> p = Post.create(:title => "Created Remotely",
  :body => "Hello from console again!")
=> #<Post:0x122df48 @prefix_options={}, @attributes={
  "updated_at"=>Wed Dec 12 15:13:53 UTC 2007,
  "body"=>"Hello from console again!", "title"=>"Created R
  "id"=>3, "created_at"=>Wed Dec 12 15:13:53 UTC 2007}>
```

Now, go back to your browser and refresh <http://localhost:3000/posts>



Got it? We now have 2 Rails applications, and one of them can speak to the other through RESTful based HTTP calls! More than that: when we created the Post ActiveResource, we gave in the username and password for the HTTP Basic Authentication we did before. That's one of the advantages of it: it is easier to make API calls later.

So, we didn't change ONE SINGLE LINE OF CODE from our application and are already able to make remote integration. All of this thanks to the RESTful way of building Rails app: you get full APIs for every RESTful resource you create. Follow the Conventions and it becomes that easy.

Conclusion

So, Rails 2.0 is packed with nice additions, lot's of optimizations and doesn't make you re-learn everything from scratch which is good. But let's say you already have a Rails 1.2 app up and running, does it makes sense to blindly update it to 2.0?

Not always. We need to use our common sense here:

- is there something in 2.0 that you want your app to have?
- is your App fully covered with a competent Test Suite?
- does your 3rd party plugins and gems already work with Rails 2.0?

Without answering those questions you should not try to update. That's why you can freeze the old gems in your app:

```
rake rails:freeze:gems -v1.2.6
```

You can also have both Rails 1.2 and 2.0 gems installed. To use the older rails command to create a 1.2 project do the following:

```
rails _1.2.6_ [project_name]
```

Make sure you have everything covered. 1.2.6 will give you several

warnings in the logs. Follow them to adjust your app to be 2.0-ready.

And if you're a novice, still learning Rails, you will not find many resources to learn just yet: every book released until now only covers 1.2. But this is not a problem: learn 1.2 first. Then you can either wait for a brand new 2.0 book to be release (I am going to release one early next year), follow tutorials like mine and you should be able to learn 2.0 in a matter of a few days, maybe only a few hours.

Rails 2.0 is not a revolution by any means, it is a very welcome evolution, polishing and refinement of what was already good. That's a good thing.

I hope my tutorial helped everybody to get up to speed in some of the new features of Rails 2.0.

And if you want to get the **full source code** for the blog we just did, download it from [here](#). Have fun!

publicado em: **Dicas e Tutoriais, English, Rails 2.0** ⋮ aceitando comentários

16 comentários