

%w(Akita On Rails) * 2.0

Rolling with Rails 2.0 – O Primeiro Tutorial Completo – Parte 1

escrito por AkitaOnRails em January 31st, 2008 @ 11:44 PM

Estou muito feliz vendo que [meu Rails 2.0 Screencast](#) foi muito bem recebido. Mais de 9.000 visitantes únicos assistiram (e centenas lêem o tutorial todos os dias). A idéia era mostrar Rails 2.0 muito rápido, apresentando o que é possível fazer em menos de 30 minutos.

Agora, irei quebrar aquele vídeo em suas partes principais e criar o **primeiro tutorial passo-a-passo sobre Rails 2.0**.

Como qualquer outro tutorial, eu não cubro 100% do Rails 2.0, apenas algumas características principais agrupadas em uma aplicação coesa. Eu recomendo o [Peepcode's Rails2 PDF](#) e o [Railscasts.com](#) de Ryan Bates para mais detalhes.

Algumas pessoas se ofereceram para traduzir o tutorial original, que eu escrevi em inglês. Um deles foi o [Lucas Húngaro](#). O problema foi que eu demorei demais a dar continuidade. Daí o [Rafael DX7](#) me enviou uma versão traduzida também. Fiz algumas alterações e finalmente temos o tutorial em português. Agradeço a colaboração.

Este tutorial possui 2 partes, para a **Parte 2**, clique [aqui](#). Cliquem [aqui](#) se quiserem imprimir esta página. O código completo está disponível [aqui](#)

Vamos começar!

Identificando o ambiente

Este tutorial é direcionado àqueles que já tem algum conhecimento do Rails 1.2. Por favor, busque os ótimos tutoriais de Rails 1.2 disponíveis na Internet.

A primeira coisa que você deve fazer é atualizar seus gems:

```
sudo gem install rails --include-dependencies
```

Provavelmente terá que atualizar o RubyGems também:

```
sudo gem update --system
```

Começando do começo. Vamos criar uma aplicação Rails:

```
rails blog -d mysql
```

Isto criará nossa estrutura de pastas padrão do Rails. A primeira coisa a reparar é o ambiente: agora temos esta estrutura principal:

- config/environment.rb
- config/initializers/inflections.rb
- config/initializers/mime_types.rb

Tudo dentro da pasta config/initializers é carregado junto com o environment.rb, e isso porque quando se está usando vários plugins diferentes e gems no seu projeto, o arquivo environment.rb tende a tornar-se desordenado e difícil de dar manutenção. Agora temos um jeito fácil para modularizar a nossa configuração.

Banco de Dados

A segunda coisa que nós temos que fazer é configurar nosso banco de dados. Isto é feito do mesmo modo que antes, no arquivo

config/database.yml:

```
development:
  adapter: mysql
  encoding: utf8
  database: blog_development
  username: root
  password: root
  socket: /opt/local/var/run/mysql5/mysqld.sock

test:
  adapter: mysql
  encoding: utf8
  database: blog_test
  username: root
  password: root
  socket: /opt/local/var/run/mysql5/mysqld.sock

production:
  adapter: mysql
  encoding: utf8
  database: blog_production
  username: root
  password: root
  socket: /opt/local/var/run/mysql5/mysqld.sock
```

Note que agora temos uma opção 'encoding' que é UTF8 por padrão. A aplicação Rails carrega com KCODE = true por padrão também, significando que ela inicia silenciosamente já com suporte Unicode, o que é ótimo. Mas a configuração 'encoding' tem uma nova função também: toda vez que o Rails conecta ao banco de dados ela será utilizada, como se estivesse definindo 'SET NAMES UTF8'.

Um truque que podemos usar pra tornar o nosso database.yml mais "DRY" (sem repetições) é isto:

```
defaults: &defaults
  adapter: mysql
  encoding: utf8
  username: root
  password: root
  socket: /opt/local/var/run/mysql15/mysqld.sock

development:
  database: blog_development
  <<: *defaults

test:
  database: blog_test
  <<: *defaults

production:
  database: blog_production
  <<: *defaults
```

Muito melhor. Nós temos novas tarefas Rake também. E algumas delas relacionam-se ao banco de dados:

db:charset	Retorna o charset para o ambiente atual de banco de dados
db:collation	Retorna o collation para o ambiente atual de banco de dados
db:create	Cria o banco de dados definido em config/database.yml para o atual RAILS_ENV
db:create:all	Cria todos os banco de dados locais definidos no config/database.yml
db:drop	Apaga (Drop) o banco de dados para o RAILS_ENV atual
db:drop:all	Apaga (Drop) todos os banco de dados locais definidos no config/database.yml
db:reset	Apaga (Drop) e recria o banco de dados do db/schema.rb para o ambiente atual
db:rollback	Faz rollback do schema para a versão anterior. Especifica o número de passos com STEP=n
db:version	Retorna o atual número de versão do schema

Nós temos um suporte de administração de banco de dados muito melhor. No Rails antigo entraríamos no administrador do banco de dados e criaríamos o banco manualmente. Agora podemos simplesmente fazer:

```
rake db:create:all
```

Se quisermos começar do zero, podemos fazer `db:drop:all`. E em meio ao desenvolvimento usar `db:rollback` para desfazer as últimas alterações feitas pelo migration.

Sensualidade

Com o banco de dados configurado e pronto, podemos criar nosso primeiro recurso (Resource). Lembre-se que agora o Rails 2.0 é RESTful por padrão (Eu estou **escrevi** um tutorial sobre RESTful também).

```
./script/generate scaffold Post title:string body:text
```

A única diferença aqui é que o `scaffold` comporta-se como o `scaffold_resource` que tínhamos antes, e o velho `scaffold` não-RESTful se foi. Nós também não temos o método de classe `scaffold` da classe ActionController que populava dinamicamente nosso controller vazio com actions padrão. Então todo `scaffold` que fazemos agora é RESTful.

Isto criará os usuais: Controller, Helper, Model, Migration, Unit Test, Functional Test.

A principal diferença está no arquivo Migration:

```
# db/migrate/001_create_posts.rb
class CreatePosts < ActiveRecord::Migration
  def self.up
    create_table :posts do |t|
      t.string :title
      t.text :body

      t.timestamps
    end
  end

  def self.down
    drop_table :posts
  end
end
```

Isto é chamado **Sexy Migrations**, primeiro inventado pelo "Err the Blog" como um plugin que encontrou seu caminho para o Core. O melhor jeito de entender a diferença é dar uma olhada no que este migration pareceria no Rails 1.2

```
class CreatePosts < ActiveRecord::Migration
  def self.up
    create_table :posts do |t|
      t.column :title, :string
      t.column :body, :text
      t.column :created_at, :datetime
      t.column :updated_at, :datetime
    end
  end

  def self.down
    drop_table :posts
  end
end
```

Isto nos livra da repetição do 't.column' e usando o formato 't.column_type' e as colunas automáticas datetime são concentradas em uma única instrução 't.timestamps' Isto não muda qualquer comportamento, mas torna o código mais 'Sexy'.

Agora, executamos o migration como antes:

```
rake db:migrate
```

Nos velhos tempos, se quiséssemos dar rollback para trocar o migration, teríamos que fazer:

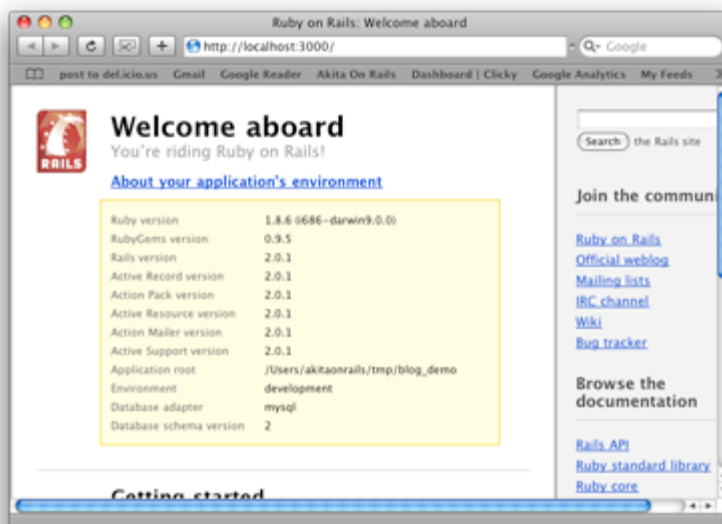
```
rake db:migrate VERSION=xxx
```

Onde 'xxx' é a versão que queremos voltar atrás. Agora, fazemos simplesmente:

```
rake db:rollback
```

Muito legal e mais elegante, com certeza. Tudo configurado, podemos iniciar o servidor como antes e dar uma olhada nas páginas geradas:

```
./script/server
```

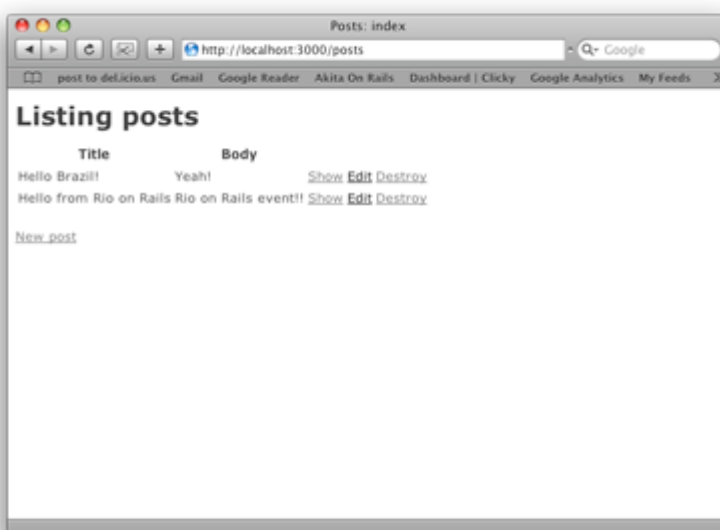


Isto carregará o Mongrel, Webrick ou Lighttpd na porta 3000. Temos uma página principal como antes, exibindo a página index.html. Uma

dica que não mostrei no screencast é:

```
# config/routes.rb
ActionController::Routing::Routes.draw do |map|
  map.root :controller => 'posts'
  map.resources :posts
end
```

Há uma nova instrução 'map.root' que tem o mesmo efeito que "map.connect ", :controller => 'posts'. Apenas um pequeno detalhe que não faz nada demais mas tenta tornar o arquivo de rotas mais simples. Uma vez configurado, não esqueça de apagar o arquivo public/index.html. Note que a raiz sempre será direcionada para o controller Posts.



Como você pode ver, tudo parece como antes. Todas as templates de scaffold são a mesma coisa. Podemos navegar, criar novas linhas e assim por diante.

Rotas Aninhadas

Então, vamos criar um recurso Comment para o Post. Isto irá completar as funcionalidades do Blog:

```
./script/generate scaffold Comment post:references bo
rake db:migrate
```

A mesma coisa aqui: scaffold no recurso Comment, configurar os nomes das colunas e seus tipos na linha de comando e o arquivo migration estará configurado. Repare outro pequeno detalhe: a palavra-chave 'references'. Como meu amigo **Arthur** me lembrou, isto faz o migrations ainda mais Sexy. Para comparar isto com a maneira

antiga de fazer a mesma coisa:

```
./script/generate scaffold Comment post_id_:integer bc
```

Chaves Estrangeiras são só detalhes de implementação que não importam. Dê uma olhada no arquivo migration:

```
def self.up
  create_table :comments do |t|
    t.references :post
    t.text :body

    t.timestamps
  end
end
```

Dê uma olhada [aqui](#) para detalhes desta nova palavra-chave 'references'. Então, executando db:migrate criamos a tabela no banco de dados. E depois, nós configuramos o model ActiveRecord e aí eles se relacionam assim:

```
# app/models/post.rb
class Post < ActiveRecord::Base
  has_many :comments
end

# app/models/comment.rb
class Comment < ActiveRecord::Base
  belongs_to :post
end
```

Ok, nada novo aqui, nós já sabemos trabalhar com associações do ActiveRecord. Mas além disso estamos trabalhando com recursos RESTful. No novo jeito Rails, gostaríamos de ter URLs como estas:

```
http://localhost:3000/posts/1/comments
http://localhost:3000/posts/1/comments/new
http://localhost:3000/posts/1/comments/3
```

Que significa: *'pegue os comentários deste post específico'* O gerador de scaffold consegue somente fazer URLs semelhantes a estas:

```
http://localhost:3000/posts/1
http://localhost:3000/comments/new
http://localhost:3000/comments/3
```

Porque no config/routes.rb temos:

```
# config/routes.rb
ActionController::Routing::Routes.draw do |map|
  map.resources :comments

  map.root :controller => 'posts'
  map.resources :posts
end
```

Como no modelo, podemos criar o que é chamado de **Nested Route – Rotas aninhadas**:

```
# config/routes.rb
ActionController::Routing::Routes.draw do |map|
  map.root :controller => 'posts'
  map.resources :posts, :has_many => :comments
end
```

Simples assim! Agora podemos fazer URLs aninhadas como mostrado acima. A primeira coisa a entender é que quando escreve-se esta URL:

`http://localhost:3000/posts/1/comments`

Rails entenderá como:

- Carregar the CommentsController
- Configurar o `params[:post_id] = 1`
- Neste caso, chamar o `'index'` action

Nós temos que preparar o CommentsController para este aninhamento. Então, o que mudaremos:

```
class CommentsController < ApplicationController
  before_filter :load_post
  ...
  def load_post
    @post = Post.find(params[:post_id])
  end
end
```

Isso deixará o `@post` já configurado para todas as ações no controller Comments. Agora temos que fazer estas alterações:

Antes	Depois
<code>Comment.find</code>	<code>@post.comments.find</code>
<code>Comment.new</code>	<code>@post.comments.build</code>
<code>redirect_to(@comment)</code>	<code>redirect_to([@post, @comment])</code>
<code>redirect_to(comments_url)</code>	<code>redirect_to(post_comments_url(@post))</code>

Isto deveria deixar o controller Comments preparado. Agora vamos

alterar as 4 views em app/views/comments. Se você abrir o new.html.erb ou o edit.html.erb, verá a nova característica abaixo:

```
# novos edit.html.erb e new.html.erb
form_for(@comment) do |f|
  ...
end
```

Este é o novo jeito de fazer esta antiga instrução do Rails 1.2:

```
# antigo new.rhtml
form_for(:comment, :url => comments_url) do |f|
  ...
end
```

```
# antigo edit.rhtml
form_for(:comment, :url => comment_url(@comment),
  :html => { :method => :put }) do |f|
  ...
end
```

Repare como a instrução form_for adapta ambas as situações 'new' e 'edit'. Isto porque o Rails pode deduzir o que fazer baseado no nome da classe do objeto @comment. Mas agora, para as Rotas Aninhadas, comentários são dependentes do Post, então isto é o que temos que fazer:

```
# novos edit.html.erb e new.html.erb
form_for([@post, @comment]) do |f|
  ...
end
```

Rails tentará ser esperto o suficiente para entender que este array representa uma Rota Aninhada. Checará routes.rb, calculará e esta é a rota nomeada **post_comment_url(@post, @comment)**.

Vamos explicar primeiro a rota nomeada. Quando configuramos um Resource Route no arquivo routes.rb, ganhamos estas rotas nomeadas:

rota	verbo HTTP	Action do Controller
comments	GET	index
comments	POST	create
comment(:id)	GET	show
comment(:id)	PUT	update
comment(:id)	DELETE	destroy
new_comment	GET	new

```
edit_comment(:id)GET      edit
"7 Ações para dominar tudo ..." :-)
```

"Lord of the Rings", para quem não entendeu a piada ;-)

Você pode acrescentar os sufixo 'path' ou 'url'. A diferença é:

```
comments_url http://localhost:3000/comments
comments_path/comments
```

Finalmente você pode acrescentar o prefixo 'formatted', dando a você:

```
formatted_comments_url(:atom) http://localhost:3000/comments.atom
formatted_comment_path(@comment, /comments/1.atom
:atom)
```

Agora, como Comments está aninhado com Post, nós somos obrigados a adicionar o prefixo 'post'. No Rails 1.2 este prefixo era opcional. Era possível dizer a diferença através do número de parâmetros passados para o helper de rota nomeada, mas isto poderia levar a muitas ambigüidades. Então agora é obrigatório ter o prefixo, como:

rota	verbo	URL
post_comments(@post)	GET	/posts/:post_id/comments
post_comments(@post)	POST	/posts/:post_id/comments
post_comment(@post, :id)	GET	/posts/:post_id/comments/:id
post_comment(@post, :id)	PUT	/posts/:post_id/comments/:id
post_comment(@post, :id)	DELETE	/posts/:post_id/comments/:id
new_post_comment(@post)	GET	/posts/:post_id/comments/new
edit_post_comment(@post, :id)	GET	/posts/:post_id/comments/edit

Logo, para resumir, temos que fazer as views de Comments se comportarem aninhados no Post. Assim temos que mudar as rotas nomeadas no código padrão gerado pelo scaffold para a forma aninhada:

```
<!-- app/views/comments/_comment.html.erb -->
<% form_for([@post, @comment]) do |f| %>
  <p>
    <b>Body</b><br />
    <%= f.text_area :body %>
  </p>

  <p>
    <%= f.submit button_name %>
  </p>
<% end %>
```

```
<!-- app/views/comments/edit.html.erb -->
<h1>Editing comment</h1>

<%= error_messages_for :comment %>

<%= render :partial => @comment,
  :locals => { :button_name => "Update"} %>

<%= link_to 'Show', [@post, @comment] %> |
<%= link_to 'Back', post_comments_path(@post) %>
```

```
<!-- app/views/comments/new.html.erb -->
<h1>New comment</h1>

<%= error_messages_for :comment %>

<%= render :partial => @comment,
  :locals => { :button_name => "Create"} %>

<%= link_to 'Back', post_comments_path(@post) %>
```

```
<!-- app/views/comments/show.html.erb -->
<p>
  <b>Body:</b>
  <%=h @comment.body %>
</p>

<%= link_to 'Edit', [:edit, @post, @comment] %> |
<%= link_to 'Back', post_comments_path(@post) %>
```

```
<!-- app/views/comments/index.html.erb -->
<h1>Listing comments</h1>

<table>
  <tr>
    <th>Post</th>
    <th>Body</th>
  </tr>

  <% for comment in @comments %>
    <tr>
      <td><%=h comment.post_id %></td>
      <td><%=h comment.body %></td>
      <td><%= link_to 'Show', [@post, comment] %></td>
      <td><%= link_to 'Edit', [:edit, @post, comment] %></
      <td><%= link_to 'Destroy', [@post, comment],
        :confirm => 'Are you sure?', :method => :delete %>
    </tr>
  <% end %>
</table>

<br />

<%= link_to 'New comment',
  new_post_comment_path(@post) %>
```

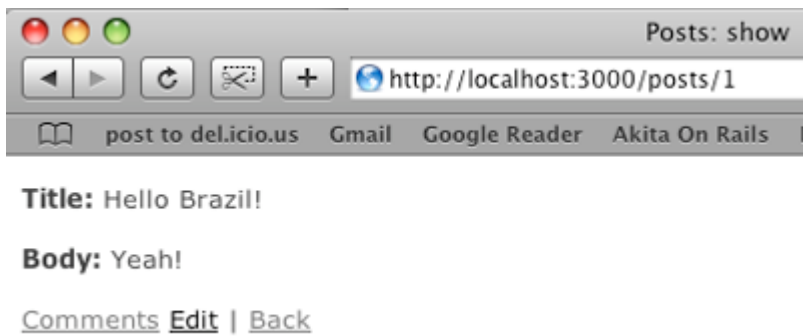
Algumas observações:

- Repare que eu criei uma partial para ser DRY nos formulários new e edit. Mas preste atenção que ao invés de :partial => 'comment', Eu fiz :partial => @comment. Então ele pode deduzir o nome do partial através do nome da classe. Se passamos uma coleção isto será equivalente à antiga instrução ':partial, :collection'.
- Eu posso usar post_comment_path(@post, @comment), ou simplesmente [@post, @comment].
- Preste bem atenção para não esquecer qualquer rota nomeada.

Finalmente, seria bom unir a lista de comentários ao post view. Logo, vamos fazer isso:

```
<!-- app/views/posts/show.html.erb -->
<%= link_to 'Comments', post_comments_path(@post) %>
<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
```

Assim, eu já adicionei um link lá. Vejamos como se parece:



Completando as views

Ok, parece bom, mas não é como um Blog deveria se comportar! A view "show" do Post já deveria ter a lista de comentários e um formulário para postar um novo comentário também! Então vamos fazer algumas pequenas adaptações. Nada de novo aqui, só o Rails tradicional. Vamos começar com a view:

```
<!-- app/views/posts/show.html.erb -->
<p>
  <b>Title:</b>
  <%=h @post.title %>
</p>

<p>
  <b>Body:</b>
  <%=h @post.body %>
</p>

<!-- #1 -->
<% unless @post.comments.empty? %>
  <h3>Comments</h3>
  <% @post.comments.each do |comment| %>
    <p><%= h comment.body %></p>
  <% end %>
<% end %>

<!-- #2 -->
<h3>New Comment</h3>
<%= render :partial => @comment = Comment.new,
  :locals => { :button_name => 'Create'}%>

<%= link_to 'Comments', post_comments_path(@post) %>
<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
```

Mais observações

1. Nada novo no iterador, apenas listando todos os comentários
2. Novamente, nós passamos a variável para a partial

Um ajuste final: sempre que nós criarmos um novo post, gostaríamos de retornar ao mesmo view "show" do Post, então mudamos o CommentsController para se comportar assim:

```
# app/controllers/comments_controller.rb
# old redirect:
redirect_to(@post, @comment)
# new redirect:
redirect_to(@post)
```

Rotas em Namespace

Ok, temos um simples mini-blog que de certo modo imita o clássico blog do screencast '15 minutes' do David em 2005. Agora vamos um passo a frente: Posts não deveriam estar disponíveis publicamente para qualquer um editar. Precisamos de uma área de administração em nosso website. Criaremos um novo controle para isso:

```
./script/generate controller Admin::Posts
```

Rails 2.0 agora suporta namespaces. Isto criará um subdiretório chamado `app/controllers/admin`.

O que queremos fazer com isso:

1. Criar uma nova rota
2. Copiar todas as actions do antigo controller Posts para o novo `Admin::Posts`
3. Copiar as views de posts para `app/views/admin*`
4. Deixar o antigo controller Posts somente com as actions `'index'` e `'show'`, isso significa deletar as views `new` e `edit` também
5. Adaptar as actions e views que há pouco copiamos para que eles entendam que estão no controller `admin`

Antes de mais nada, vamos editar `config/routes.rb` novamente:

```
map.namespace :admin do |admin|
  admin.resources :posts
end
```

Na prática isso significa que temos rotas nomeadas para Posts com o prefixo `'admin'`. Isto tornará claro as antigas rotas de posts para as novas rotas de posts do `admin`, semelhante a isso:

```
posts_path           /posts
post_path(@post)    /posts/:post_id
admin_posts_path     /admin/posts
admin_post_path(@post) /admin/posts/:post_id
```

Vamos copiar as actions do antigo controller Posts e adaptar as rotas para o novo namespace:

```
# app/controllers/admin/posts_controller.rb
...
def create
  # old:
  format.html { redirect_to(@post) }
  # new:
  format.html { redirect_to([:admin, @post]) }
end

def update
  # old:
  format.html { redirect_to(@post) }
  # new:
  format.html { redirect_to([:admin, @post]) }
end

def destroy
  # old:
  format.html { redirect_to(posts_url) }
  # new:
  format.html { redirect_to(admin_posts_url) }
end
...
```

Não esqueça de deletar todos os métodos do `app/controllers/posts_controller.rb`, deixando apenas os métodos `'index'` e `'show'`.

Vamos copiar as views (assumindo que seu shell já está na pasta principal do projeto):

```
cp app/views/posts/*.erb app/views/admin/posts
rm app/views/posts/new.html.erb
rm app/views/posts/edit.html.erb
```

Agora vamos editar as views de `app/views/admin/posts`:

```
<!-- app/views/admin/posts/edit.html.erb -->
<h1>Editing post</h1>

<%= error_messages_for :post %>

<% form_for([:admin, @post]) do |f| %>
  ...
<% end %>

<%= link_to 'Show', [:admin, @post] %> |
<%= link_to 'Back', admin_posts_path %>
```

```
<!-- app/views/admin/posts/new.html.erb -->
<h1>New post</h1>

<%= error_messages_for :post %>

<% form_for([:admin, @post]) do |f| %>
  ...
<% end %>

<%= link_to 'Back', admin_posts_path %>
```

```
<!-- app/views/admin/posts/show.html.erb -->
<p>
  <b>Title:</b>
  <%=h @post.title %>
</p>

<p>
  <b>Body:</b>
  <%=h @post.body %>
</p>

<%= link_to 'Edit', edit_admin_post_path(@post) %> |
<%= link_to 'Back', admin_posts_path %>
```

```
<!-- app/views/admin/posts/index.html.erb -->
...
<% for post in @posts %>
  <tr>
    <td><%=h post.title %></td>
    <td><%=h post.body %></td>
    <td><%= link_to 'Show', [:admin, post] %></td>
    <td><%= link_to 'Edit', edit_admin_post_path(post) %>
    <td><%= link_to 'Destroy', [:admin, post],
      :confirm => 'Are you sure?', :method => :delete %>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New post', new_admin_post_path %>
```

Quase pronto: se você testar <http://localhost:3000/admin/posts> deverá funcionar corretamente agora. Porém estará feio, e isso porque não temos um layout global da aplicação. Quando fizemos o primeiro scaffolds, Rails gerou um layout específico para Post e Comment somente. Vamos apagá-lo e criar um genérico:

```
cp app/views/layouts/posts.html.erb \
  app/views/layouts/application.html.erb
rm app/views/layouts/posts.html.erb
rm app/views/layouts/comments.html.erb
```

Logo, vamos mudar o título para:

```
<!-- app/views/layouts/application.html.erb -->
...
<title>My Great Blog</title>
...
```

Só permanecem as antigas páginas 'index' e 'show' do controller Posts. Eles também têm links para os métodos que deletamos, então vamos retirá-los:

```
<!-- app/views/posts/index.html.erb -->
<h1>My Great Blog</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Body</th>
  </tr>

  <% for post in @posts %>
    <tr>
      <td><%=h post.title %></td>
      <td><%=h post.body %></td>
      <td><%= link_to 'Show', post %></td>
    </tr>
  <% end %>
</table>
```

```
<!-- app/views/posts/show.html.erb -->
<p>
  <b>Title:</b>
  <%=h @post.title %>
</p>

<p>
  <b>Body:</b>
  <%=h @post.body %>
</p>

<% unless @post.comments.empty? %>
  <h3>Comments</h3>
  <% @post.comments.each do |comment| %>
    <p><%= h comment.body %></p>
  <% end %>
<% end %>

<h3>New Comment</h3>

<%= render :partial => @comment = Comment.new,
  :locals => { :button_name => 'Create'}%>

<%= link_to 'Back', posts_path %>
```

Nós podemos testar tudo do browser, indo em <http://localhost:3000/admin/posts> e vendo que tudo está funcionando corretamente agora. Mas, nós também temos uma coisa errada: uma área administrativa não deveria estar disponível publicamente. No momento você pode entrar e editar qualquer coisa. Precisamos de autenticação.

Autenticação HTTP Básica

Há várias maneiras de implementar autenticação e autorização. Um plugin que é muito usado para isto é **restful_authentication**.

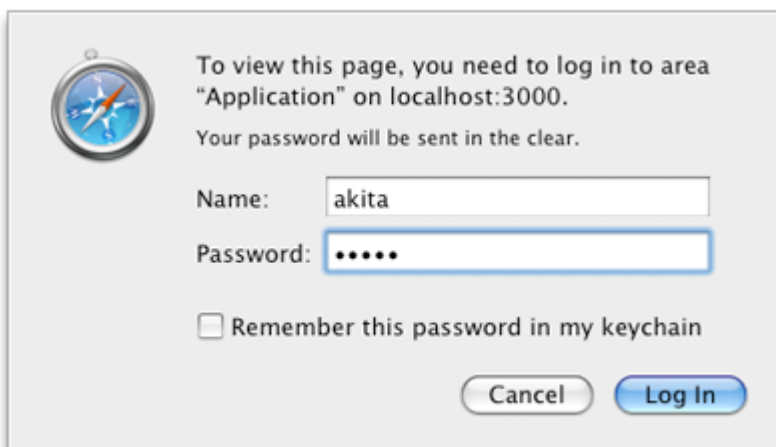
Porém nós não queremos nada demais aqui. E para isto o Rails 2.0 nos dá um ótima maneira de fazer autenticação. A idéia é: vamos usar o que HTTP já nos dá: **Autenticação HTTP Básica**. A desvantagem é: nós definitivamente precisamos usar SSL quando utilizarmos em produção. Mas, claro, você deveria fazer isso de qualquer maneira. Formulário de autenticação HTML não é protegido sem SSL também.

Assim, vamos editar nosso controller `Admin::Posts` para adicionar autenticação:

```
# app/controllers/admin/posts.rb
class Admin::PostsController < ApplicationController
  before_filter :authenticate
  ...
  def authenticate
    authenticate_or_request_with_http_basic do |name, pa
      #User.authenticate(name, pass)
      name == 'akita' && pass == 'akita'
    end
  end
end
```

Você já sabe o que o 'before_filter' faz: ele executa o método configurado antes de qualquer ação no controller. Se você configurar na classe ApplicationController então ele vai executar antes de qualquer ação de qualquer outro controller. Porém nós queremos somente proteger Admin::Posts aqui

Logo, implementamos este método e o segredo é o método 'authenticate_or_request_with_http_basic' que vamos bloquear. Isto nos dará o username e senha que o usuário digitou no browser. Nós normalmente teríamos um modelo de Usuário de algum tipo para autenticar estes dados, mas para nosso exemplo muito, muito simples eu estou fazendo a verificação diretamente, mas você entendeu a idéia.



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

0 comentários