%w(Akita On Rails) * 2.0

Rolling with Rails 2.0 – O Primeiro Tutorial Completo – Parte 2

escrito por AkitaOnRails em January 31st, 2008 @ 11:45 PM

Para o screencast que eu fiz, que inspirou este tutorial, clique **aqui**. Para ler a primeira parte deste tutorial, clique **aqui**. O código completo está disponível **aqui**. E cliquem **aqui** se quiserem imprimir esta página.

Espero que todos aproveitem!

Outros detalhes

Nós temos basicamente feito até aqui: um sistema de blog completo, com suporte autenticado a tarefas administrativas. Agora vamos avançar um pouco e falar sobre outros detalhes no Rails 2.0, alguns deles invisíveis para o usuário.

Query Cache

Minha alteração favorita no ActiveRecord é **Query Cache**. A idéia é bem simples: enquanto processa uma requisição, você pode acabar fazendo a mesma query SQL mais de uma vez. Às vezes você faz isso:

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

Significa que você está fazendo 'caching' manualmente. Mas às vezes você tem condições bem complexas e pode não ser óbvio como fazer cache disto. Isto é o que Quary Cache resolve. Nós podemos ver isto assim:

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

Se chamarmos http://localhost:3000/posts e olharmos o log/development.log, isto é um trecho do que vamos ver:

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

A primeira busca emite uma query normal para o banco de dados. Mas a segunda, sendo idêntica, não irá ao banco de dados novamente, trará os resultados do cache interno! Isto é inteligente.

Isto não é um cache de banco de dados completo. Isto não é uma substituição de soluções como cache de memória. Isto é somente um pequeno acréscimo ao conjunto de ferramenta do Rails que nós faríamos manualmente de qualquer jeito.

Vantagens

Falando em performance, há uma outra coisa que perturbam administradores web. Vamos imaginar que nós adicionamos algum Ajax ao nosso blog. A primeira coisa a fazer é adicionar as bibliotecas Scriptaculous e Prototype necessárias:

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

Se nós recarregarmos o browser e verificarmos o código fonte HTML gerado, isto é o que teremos:

```
<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
```

Isto é horrível! E pode ficar pior porque provavelmente adicionaremos mais bibliotecas dependendo da complexidade da interface que estamos construindo. O problema é: cada uma destas linhas representa uma

requisição HTTP ao servidor. Nós estamos acionando o servidor web com pelo menos 5 requisições para construir uma única página.

Modificaremos isto um pouco:

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

Repare a opção :cache no método javascript_include_tag. Para ver isto funcionando, temos que reiniciar nosso servidor em modo produção. Lembrando que não criamos tabelas lá ainda, então, antes de tudo, temos que migrar tudo que fizemos antes:

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

Agora, recarregue seu browser e verifique o código HTML novamente:

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

Muito bom! Todas aquelas 5 ou mais requisições HTTP individuais para carregar javascript foram reduzidas a uma única. Rails 2.0 empacota todos os javascripts com a opção :cache dentro de um único arquivo. Dependendo do tamanho do seu website isto pode significar um carregamento mais rápido no browser do cliente.

Não esqueça de sair do servidor que está em modo produção e iniciá-lo novamente no modo desenvolvimento.

Ajax Helper

Falando sobre Ajax, há alguns novos helpers também. Um deles nos facilita a identificar elementos DOM individuais no browser. Assim:

```
<!-- 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 %>
...
```

Preste atenção para o helper div_for. Criamos um Post, por exemplo, e

quando recarregamos o browser em http://localhost:3000/posts, isto é o que temos no código fonte HTML:

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

Entendeu? Você pegou todas as tags div com class e id já configurados por padrão. Agora você pode usar Prototype para capturar este item individualmente assim:

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

E assim por diante. Para Ajax, isto é ótimo. Mais uma DRY-ificação.

RETRATAÇÃO: Eu estou bem ciente que isto NÃO É WEB STANDARDS! Ou seja: nunca coloque uma tag <tr> entre tags <div>. Primeiro, evite o uso de tabelas quando você não necessita delas. Razão pela qual estou fazendo esta heresia aqui: para ser mais rápido e direto ao ponto. Como o scaffold já gerou tabelas, eu não quis perder tempo trocando o layout. Não faça isto.

Atom Feeds

Vamos imaginar que queremos que nosso website tenha Atom Feeds! Claro, todo blog tem um feed. Como fazemos isto? Felizmente Rails já entende o formato atom. Isto é o que temos que fazer:

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

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

Adicionando aquela chamada format.atom, sem qualquer bloco depois, vamos renderizar automaticamente o index.atom.builder. Vamos criar:

```
# 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
```

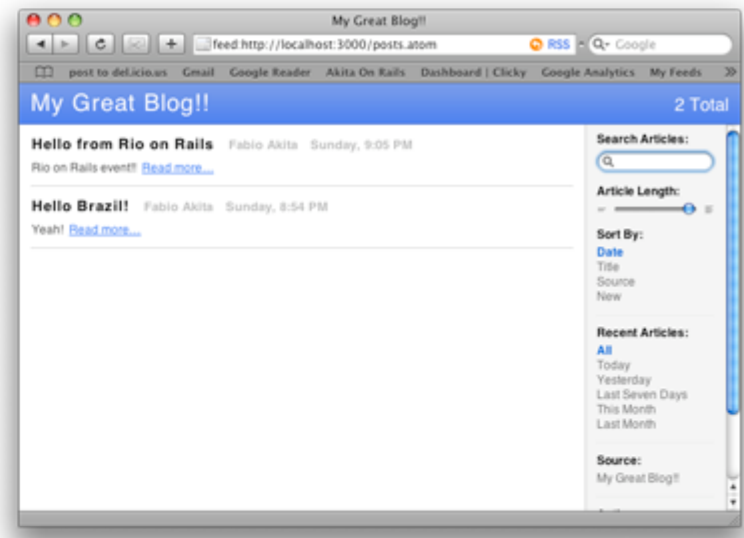
Aqui, estamos usando o Helper Atom Feed, que é novo no Rails 2.0. Ele é uma simplificação do velho e bom XMLBuilder que já tínhamos. Isto é uma total DSL para construir feeds Atom.

Agora uma explicação. Você provavelmente reparou que não temos templates com a extensão 'rhtml'. O Scaffold gerou extensões 'html.erb'. Isto porque a nova convenção em Rails é:

```
engine]
```

Então, index.html.erb significa, o template para o action 'index', retornando conteúdo 'html' e renderizado usando o engine 'erb' para templates. Logo, index.atom.builder significa outro template para o action 'index', retornando conteúdo Atom e renderizado usando o engine 'builder'. Rails 2.0 ainda reconhecerá os antigos nomes de arquivo por enquanto, mas você deverá usar este novo formato assim que possível.

Com isto tudo pronto, podemos chamar http://localhost:3000/posts.atom (repare a extensão na URL):



Assim, o Safari entendeu como um feed Atom e mostrou corretamente. Outros browsers se comportarão diferente mas isto é o que o Atom Feed Helper gerará:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Ato
  <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>
```

Legal, nosso blog tem também um Feeds. Mas não precisamos parar por aí.

Suporte ao iPhone

Vamos fazer algo mais complexo. O que quero de diferente na homepage quando usuários do iPhone acessarem o blog? Nós podemos criar Mime-Types customizáveis pra ele. Primeiro vamos configurar o mime type:

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

Como eu expliquei antes, esta é uma configuração de ambiente modular só para mime types. a linha acima registrará o tipo :iphone customizado para ser HTML. Temos que reiniciar o servidor agora para isto funcionar corretamente.

Então, precisamos alterar o controller Posts para identificar quando um browser iPhone chegar a ele.

```
# 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
```

RETRATAÇÃO: Parece que a Apple recomenda procurar pela string "Mobile Safari" em vez de "iPhone" como eu fiz no pedaço de código acima. Eu não testei isso mas faz sentido.

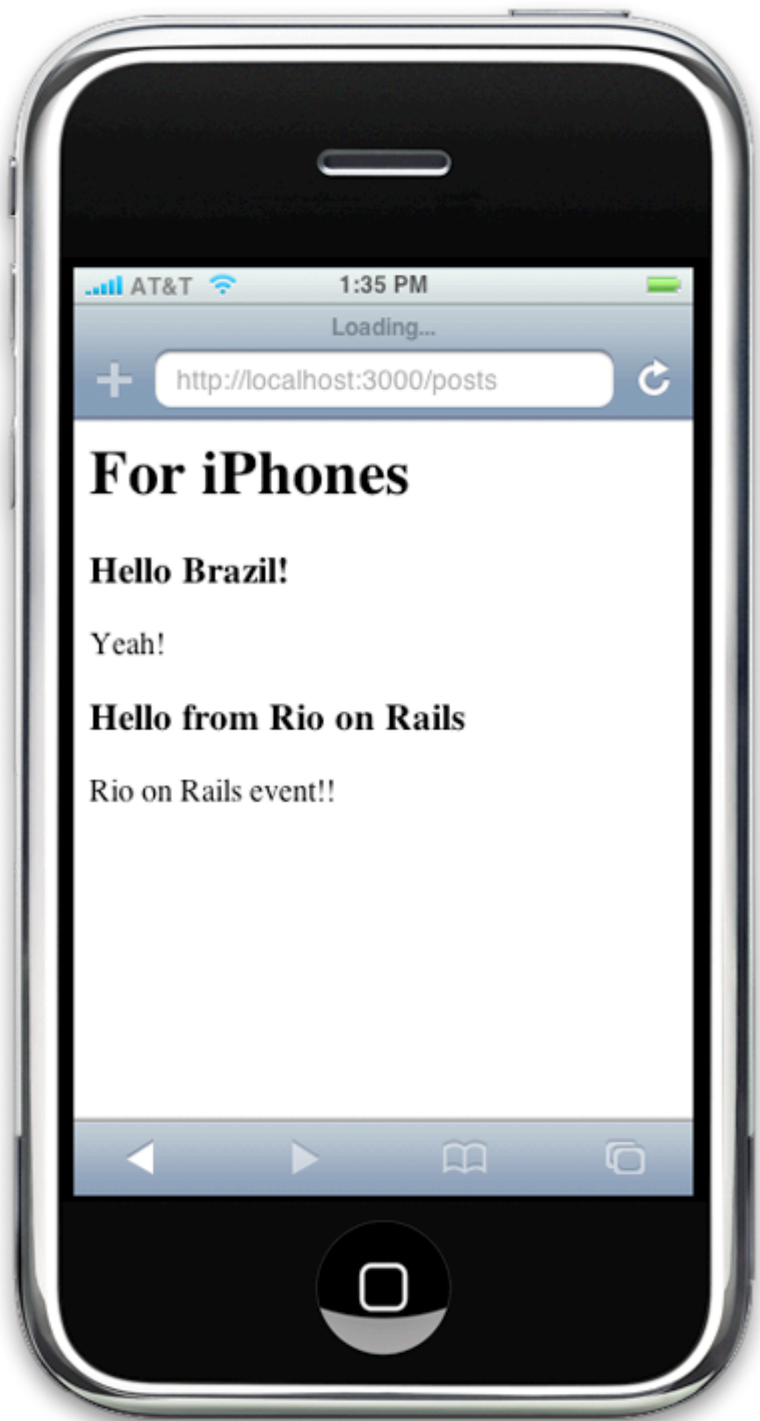
Neste caso particular nós estamos analisando o header HTTP que foi retornado do browser para o servidor. Este dado está disponível no hash de requisição. Assim, se o User Agent mencionar iPhone, nós deveríamos agir corretamente trocando o formato da requisição padrão de :html para :iphone.

E na action 'index' ou qualquer outra action que precise se comportar diferentemente. Agora é só uma questão de responder no formato correto. Como explicado anteriormente, isto renderizará o template index.iphone.erb. Vamos criar um template simplificado pra ele:

```
<!-- 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 %>
```

Muito simples. Para testar podemos usar um grande número de plugins de muitos browsers que enviarão o user agent correto. Podemos usar próprio iPhone, ou se você está num Mac, baixar o **iPhone** simulator. Isto é o que teremos:



Ótimo. Nosso blog está se tornando mais e mais completo que antes. Não somente conseguimos ter feeds Atom mas um suporte completo para o iPhone também. E assim é como podemos tratar muitos outros dispositivos ou formatos de arquivos como PDFs, CSVs e por aí vai.

Debugando

O antigo Rails tem algo chamado Breakpointer. Mas ninguém usa porque não foi bem implementado. Há poucos meses atrás um outro foi incorporado ao conjunto que é o **ruby-debug**. Ele é tão bom que foi integrado ao Rails. Antes de tudo, você tem que instalá-lo:

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

Agora temos que reiniciar o servidor com uma nova opção:

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

Finalmente podemos adicionar o comando ‘debugger’ em qualquer lugar que quisermos. Por exemplo, vamos fazer isto aqui:

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

Se recarregarmos o browser agora em http://localhost:3000/posts iremos reparar que ele parou. E se olharmos no terminal, isto é o que temos:

```
/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
```

Temos uma mistura do shell script/console e do shell ruby-debug juntos. Assim, podemos manipular todos os objetos Rails em tempo real, modificando-os. Podemos andar através do código usando o comando ’s’(tep) ou continuar o processamento usando o comando ‘c’(ontinue). O comando ‘list’ também ajuda a mostrar onde estamos no código fonte. Ruby-debug tem dúzias de opções e comandos que são de suma importância.

Nada substitui um bom conjunto de testes. Não substitua testes por debugging. Ambos fazem o trabalho juntos. E falando em testes, Fixtures receberam algum cuidado também. Rails 2.0 incorpora o chamado **Foxy Fixtures**.

Foxy Fixtures

Uma das coisas mais chatas de fazer é ficar de olho nas chaves primárias e estrangeiras em fixtures enquanto desenvolvemos. Se você tem fixtures simples, isto não é muito importante. Mas quando você tem dúzias, cada qual com outras dúzias de linhas e vários tipos de associações existentes no modelo, especialmente many-to-many, então torna-se difícil manter tudo funcionando.

Agora dê uma olhada em como Fixtures se parecem no 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 uma vez disse que *nomear uma linha fixture é uma arte* e isto é ainda mais importante agora. Dar às linhas da fixtures um nome significativo, um que torne fácil saber de qual teste ela é.

A novidade aqui é que podemos suprimir a chave primária (campo id). A outra coisa é que podemos fazer associações por nome ao invés de ids. Repare como fazemos um comment associado ao post: pelo nome do post.

E quando temos várias linhas com o mesmo valor nas mesmas colunas, podemos destacá-las e reusá-las. No exemplo acima eu destaquei a

coluna datetime que toda linha usa. Isto pode parecer apenas firula, mas não é: isto faz os testes irem para um novo nível. Desenvolvedores geralmente não gostam de escrever testes, então tornando-os mais fáceis isso se torna um motivador.

Armazenamento de Cookie

A maneira padrão para controlar o estado de aplicações web entre requisições é salvar uma session e enviar seu id para o browser do usuário através de Cookies. Isto funciona mas nos trás um problema: onde armazenar aquelas sessions? Até a versão 1.2 Rails armazenava todas como arquivos individuais na pasta 'tmp'. Você pode também usar SQL Session Store (armazenar sessions em uma tabela no banco de dados).

A primeira opção tem a vantagem de ser inicialmente mais rápida, mas você acaba tendo problemas porque não há nada compartilhado: os arquivos estão em um servidor e se você tiver muitos servidores tem que usar algo sórdido como NFS. O banco de dados é fácil de gerenciar mas você paga o preço de acessá-lo a cada requisição. Arquivos também não são uma boa idéia se você esquecer de limpar a pasta tmp de tempos em tempos. Sistemas de arquivos podem engasgar facilmente quando começa a ter centenas de milhares de arquivos pequenos.

Então, Rails 2.0 propõe uma solução muito legal e limpa chamada 'Armazenamento de Session em Cookie'. Esta é a opção padrão. Tudo começa no config/enviroment.rb onde encontrará algo como:

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

É utilizada criptografia forte para gerar uma chave secreta (versões betas iniciais eram inseguras, mas a versão lançada é bem forte).

Se você analisar os pacotes HTTP sendo transferidos, encontrará este cabeçalho:

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

Encurtei as strings aqui: elas são bem maiores. O conceito é: não armazene sessões no servidor, mande-as de volta ao browser. É o tipo de dado que o usuário não liga. Os dados transferidos consistem de um blob Base64 protegido por um Hash para evitar corrompimento.

Isto segue os mantras **NÃO ARMAZENE OBJETOS GRANDES NA SESSÃO** e **NÃO ARMAZENE DADOS CRÍTICOS OU SECRETOS NA SESSÃO**. Se você está armazenando grandes coleções de estruturas de dados complexas em cache, provavelmente está fazendo algo bem errado. Uma sessão Rails típica não deveria conter mais do que o id do usuário e, talvez, uma mensagem Flash. Então ela é bem pequena.

Comparada a outros recursos na página web, ela praticamente não tem peso.

A vantagem é que você se livra da manutenção de arquivos de sessão no servidor e não sofre perda de performance no banco de dados. É uma excelente opção e deve ser usada a menos que se tenha alguma condição extrema. Se você simplesmente começar a usar Rails 2.0 e nunca ligar para isso, estará automaticamente usando o Armazenamento em Cookie. É melhor que usar em arquivos e esqueçer de limpar o diretório de sessões para apenas ser mordido por isso um dia.

Deprecações

Muitas coisas foram adicionadas. Há muito mais coisas que não estou falando neste tutorial. Se você está rodando o Rails 1.2.6, dê uma olhada em seus logs: eles mostrarão muitas coisas que você usa e que não existem mais no Rails 2.0.

As principais coisas que você provavelmente ainda usa:

- No controller, não use @params, @session, @flash, @request ou @env. Você agora tem métodos com nomes equivalentes, respectivamente: params, session, flash, request and env. Use-os no lugar dos outros.
- Nos models, não use find_one ou find_all, use find(:first) e find(:all)

Os principais componentes que foram retirados:

- acts_as_list
- acts_as_tree
- acts_as_nested_set
- Todos os adapters de banco de dados exceto MySQL, SQLite, PostgreSQL
- 'Classic' Pagination
- ActionWebService

Os plugins acts_as_* são opcionais e você ainda pode tê-los. Eles estão todos aqui: http://svn.rubyonrails.org/rails/plugins/

A antiga paginação ainda é mantida pelo "Err the Blog", então faça:

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

Mas claro, não use mais isto, use **will_paginate**.

Os adapters de bancos de dados agora estão separados em gems. Isto é uma boa idéia porque suas evoluções individuais não ficam presas aos lançamentos do Rails. Eles podem liberados mais freqüentemente e você pode escolher se precisa deles ou não. O jeito padrão para instalá-los é:

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

A outra boa coisa sobre todas aquelas coisas serem separadas é que o Rails Core pode ficar mais leve, sem carregar coisas que muitos nunca usarão.

SOAP vs REST

A última deprecação que eu gostei foi retirar o ActionWebService em favor do ActiveResource. Criar APIs SOAP não era difícil em Rails mas não era divertido também. Deixe-me mostrar o que se pode fazer com ActiveResource.

Mantenha o Blog que criamos rodando. De outro terminal shell, vamos criar outro projeto Rails:

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

Isto é uma aplicação Rails mínima. Para este exemplo o console Rails sozinho basta. Nele digite o seguinte:

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

Agora, fique surpreso! No mesmo console, digite isto agora:

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

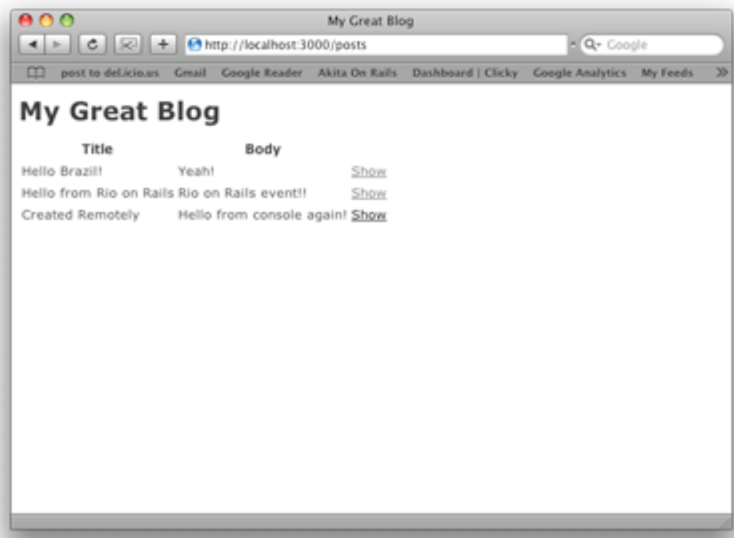
Isso Mesmo! Lembre: neste novo projeto Rails nós não configuramos o banco de dados ou criamos qualquer model. Isto é um resultado de exemplo da chamada que fizemos:

```
from (irb):4>> Post.find(:all)
=> [#<Post:0x1254ea4 @prefix_options={}, @attributes={
  "updated_at"=>Sun Dec 09 22:55:31 UTC 2007,
  "body"=>"Yeah!", "title"=>"Hello Brazil!", "id"=>1,
  "created_at"=>Sun Dec 09 22:54:15 UTC 2007}>,
  #<Post:0x1254e90 @prefix_options={}, @attributes={
  "updated_at"=>Sun Dec 09 23:05:32 UTC 2007,
  "body"=>"Rio on Rails event!!",
  "title"=>"Hello from Rio on Rails", "id"=>2,
  "created_at"=>Sun Dec 09 23:05:32 UTC 2007}>]
```

Podemos ainda criar um novo post, assim:

```
>> p = Post.create(:title => "Created Remotely", :body
=> #<Post:0x122df48 @prefix_options={}, @attributes={"up
```

Agora, voltamos ao nosso browser e recarregamos http://localhost:3000/posts



Entendeu? Temos 2 aplicações Rails e uma delas pode conversar com a outra através de chamadas HTTP baseadas em REST! Mais que isto: quando criamos o ActiveResource Post, fornecemos o username e password para a Autenticação Básica HTTP que fizemos antes. Esta é uma das vantagens disto: é fácil de fazer chamadas API depois.

Não trocamos UMA SIMPLES LINHA DE CÓDIGO de nossa aplicação e já somos capazes de fazer integração remota. Por tudo isto agradeça ao jeito RESTful de construir aplicações Rails: você ganha APIs completas para qualquer recurso RESTful que criar. Siga as convenções e isto torna-se fácil.

Conclusão

Então, Rails 2.0 está reunindo adições legais, muitas são otimizações e não fazem você reaprender tudo do início, o que é bom. Mas vamos dizer que você já tenha uma aplicação Rails 1.2 funcionando, faz sentido atualizá-la cegamente para Rails 2.0?

Nem sempre. Precisamos usar nosso bom senso aqui:

- há algo no 2.0 que você quer que sua aplicação tenha?
- sua aplicação é totalmente coberta com uma Suíte de Testes adequada?
- seus plugins e gems de terceiros já funcionam com Rails 2.0?

Sem responder a estas questões você não deveria tentar atualizar. Você pode dar congelar (freeze) os gems antigos na sua aplicação:

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

Você pode ter também ambos os gems, Rails 1.2 e 2.0, instalados. Para usar o comando Rails para criar um projeto 1.2 faça o seguinte:

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

Tenha certeza de que se tem tudo coberto. 1.2.6 dará várias mensagens de aviso nos logs. Verifique-as para ajustar sua aplicação para ser 2.0 compatível.

E se você é um novato, ainda aprendendo Rails, não encontrará muitos recursos para aprender ainda: todos os livros lançados até agora somente abordam até a versão 1.2. Mas isto não é um problema: aprenda 1.2 primeiro. Você pode ainda aguardar o lançamento de um novo livro do Rails 2.0 (eu vou lançar um em breve, em 2008), procurar tutoriais como o meu e você será capaz de aprender 2.0 em uma questão de poucos dias, talvez até em poucas horas.

Rails 2.0 não é uma revolução, mas é uma evolução muito bem vinda. Um polimento e refinamento do que já era bom. Isto é ótimo.

Eu espero que meu tutorial ajude todos a entender rapidamente algumas das novas características do Rails 2.0.

E se você quiser pegar o **código fonte completo** do blog que fizemos, pegue **aqui**. Divirta-se!

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

0 comentários