



Tutorial de Rails

Desenvolvido por
Eustáquio “TaQ” Rangel

Dedicado para a minha filhinha Ana Isabella e para a minha esposa Ana Carolina que falou que me mata se eu não dedicar alguma coisa para ela depois de dedicar o livro de Ruby só para a Ana Isabella. ;-)

"Rails", "Ruby on Rails", e o logotipo do Rails são marcas registradas de David Heinemeier Hansson. Todos os direitos reservados. Obrigado ao David por autorizar o uso do logotipo do Rails. :-)

Revisão 2 – 20/05/2006

Este trabalho está licenciado sob uma Licença Creative Commons Atribuição-Uso Não-Comercial-Compatilhamento pela mesma licença.

Para ver uma cópia desta licença, visite
<http://creativecommons.org/licenses/by-nc-sa/2.5/br/>
ou envie uma carta para
Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.



Sumário

Sobre esse tutorial.....	3
O que é o Rails?.....	4
O que é Ruby?.....	4
Instalando o Rails.....	6
Configurando o banco de dados.....	6
Criando o projeto.....	7
Scaffold.....	8
Modelos.....	9
Visualizadores.....	11
Controladores.....	14
Validação.....	17
Relacionamentos.....	20
Personalizando o código gerado.....	24
Arquivos binários.....	28
Tarefas administrativas.....	36
Criando a livreria “externa”.....	43
Melhorando nossas listagens.....	48
Pesquisando.....	55
Renderizações parciais.....	56
Renderizando parcialmente com Ajax.....	61
Finalizando.....	65

Sobre esse tutorial

Além de tudo que escrevi ali em cima, gostaria de deixar claro algumas coisas antes de começarmos a meter a mão na massa:

1. O David realmente me deixou usar o logotipo. Guardei o email de prova ehehe.
2. Minha esposa disse que esse tipo de dedicatória não vale, mas tudo bem, ficou engraçado. ;-)
3. Fiz esse tutorial de maneira bem descompromissada de acordo com várias anotações que eu havia feito. Não esperem um esmero de código, a intenção foi liberar algo rápido e prático para dar uma noção do framework. Tem muita coisa que pode ser melhorada depois de entender os conceitos básicos e sujar as mãos no código.
4. Por causa da razão acima podem ser encontrados vários códigos CSS meio estranhos, e pior, algumas partes que nem usam CSS. Eu continuo usando tabelas em alguns pontos, mas isso pode ser melhorado também.
5. Também não me preocupei muito com os layouts. Tem umas coisas bem feinhas por aí.
6. Deve ter um ou outro erro (espero que poucos!) perdidos por aí. Se alguém encontrar por favor me avise (eustaquiorangel at yahoo.com) por favor!
7. Se alguém quiser armazenar o tutorial para download no seu servidor, por favor me informe o link para que eu possa mencionar no meu site.
8. Podem usar o tutorial para ler, imprimir, mandar para o amigo, etc, etc, etc, só não podem usar para uso comercial, seus espertinhos. ;-)
9. Sugestões serão bem-vindas e podem ser enviadas para o meu email (eustaquiorangel at yahoo.com).
10. Espero que se divirtam. :-)

O que é o Rails?

Rails é um framework¹ feito em **Ruby** que funciona no conceito MVC – Model, View, Controller - onde é separado o modelo de dados, a interface do usuário e o controle lógico do programa, permitindo que alterações em qualquer uma dessas partes tenham pouco impacto nas outras.

O que é Ruby?

Para explicar o que é Ruby, eu faço uma cópia descarada do mesmo texto que está no meu livro, uma tradução livre do que Yukihiro “Matz” Matsumoto, seu criador, diz a respeito dela em

<http://www.ruby-lang.org/en/20020101.html>:

“Ruby é uma linguagem de script interpretada para programação orientada a objetos de um modo fácil e rápido. Ela tem vários recursos para processar arquivos de texto e para fazer tarefas de gerenciamento de sistema (assim como o Perl). Ela é simples, direto ao ponto, extensível e portátil. Oh, preciso mencionar, é totalmente livre, o que significa não só livre de precisar pagar para usá-la, mas também a liberdade de usar, copiar, modificar e distribuí-la.”

Recursos da linguagem

- Ruby tem uma sintaxe simples, parcialmente inspirada por Eiffel e Ada.
- Ruby tem recursos de tratamento de exceções, assim como Java e Python, para deixar mais fácil o tratamento de erros.
- Os operadores do Ruby são açúcar sintático para os métodos. Você pode redefini-los facilmente.
- Ruby é uma linguagem completa e pura orientada à objetos. Isso significa que todo dado em Ruby é um objeto, não do jeito de Python e Perl, mas mais do jeito do SmallTalk: sem exceções. Por exemplo, em Ruby, o número 1 é uma instância da classe *Fixnum*.
- A orientação à objetos do Ruby é desenhada cuidadosamente para ser completa e aberta à melhorias. Por exemplo, Ruby tem a habilidade de adicionar métodos em uma classe, ou até mesmo em uma instância durante o runtime! Então, se necessário, a instância de uma classe pode se comportar diferente de outras instâncias da mesma classe.
- Ruby tem herança única, de propósito. Mas entende o conceito de módulos (chamados de Categories no Objective-C). Módulos são coleções de métodos. Toda classe pode importar um módulo e pegar seus métodos. Alguns de nós acham que isso é um jeito mais limpo do que herança múltipla, que é complexa e não é usada tanto comparado com herança única (não conte C++ aqui, pois lá não se tem muita escolha devido a checagem forte de tipo!).
- Ruby tem closures² verdadeiras. Não apenas funções sem nome, mas com bindings de variáveis verdadeiras.
- Ruby tem blocos em sua sintaxe (código delimitado por `{...}` ou `do...end`). Esses blocos podem ser passados para os métodos, ou convertidos em closures.

1 Um *framework* pode ser definido como uma software estrutura de auxílio ao desenvolvimento de outros softwares, visando prover agilidade e eficiência para que o programador se livre da implementação de código repetitivo e ... chato.

2 *Closures* podem ser definidas como funções criadas dentro de outras funções, e que referenciam o ambiente (variáveis) da função externa mesmo depois dela ter saído de escopo, retendo a sua referência.

- Ruby tem um *garbage collector* que realmente é do tipo marca-e-limpa. Ele atua em todos os objetos do Ruby. Você não precisa se preocupar em manter contagem de referências em libraries externas. É melhor para a sua saúde.
- Escrever extensões em C para Ruby é mais fácil que em Perl ou Python, em grande parte por causa do garbage collector, e em parte pela boa API de extensões. A interface SWIG também está disponível.
- Inteiros em Ruby podem (e devem) ser usados sem contar sua representação interna. Existem inteiros pequenos (instâncias da classe *Fixnum*) e grandes (*Bignum*), mas você não precisa se preocupar em qual está sendo utilizado atualmente. Se um valor é pequeno o bastante, um inteiro é um *Fixnum*, do contrário é um *Bignum*. A conversão ocorre automaticamente.
- Ruby não precisa de declaração de variáveis. Apenas usa a convenção de nomenclatura para delimitar o escopo das variáveis. Por exemplo: *var* = *variável local*, *@var* = *variável de instância*, *\$var* = *variável global*. E não precisa do uso cansativo do *self* em cada membro da instância.
- Ruby pode carregar bibliotecas de extensão dinamicamente, se o sistema operacional permitir.
- Ruby tem um sistema de threading independente do sistema operacional. Então, para cada plataforma que você roda o Ruby, você tem multithreading de qualquer jeito, até no MS-DOS! ;-)
- Ruby é altamente portátil: ela é desenvolvida em sua maioria no Linux, mas funciona em muitos tipos de UNIX, DOS, Windows 95/98/Me/NT/2000/XP, MacOS, BeOS, OS/2, etc.

Você pode encontrar Ruby no seu site oficial na internet:

<http://www.ruby-lang.org>

Lá você encontra o código-fonte e versões instaláveis para Windows[™]. Compilar o código é rápido e fácil, no velho esquema

```
tar xvfz ruby-<versão>.tgz
./configure
make
make install
```

Procure na sua distribuição (se você usa GNU/Linux) algum pacote do Ruby, e se você usa Windows ou Mac, nos links correspondentes no site.

Se você quiser conhecer mais de Ruby, pode baixar o meu tutorial

<http://beam.to/taq/tutorialruby.php>

ou comprar o meu livro (compra,compra,compra! :-):



Você pode encontrá-lo clicando ali na foto ou no site da Brasport (<http://www.brasport.com.br>) e nas melhores livrarias do ramo (ó! que chique!).

Instalando o Rails

Atenção!

Vou basear o tutorial de Rails **inteiro** em ambiente GNU/Linux. Por favor, eu não utilizo Windows e nem tenho computador com ele instalado para testar a instalação nesse ambiente. Mesmo se tivesse alguma disponível, eu prefiro ficar longe. :-)

Para instalar o Rails, antes vamos precisar instalar a linguagem Ruby, que pode ter seu download feito em

<http://www.ruby-lang.org>

Lá encontramos os fontes da linguagem. Se você utiliza alguma distribuição que utiliza pacotes, por favor verifique se o pacote do Ruby se encontra disponível.

Depois precisamos do **RubyGems**, que é um gerenciador de pacotes para o Ruby. Pode ter seu download feito em

<http://docs.rubygems.org/>

Após instalados o Ruby e o RubyGems, podemos instalar o pacote do Rails facilmente com o comando

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

Digite *rails* no seu console e verifique o que acontece. Se não acontecer nada, houve algum problema. Verifique a sua instalação.

Falando em console, acostume-se a digitar coisas lá, pois vamos o utilizar muito durante o tutorial.

Configurando o banco de dados

Vamos usar nos nossos exemplos o banco de dados [MySQL](#), que é de longe o banco de dados gratuito mais comum tanto em GNU/Linux como Windows.

Criando as tabelas básicas dos produtos e usuários (sinta-se livre para criar essas tabelas com a ferramenta que quiser):

```
[taq@~]mysql -u taq -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9 to server version: 4.1.14

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database livraria_development;
Query OK, 1 row affected (0,38 sec)

mysql> use livraria_development;
Database changed

mysql> create table usuarios (
  -> id int not null auto_increment,
  -> nome varchar(75) not null,
  -> email varchar(75) not null,
  -> senha varchar(10) not null,
  -> admin int not null,
  -> img blob,
  -> primary key(id));
Query OK, 0 rows affected (0,08 sec)
```

```
mysql> create table tipos (
  -> id int not null auto_increment,
  -> descricao varchar(50) not null,
  -> primary key(id)
  -> );
Query OK, 0 rows affected (0,01 sec)

mysql> create table categorias (
  -> id int not null auto_increment,
  -> descricao varchar(50) not null,
  -> primary key(id)
  -> );
Query OK, 0 rows affected (0,01 sec)

mysql> create table produtos (
  -> id int not null auto_increment,
  -> descricao varchar(100) not null,
  -> tipo_id int not null,
  -> categoria_id int not null,
  -> primary key(id),
  -> constraint fk_tipo_produto foreign key(tipo_id) references
tipos(id),
  -> constraint fk_cate_produto foreign key(categoria_id) references
categorias(id)
  -> );
Query OK, 0 rows affected (0,00 sec)
```

Convém notar na última tabela criada, *produtos*, que eu criei algumas *foreign keys* referenciando os campos de outras duas tabelas, *tipos* e *categorias*. Uma convenção do Rails é usar a palavra *id* em certos campos chaves, como o identificador único de uma tabela (no caso, todas as tabelas usam esse campo) e nos nomes dos campos que servem de referências para outras tabelas (*tipo_id* e *categoria_id*).

Criando o projeto

Tudo pronto no banco de dados, vamos criar o projeto. Vá para o diretório onde você quer que o servidor web tenha acesso no seu projeto (no meu caso, */var/www/htdocs*) e execute o comando:

```
[taq@/var/www/htdocs]rails livraria
create
create  app/controllers
create  app/helpers
create  app/models
create  app/views/layouts
create  config/environments
create  components
create  db
create  doc
create  lib
create  lib/tasks
create  log
...
```

Agora vamos configurar o Rails para acessar o banco de dados que criamos (abra o arquivo com o seu editor favorito, eu uso o *vim*):

```
[taq@/var/www/htdocs/livraria]vim config/database.yml
development:
```

```
adapter: mysql
database: livraria_development
username: taq
password: *****
```

Agora vamos inicializar o nosso servidor web. Para esse tutorial, vamos utilizar o WEBrick, que já vem instalado:

```
[taq@/var/www/htdocs/livraria]ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-04-19 18:14:41] INFO  WEBrick 1.3.1
[2006-04-19 18:14:41] INFO  ruby 1.8.4 (2005-12-24) [i686-linux]
[2006-04-19 18:14:41] INFO  WEBrick::HTTPServer#start: pid=4174 port=3000
```

Se vocês repararem no arquivo de configuração do banco de dados, vão ver que ele tem definidos **development**, **test** e **production**, com configurações individuais para cada um. Isso nos permite usar diferentes bancos de dados para cada característica de nosso projeto, além de ter diferenças de performance. Para o passo de criar o site, vamos usar o default **development**. Se precisarmos utilizar **test** ou **production**, basta rodar

```
ruby script/server -e development | test | production
```

Escolha uma das três opções ou deixe em branco para utilizar o **development**, que é o default.

Também podemos utilizar o [Apache](#) ou o [Lighttpd](#) para rodar o Rails.

Scaffold

Criando o primeiro *scaffold*³, vamos começar com gerenciamento de usuários, afinal, todo sistema precisa disso:

```
[taq@/var/www/htdocs/livraria]ruby script/generate scaffold Usuario ←
Admin::Usuario
  exists  app/controllers/admin
  exists  app/helpers/admin
  exists  app/views/admin/usuario
  exists  test/functional/admin
  ...
```

Tentando explicar um pouco esse comando:

- **ruby** – Sem a linguagem Ruby instalada não conseguiremos fazer nada. Esse comando chama o interpretador da linguagem passando os parâmetros, que são ...
- **script/generate** – O programa *generate* que se encontra dentro do diretório *script*.
- **scaffold** – O parâmetro para o programa indicando que queremos que ele gere um *scaffold*.

³ Um *scaffold* é um meio de criar código para um determinado modelo (que indicamos) através de um determinado controlador (que indicamos também). É um meio de começarmos rapidamente a ver os resultados no nosso navegador web e um método muito rápido de implementar o CRUD (Create, Retrieve, Update, Delete) na sua aplicação. Lembrando que o scaffold cria código que, fatalmente, vai ser alterado depois, a não ser que você deseje manter o nível *muito* básico de interface e controle padrão de campos que ele proporciona.

- **Usuario** – O **modelo** (*model*), ou seja, o objeto que vai cuidar dessa tabela específica, *usuarios* (notem o plural), no banco de dados.⁴

- **Admin::Usuario** – O **controlador**, que administra as ações recebidas.⁵

Aqui já aparece o controlador, a terceira parte do MVC. Nosso controlador vai ser chamado de **Admin::Usuario**. Isso vai ser traduzido pelo Rails como uma estrutura de diretórios `<aplicação>/controllers/admin/usuario`. Eu preferi manter toda a estrutura de administração do site abaixo do diretório *admin* para dar uma separação mais lógica para a coisa.

Testando:

```
[taq@/var/www/htdocs/livraria]ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
```



Ilustração 1: Primeiro encontro com o scaffold

Modelos

Antes de darmos uma olhada nos *visualizadores*, vamos brincar um pouco com o *modelo*. Como mencionei ali em cima, foi criado o modelo para a tabela de *usuários*, sendo que o modelo chama *Usuario*. Reparem como que a tabela está no plural, e o modelo, no singular, com uma letra maiúscula no início. Isso segue o modelo de mapeamento objeto-relacional (ORM⁶) suportado pelo Rails. Vamos dar uma olhada no conteúdo do arquivo do modelo, que está em `<aplicacao>/app/models/usuario.rb`:

```
class Usuario < ActiveRecord::Base
end
```

A única coisa, por enquanto, em nosso modelo é a declaração da classe *Usuario* a partir da classe *ActiveRecord::Base*. Mas isso vai mudar no decorrer do tutorial.

No nosso caso, a classe *Usuario* se relaciona com a tabela *usuarios*. Podemos ter nomes como *EncomendaCliente*, separando as palavras com maiúsculas, que vão ser mapeados para tabelas com as palavras separadas por um sublinhado, ficando *encomendas_clientes*, por exemplo. Inclusive, o Rails tenta converter de singular para plural utilizando o Active Support, que é um de seus componentes. “Ei, mas como ele sabe converter para plural certas palavras específicas da

⁴ Os modelos se encontram no diretório `<aplicação>/models/modelo.rb`.

⁵ Os controladores se encontram em `<aplicação>/controllers/<controlador>_controler.rb`.

⁶ ORM, Object Relational Mapping é uma técnica que mapeia bancos de dados para conceitos de orientação á objetos. Podemos imaginar que cada tabela no banco de dados é uma classe, cada linha de dados é um objeto, e cada coluna é um atributo do objeto.

nossa língua, o Português?” vocês podem me perguntar. Bem, na verdade ele não sabe, e podemos forçar o nome da nossa tabela no modelo criado, alterando o modelo (que só existe no nosso exemplo hipotético) assim:

```
class EncomendaCliente < ActiveRecord::Base
  set_table_name "encomendas_clientes"
end
```

Agora que temos um modelo de *Usuario* definido (lembrem-se de não usar acentos nos seus modelos ou tabelas!), podemos brincar um pouco com as características do Active Record, outro componente do Rails, antes de darmos mais uma olhada no navegador.

O Rails vem com um console que nos permite interagir com o modelo. Primeiro vamos conectar no nosso banco de dados e verificar se existe algum usuário com o código 999:

```
[taq@~]mysql -u taq -p livraria_development
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 5.0.19

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> select * from usuarios where id=999;
Empty set (0,00 sec)
```

Ok, já sabemos que não tem. Agora vamos abrir um console, criar um usuário novo já que definimos o modelo, e salvar o dito cujo:

```
[taq@/var/www/htdocs/livraria]ruby script/console
Loading development environment.
u >> u = Usuario.new
=> #<Usuario:0xb72ef1fc @new_record=true, @attributes={"img"=>nil,
"nome"=>"", "admin"=>0, "senha"=>"", "email"=>""}>
>> u.id = 999
=> 999
>> u.nome = "Teste"
=> "Teste"
>> u.email = "teste@teste.com"
=> "teste@teste.com"
>> u.senha = "teste"
=> "teste"
>> u.admin = 0
=> 0
>> u.save
=> true
```

Verificando novamente no banco de dados:

```
mysql> select * from usuarios where id=999;
+-----+-----+-----+-----+-----+-----+
| id  | nome  | email          | senha | admin | img  |
+-----+-----+-----+-----+-----+-----+
| 999 | Teste | teste@teste.com | teste | 0     |     |
+-----+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

Legal hein? Tivemos um exemplo claro da ORM funcionando. Há um atalho para salvar um registro, que é utilizar o método **create** que vai pegar como parâmetros uma *hash* com o conteúdo do registro, tendo seus campos como chaves e os valores, como valores, daí:

```
>> u = Usuario.create(:nome=>"Outro teste!", :email=>"outro.teste@teste.com",
:senha=>"outroteste", :admin=>0)
=> #<Usuario:0xb75a52fc @new_record=false,
@errors=#<ActiveRecord::Errors:0xb75a4140 @errors={} ,
@base=#<Usuario:0xb75a52fc ...>, @attributes={"img"=>nil, "nome"=>"Outro
teste!", "admin"=>0, "id"=>1002, "senha"=>"outroteste",
"email"=>"outro.teste@teste.com"}>

>> u.id
=> 1002

>> Usuario.find(1002)
=> #<Usuario:0xb7593818 @attributes={"img"=>nil, "nome"=>"Outro teste!",
"admin"=>"0", "id"=>"1002", "senha"=>"outroteste",
"email"=>"outro.teste@teste.com"}>
```

O que fiz foi criar uma *hash* com os dados e passei para o método **create**, que me retornou um objecto do tipo *Usuario*, com o *id* novo criado, no caso, 1002, e logo em seguida efetuei uma pesquisa, que me retornou os dados inseridos.

Inclusive podemos pesquisar novamente e vamos ter como resultado o registro que acabamos de salvar:

```
>> Usuario.find(999)
=> #<Usuario:0xb762ddc8 @attributes={"img"=>nil, "nome"=>"Teste",
"admin"=>"0", "id"=>"999", "senha"=>"teste", "email"=>"teste@teste.com"}>
```

Uma coisa bem interessante que o Active Record permite é usar métodos de procura dinâmicos, que são métodos que são criados de acordo com os atributos do objeto em questão. Por exemplo, temos no nosso modelo de *Usuario* os atributos *id*, *nome*, *senha*, *email*, *admin* e *img*. Para procurar por nome, podemos usar:

```
>> u = Usuario.find_by_nome("Teste")
=> #<Usuario:0xb7605e04 @attributes={"img"=>nil, "nome"=>"Teste",
"admin"=>"0", "id"=>"999", "senha"=>"teste", "email"=>"teste@teste.com"}>
```

Ou procurando por email:

```
>> u = Usuario.find_by_email("teste@teste.com")
=> #<Usuario:0xb75fd8a8 @attributes={"img"=>nil, "nome"=>"Teste",
"admin"=>"0", "id"=>"999", "senha"=>"teste", "email"=>"teste@teste.com"}>
```

Ou seja, os campos da tabela foram mapeados como atributos do objeto, que podem ser consultados com métodos dinamicamente criados com o poder do Ruby.

Só para mostrar a natureza dinâmica desse tipo de operação, podemos perguntar para o nosso objeto do modelo se existe o método **find_by_nome** nele:

```
>> Usuario.respond_to?(:find_by_nome)
=> false
>> Usuario.respond_to?(:find_by_sql)
=> true
```

Não existe. Logo depois perguntei de **find_by_sql**, e ele está lá. Ou seja, o método dinâmico é criado no momento de sua necessidade.

Visualizadores

Para personalizar as telas, vou alterar o arquivo de layout padrão do controlador. Agora entra a

segunda parte do MVC, que são os **visualizadores** (views).

No caso da URL acima, o Rails vai procurar o arquivo de layout padrão em

```
<aplicação>/app/views/layouts/usuario.rhtml
```

Que vai conter:

```
<html>
<head>
  <title>Admin::Usuario: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

<p style="color: green"><%= flash[:notice] %></p>

<%= @content_for_layout %>

</body>
</html>
```

A primeira coisa que vou fazer é **apagar** (sim, exterminar sem misericórdia!) o arquivo acima (*usuario.rhtml*) e criar um arquivo de layout padrão para o controlador *admin* **todo**, e depois vou inserir indicações desse arquivo para os controladores abaixo do *admin*. No exemplo, vou criar o arquivo *admin.rhtml* :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" ↵
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <title>Tela de administração <%= controller.action_name %></title>
    <%= stylesheet_link_tag 'scaffold' %>
    <%= stylesheet_link_tag 'livraria' %>
  </head>
  <body>
    <div id="topo">
      <strong>Livraria Acme</strong>
    </div>
    <p style="color:green;"><%= flash[:notice] %></p>
    <%= @content_for_layout %>
  </body>
</html>
```

E indicar que o layout⁷ que o controlador *usuario* deve usar é o “*admin*”. Para isso, vou alterar o controlador, que se encontra em

```
<aplicação>/app/controllers/admin/usuario
```

```
class Admin::UsuarioController < ApplicationController
  layout "admin"
  def index
    list
    render :action => 'list'
  end
  ...
end
```

Algumas coisas destacadas em vermelho nos arquivos acima:

⁷ Os layouts se encontram em <aplicação>/app/views/layouts

- O charset tem que ser o **utf-8**. O servidor Webrick funciona nele e se você inserir alguma acentuação e não especificar que seu arquivo está nesse encoding, e **salvar** ele nesse encoding, você vai ver um monte de sinaisinhos estranhos na tela.
- A instrução **stylesheet_link_tag 'livraria'** está criando um link com o arquivo CSS *livraria.css* que vamos ver logo em seguida.
- A instrução **@content_for_layout** especifica que nesse ponto é que vai ser inserido o conteúdo do arquivo da ação desejada (nesse caso, **list**, que vai estar em *aplicação/app/views/admin/tipo/list.rhtml*).
- Indiquei o layout com o método, oras, vejam, *layout*.

Vamos dar uma olhada no arquivo *livraria.css*, que se encontra em

```
<aplicação>/public/stylesheets/

body {
  margin:0;
}

#topo {
  background:#c00000;
  color:white;
  width:100%;
  height:100px;
  margin:0;
}

#topo strong {
  font:bold 24px verdana,arial,helvetica,sans-serif;
  margin:25px;
  float:left;
}

#conteudo {
  margin:25px;
}

.novo {
  padding-left:20px;
  background:url(../images/mais.png) 3px center no-repeat;
  font-weight:bold;
}
```

E agora em na *view list.rhtml*⁸, vai o “miolo” do arquivo, já devidamente modificado com o texto traduzido:

```
<div id="conteudo">
  <h1>Listando os usuários</h1>
  <table>
    <tr>
      <% for column in Usuario.content_columns %>
        <th><%= column.human_name %></th>
      <% end %>
    </tr>
    <% for usuario in @usuarios %>
      <tr>
        <% for column in Usuario.content_columns %>
          <td><%=h usuario.send(column.name) %></td>
        <% end %>
      </tr>
    </tr>
  </table>
</div>
```

8 As views se encontram em *<aplicação>/app/views/<controlador>/<ação>*

```

        <td><%= link_to 'Mostra', :action => 'show', :id => usuario %></td>
        <td><%= link_to 'Edita' , :action => 'edit', :id => usuario %></td>
        <td><%= link_to 'Apaga' , { :action => 'destroy', :id => usuario },
:confirm => 'Tem certeza?' %></td>
    </tr>
    <% end %>
</table>

<p>
    <%= link_to 'Página anterior', { :page => ←
@usuario_pages.current.previous } if @usuario_pages.current.previous %>
    <%= link_to 'Próxima página', { :page => @usuario_pages.current.next } ←
if @usuario_pages.current.next %>
</p>
<p>
    <%= link_to 'Novo usuário', { :action => 'new'}, :class => 'novo' %>
</p>
</div>

```

Um ponto a ser observado aqui: o uso do método **h**, que é um atalho para **html_escape**, um método para converter strings com caracteres especiais (como &) em suas entidades HTML (nesse caso, &).

Apesar da *demonização* das tabelas hoje em dia, para esse caso elas servem muito bem, pois é pouca coisa e não vamos nos ater em mexer muito no CSS por hora. O resultado disso tudo é:



Ilustração 2: Código do scaffold personalizado com CSS

Controladores

Uma coisa interessante pode-se notar no título da página: a ação está especificada como *index*. Lembrem-se do conteúdo do controlador *tipo*:

```

def index
  list
  render :action => 'list'
end

```

No controlador estão especificadas todas as ações que acharmos necessárias. Sempre que não for especificada uma ação, o Rails aciona a *index*, e nesse caso, *index* chama *list*.

Agora vamos clicar em “Novo usuário” e personalizar o arquivo *new.rhtml*, que se encontra em `<aplicação>/app/views/admin/usuario/`:

```

<div id="conteudo">
  <h1>Novo usuário</h1>

  <%= start_form_tag :action => 'create' %>
  <%= render :partial => 'form' %>
  <%= submit_tag "Cria novo usuário", :class => "submit" %>
  <%= end_form_tag %>
  <p>
    <%= link_to 'Voltar', {:action => 'list'}, :class => 'voltar' %>
  </p>
</div>

```

Uma observação importante ali na chamada de **link_to**: o primeiro parâmetro é o texto do link, o segundo é uma *hash* onde fica a ação de destino do link e mais algumas opções de controle (nesse caso, tendo o controlador omitido pois é o que estamos), e o terceiro é uma *hash* com as propriedades HTML do link, onde nesse caso especifiquei que a classe CSS do link. Como são duas *hashes*, tive que separar a primeira da segunda com as chaves ({}), do contrário iria ficar as duas instruções na primeira *hash*. Por exemplo:

Se a definição do método for: *link_to(titulo,hash1,hash2)* então

```

<%= link_to 'Voltar', :action => 'list', :class => 'voltar' %>

```

Seria traduzido como

```

link_to('Voltar',{:action=>'list',:class=>'voltar'})

```

Ao passo que

```

link_to('Voltar',{:action=>'list'},{:class=>'voltar'})

```

é o correto, pois informamos as duas *hashes* ali. Então, preste atenção nesses detalhes.

Também vamos alterar um pouco o arquivo *livraria.css*:

```

.novo, .voltar {
  padding-left:20px;
  background:transparent url(../images/mais.png) 3px center no-repeat;
  font-weight:bold;
}

a:hover {
  color:red;
  background-color:white;
}

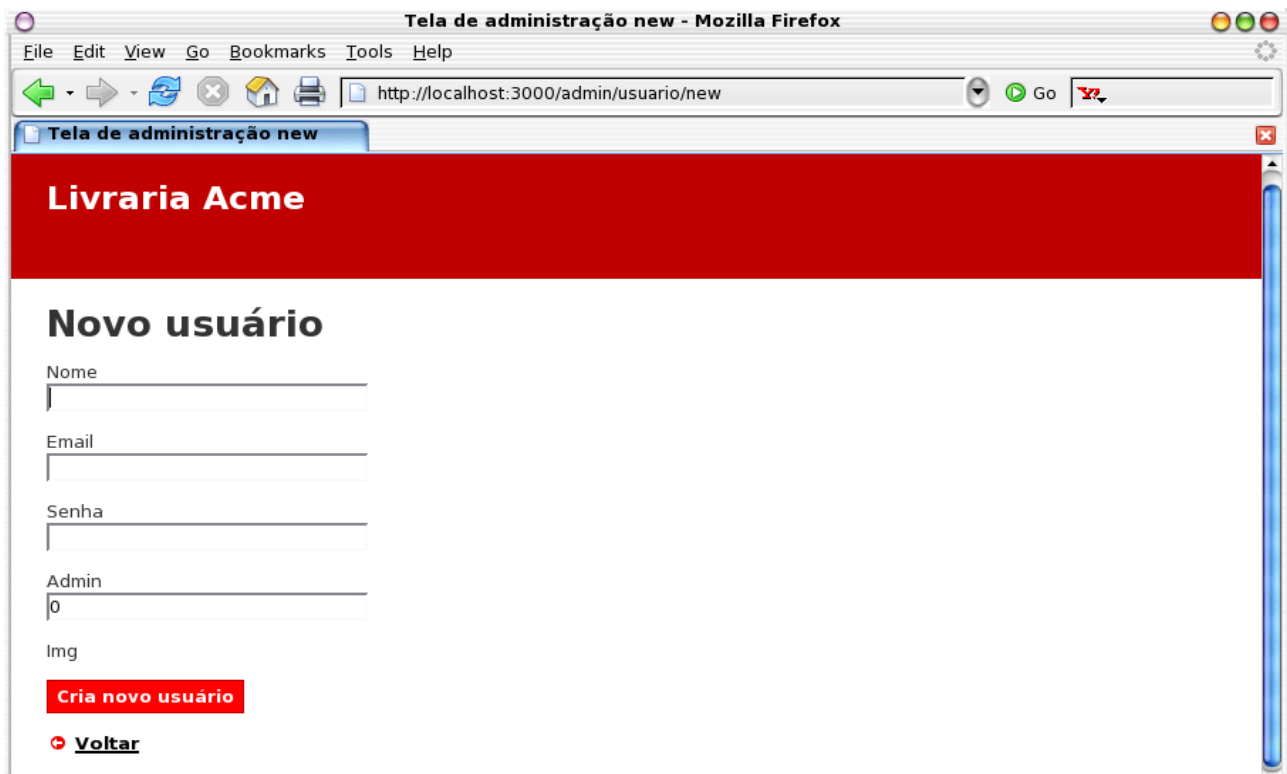
.voltar {
  background:url(../images/voltar.png) 3px center no-repeat;
}

.submit {
  border:1px solid #c00000;
  background:red;
  color:white;
  font-weight:bold;
  padding:3px;
}

```

Tivemos que alterar o *a:hover* pois o Rails especifica um formato próprio no arquivo *scaffold.css*. Como não queria daquela maneira, sobreescrevi a regra. Também adicionei duas pequenas imagens em *<aplicação>/public/images*.

Vamos ter agora:



The screenshot shows a Mozilla Firefox browser window titled 'Tela de administração new - Mozilla Firefox'. The address bar shows 'http://localhost:3000/admin/usuario/new'. The page has a red header with 'Livraria Acme' in white. Below the header, the title 'Novo usuário' is displayed. The form contains five input fields: 'Nome', 'Email', 'Senha', 'Admin' (with a dropdown arrow), and 'Img'. Below the fields is a red button labeled 'Cria novo usuário' and a link labeled 'Voltar' with a red arrow icon.

Ilustração 3: Cadastro de usuários

Ficou legal, mas reparem no campo *admin* e no campo *img*. Não era bem aquilo que esperávamos ali. Antes de complicar um pouco mais (sim, vamos ter que sair um pouco do arroz-com-feijão do *scaffold* nesse caso), vamos ver algumas coisas mais básicas definindo o que precisamos para a tabela *tipos*, e já-já voltamos ao usuário, após absorver mais alguns outros conceitos.

Vamos criar tudo que precisamos para interagir com a tabela *tipos*. Primeiro, o *scaffold*:

```
[taq@/var/www/htdocs/livraria]ruby script/generate scaffold Tipo ←  
Admin::Tipo
```

Apagamos `<aplicação>/app/controllers/admin/tipo.rhtml` e inserimos o layout no controlador:

```
class Admin::TipoController < ApplicationController  
  layout "admin"
```

E alteramos `<aplicação>/app/views/admin/tipo/list.rhtml` de maneira similar ao que fizemos com `<aplicação>/app/views/admin/usuario/list.rhtml`. Vamos ter:



Ilustração 4: Listagem de tipos

E, após configurado o *new.rhtml*:

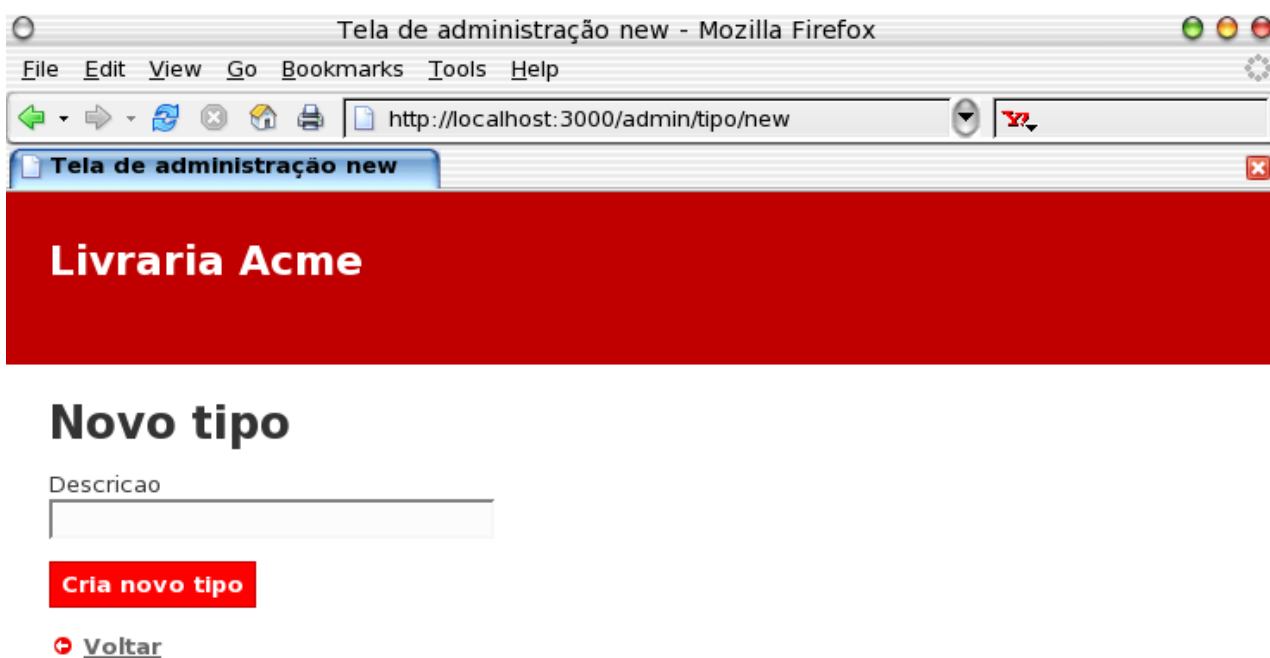


Ilustração 5: Cadastro de tipos

Validação

Um problema que há nessa tela é que se clicarmos no botão “Cria novo tipo”, e a descrição estiver vazia, a operação vai ser concluída. Temos que especificar que o campo tem que conter valor. Para isso alteramos nosso **modelo** (*<aplicação>/models/tipo.rb*):

```
class Tipo < ActiveRecord::Base
  validates_presence_of :descricao
end
```

Inseri a linha em vermelho, indicando que a presença de conteúdo no campo *descrição* é obrigatória. Se clicarmos no botão com o campo vazio, vamos ter:



Ilustração 6: Regra de validação em ação

Uma coisa chatinha ali é que o Rails ainda não é internacionalizado. Podemos fazer uma “gambiarra” e traduzir as mensagens de erro. Para isso, temos que criar um arquivo para sobrescrever algumas coisas do framework, nesse caso, as mensagens de erro. Crie um diretório chamado `<aplicação>/overrides` e crie um arquivo chamado `messages.rb` com o seguinte conteúdo:

```
module ActiveRecord
  class Errors
    @@default_error_messages = {
      :inclusion => "não está incluído na lista",
      :exclusion => "está reservado",
      :invalid => "é inválido.",
      :confirmation => "não corresponde á confirmação",
      :accepted => "deve ser aceito",
      :empty => "não pode estar sem conteúdo",
      :blank => "não pode estar em branco",
      :too_long => "muito longo (máximo %d caracteres)",
      :too_short => "muito curto (mínimo %d caracteres)",
      :wrong_length => "de comprimento errado (deveria ter %d caracteres)",
      :taken => "já está em uso",
      :not_a_number => "não é um número"
    }
  end
end
```

Isso vai sobrescrever as mensagens de erro do `ActiveRecord`. Para carregar o conteúdo desse arquivo altere o arquivo `<aplicação>/config/environment.rb` inserindo a seguinte linha no final:

```
require "#{RAILS_ROOT}/app/overrides/messages"
```

Vamos tentar inserir novamente o registro, após **reiniciarmos o servidor para carregar as novas mensagens**:



Ilustração 7: Algumas mensagens de erro traduzidas

Não ajudou *tanto* mas já deu para contornar a causa do erro e ver como podemos reescrever algumas coisas do Rails de acordo com nossa necessidade. Vamos ver mais tarde mais algumas conversões necessárias. Em tempo: para traduzir o resto das mensagens teríamos que alterar os fontes do Rails, e não que seja difícil, mas a cada nova versão teríamos que aplicar esse “patch”. Outro lugar que pode ser traduzido é o arquivo do controlador, onde existem algumas poucas mensagens também.

Vamos criar alguns tipos, “Livro” e “CD”, e olharmos como ficou nossa tela de listagem:

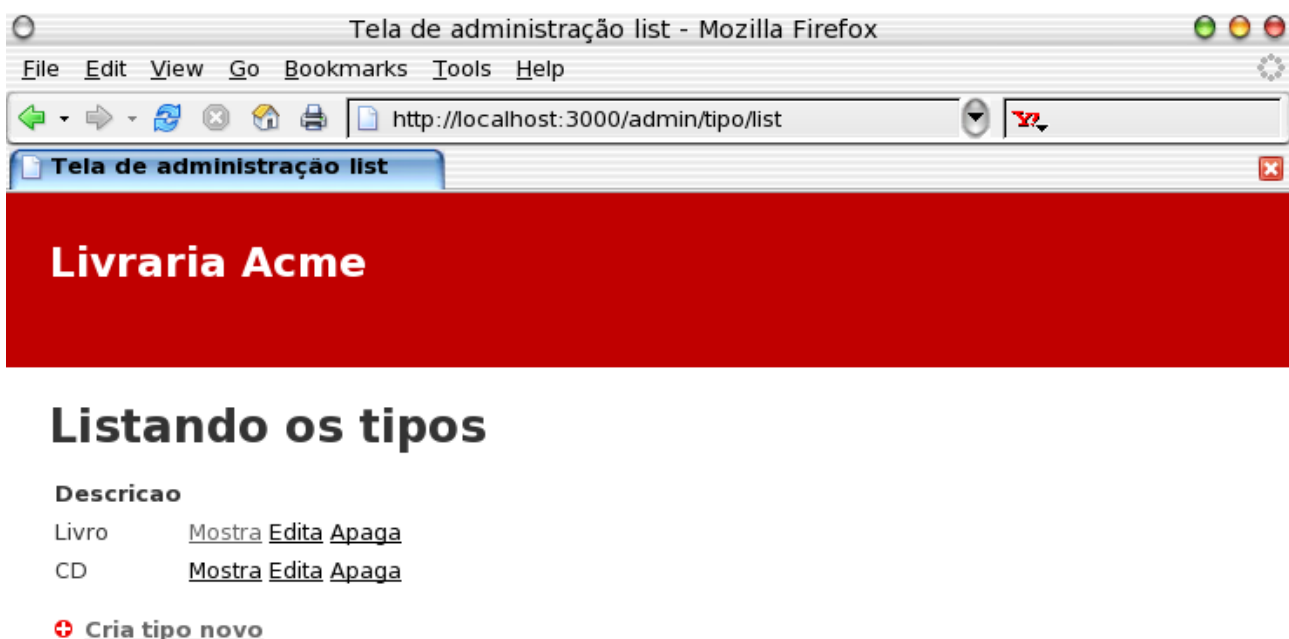


Ilustração 8: Listagem de tipos com algum conteúdo

Vamos precisar alterar, dentro de `<aplicação>/views/admin/tipo`, os arquivos `show.rhtml` e `edit.rhtml`, para terminar a parte administrativa dos tipos.

Agora é criar o controlador e modelo das categorias e fazer a mesma coisa que nos tipos:

```
[taq@/var/www/htdocs/livraria]ruby script/generate scaffold Categoria ←  
Admin::Categoria
```

Depois de tudo pronto, vamos ter:

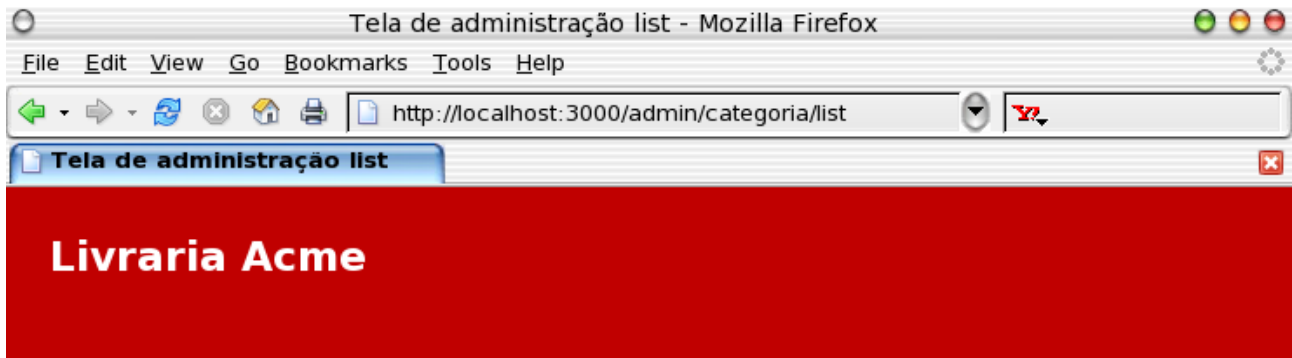


Ilustração 9: Listagem de categorias com algum conteúdo

Relacionamentos

Agora vem uma parte interessante e diferente. Criar o modelo e o controlador para *produto*:

```
[taq@/var/www/htdocs/livraria]ruby script/generate scaffold Produto ←  
Admin::Produto
```

Digo interessante pois cada *produto* tem um *tipo* e uma *categoria*. Então vamos ter que alterar algumas coisinhas. Primeiro nos *modelos*; temos que especificar que o cada tipo e categoria podem estar presentes em vários produtos. Para isso abrimos `<aplicação>/models/tipo.rb` e `<aplicação>/models/categoria.rb` e inserimos:

```
class Tipo < ActiveRecord::Base  
  has_many :produto  
  ...  
end
```

Depois temos que especificar que *produto* tem um tipo e uma categoria, ou seja, ele está relacionado com essas tabelas (tanto que tem *tipo_id* e *categoria_id* como *foreign keys*), então ele depende e pertence de certo modo à essas tabelas. Vamos usar:

```
class Produto < ActiveRecord::Base  
  belongs_to :tipo  
  belongs_to :categoria  
end
```

Dica

A associação *belongs_to* está claramente definida em http://wiki.rubyonrails.com/rails/pages/belongs_to:

“In general, the Foo model belongs_to :bar if the foo table has a bar_id foreign key column.”

Agora, se olharmos na URL de criação de produtos, vamos ver:



Ilustração 10: Cadastro de produtos

Opa! Mas onde estão *tipo* e *categoria*? O scaffold não gera os objetos correspondentes à esses campos no formulário da página, então vamos ter que fazer “na unha”.

A primeira coisa que teremos que fazer é alterar o controlador para informar que para cada novo *tipo* e nova *categoria* (ou quando formos editar algum desses), vamos ter que pegar os valores das tabelas correspondentes. No controlador de *produto* vamos inserir:

```
def new
  @produto = Produto.new
  @tipos = Tipo.find_all.collect{ |t| [t.descricao,t.id] }
  @categorias = Categoria.find_all.collect{ |c| [c.descricao,c.id] }
end

def edit
  @produto = Produto.find(params[:id])
  @tipos = Tipo.find_all.collect{ |t| [t.descricao,t.id] }
  @categorias = Categoria.find_all.collect{ |c| [c.descricao,c.id] }
end
```

Criei um *Array* com a *descrição* e o *id* de cada *tipo* e *categoria*. Esse array já está no formato esperado pelo método *select*, que vai gerar um componente XHTML com todas as opções disponíveis da tabela requerida. E como vamos inserir essa informação em ambos os *forms*, o de criação e o de edição?

O Rails criou para nós vários arquivos no diretório de *views*, inclusive o arquivo *_form.rhtml*, que **vai ser compartilhado para a criação e edição**. Justamente o que precisávamos. O conteúdo *default* do arquivo é:

```

<%= error_messages_for 'produto' %>

<!--[form:produto]-->
<p><label for="produto_descricao">Descricao</label><br/>
<%= text_field 'produto', 'descricao' %></p>

<!--[eoform:produto]-->

```

E vamos trocá-lo para

```

<%= error_messages_for 'produto' %>

<!--[form:produto]-->
<p><label for="produto_descricao">Descricao</label><br/>
<%= text_field 'produto', 'descricao' %></p>

<p><label for="produto_tipo">Tipo</label><br/>
<%= select("produto","tipo_id", @tipos ) %></p>

<p><label for="produto_categoria">Categoria</label><br/>
<%= select("produto","categoria_id", @categorias ) %></p>

<!--[eoform:produto]-->

```

Inserimos duas instruções novas para criarmos os dois elementos *select* XHTML. Agora, se recarregarmos a URL de criação de um novo produto, vamos ter:



The screenshot shows a Mozilla Firefox browser window titled 'Tela de administração new - Mozilla Firefox'. The address bar shows 'http://localhost:3000/admin/produto/new'. The page has a red header with the text 'Livraria Acme'. Below the header, the title 'Novo produto' is displayed. The form contains a text input for 'Descricao', a dropdown menu for 'Tipo' with 'Livro' selected, and another dropdown menu for 'Categoria' with 'Ficção' selected. At the bottom of the form, there is a red button labeled 'Cria' and a link labeled 'Voltar' with a red arrow icon.

Ilustração 11: Cadastro de produtos com tipo e categoria

Olhem lá os dois campos novos! Vamos criar e editar algum produto:



Tela de administração edit - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:3000/admin/produto/edit/2

Livraria Acme

Editando produto

Descricao
Reign in Blood

Tipo
CD

Categoria
Thrash Metal

Salvar

+ [Mostrar](#) | + [Voltar](#)

Os dois campos estão na tela de edição ainda, pois como mencionei, o arquivo `_form.rhtml` é compartilhado por ambas as telas de criação e edição.

A tela da ação `list` tem algumas características:



Tela de administração list - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:3000/admin/produto/list

Livraria Acme

Listando produtos

Descricao	Tipo	Categoria	
Ponto de Impacto	Livro	Ficção	Mostra Edita Apaga
Reign in Blood	CD	Thrash Metal	Mostra Edita Apaga

+ [Cria produto novo](#)

Ilustração 12: Listagem de produtos

Como que eu listei ali a *descrição* de *tipo* e *categoria*, sendo que na tabela do *produto* só tenho o *id* (*tipo_id* e *categoria_id*) de cada um? É aí que entra as regras do *has_many* e *belongs_to* que especificamos acima. O Rails automaticamente criou propriedades *tipo* e *categoria* para cada *produto*, então na listagem eu usei:

```
<% for produto in @produtos %>
<tr>
  <% for column in Produto.content_columns %>
```

```

<td><%=h produto.send(column.name) %></td>
<% end %>

<td><%=h produto.tipo.descricao %></td>
<td><%=h produto.categoria.descricao %></td>

<td><%= link_to 'Mostra', :action => 'show', :id => produto %></td>
<td><%= link_to 'Edita', :action => 'edit', :id => produto %></td>
<td><%= link_to 'Apaga', { :action => 'destroy', :id => produto }, :confirm
=> 'Tem certeza?' %></td>
</tr>
<% end %>

```

Que me deu acesso às propriedades do *tipo* e da *categoria*.

Personalizando o código gerado

Agora que vimos mais alguns conceitos, vamos voltar na tela do usuário. O *scaffold* quebra o galho em muitas coisas, mas em algumas temos que meter a mão na massa e alterar o código gerado. Vamos pegar o modelo do formulário do usuário, alterando os arquivos

<aplicação>/app/views/admin/usuario/new.rhtml

```

<div id="conteudo">
  <h1>Novo usuário</h1>

  <%= form_tag(:action => 'create'), :multipart => true) %>
  ...

```

e <aplicação>/app/views/admin/usuario/_form.rhtml para:

```

<%= error_messages_for 'usuario' %>

<!--[form:usuario]-->
<table>
  <tr>
    <td>Nome</td><td><%= text_field 'usuario', 'nome' %></td>
  </tr>
  <tr>
    <td>Email</td><td><%= text_field 'usuario', 'email' %></td>
  </tr>
  <tr>
    <td>Senha</td><td><%= password_field 'usuario', 'senha' %></td>
  </tr>
  <tr>
    <td>Administrador</td><td><%= check_box 'usuario', 'admin' %></td>
  </tr>
  <tr>
    <td>Imagem</td><td><%= file_field("usuario","img") %></td>
  </tr>
</table>
<!--[eoform:usuario]-->

```

Alguns pontos importantes:

- **:multipart => true** – A grosso modo isso indica que estamos enviando mais do que campos texto no nosso formulário, vamos enviar dados binários provenientes de um arquivo. Preste atenção nisso, senão não conseguiremos enviar os dados binários!
- **text_field** – Vai gerar um campo texto no nosso formulário, tendo o seu conteúdo preenchido com o valor de **@usuario.nome**.

- **password_field** – Gera um campo texto similar ao anterior, mas com asteriscos no lugar dos caracteres, para informarmos uma senha.
- **check_box** – Gera um campo tipo *checkbox* através do valor **@usuario.admin**
- **file_field** – Gera um campo texto com um botão para *upload* de arquivos. É através desse campo que serão transmitidos os dados binários esperados pelo *multipart*.

Com tudo isso, vamos conseguir configurar um usuário com nome, email, senha, indicar se ele é um administrador do sistema e ainda ter a opção de efetuar o upload de uma foto. Vai ser parecido com isso:



The screenshot shows a Mozilla Firefox browser window with the title 'Tela de administração new - Mozilla Firefox'. The address bar displays 'http://localhost:3000/admin/usuario/new'. The page has a red header with the text 'Livraria Acme'. Below the header, the main content area is titled 'Novo usuário'. It contains a form with the following fields: 'Nome', 'Email', 'Senha', and 'Imagem'. The 'Senha' field is masked with asterisks. There is a checkbox labeled 'Administrador' next to the 'Senha' field. The 'Imagem' field has a 'Browse...' button next to it. Below the form fields, there is a red button labeled 'Cria novo usuário' and a link labeled 'Voltar' with a plus icon.

Ilustração 13: Cadastro de usuário com campo de imagem

Agora vamos inserir um usuário:

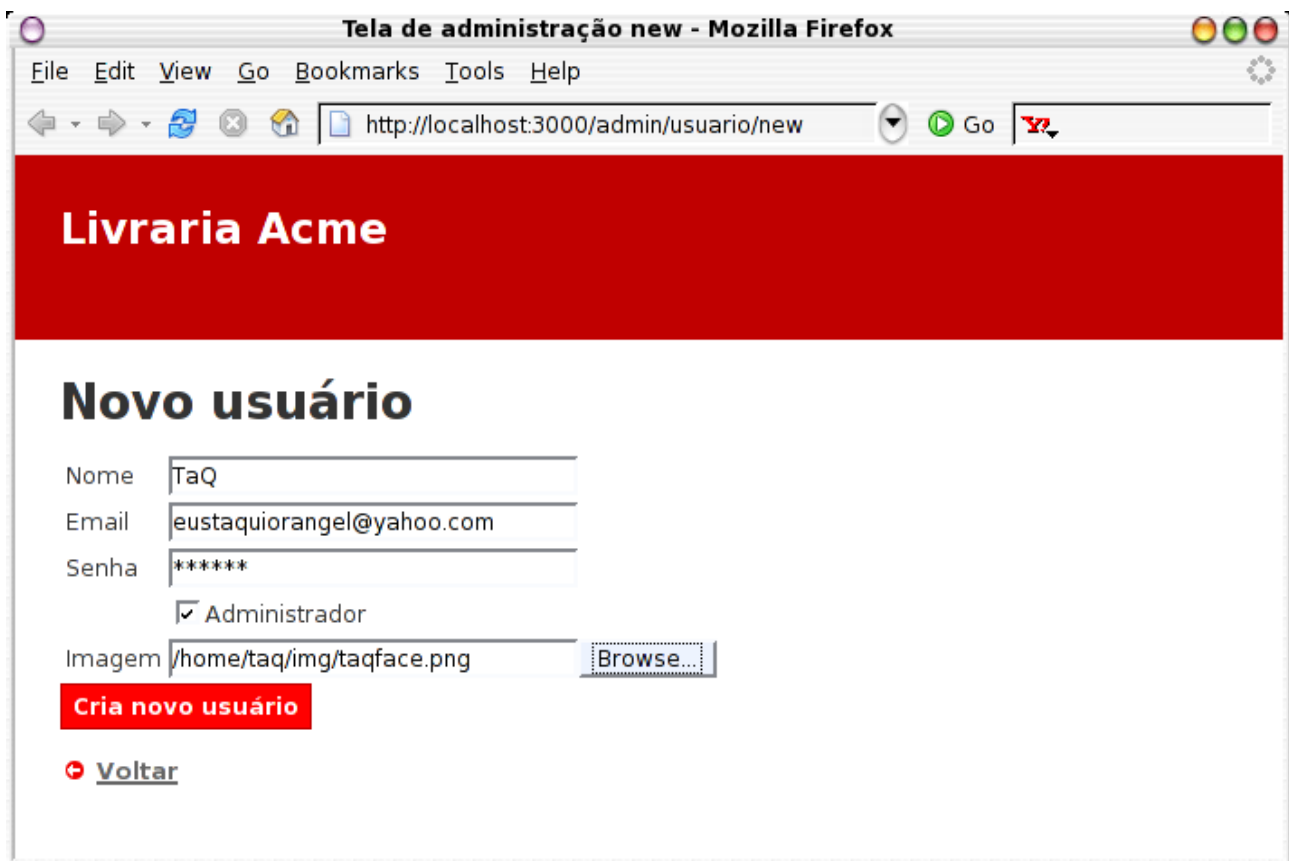


Ilustração 14: Cadastrando um usuário

Porém, na listagem:

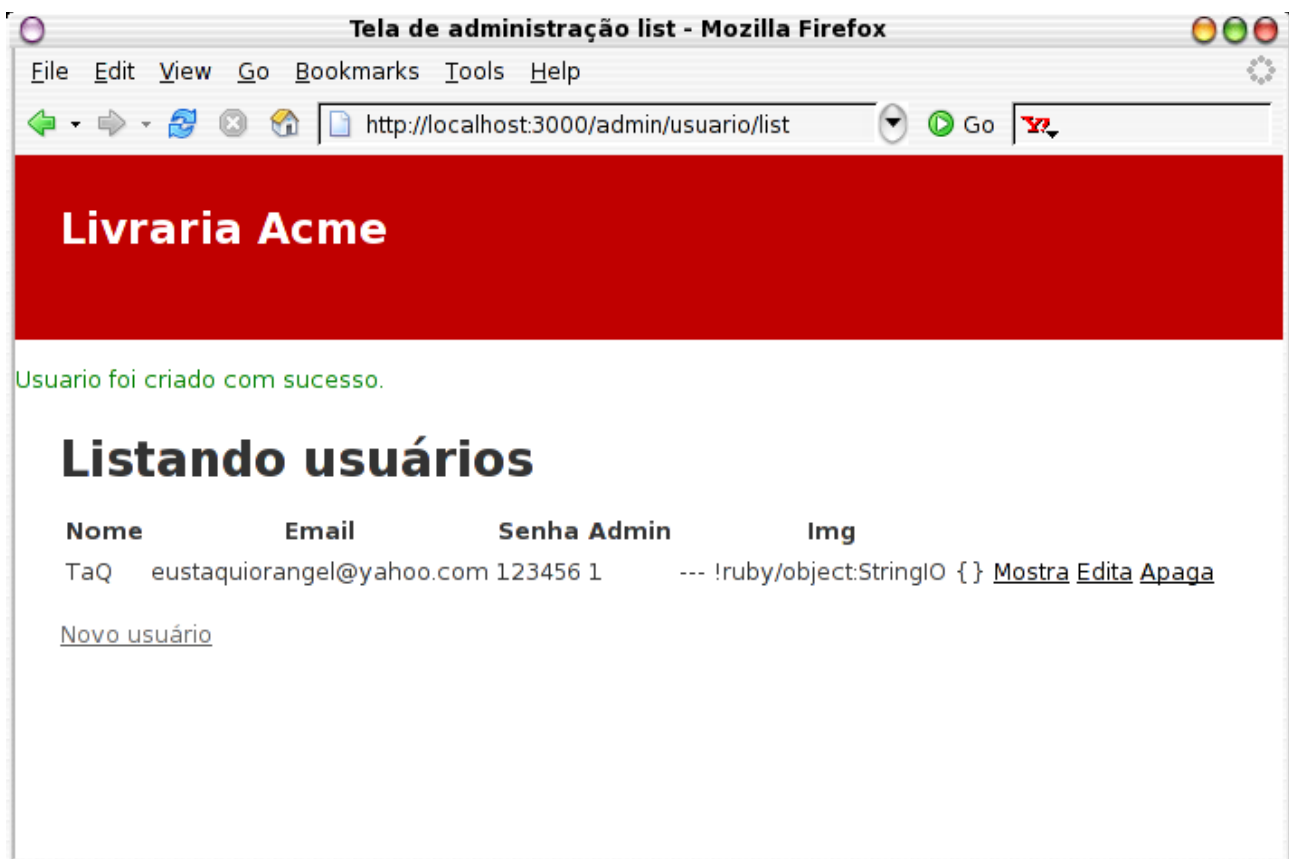


Ilustração 15: Listagem estranha de usuários

Algumas coisas a serem notadas:

1. Ei! De que adianta uma senha se ela é apresentada na listagem? Isso não pode aparecer ali.
2. Ter o campo como administrador como "1" pode significar que o usuário em questão tem esse tipo de permissão, mas fica meio confuso.
3. Essa imagem está esquisita. Não parece efetivamente que um arquivo binário foi gravado ali.

Vamos dar uma arrumada nisso, customizando o arquivo

`<aplicação>/app/views/admin/usuario/list.rhtml:`

```
<div id="conteudo">
  <h1>Listando usuários</h1>
  <table>
    <tr>
      <th>Nome</th>
      <th>Email</th>
      <th>Administrador?</th>
      <th>Imagem</th>
    </tr>
    <% for usuario in @usuarios %>
      <tr>
        <td><%=h usuario.nome %></td>
        <td><%=h usuario.email %></td>
        <td><%=h usuario.admin==1 ? "Sim" : "Não" %></td>
        <td> usuario.id) %>
%>"/></td>
        <td><%= link_to 'Mostra', :action => 'show', :id => usuario %></td>
        <td><%= link_to 'Edita', :action => 'edit', :id => usuario %></td>
        <td><%= link_to 'Apaga', { :action => 'destroy', :id => usuario }, %>
:confirm => 'Are you sure?', :post => true %></td>
      </tr>
    <% end %>
  </table>
  <p>
    <%= link_to 'Página anterior', { :page => %>
@usuario_pages.current.previous } if @usuario_pages.current.previous %>
    <%= link_to 'Próxima página', { :page => @usuario_pages.current.next } %>
if @usuario_pages.current.next %>
  </p>
  <p>
    <%= link_to 'Novo usuário', :action => 'new' %>
  </p>
</div>
```

Isso vai nos mostrar:



Ilustração 16: Listagem de usuários melhorada

Não melhorou lá aquelas coisas. Algumas coisas a serem notadas:

1. Especifiquei os nomes das colunas “na unha” agora, pois são poucas e quero fazer um filtro, para, por exemplo, não mostrar a senha.
2. Os valores dos campos posso mostrar com *usuario.campo*, pois já estão *hardcoded*, ou seja, não preciso mais usar o método *send* com uma string para recuperar o valor de um campo.
3. Na imagem, usei `` com *url_for* usando como parâmetros *action="img"* e *usuario.id*

Arquivos binários

No caso da *action="img"*, precisamos de uma ação *img* no controlador para recuperarmos a nossa imagem do banco de dados. Vamos inserir esse método no controlador do usuário:

```
def img
  @usuario = Usuario.find(params[:id])
  send_data(@usuario.img, :type=>"image/png", :disposition=>"inline")
end
```

Recarregando a página de listagem de usuários, não vamos notar muita diferença. Isso por que a nossa imagem foi gravada de maneira errada no banco de dados. Foi gravada como uma string, e não como dados binários. Para consertar isso, vamos alterar o modelo do usuário, para quando receber a informação de onde carregar os dados, ler e guardar no banco de dados. Para isso vamos fazer algumas “jogadinhas”.

A primeira delas, é trocar o nome do campo que recebe o arquivo na view `<aplicacao>/app/views/admin/usuario/_form.rhtml`. Vamos trocar de

```
<td>Imagem</td><td><%= file_field("usuario","img") %></td>
```

Para

```
<td>Imagem</td><td><%= file_field("usuario","imagem") %></td>
```

Qual o motivo? Nem temos um campo chamado **imagem** em nossa tabela do banco de dados! E,

como não temos, o que vai ser feito com o que for recebido do campo do formulário? Vamos alterar nosso modelo em `<aplicacao>/app/models/usuario.rb` e adicionar:

```
def imagem=(img_field)
  self.img = img_field.read
end
```

O que vai acontecer é que quando processarmos o nosso formulário, o conteúdo do campo *imagem* vai ser passado para um método chamado *imagem=(i)* no nosso modelo, que é o que acabamos de definir. Após receber o conteúdo do campo, utilizamos o método **read** para ler o conteúdo e armazenar em **self.img**, que referencia o campo **img**, que realmente existe em nosso banco de dados. Vocês podem perguntar por que não reescrevemos o método **img=(i)**, afinal, ele já está no modelo, mas pensem o que iria acontecer:

```
def img=(img_field)
  self.img = img.read
end
```

O método **img** carregaria os dados e chamaria **img** novamente (afinal, estamos atribuindo valor para **img** com o sinal de =) e iríamos cair em chamadas recursivas infinitas (na verdade, não, pois o script iria ser interrompido após um certo número – elevado – de chamadas recursivas).

Vamos apagar o cadastro que fizemos e adicionar os dados novamente. Por enquanto, insira uma imagem PNG. Vamos ver como ficou:

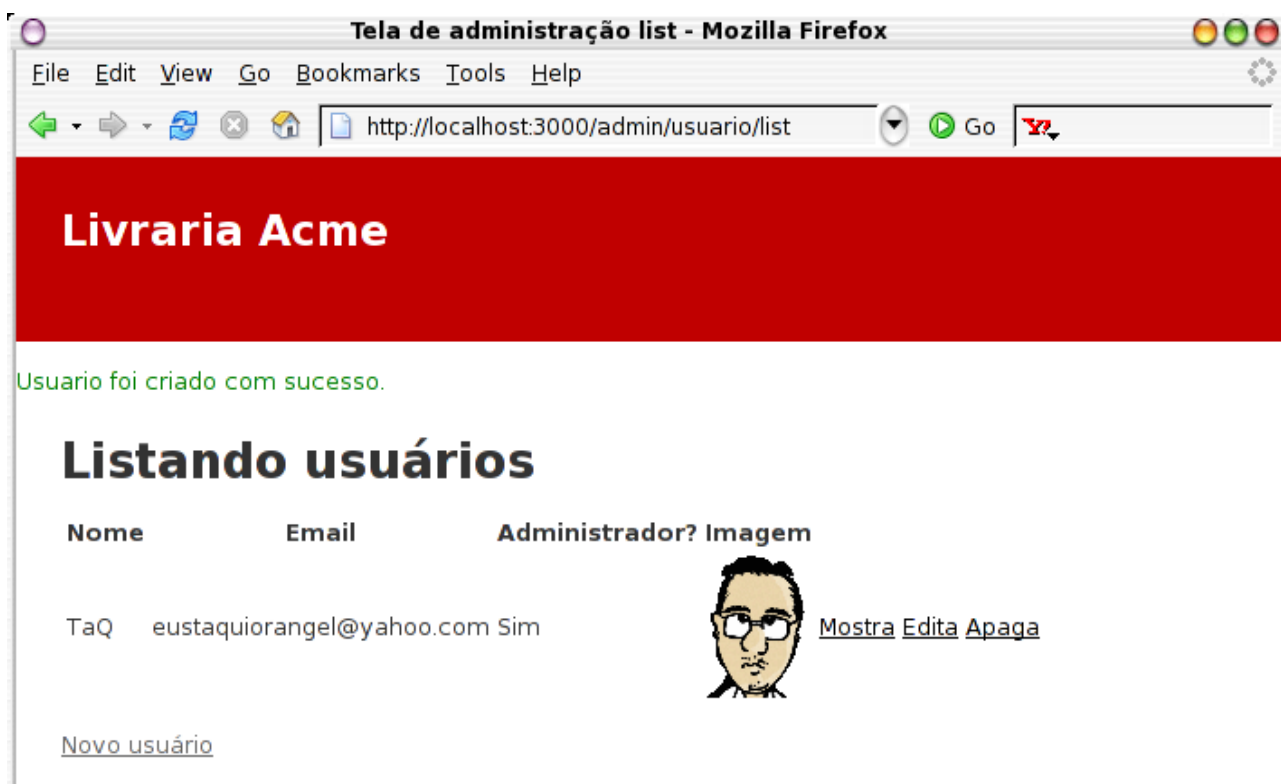


Ilustração 17: Listagem de usuário com imagem

Ah, agora sim! A imagem está corretamente grava em nosso banco de dados, e foi visualizada corretamente também.

Vamos mexer mais um pouco na tela de listagem de usuários:

```
<div id="conteudo">
  <h1>Listando usuários</h1>
  <table>
    <% for usuario in @usuarios %>
```

```

<tr>
  <td rowspan="4"> @
usuario.id) %>"/></td>
</tr>
<tr>
  <td><h2><%=h usuario.nome %></h2></td>
</tr>
<tr>
  <td><%=h usuario.email %></td>
</tr>
<tr>
  <td>
    <%= link_to 'Mostra', :action => 'show', :id => usuario %>
    <%= link_to 'Edita', :action => 'edit', :id => usuario %>
    <%= link_to 'Apaga', { :action => 'destroy', :id => usuario },
:confirm => 'Are you sure?', :post => true %>
  </td>
</tr>
<% end %>
</table>
<p>
  <%= link_to 'Página anterior', { :page => @
@usuario_pages.current.previous } if @usuario_pages.current.previous %>
  <%= link_to 'Próxima página', { :page => @usuario_pages.current.next }
if @usuario_pages.current.next %>
</p>
<p>
  <%= link_to 'Novo usuário', { :action => 'new'}, :class => 'novo' %>
</p>
</div>

```

A listagem vai ficar assim:



Ilustração 18: Listagem de usuário modificada

Um problema que ocorre com upload de imagens (e outros arquivos binários) é que temos que guardar o MIME type para enviar para o browser quando formos recuperar os dados. No nosso caso, na ação *img* do controlador, está fixo como *image/png*.

Podemos armazenar outros tipos, é claro, mas teríamos que ter mais campos para armazenar o conteúdo do MIME type. No nosso caso, vamos optar por armazenar somente imagens PNG. Para criarmos um filtro eficiente, vamos inserir essa trava no modelo.

Para isso, temos que guardar em algum lugar o tipo de arquivo que foi feito o upload. Como não temos nenhum lugar para guardar isso no banco de dados, vamos criar um atributo temporário para guardarmos o MIME type.

```
class Usuario < ActiveRecord::Base
  attr_accessor :content_type
  validates_presence_of :nome, :email, :senha
  def imagem=(img_field)
    self.content_type = img_field.content_type
    self.img = img_field.read
  end
end
```

Com isso já estamos armazenando o MIME type. Mas como validar? Como optamos por armazenar *ou não* uma imagem, temos que verificar se o usuário está efetuando upload de algum arquivo, e podemos verificar isso no atributo **filename**, armazenando-o também.

```
class Usuario < ActiveRecord::Base
  attr_accessor :content_type, :filename
  validates_presence_of :nome, :email, :senha
  def imagem=(img_field)
    self.content_type = img_field.content_type
    self.filename = img_field.original_filename
    self.img = img_field.read
  end
end
```

Agora podemos fazer a validação. No modelo, podemos (re)definir o método **validate**, que será protegido (*protected*) e será chamado para validar o que enviamos através do formulário. Vamos escrevê-lo:

```
class Usuario < ActiveRecord::Base
  attr_accessor :content_type, :filename
  validates_presence_of :nome, :email, :senha

  protected
  def validate
    if self.filename.length>0 and self.content_type !~ /^image\/png/
      errors.add(:img, "A imagem tem que ser um PNG! Foi enviado ↵")
    end
  end

  public
  def imagem=(img_field)
    self.content_type = img_field.content_type
    self.filename = img_field.original_filename
    self.img = img_field.read
  end
end
```

Agora, tentando efetuar upload de um JPEG:

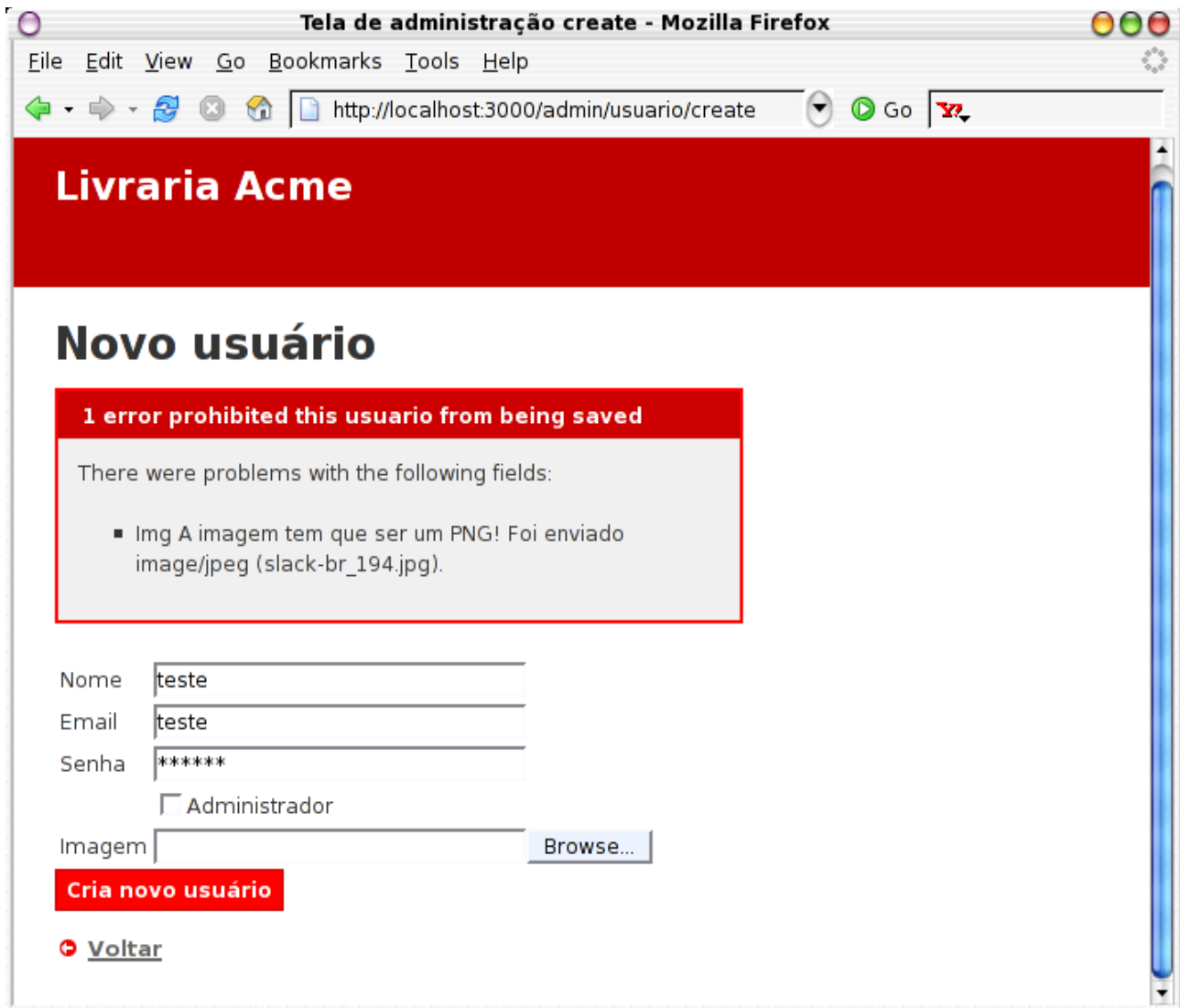


Ilustração 19: Barrando o upload de um JPEG

Se enviarmos um PNG, tudo funciona direito:

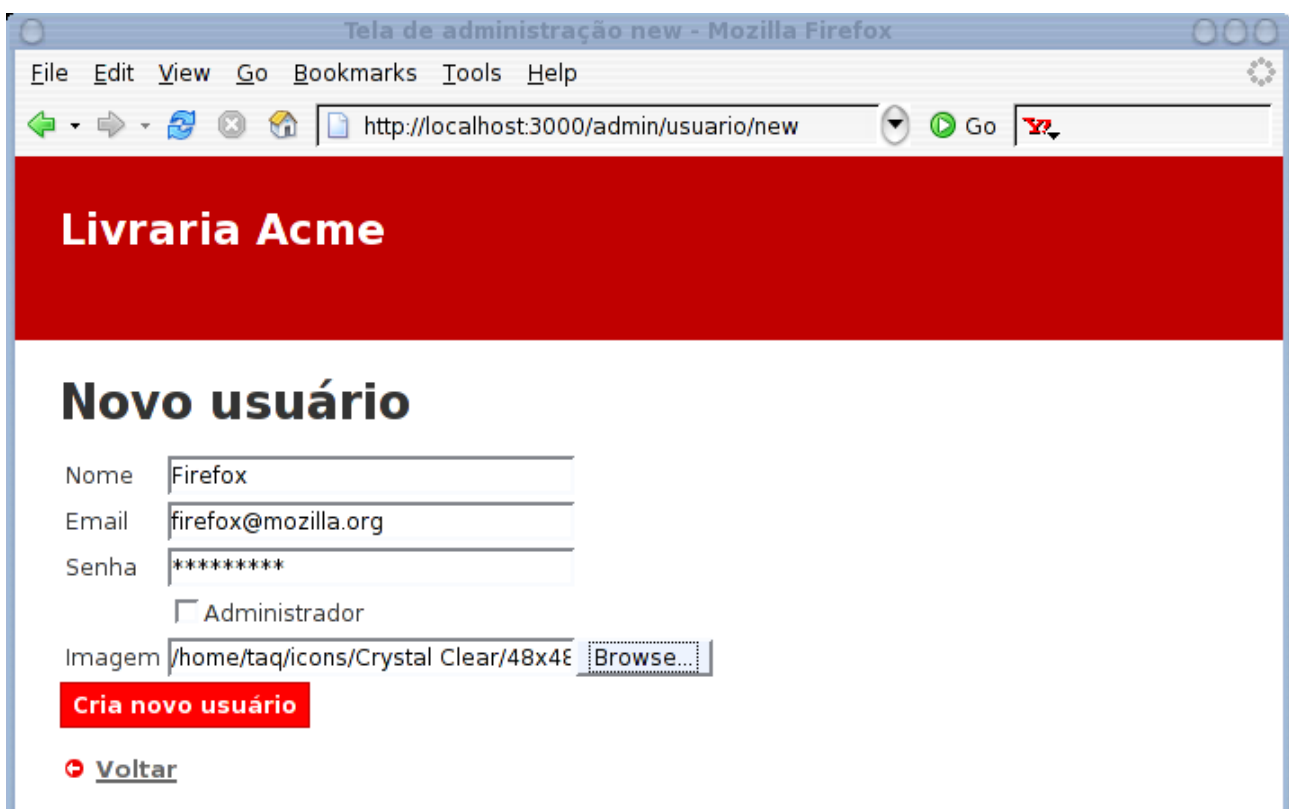


Ilustração 20: Efetuando upload de um PNG



Ilustração 21: Listagem de usuários com imagens

Para dar uma garibada na tela, vamos limitar os valores informados no formulário, verificando que nome e email são valores únicos, os tamanhos do nome e senha e o formato do email. Alteramos o modelo do usuário para:

```
class Usuario < ActiveRecord::Base
  attr_accessor :content_type, :filename
  validates_presence_of :nome, :email, :senha
  validates_uniqueness_of :nome, :email
  validates_length_of :nome, :in => 3..75
  validates_length_of :senha, :in => 3..10
  validates_format_of :email, :with => /[a-zA-Z0-9_.-]+@[a-zA-Z0-9_-]
  +\.)+[a-zA-Z]{2,4}/
```

Tentando efetuar alguns cadastros inválidos:

Tela de administração create - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:3000/admin/usuario/create

Livraria Acme

Novo usuário

3 errors prohibited this usuario from being saved

There were problems with the following fields:

- Nome muito curto (mínimo 3 caracteres)
- Senha muito curto (mínimo 3 caracteres)
- Email é inválido.

Nome

Email

Senha

Administrador ☐

Imagem Browse...

Cria novo usuário

[Voltar](#)

Ilustração 22: Validação no cadastro de usuários - campos muito curtos

Tela de administração create - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:3000/admin/usuario/create

Livraria Acme

Novo usuário

2 errors prohibited this usuario from being saved

There were problems with the following fields:

- Nome já está em uso
- Email já está em uso

Nome

Email

Senha

Administrador ☒

Imagem Browse...

Cria novo usuário

[Voltar](#)

Ilustração 23: Validação no cadastro de usuários – nome e email repetidos

Agora alterando a ação *edit* para que mostre a foto atual quando quisermos alterar algum dado:

```
<div id="conteudo">
  <h1>Editando usuario</h1>

  <%= form_tag(:action => 'update', :id => @usuario},:multipart => true) %>
  <%= render :partial => 'form' %>
  <%= submit_tag 'Salvar', :class => 'submit' %>
  <%= end_form_tag %>
  <p>
    <b>Foto atual<b/><br/>
    @usuario.id) %>" />
  </p>
  <p>
    <%= link_to 'Mostra', :action => 'show', :id => @usuario %> |
    <%= link_to 'Volta', :action => 'list' %>
  </p>
</div>
```

Vai resultar em:



Ilustração 24: Mostrando a foto atual

Tarefas administrativas

Agora que já temos um cadastro de usuários, podemos fazer uma tela de login. Vamos criar um controlador de login para isso, com duas ações, **login** e **logout**:

```
[taq@/var/www/htdocs/livraria]ruby script/generate controller Login ↵  
login logout
```

Vamos precisar de uma view (<aplicação>/app/views/login/login.rhtml) para a tela de login:

```
<div id="conteudo">  
  <h1>Identificação de usuário</h1>  
  <%= form_tag %>  
  <table>  
    <tr>  
      <td>Email</td>  
      <td><%= text_field("usuario","email") %></td>  
    </tr>  
    <tr>  
      <td>Senha</td>  
      <td><%= password_field("usuario","senha") %></td>  
    </tr>  
  </table>  
  <%= submit_tag "Entrar no sistema", :class => "submit" %>  
  <%= end_form_tag %>  
</div>
```

Por enquanto, vamos usar o layout de *admin*, depois mudaremos isso. Essa view vai ser acessada de duas maneiras: visualizando normalmente e tendo os dados do seu formulário enviados. Vejamos como fica visualizando normalmente:

```
class LoginController < ApplicationController  
  layout "admin"  
  def login  
  end  
  def logout  
  end  
end
```



The screenshot shows a web browser window with the address bar displaying 'http://localhost:3000/login/login'. The page has a red header with the text 'Livraria Acme'. Below the header, the title 'Identificação de usuário' is displayed. There are two input fields: 'Email' and 'Senha'. Below these fields is a red button with the text 'Entrar no sistema'.

Ilustração 25: Tela de login

Do jeito que está, essa tela não faz nada. O que queremos que ela faça é armazenar o id e o

status do usuário em uma **sessão**, se a senha estiver correta. Se o usuário processou o formulário, vamos consultar e verificar a senha no banco de dados. Se estiver visualizando a tela, vamos inicialmente zerar as informações da sessão. Podemos diferenciar se o formulário foi processado ou a página visualizada utilizando **request.get?**:

```
class LoginController < ApplicationController
  layout "admin"
  def login
    if request.get?
      session[:user_id] = nil
      session[:user_nome]= nil
      session[:user_adm] = nil
      @usuario = Usuario.new
    end
  end
  def logout
  end
end
```

Para armazenar as informações na sessão, é só utilizar **session[:variavel]**.

Já podemos verificar se o formulário foi processado ou não inserindo um **else** ali. O que vamos fazer é criar um objeto *Usuario* com os parâmetros que foram enviados, procurar no banco de dados se há algum registro que tenha o email e senha informados, e se encontrados, armazenar na sessão e redirecionar o usuário para alguma página condizente com o seu status. Se o usuário tiver status de administrador, por exemplo, iremos para a página inicial de administração do site.

```
class LoginController < ApplicationController
  layout "admin"
  def login
    if request.get?
      session[:user_id] = nil
      session[:user_nome]= nil
      session[:user_adm] = nil
      @usuario = Usuario.new
    else
      @usuario = Usuario.new(params[:usuario])
      dados = Usuario.find(:first, :conditions => [" email = ? and senha ←
= ?", @usuario.email, @usuario.senha])
      if dados
        session[:user_id] = dados.id
        session[:user_nome]= dados.nome
        session[:user_adm] = dados.admin
        if dados.admin==1
          redirect_to(:controller => "admin/admin", :action => "index")
        end
      else
        flash[:notice] = "Dados inválidos!"
      end
    end
  end
  def logout
  end
end
```

Uma coisa interessante de se notar: na construção da condição do *find*, utilizei um array com

```
[" email = ? and senha = ?", @usuario.email, @usuario.senha]
```

Isso vai fazer com que, quando o comando for montado, seja “dado uma geral” nele para evitar alguma coisa estranha que possa permitir algum ataque de SQL Injection ⁹. Seria fácil fazer esse tipo de coisa se utilizássemos

```
"email = '#{@usuario.email}' and senha='#{@usuario.senha}'"
```

Do jeito que fizemos no primeiro (e correto) exemplo, estamos utilizando alguns *bindings*, nesse caso, pontos de interrogação, indicando para o Rails onde inserir os valores das variáveis que indicamos a seguir, dando tratamento adequando para as variáveis, para evitar que alguém tente fazer alguma gracinha.

Se os dados foram encontrados usando *find*, já podemos armazenar os dados na sessão e redirecionar o usuário para ... opa, para onde? Quero uma página inicial de administração, nesse caso. Para o usuário “normal”, ainda não fizemos nada. Nesse caso, vamos criar uma ação default para o controlador *admin*:

```
[taq@var/www/htdocs/livraria]ruby script/generate controller Admin::Admin
exists  app/controllers/admin
exists  app/helpers/admin
create  app/views/admin/admin
exists  test/functional/admin
create  app/controllers/admin/admin_controller.rb
create  test/functional/admin/admin_controller_test.rb
create  app/helpers/admin/admin_helper.rb
```

Vamos alterar `<aplicação>/app/controllers/admin/admin_controller.rb` para:

```
class Admin::AdminController < ApplicationController
  layout "admin"

  def index
    render :action => 'index'
  end
end
```

Desse modo vamos ter uma ação (`<aplicação>/app/views/admin/admin/index.rhtml`) default *index*, que vai conter

```
<div id="conteudo">
  <h1>Tela de administração</h1>
  <h2>Bem-vindo, <%= session[:user_nome] %></h2>
</div>
```

que, após digitada a senha correta, vai nos mostrar



Ilustração 26: Dando as boas vindas ao administrador

⁹ SQL Injection é uma técnica que permite “craquear” os comandos enviados para um banco de dados através de alguns caracteres específicos para essa finalidade.

Agora já podemos alterar o layout de admin. Vamos inserir um menu em <aplicação>/app/views/layouts/admin.rhtml:

```
<div id="menu">
  <ul>
    <li><%= link_to "Inicial",:controller=>"admin",:action=>"index" %></li>
    <li><%= link_to "Tipos",:controller => "tipo" , :action => "list" %></li>
    <li><%= link_to "Categorias",:controller=>"categoria",:action=>"list"
%></li>
    <li><%= link_to "Produtos",:controller=>"produto",:action=>"list" %></li>
    <li><%= link_to "Usuários",:controller=>"usuario",:action=>"list" %></li>
    <li><%= link_to "Logout "+(session[:user_nome].nil? ? "" :
session[:user_nome]),{:controller => "../login" ,:action => "logout"},
:confirm => "Tem certeza que deseja fazer logout?" %></li>
  </ul>
</div>
<div id="msgs">
  <%= flash[:notice] %>
</div>
```

E alterando o arquivo CSS:

```
#conteudo {
  float:left;
}

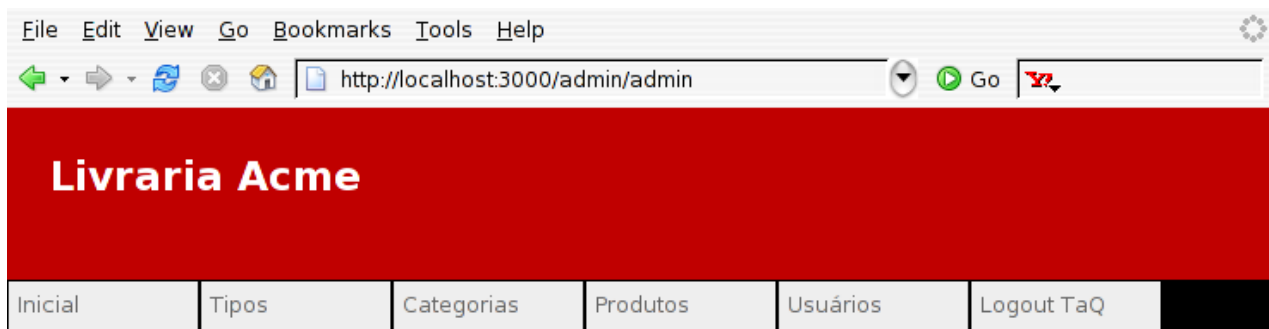
#msgs {
  color:green;
  font-weight:bold;
}

#menu ul {
  padding:0;
  margin:0;
  float:left;
  width:100%;
  list-style:none;
  background:black;
}

#menu li {
  display:inline;
}

#menu li a {
  padding:5px;
  border:1px solid black;
  text-decoration:none;
  width:100px;
  background:#eee;
  float:left;
}
```

Vamos ter:



Tela de administração

Bem-vindo, TaQ

Ilustração 27: Tela de administração com menu

Para navegar no que já fizemos até agora, é só clicar no menu horizontal.

Fica uma observação importante na linha

```
<li><%= link_to "Logout "+(session[:user_nome].nil? ? " " :  
session[:user_nome]),{:controller => "../login" ,:action => "logout"},  
:confirm => "Tem certeza que deseja fazer logout?" %></li>
```

onde referenciamos o controlador no diretório *acima* do controlador atual, e inserimos uma mensagem de confirmação de logout. Falando em logout, vamos implementar a ação de logout, que vai apenas limpar os dados da sessão e redirecionar para a página de login, em `<aplicacao>/app/controllers/login_controller.rb`:

```
def logout  
  session[:user_id] = nil  
  session[:user_nome] = nil  
  session[:user_adm] = nil  
  redirect_to(:action => "login")  
end
```

Agora uma coisa **muito importante**. Não podemos de modo algum dar acesso a qualquer um nas nossas telas de administração de sistema. Já que temos os usuários cadastrados e métodos eficientes de login e logout, podemos verificar se o usuário está logado e autorizado a usar as telas de administração. Para isso vamos alterar os controladores, primeiro da aplicação toda usando o arquivo `<aplicação>/app/controllers/application.rb`:

```
class ApplicationController < ActionController::Base  
  def admin?  
    if logged? and session[:user_adm] < 1  
      flash[:notice] = "Você não tem autoridade para isso."  
      redirect_to(:controller=> "../login", :action=> "logout")  
    end  
  end  
  def logged?  
    if not session[:user_id]
```



```

        flash[:notice] = "Por favor efetue o login."
        redirect_to(:controller=> "../login", :action=> "login")
        return false
      end
      return true
    end
  end
end

```

Definimos dois métodos, um para verificar se o usuário é um administrador e outro para verificar se está logado no sistema.

Agora vamos alterar os controladores específicos, começando com *tipo*:

```

class Admin::TipoController < ApplicationController
  layout "admin"
  before_filter :admin?
  ...
end

```

e por aí vai. Uma das coisas que você vai notar é que, na tela de login, enquanto não efetuado o login, tentarmos clicar em alguma das opções, vamos ter um erro, pois não estamos dentro do controlador *admin* para especificarmos as ações abaixo dele. Inclusive no caso de usarmos controladores com ... subcontroladores, que seja esse o nome de um controlador que se encontra em um subdiretório, é bom utilizarmos o caminho completo dos controladores na *url_to*, levando em conta que a barra (/) vai significar o diretório “raiz” dos controladores (<aplicacao>/app/controllers).

Vamos alterar a view *admin* para:

```

<div id="menu">
  <ul>
    <li><%= link_to "Inicial" ,:controller => "/admin/admin"
, :action => "index" %></li>
    <li><%= link_to "Tipos" ,:controller => "/admin/tipo"
, :action => "list" %></li>
    <li><%= link_to "Categorias",:controller =>
"/admin/categoria",:action => "list" %></li>
    <li><%= link_to "Produtos" ,:controller => "/admin/produto"
, :action => "list" %></li>
    <li><%= link_to "Usuários" ,:controller => "/admin/usuario"
, :action => "list" %></li>
    <li><%= link_to "Logout "+(session[:user_nome].nil? ? " " :
session[:user_nome]),{:controller => "/login" ,:action => "logout"}, :confirm
=> "Tem certeza que deseja fazer logout?" %></li>
  </ul>
</div>

```

Agora, quando vamos para o controlador *admin*, podemos acessar o controlador *login* normalmente. Mas, como o processo de login não é exclusivo dos administradores do sistema, vamos alterar o layout para o layout externo da nossa livraria, que vai começar a ser construído agora.

O conteúdo do arquivo <aplicação>/app/views/layouts/livraria.rhtml ficou:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>Livraria ACME</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <%= stylesheet_link_tag "livraria" %>
  </head>
  <body>
    <div id="topo_livraria">

```

```

    <%= image_tag("acme.png") %>
    A livraria nos trilhos do sucesso
  </div>
  <%= @content_for_layout %>
</body>
</html>

```

Após alguma manipulação no arquivo CSS, temos:

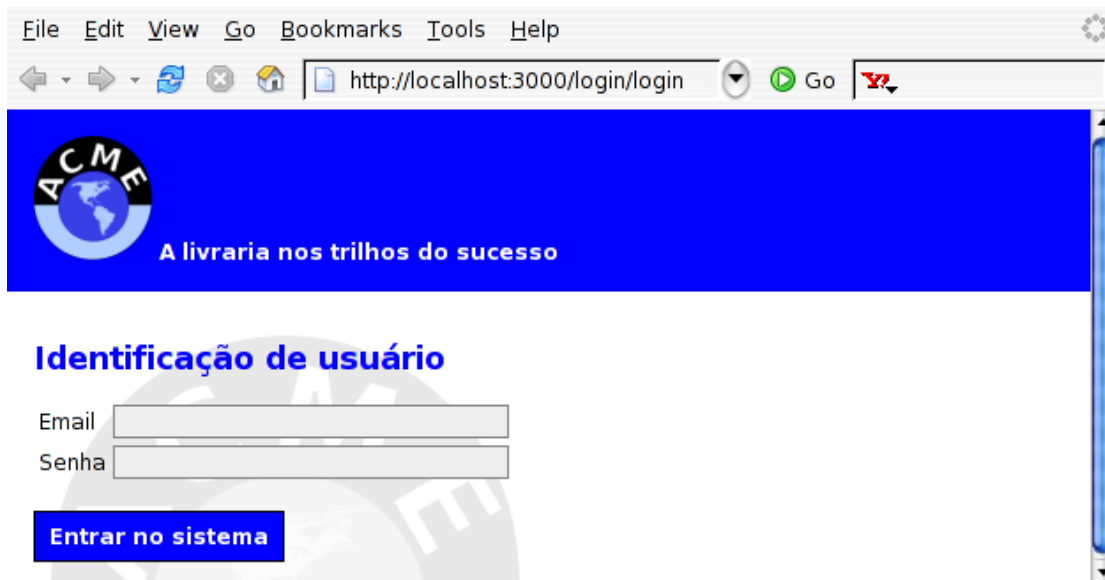


Ilustração 28: Tela de login de usuário na livraria

E criando agora o menu do site, de acordo com os tipos que cadastramos:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>Livraria ACME</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <%= stylesheet_link_tag "livraria" %>
  </head>
  <body>
    <div id="topo_livraria">
      <%= image_tag("acme.png") %>
      A livraria nos trilhos do sucesso!
    </div>
    <div id="menu_livraria">
      <ul>
        <% for tipo in Tipo.find_all %>
          <li><%= link_to tipo.descricao.pluralize, :controller => "produto", ↵
:action => "list", :id => tipo.descricao %></li>
        <% end %>
      </ul>
    </div>
    <%= @content_for_layout %>
  </body>
</html>

```

Olhem o menu (após ajustar o arquivo CSS praticamente inserindo as regras de **_livraria* nas regras do menu do *admin*:

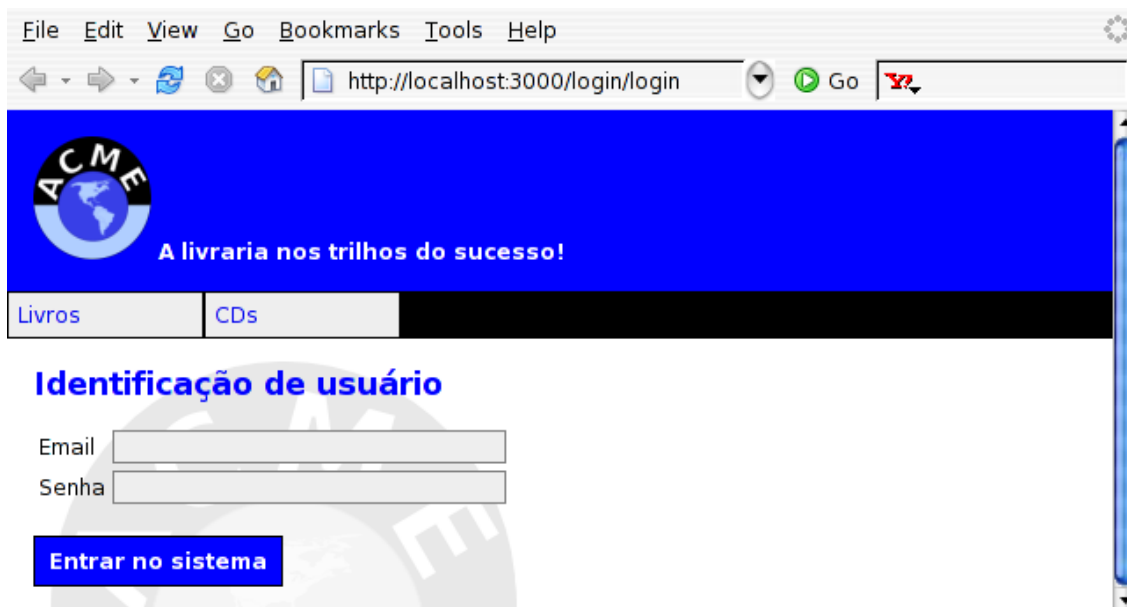


Ilustração 29: Tela de login de usuário na livraria, com menu

Ficou bonitinho, mas não esperem um clássico (ou mesmo uma coisa meio termo, vá lá) de design. Vamos fazer somente para quebrar um galho. :-)

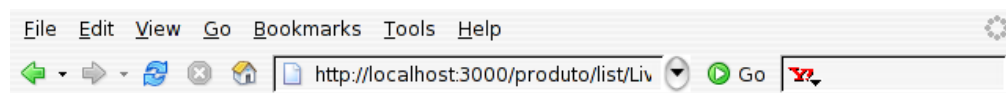
Uma coisa a ser notada é que eu utilizei o método **pluralize** na *descrição* do *tipo*. Esse método não é um método padrão de Strings do Ruby, e sim um que foi adicionado à todas as Strings processadas no Rails, através do Active Support e da tremenda flexibilidade do Ruby.

Criando a livraria “externa”

Vamos criar um controlador para *produto*:

```
[taq@var/www/htdocs/livraria]ruby script/generate controller Produto list
exists app/controllers/
exists app/helpers/
create app/views/produto
exists test/functional/
create app/controllers/produto_controller.rb
create test/functional/produto_controller_test.rb
```

Clicando em *Livros*:



Produto#list

Find me in app/views/produto/list.rhtml

Ilustração 30: Listagem de produtos

Ooops, esquecemos de personalizar as ações. A primeira coisa é alterar o controlador para que

ele use o layout *livraria*, depois personalizando `<aplicacao>/app/controllers/produto_controller.rb`. Vamos indicar que queremos usar o layout *livraria*:

```
class ProdutoController < ApplicationController
  layout "livraria"
```

e agora vamos configurar a ação *list*, primeiro verificando qual foi o *id* enviado no menu (reparem acima que eu especifiquei o *id* do menu como *tipo.descricao*, onde vai ser enviado, por exemplo, *Livro*), encontrando o *id* do tipo de acordo com a descrição na tabela *Tipo* e paginando os produtos de acordo com esse tipo:

```
def list
  tipo = Tipo.find(:first, :conditions => ["descricao = ?", params[:id]])
  @produto_pages, @produtos = paginate :produtos, :per_page => 10, <
:conditions => ["tipo_id = ?", tipo.id]
end
```

A ação *list* vai retornar páginas com 10 produtos cada, que tenham como *tipo_id* o valor encontrado como *id* do tipo (*Livro*, *CD*, etc.) enviado como parâmetro.

Também alterei a view *list* com o *id* enviado:

```
<div id="conteudo_livraria">
  <h1>Listando <%= params[:id] %>s</h1>
```

Olhem nossos produtos aparecendo (após cadastrar mais alguns – sintam-se a vontade para criarem quantos tipos, categorias e produtos quiserem):



Ilustração 31: Listando os livros



Ilustração 32: Listando os CDs

Vamos dar mais uma incrementada nessa página. Sabemos que os produtos tem categorias,

então vamos criar uma listagem de categorias por produto. A primeira coisa que precisamos fazer é providenciar uma listagem das categorias dos produtos do tipo requisitado. Temos duas opções:

1. Utilizar os objetos já criados no modelo e fazer um filtro dos seus resultados;
2. Utilizar uma consulta convencional no modelo relacional do banco de dados;

Fica a critério de cada um qual opção escolher. Vamos ver primeiro como ficaria a listagem das categorias já filtradas, alterando a view *list* (e o arquivo CSS, a gosto do freguês, também):

```
<div id="categorias">
  <h2>Categorias</h2>
  <ol>
    <% for categoria in @categorias %>
      <li><%= link_to categoria, :action => "list", :categoria => categoria
    %></li>
    <% end %>
  </ol>
</div>
<div id="conteudo_livraria">
  <h1>Listando <%= params[:id] %>s</h1>
```

Vamos atentar nesse momento somente o parâmetro **:categoria** que enviei no método **link_to**: ele vai criar uma URL com o tipo corrente e a categoria corrente. Vamos ver isso logo que verificarmos como ficou nossa tela:

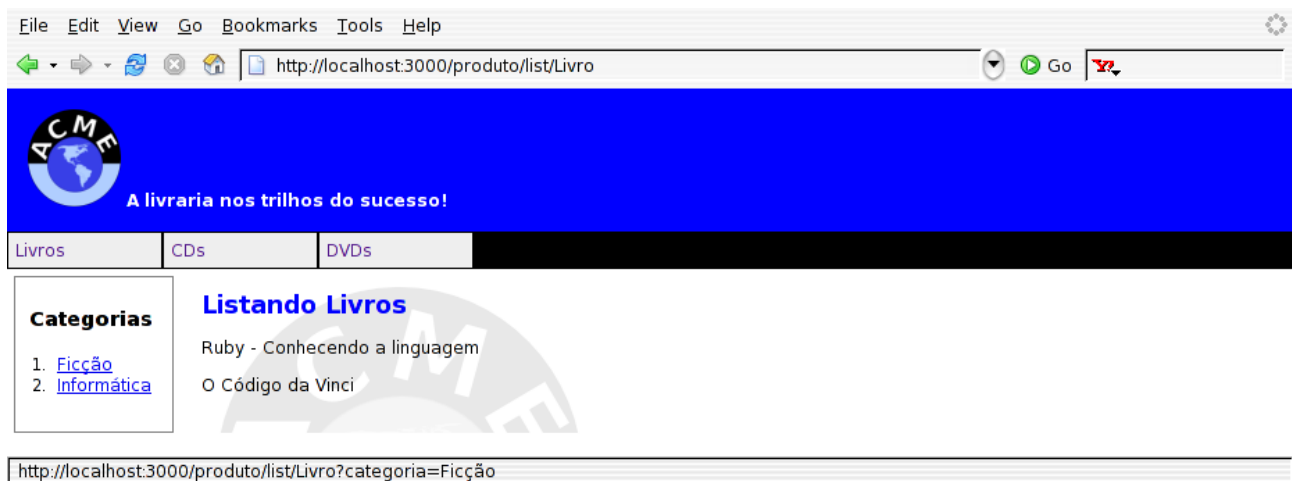


Ilustração 33: Listando produtos com categoria

Habilitei a barra de status no navegador, olhem o link como ficou:

<http://localhost:3000/produto/list/Livro?categoria=Ficção>

O que podemos reparar é que na listagem do *tipo*, estamos vendo os produtos daquele tipo e de todas as suas categorias. Vamos ver como filtrar isso logo, mas agora vamos ver como que filtramos as categorias.

Utilizando os objetos do modelo, vamos ter isso no controlador de *produto*:

```
def list
  tipo = Tipo.find(:first, :conditions => ["descricao = ?" <
  ",params[:id]]))
  @categorias = Produto.find(:all,:conditions => ["tipo_id = ?" <
  ",tipo.id]).map{ |e| e.categoria.descricao }.uniq.sort
  @produto_pages, @produtos = paginate :produtos, :per_page => 10, <
  :conditions => ["tipo_id = ?",tipo.id]
end
```

Destrinchando o que aconteceu ali, com uma aulinha rápida de Ruby (ei, [comprem o meu livro!](#) :-):

1. Produto.find trouxe uma lista de todos (:all) os produtos que correspondiam á condição especificada (tipo_id=<id>) em um array;
2. Utilizando *map*, selecionei os valores das *descrições* dos tipos;
3. Utilizando *uniq*, selecionei os valores únicos das descrições (sem repetir);
4. Utilizando *sort*, ordenei os valores;

O resultado disso tudo foi apresentado na view *list*, como mostrado acima.

A segunda opção é utilizar o método **find_by_sql**, consultando diretamente a base de dados (vou comentar o primeiro modo):

```
def list
  tipo = Tipo.find(:first, :conditions => ["descricao = ?",params[:id]])

  # @categorias = Produto.find(:all,:conditions => ["tipo_id = ? <-
",tipo.id]).map{ |e| e.categoria.descricao }.uniq.sort

  @categorias = Produto.find_by_sql ["select distinct a.descricao from <-
categorias a, produtos b where a.id=b.categoria_id and b.tipo_id=? order by
a.descricao",tipo.id]

  @produto_pages, @produtos = paginate :produtos, :per_page => 10,
:conditions => ["tipo_id = ?",tipo.id]
end
```

Vejam que eu enviei a query completa (eu podia ter usado uma *subquery* ali mas a versão do MySQL que tenho aqui não suporta isso) e um ponto bem importante, **são retornados objetos** com uma propriedade *descricao* e não mais *Strings* como no nosso primeiro modo. Então temos que alterar a view da *list* também:

```
<div id="categorias">
  <h2>Categorias</h2>
  <ol>
    <% for categoria in @categorias %>
      <li><%= link_to categoria.descricao, :action => "list", :categoria => categoria.descricao <-
%></li>
    <% end %>
  </ol>
</div>
<div id="conteudo_livraria">
  <h1>Listando <%= params[:id] %>s</h1>
  ...
```

Os resultados são similares em ambos os modos, vai do gosto do freguês e da quantidade de produtos versus velocidade de filtragem dos objetos.

Agora podemos limitar nossa listagem por categoria, se desejado. Vamos alterar o controlador de *produto*:

```
def list
  tipo = Tipo.find(:first, :conditions => ["descricao = ? <- ",params[:id]])
  categoria = Categoria.find(:first, :conditions => ["descricao = ? <-
",params[:categoria]])
  # @categorias = Produto.find(:all,:conditions => ["tipo_id = ? <-
",tipo.id]).map{ |e| e.categoria.descricao }.uniq.sort
  @categoria = params[:categoria]
  @categorias = Produto.find_by_sql ["select distinct a.descricao from <-
categorias a, produtos b where a.id=b.categoria_id and b.tipo_id=? order by <-
a.descricao",tipo.id]
  conditions = categoria.nil? ? ["tipo_id = ?",tipo.id] : ["tipo_id = ? <-
```

```

and categoria_id = ?",tipo.id,categoria.id]
  @produto_pages, @produtos = paginate :produtos, :per_page => 10, ←
  :conditions => conditions
end

```

Primeiro tentei retornar o id da categoria que talvez foi passada como parâmetro. Se não foi, retorna nulo. Aproveitei e salvei como uma variável de instância (leia o livro de Ruby ;-).

Depois verificando se houve ou não uma categoria como parâmetro, alterei a condição de paginação dos produtos e a view de *list*:

```

<div id="conteudo_livraria">
  <h1>Listando <%= params[:id] %>s</h1>
  <% if ! @categoria.nil? %>
  <h2><%= @categoria %></h2>
  <% end %>
  <% for produto in @produtos %>
  ...

```

Ou seja, se houve uma categoria especificada, vai ser mostrada ali na página. Os resultados são:

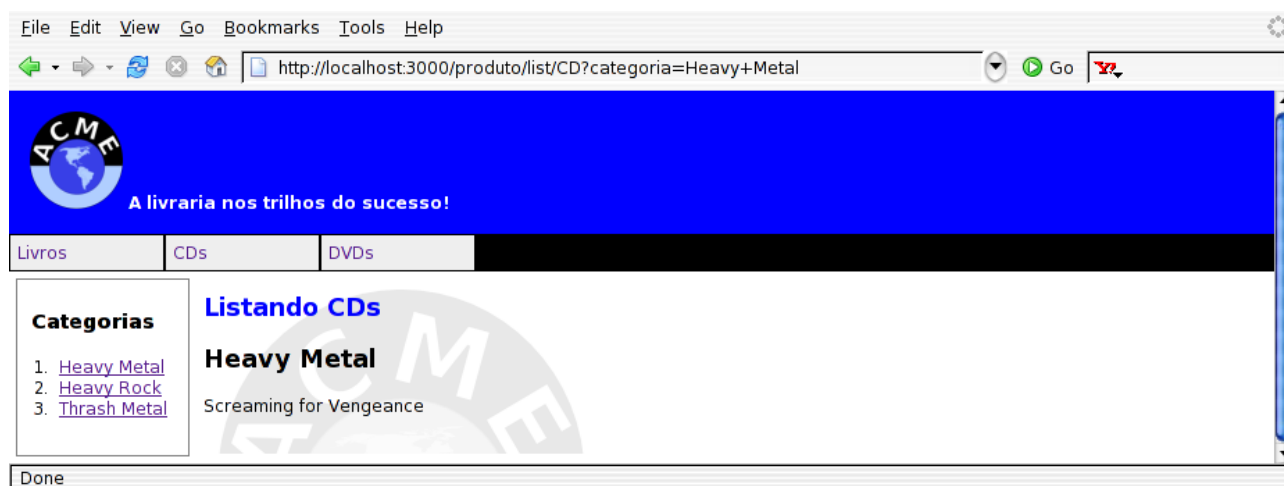


Ilustração 34: Listando os CDs de Heavy Metal

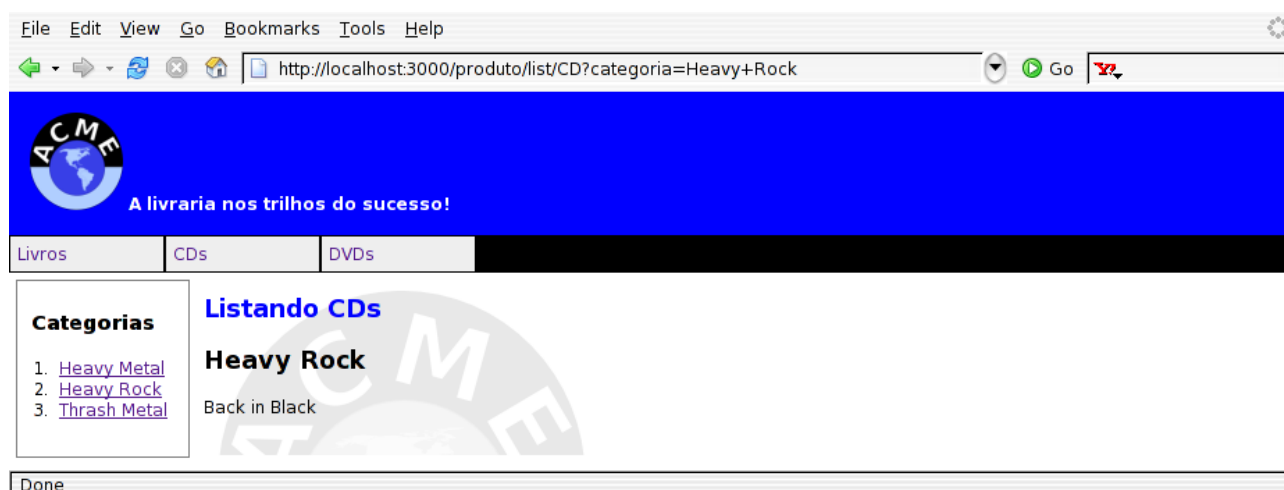


Ilustração 35: Listando os CDs de Heavy Rock

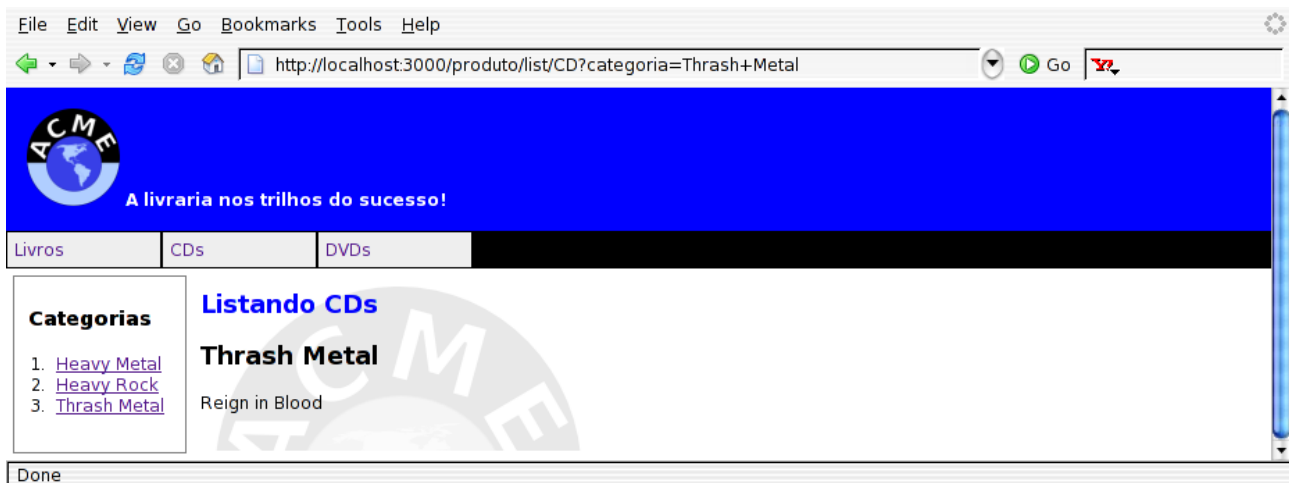


Ilustração 36: Listando os CDs de Thrash Metal

Melhorando nossas listagens

Agora temos que melhorar um pouco esses produtos, não? Que tal inserirmos o nome dos autores e uma imagem do produto? Pelo que aprendemos até agora fica fácil, e vamos alterar o banco de dados mesmo depois de ter criado o modelo. Vale lembrar que estamos fazendo uma coisa mais descompromissada para efeito didático. Os nomes dos autores ficariam melhor se armazenados em uma tabela de autores blá blá blá mas vocês entenderam né? :-)

```
[taq@~]mysql -u taq -p livraria_development
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 5.0.19

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> alter table produtos add autor varchar(75) not null default "";
Query OK, 5 rows affected (0,14 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> alter table produtos add img mediumblob;
Query OK, 5 rows affected (0,08 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

No caso do produto, estou supondo que vamos armazenar imagens maiores, então usei o tipo de dado **mediumblob** para o campo da imagem.

Inserindo no controlador *admin/produto* um método para retornar a imagem:

```
def img
  @produto = Produto.find(params[:id])
  send_data(@produto.img, :type=>"image/png", :disposition=>"inline")
end
```

Alterando a view *new* de *produto* para incluir o multipart:

```
<div id="conteudo">
  <h1>Novo produto</h1>
```



```
<%= form_tag({:action => 'create'}, :multipart => true) %>
```

Outra coisa para alterarmos, especificamente no caso de *produto*, é inserirmos os arrays de *tipo* e *categoria* na ação *update*:

```
def update
  @produto = Produto.find(params[:id])
  @tipos = Tipo.find_all.collect {|t| [t.descricao,t.id]}
  @categorias = Categoria.find_all.collect {|c| [c.descricao,c.id]}
  ...
end
```

A razão disso é que inserimos um método *validate* também no modelo, para aceitar apenas PNGs (sim, eu gosto de PNGs ;-):

```
protected
def validate
  if self.filename.length>0 and self.content_type !~ /^image\/png/
    errors.add(:img, "A imagem tem que ser um PNG! Foi enviado ←
#{self.content_type} (#{self.filename}).")
  end
end
```

Dando uma olhada na ação *update*:

```
if @produto.update_attributes(params[:produto])
  flash[:notice] = 'Produto foi atualizado com sucesso.'
  redirect_to :action => 'show', :id => @produto
else
  render :action => 'edit'
end
```

O que acontece é que, se houver algum problema na atualização, será executado o método *render* com a ação *edit*. Nesse ponto, a tela será recarregada, e se não constarem os arrays com as opções dos *selects*, vai ser gerado um erro, pois os *selects* de *tipo* e *categoria* não poderão ser montados com valores nulos. Apesar da ação *edit* constar com

```
def edit
  @produto = Produto.find(params[:id])
  @tipos = Tipo.find_all.collect {|t| [t.descricao,t.id]}
  @categorias = Categoria.find_all.collect {|c| [c.descricao,c.id]}
end
```

que cria os arrays, isso só funcionaria se ao invés de

```
render :action => 'edit'
```

tivéssemos

```
redirect_to :action => 'edit', :id => @produto
```

mas nesse caso, perderíamos as mensagens de erro da validação.

Alterando o modelo de *produto*, da mesma maneira que fizemos com o modelo de *usuario* :

```
public
def imagem=(img_field)
  self.content_type = img_field.content_type
  self.filename = img_field.original_filename
  self.img = img_field.read
end
```

Alterando o form de administração de *produto*:

```

...
<p><label for="produto_imagem">Imagem</label><br/>
<%= file_field("produto","imagem") %></p>

<!--[eoform:produto]-->

```

Alterando a view *show* de administração de *produto*:

```

<div id="conteudo">
  <p><b>Descrição:</b> <%=h @produto.descricao %></p>
  <p><b>Autor:</b> <%=h @produto.autor %></p>
  <p><b>Tipo:</b> <%=h @produto.tipo.descricao %></p>
  <p><b>Categoria:</b> <%=h @produto.categoria.descricao %></p>
  <p>
     @produto.id) %>"/>
  </p>
  <p>
    <%= link_to 'Editar', :action => 'edit', :id => @produto %> |
    <%= link_to 'Voltar', :action => 'list' %>
  </p>
</div>

```

Alterando a view *list* de administração de *produto*:

```

<table>
<% for produto in @produtos %>
  <tr>
    <td> produto.id) %>"/></td>
    <td valign="top">
      <b><%=h produto.descricao %></b><br/>
      <%=h produto.autor %><br/>
      <%=h produto.tipo.descricao %><br/>
      <%=h produto.categoria.descricao %><br/>
      <%= link_to 'Mostrar', :action => 'show', :id => produto %>
      <%= link_to 'Editar', :action => 'edit', :id => produto %>
      <%= link_to 'Apagar', { :action => 'destroy', :id => produto },
      :confirm => 'Tem certeza?', :post => true %>
    </td>
  </tr>
</table>

```

Isso vai nos dar:



Ilustração 37: Listando produtos com imagens na tela de administração

Na listagem externa de produtos, vamos ter agora:

```
...
<div id="conteudo_livraria">
  <h1>Listando <%= params[:id] %>s</h1>
  <% if ! @categoria.nil? %>
    <h2><%= @categoria %></h2>
  <% end %>
  <% for produto in @produtos %>
    <p>
       produto.id) %>"/>
      <b><%= produto.descricao %></b><br/>
      <i><%= produto.autor %></i><br/>
      <%= link_to "Ampliar a capa", {:action => "img", :id => produto.id},
      :popup => true %>
    </p>
    <hr style="clear:both;"/>
  <% end %>
  <p><%= pagination_links(@produto_pages) %>
</div>
```

Que vai resultar em:

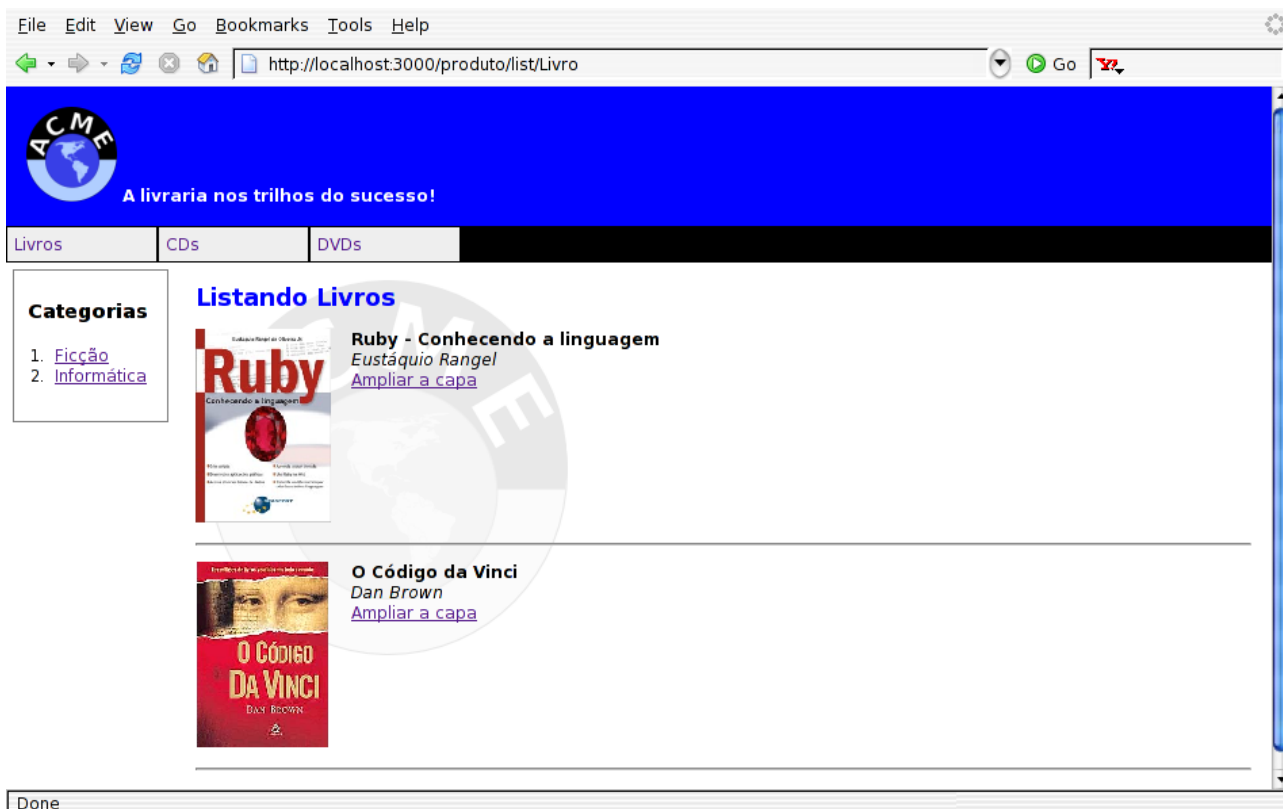


Ilustração 38: Listagem externa com categorias e figuras

Listando todos os CDs de *Thrash Metal*, clicando em CDs e nessa categoria:

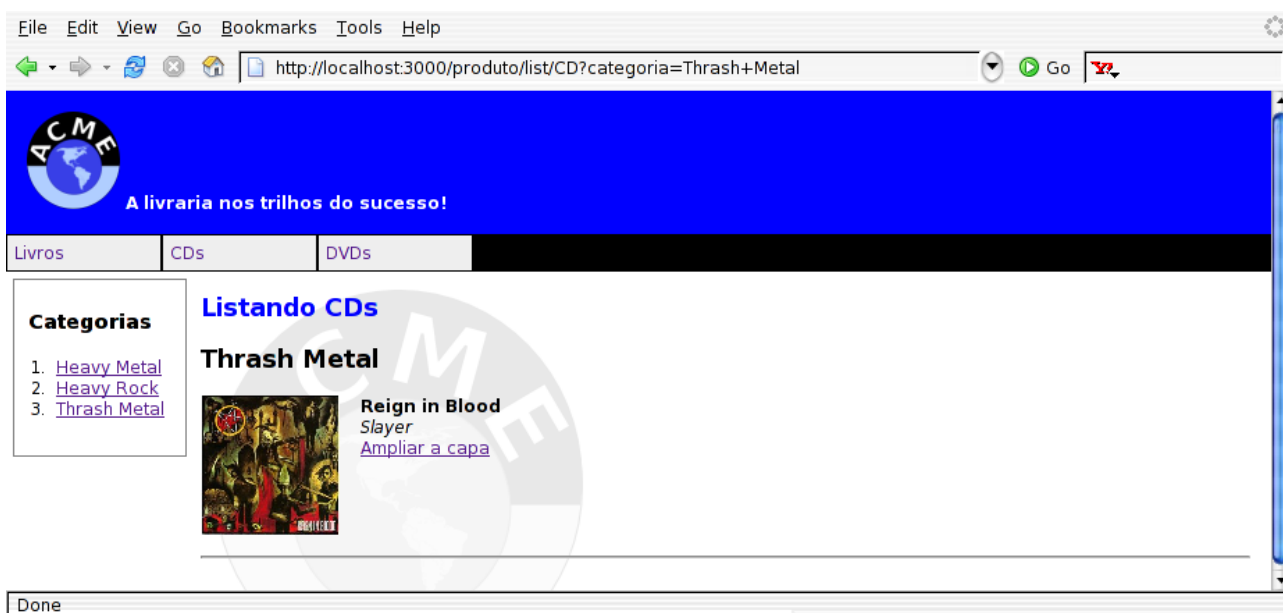


Ilustração 39: Listando apenas uma categoria

A novidade ficou por conta do link para ampliar a capa, que foi destacado em vermelho na listagem da view *list*.

Vamos agora alterar a ordem por *autor* e *descrição* alterando o controlador:

```
def list
  ...
  @produto_pages, @produtos = paginate :produtos, :per_page => 10, ←
  :conditions => conditions, :order => "autor,descricao"
```

end

Cadastrei alguns produtos novos, e dando uma nova verificada:

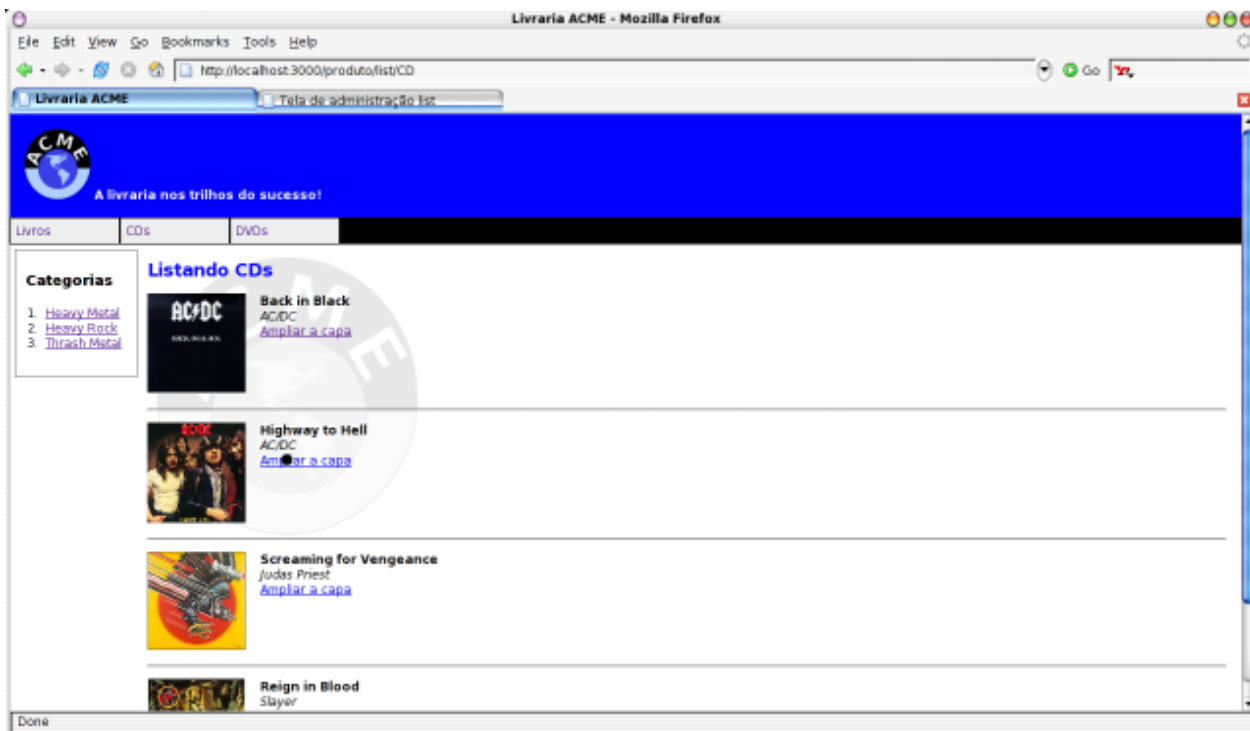


Ilustração 40: Listagem por tipo em ordem alfabética

Uma coisa a ser arrumada já. Vamos supor que acessamos a URL

<http://localhost:3000/produto/list>

vamos ter algo parecido com isso:

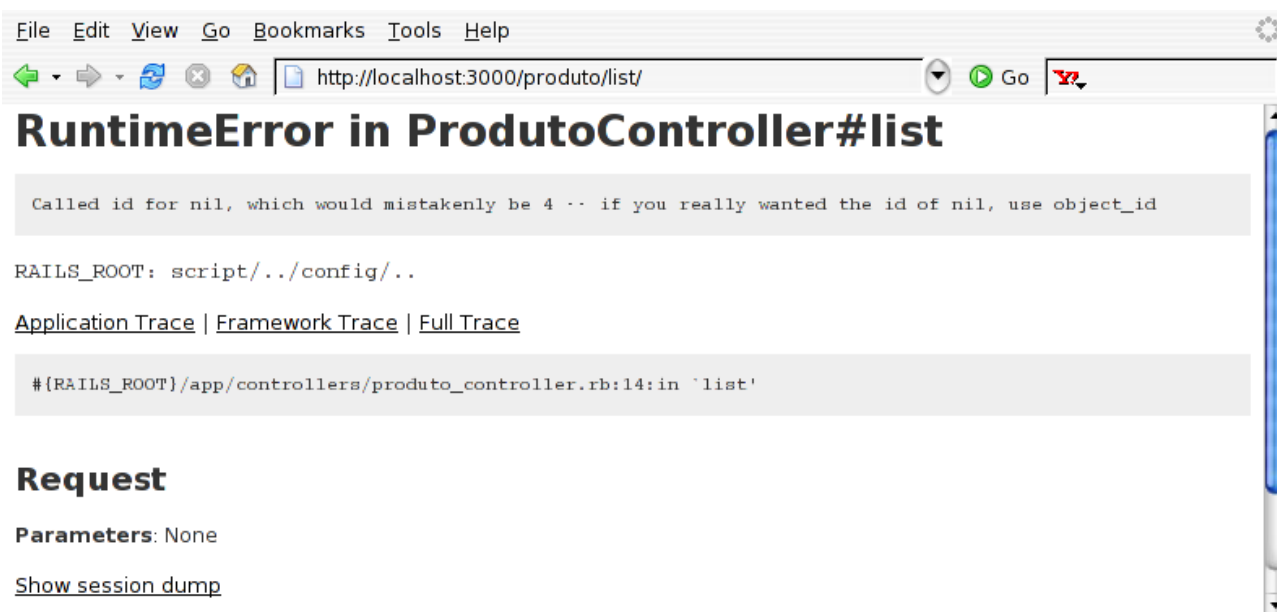


Ilustração 41: Erro na listagem sem id

pois não foi especificado nenhum id para listar. Para especificar um default, podemos usar no controlador:

```
def list
  params[:id] ||= "Livro"
```

```

    tipo = Tipo.find(:first, :conditions => ["descricao = ?" <
",params[:id]])
    ...

```

Isso vai redirecionar automaticamente para a seção de livros. Podemos dar uma monitorada nas vezes que o usuário acessou a URL sem o *id*, usando o recurso dos *loggers*, que vão gravar mensagens que apontamos no log do sistema. É muito fácil usar esse tipo de recurso, vamos alterar a ação *list* novamente:

```

def list
  logger.warn("Tentativa de listar produtos sem tipo definido") if <
params[:id].nil?
  params[:id] ||= "Livro"

```

Se dermos uma olhada em `<aplicacao>/log/development.log` (lembrando que estamos vendo o *development* pois é a opção que estamos rodando o servidor, poderia ser o *test* ou *production*), vamos ter, perdido lá no meio:

```

Processing ProdutoController#list (for 127.0.0.1 at 2006-04-22 10:11:59)
[GET]
  Session ID: de27ea7a95e42ec73a8e3ee22a74e7be
  Parameters: {"action"=>"list", "controller"=>"produto"}
  Tentativa de listar produtos sem tipo definido
  Tipo Load (0.000720)  SELECT * FROM tipos WHERE (descricao = 'Livro')
  LIMIT 1

```

Podemos utilizar também **logger.info**, **logger.error** e **logger.fatal**.

E o que acontece se acessarmos

<http://localhost:3000>

Ah-há, vamos dar de cara com a página padrão do Rails:

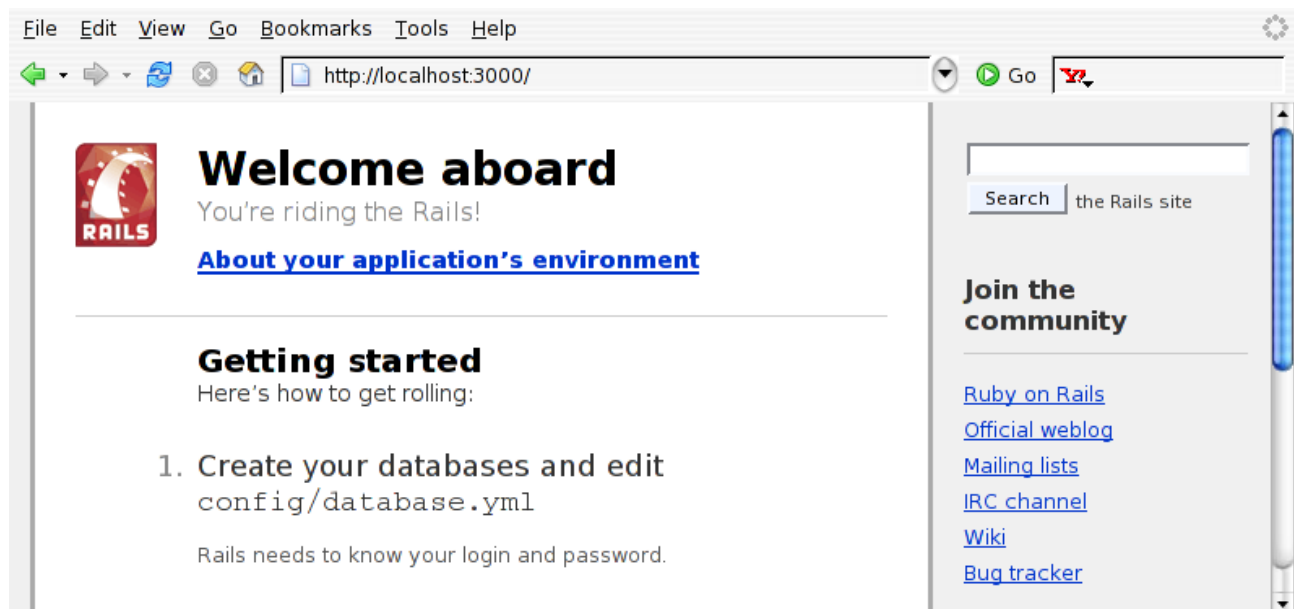


Ilustração 42: Tela inicial padrão

Para alterarmos isso, primeiro temos que apagar o arquivo *index.html* no diretório `<aplicação>/public` e inserir em `<aplicacao>/config/routes.rb`:

```

map.connect '', :controller => "produto", :action => "list", :id => "Livro"

```

Com isso estamos redirecionando a URL root para a ação *list* do controlador *produto*.

Pesquisando

Vamos implementar um campo de pesquisa agora na nossa livraria. Dei uma garibada no layout de livraria:

```
<div id="topo_livraria">
  <table width="100%">
    <tr>
      <td width="100px;"><%= image_tag("acme.png") %></td>
      <td><h1>A livraria nos trilhos do sucesso!</h1></td>
    </tr>
    <tr>
      <td colspan="2" align="right">
        <%= start_form_tag :action => "search" %>
        <label for="busca_str">Buscar</label>
        <%= text_field "busca", "str" %>
        <%= end_form_tag %>
      </td>
    </tr>
  </table>
</div>
```

Isso nos dá um campo texto no topo do site:



Ilustração 43: Campo de busca

Agora vamos inserir um método *search* no controlador de *produto*:

```
def search
  @term      = params[:busca][:str]
  @tipos     = Tipo.find(:all).map{|e| e.descricao}.uniq
  @produtos = Produto.find(:all, :conditions => ["descricao like ? ←", "%#{@term}%"], :order => "tipo_id,categoria_id,descricao")
end
```

O conteúdo do campo texto do formulário criado no topo do site será salvo em **@term**. Também criamos uma lista de tipos de produtos disponíveis no site, para apresentar do lado esquerdo da busca (para que replicar o menu logo acima não sei, mas já fica como exercício descobrir alguma coisa de útil para por ali e não ficar vazio ;-)) e em seguida procuramos todos os produtos que “casam” com o que procuramos no campo texto. Vejam que eu usei os *bindings* de variáveis mas fiz uma interpolação de expressão (aquele negócio com os `#{ }`) usando o termo.

Também devemos criar uma view de *search* para que possa interpretar os resultados que acabamos de criar no controlador. Inserir isso na view:

```
<div id="categorias">
  <h2>Produtos</h2>
  <ul>
    <% for tipo in @tipos %>
    <li><%= link_to tipo.pluralize, :action => "list" %></li>
    <% end %>
  </ul>
</div>
<div id="conteudo_livraria">
  <h1>Busca de produtos - procurando '<%= @term %>'
```

E obtive, após procurar “*ruby*”:



Ilustração 44: Resultado da busca

Renderizações parciais

Podemos notar uma certa repetição ali, mostrando o produto. Vamos fazer o seguinte, retirar aquele *for* e aprender sobre *partials*. O que seria isso? Podemos pensar como um desvio de renderização da página corrente para outra página, com os parâmetros que quisermos enviar. No caso do nosso produto, primeiro temos que definir um arquivo com o conteúdo do que será renderizado. Os *templates* parciais tem que começar com um sublinhado (*_*) e ficar abaixo de

<aplicação>/app/views.

No nosso caso, vamos definir <aplicacao>/app/views/produto/_produto.rhtml:

```
<p>
   produto.id) %>" />
  <b><%= produto.descricao %></b><br/>
  <i><%= produto.autor %></i><br/>
  <%= link_to "Ampliar a capa", {:action => "img", :id => produto.id}, ←
:popup => true %>
</p>
<hr style="clear:both;" />
```

Prestando atenção, vemos que é o código que estava dentro do *for* na *list*, e podemos inclusive eliminar todo o código do *for* da *view search* que implementamos acima!

Vamos dar uma olhada como ficou o <aplicacao>/app/views/produto/list.rhtml agora:

```
<div id="categorias">
  <h2>Categorias</h2>
  <ol>
    <% for categoria in @categorias %>
      <li><%= link_to categoria.descricao, :action => "list", :categoria => ←
categoria.descricao %></li>
    <% end %>
  </ol>
</div>
<div id="conteudo_livraria">
  <h1>Listando <%= params[:id] %>s</h1>
  <% if ! @categoria.nil? %>
    <h2><%= @categoria %></h2>
  <% end %>
  <%= render(:partial => "produto", :collection => @produtos) %>
  <p><%= pagination_links(@produto_pages) %></p>
</div>
```

Uau! Bem mais enxuto hein? O que acontece é que o **render** usa como parâmetro o *partial* que vai renderizar (sem o sublinhado) e armazena o valor de cada item da coleção de objetos (indicada por *:collection*) que enviamos em *@produtos* em uma variável com o nome do template (indicado em *:partial*) em chamada *produto* também.

Para mostrar como podemos usar o *partial* com apenas um objeto, vamos primeiro fazer algumas alterações nos nossos produtos. Vamos inserir um campo na tabela do banco de dados chamado *detalhes*, onde vamos dar um pouco mais de detalhes sobre o produto:

```
alter table produtos add detalhes text
```

Isso vai nos dar um campo de tamanho considerável para uma boa descrição (65535 caracteres).

Agora vamos alterar <aplicacao>/app/view/admin/produto/_form.rhtml para contemplar o campo novo, vamos criá-lo como uma área de texto com 100 colunas e 5 linhas:

```
<p><label for="produto_detalhes">Detalhes</label><br/>
<%= text_area "produto", "detalhes", :cols => "100", :rows=>"5" %>
```

Vamos inserir alguns dados ali:

Editando produto

Descricao	Reign in Blood
Autor	Slayer
Tipo	CD
Categoria	Thrash Metal
Detalhes	
<p>Esse foi um dos discos de maior influência na década de 80, e com certeza é o grande clássico do Slayer.</p> <p>Petardos como "Angel of Death", "Raining Blood" e "Postmortem" são tocados até hoje, mais de 20 anos depois, mantendo todo o peso e a agressividade original com que foram criados.</p> <p>A formação da época (1986) é a formação clássica do Slayer: Tom Araya no baixo e vocal, Jeff Hanneman e Kerry King nas guitarras e Dave Lombardo na bateria.</p>	

Ilustração 45: Inserindo detalhes no produto

Agora precisamos de um jeito de ver isso no site externo. Podemos configurar a ação *show* no nosso controlador de produto:

```
def show
  @produto = Produto.find(params[:id])
end
```

Agora vamos alterar a view do *partial*:

```
<%= link_to "Ampliar a capa", {:action => "img" , :id => produto.id}, :
:popup => true %><br/>
<%= link_to "Mais detalhes" , {:action => "show", :id => produto.id} %>
```

E finalmente alterando a view *show*. Nesse caso, poderíamos até já deixar essa view responsável por mostrar o produto, mas fica mais prático criar uma *partial* que pode ser renderizada em qualquer outro ponto que necessitamos, ajudando no DRY¹⁰ (apesar de falar do DRY, não me preocupei muito com isso em certos pontos por achar que para efeito didático às vezes é melhor fazer uma “lambança” em certos pontos).

Vamos criar uma *partial* chamada *_produtodetalhe.rhtml*:

```
<p>
   produtodetalhe.id) %>"/>
  <b><%= produtodetalhe.descricao %></b><br/>
  <i><%= produtodetalhe.autor %></i><br/>
  <%= link_to "Ampliar a capa", {:action => "img" , :id =>
produtodetalhe.id}, :popup => true %>
  <br style="clear:both;"/>
  <%= produtodetalhe.detalhes.nil? ? "" :
produtodetalhe.detalhes.split("\n").map {|v| v =~ /^<li>/ ?
v : v+<br/>} .join %>
</p>
```

Aqui fiz uma jogadinha interessante: quando cadastramos os detalhes do produto, a quebra de linha é armazenada no banco de dados com o caracter `\n`, que não serve para nada em uma

¹⁰ Don't Repeat Yourself, ou seja, não faça a mesma coisa mais de uma vez gerando retrabalho e redundância. Não é aplicável em situações, tipo, na sua lua-de-mel.

página HTML, tendo que ser substituído por uma tag **
** para que a linha seja quebrada. Só que no caso do produto que cadastrei, usei tags HTML para negrito (****) em algumas partes e uma lista ordenada (****) para as músicas. Como cada item da lista (****) já gera uma quebra de linha automática, eu tive que substituir o caracter **\n** com **
** apenas nas linhas que não começam com ****. O que fiz foi:

1. Primeiro, verificar se tem conteúdo nos detalhes, se não tiver, não imprime nada.
2. Criar um array “quebrando” a string do detalhe sempre que encontrar um **\n** (`detalhes.split("\n")`);
3. Usar *map* para processar todos os elementos do Array e retornar um Array novo;
4. Se o elemento corrente iniciar com ****, fica sem alteração;
5. Se não, retorna o elemento com um **
** no final;
6. Usamos o método **join** para transformar o Array em uma String.

E agora alterar a view *show* para chamar o *partial*:

```
<div id="conteudo_livraria">
  <h1>Detalhes do produto</h1>
  <%= render(:partial => "produtodetalhe", :object => @produto) %>
</div>
```

Ah-há, aqui vemos que ao invés de *:collection*, como visto anteriormente quando tínhamos vários objetos, passamos o objeto retornado pela ação *show* e o enviamos para a *partial* com *:object*.

Agora, e se precisarmos dessa *partial* fora de *produto*, vamos imaginar, em uma ação onde se lista os produtos no carrinho de compras? Para isso, podemos criar um diretório chamado, digamos, *partials*, abaixo de *<aplicação>/app/views*, movemos *_produtodetalhe.rhtml* para lá e alteramos a view *show* que acabamos de mostrar para:

```
<div id="conteudo_livraria">
  <h1>Detalhes do produto</h1>
  <%= render(:partial => "partials/produtodetalhe", :object => @produto) %>
</div>
```

Se a *partial* vê que foi passado um *path* (identificando pela presença de uma barra), procura automaticamente abaixo de *<aplicação>/app/view* o diretório e a *partial* especificada. Assim podemos criar *partials* comuns para toda a aplicação. Isso ajuda no DRY também. Podemos mover o nosso *partial _produtos.rhtml* para esse diretório também e alterar as ações *list* e *search*.

Uma *screenshot* da tela de detalhes do produto:

Detalhes do produto



Reign in Blood
Slayer
[Ampliar a capa](#)

Esse foi um dos discos de maior influência na década de 80, e com certeza é o grande clássico do Slayer. Petardos como "Angel of Death", "Raining Blood" e "Postmortem" são tocados até hoje, mais de 20 anos depois, mantendo todo o peso e a agressividade original com que foram criados. A formação da época (1986) é a formação clássica do Slayer: **Tom Araya** no baixo e vocal, **Jeff Hanneman** e **Kerry King** nas guitarras e **Dave Lombardo** na bateria.

Músicas

1. Angel of Death
2. Piece by Piece
3. Necrophobic
4. Altar of Sacrifice
5. Jesus Saves
6. Criminally Insane
7. Reborn
8. Epidemic
9. Postmortem
10. Raining Blood

Ilustração 46: Detalhes do produto

Renderizando parcialmente com Ajax

Não, não é aquele produto de limpeza que a sua mãe usava. Se você ficou fora do planeta em 2005 ou por acaso não leu nenhum site de tecnologia e não sabe o que é Ajax, a grosso modo é a designação para Asynchronous Javascript and XML, uma técnica que permite desenvolver aplicações interativas que alteram elementos de uma página sem precisar recarregar a mesma usando a tradicional metodologia de envio dos dados através de um POST. Isso permite que atualizemos apenas alguns determinados elementos de nossa página através de requisições para scripts externos.

Para um exemplo fácil, vamos imaginar que os responsáveis pelo nosso site podem inserir comentários nos produtos (não vamos deixar os usuários fazerem isso a não ser que tenhamos pessoal para revisar as barbaridades que eles podem escrever lá, “*kd vc kr me manda um scrap blz vlw fuizzzzzz*, argh” certo?) e queremos que esses comentários sejam mostrados quando o produto for visualizado, através do click em um link. Antes de mais nada precisamos da estrutura para manipular nossos comentários, através de uma tabela como:

```
create table comentarios (
  id          int auto_increment,
  produto_id  int not null,
  data        date,
  comentario  varchar(250) not null,
  primary key(id)
);
```

Vamos usar um campo data para marcar quando o comentário foi inserido e ordenar em ordem decrescente. Toda a estrutura de manutenção eu deixo por conta de vocês – basta seguir o que fizemos com *tipo* e *categoria*. Só lembre de especificar as relações, em produto:

```
class Produto < ActiveRecord::Base
  has_many :comentarios
  ...
```

e comentário:

```
class Comentario < ActiveRecord::Base
  belongs_to :produto
end
```

Isso vai permitir que os comentários sejam recuperados com *Produto.comentarios*.

Logo após criar toda a estrutura CRUD dos comentários, vamos criar uma ação no controlador do *Produto* chamada **comentários**:

```
def comentarios
  @produto = Produto.find(params[:id])
  render(:partial => "partials/produtocomentarios", :layout => false, :object
=> @produto)
end
```

Uma parte muito importante ali: estou pedindo que seja renderizado **sem** a aplicação do layout. Faz sentido, se lembrarmos que estamos atualizando somente uma parte da tela e não precisamos de todo o código que é gerado no layout, pois o nosso elemento ficaria praticamente com toda a estrutura de uma página completa se deixássemos o layout!

Vamos dar uma olhada no nosso *partial* dos comentários:

```
<b>Comentário(s)</b><br/>
<% for comentario in produtocomentarios.comentarios.sort_by{|o|
o.data}.reverse %>
"<%= comentario.comentario %>"<br/>
<% end %>
<% if produtocomentarios.comentarios.size < 1 %>
<i>Sem comentários cadastrados.</i>
```

```
<% end %>
```

Um pequeno exercício com Ruby que fiz, em destaque em vermelho, foi ordenar os comentários pela sua data (com **sort_by**) e inverter a ordem do array gerado (com **reverse**).

E agora no *partial* do *Produto*, onde vamos inserir um link para que sejam visualizados os comentários do produto em questão:

```
<p>
   produto.id) %>"/>
  <b><%= produto.descricao %></b><br/>
  <i><%= produto.autor %></i><br/>
  <%= link_to "Ampliar a capa", {:action => "img" , :id => produto.id},
:popup => true %><br/>
  <%= link_to "Mais detalhes" , {:action => "show", :id => produto.id} %>
  <p id="comentario_<%= produto.id %>">
    <%= link_to_remote "Clique aqui para comentários", :update => ←
"comentario_#{produto.id}", :complete => "new
Effect.Highlight('comentario_#{produto.id}')" , :url => {:action => ←
:comentarios, :id => produto.id} %>
  </p>
</p>
<hr style="clear:both;"/>
```

Vamos dar uma analisada ali. Eu criei um novo parágrafo (<p>) dentro do parágrafo do produto, com um id chamado “comentário_” concatenado com o id do produto (ou seja, *comentario_1*, *comentario_2*, etc.).

Dentro do parágrafo criei um link com **link_to_remote** com o texto “Clique aqui para comentários”, indiquei que o elemento que vai ser atualizado quando clicarmos no link é o que acabamos de criar acima (comentario_<id>), e fiz uma firulinha, indiquei que quando o conteúdo tiver sido completamente atualizado devemos acionar um dos efeitos de apresentação disponibilizados pelo Rails, nesse caso, **Effect.Highlight(<elemento>)**, que vai mostrar o elemento indo de uma coloração amarela para branca.

Temos mais efeitos, segue uma lista descritiva:

- **Effect.Appear** - Mostra gradativamente o elemento.
- **Effect.Fade** - Apaga gradativamente o elemento.
- **Effect.Puff** - Apaga o elemento através de uma animação.
- **Effect.BlindDown** - Efeito “persiana abaixo”.
- **Effect.BlindUp** - Efeito “persiana acima”.
- **Effect.SwitchOff** - Apaga o elemento “dobrando” o mesmo.
- **Effect.SlideDown** - Desliza o elemento para baixo.
- **Effect.SlideUp** - Desliza o elemento para cima.
- **Effect.DropOut** - Apaga o elemento deslizando para baixo.
- **Effect.Shake** - Dá uma “tremida” no elemento.
- **Effect.Pulsate** - Faz o elemento “pulsar”, sumindo e aparecendo gradativamente.
- **Effect.Squish** - Apaga o elemento reduzindo seu tamanho até sumir.
- **Effect.Fold** - Apaga o elemento deslizando para cima e á esquerda.
- **Effect.Grow** - Mostra o elemento aumentando o seu tamanho.
- **Effect.Shrink** - Apaga o elemento reduzindo o seu tamanho.
- **Effect.Highlight** - Altera o background de amarelo para branco gradativamente.

Mas e aí, onde vamos pegar o conteúdo para atualizar toda essa estrutura que fizemos aí em cima? Vamos pegar através do que foi indicado por **:url**, nesse caso a ação (:action) **comentarios** (no mesmo controlador que estamos), enviando o id do produto atual com **:id** (produto.id).

Mas calma! Antes de testar o que fizemos, temos que inserir o código milagroso que vai fazer

nossas requisições do Ajax e os efeitos do Javascript. Vamos alterar o layout da livraria:

```
<head>
  <title>Livraria ACME</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <%= stylesheet_link_tag "livraria" %>
  <%= javascript_include_tag "prototype", "effects" %>
</head>
```

Agora sim. Vamos dar uma olhada na listagem dos produtos:



Ilustração 47: Link para comentários

Olhem os links para os comentários! Vamos clicar em algum (após inserir algum comentário no produto, lógico) e teremos:



Ilustração 48: Comentários atualizados

Agora o último detalhe: vamos inserir o *partial* dos comentários dentro do *partial* dos detalhes do produto:

```
<p>
   produtodetalhe.id) %>"/>
  <b><%= produtodetalhe.descricao %></b><br/>
  <i><%= produtodetalhe.autor %></i><br/>
  <%= link_to "Ampliar a capa", {:action => "img" , :id => ←
produtodetalhe.id}, :popup => true %>
  <br style="clear:both;" />
  <%= produtodetalhe.detalhes.nil? ? "" :
produtodetalhe.detalhes.split("\n").map {|v| v =~ /^<li>/ ? v : v+"<br/>" ←
}.join %>
  <%= render(:partial => "partials/produtocomentarios", :object => ←
produtodetalhe) %>
</p>
```

Parecem blocquinhos de LEGO™ né? Dando uma olhada em como ficaram os detalhes do produto:

Detalhes do produto



Back in Black
AC/DC
[Ampliar a capa](#)

Um dos grandes clássicos do AC/DC, "Back in Black" foi o primeiro registro com Brian Johnson, que substituiu o grande Bon Scott nos vocais, após a sua trágica morte. Esse CD é repleto de clássicos como "Back in Black", "You Shook Me All Night Long" e "Hell's Bells", onde Brian mostrou foi a escolha perfeita para a banda.

Formação: Brian Johnson (vocalis), Angus Young (guitarra), Malcolm Young (guitarra), Cliff Williams (baixo), Phil Rudd (bateria).

Músicas

1. Hells Bells
2. Shoot To Thrill
3. What Do You Do For Money Honey
4. Given The Dog A Bone
5. Let Me Put My Love Into You
6. Back In Black
7. You Shook Me All Night
8. Have A Drink On Me
9. Shake A Leg
10. Rock And Roll Ain't Noise Pollution

Comentário(s)

"Um dos grandes albuns da década de 80!"

"Recomendado! Um dos grandes clássicos do AC/DC"

Ilustração 49: Detalhes do produto já com os comentários

Olhem os comentários ali no final dos detalhes.

Usamos o Ajax para alterar dinamicamente um elemento da nossa página e ainda criamos mais uma “pecinha” para a visualização do nosso produto. Bom. :-)

Finalizando

Lógico que em uma livraria virtual real precisamos de muito mais coisas: carrinho de compras, opções de pagamento, cálculo de frete, etc. etc. etc. Isso vai um pouco além do escopo desse tutorial (uau, já chegamos em mais de 50 páginas), mas espero que vocês tenham aproveitado o que foi apresentado até aqui para ter uma idéia (boa, espero ;-) do que o Rails (e Ruby!) pode fazer por vocês. A partir do ponto em que paramos vocês podem criar todas as *features* que eu mencionei acima e muitas outras mais, além de fazer melhorias no layout e no CSS do que foi apresentado, pois fiz meio rapidinho para poder escrever mais. :-)

É isso, obrigado por me acompanhar até aqui. ;-)

Índice alfabético

A	
Active Record.....	10, 11
Ajax.....	61, 63
Apache.....	8
Application.rb.....	40
B	
Banco de dados.....	6, 7, 8, 9, 10, 28, 29, 31, 37, 45, 48
Belongs_to.....	20, 21, 23
Bindings de variáveis.....	4, 55
C	
Charset.....	13
Check_box.....	24, 25
Controlador.....	9, 11, 12, 14, 15, 16, 19, 20, 21, 28, 30, 36, 38, 40, 41, 43, 45, 46, 48, 52, 53, 54, 55, 56
CRUD.....	61
D	
Download.....	6
DRY.....	58, 59
E	
Edit.rhtml.....	20
Effect.....	62
F	
File_field.....	24, 25, 28, 50
Find_by_sql.....	11, 46
Foreign keys.....	7, 20
G	
GNU/Linux.....	5, 6
H	
H.....	14
Has_many.....	20, 23
Hash.....	15
Html_escape.....	14
J	
Javascript.....	61, 63
L	
Layout.....	11, 12, 13, 16, 36, 37, 38, 39, 41, 42, 44
Lighttpd.....	8
Link_to.....	15, 24, 27, 30, 35, 39, 40, 41, 42, 45, 46, 50, 51
List.rhtml.....	13, 16, 27
Livro.....	1, 4, 5, 19, 44, 45, 46, 47, 53, 54
Loggers.....	54
M	
Métodos de procura dinâmicos.....	11
MIME.....	30, 31
Modelo.....	4, 9, 10, 11, 17, 20, 24, 28, 29, 31, 33, 45, 48, 49
Multipart.....	24, 25, 35, 48, 49

MVC.....	4
Mysql.....	6, 7, 8, 48
MySQL.....	6, 46, 48
N	
New.rhtml.....	14, 17, 24
O	
ORM.....	9, 10
P	
Partial.....	15, 35, 57, 58, 59, 61, 62, 63, 64
Partials.....	56, 59
Password_field.....	24, 25, 36
Pluralize.....	42, 43, 56
R	
Render.....	12, 14, 15, 35, 38, 49, 57, 59
Request.get?.....	37
Reverse.....	61, 62
Ruby.....	1, 4, 5, 6, 8, 9, 10, 11, 16, 20, 36, 38, 43, 46, 47, 56, 65
S	
Scaffold.....	8, 12, 15, 16, 20, 21, 24
Session.....	37, 38, 39, 40, 41
Show.rhtml.....	20
Sort_by.....	61, 62
SQL Injection.....	38
T	
Text_field.....	22, 24, 36, 55
U	
Upload.....	25, 30, 31
Url_to.....	41
V	
Visualizadores.....	12
W	
Webrick.....	13
WEBrick.....	8, 9
_form.rhtml.....	21, 23, 24
.rhtml.....	28