



APRENDENDO RUBY ON RAILS

Índice

O que é Ruby on Rails.	4
A linguagem Ruby.	4
A quem se destina este livro.	5
Configuração e instalação.	6
Instalação em Windows.	7
Instalação em Mac OS.	8
Instalação em Linux.	9
Iniciando em Rails.	10
Desenvolvendo uma Aplicação completa.	10
Configurando o banco de dados.	13
MVC – Models, Views e Controllers.	16
O primeiro controller.	18
A primeira View.	21
Incrementando as Views com uso de layouts.	23
Models.	29
Gerando migrações.	30
Rake.	32
Cadastro manual no banco de dados.	36
Primeira listagem de dados do banco	39
REST.	46
O que é REST.	46
map.resources	48
Relacionamento entre models	51
Inserção, edição e exclusão de registros.	54
Helpers para formulários (form_for).	55
Inserção de dados.	59

Edição de dados.	63
Exclusão de dados	68
Validações.	72
Scaffolding.	79
Entendendo os arquivos gerados pelo Scaffold: O controller	
Categorias	82
Entendendo os arquivos gerados pelo Scaffold: As Views	85
Avançando no Rails.	90
Roteamento padrão	90
URLs Amigáveis.	93
Reaproveitamento de código com Partial.	97
Ajax.	104
Preparando as páginas para utilizar AJAX.	105
link_to_remote.	106
Templates JS.RJS.	108
remote_form_for.	110
Desenvolvimento baseado em Testes.	114
Testes de unidade e testes funcionais.	115
Criando nosso primeiro teste de unidade.	115
Executando os testes.	118
Fixtures.	119
Testes Funcionais.	121

O que é Ruby on Rails

Ruby on Rails é uma tecnologia que permite desenvolver sites dinâmicos orientados a banco de dados e aplicações para a web - de forma semelhante a tantas outras linguagens de programação como PHP ou ASP.

Porém, apesar de ser mais novo que estas duas linguagens, o Ruby on Rails vêm crescendo de forma espantosa e têm chamado a atenção de desenvolvedores de todo o mundo, isso porque ele permite aumentar a velocidade e facilidade no desenvolvimento de projetos.

Tecnicamente falando, Rails é um framework criado na linguagem de programação Ruby (daí o nome **Ruby** on Rails),

Um framework é como um esqueleto em cima do qual se desenvolve uma aplicação completa. Existem dezenas de frameworks disponíveis e muitos deles existem a mais tempo que o Rails, então o que faz do Rails algo tão importante? A resposta é simples: O Rails foi criado com o intuito de permitir o desenvolvimento ágil, com alta produtividade, escrevendo poucas linhas de código e tendo muito resultado como consequência. Aplicações que levam semanas ou meses para desenvolver em linguagens tradicionais podem ser feitas em horas ou dias com Ruby on Rails.

A LINGUAGEM RUBY

Ruby é uma linguagem de script interpretada para programação orientada a objetos com uma filosofia e sintaxe muito limpa que fazem com que programar seja elegante e divertido.

A linguagem foi criada no início da década de 90 no Japão e rapidamente ganhou popularidade no mundo inteiro por sua filosofia de ter foco nas pessoas.

Para se tornar um desenvolvedor Ruby on Rails pleno é importante conhecer à fundo a linguagem Ruby. Existem dezenas de livros, tutoriais e sites que podem lhe

ajudar, com grande destaque para o próprio site da linguagem Ruby (<http://www.ruby-lang.org>), e o livro virtual “Why's (Poignant) Guide to Ruby” (<http://poignantguide.net/ruby/> - com versão em português em <http://github.com/carlosbrando/poignant-br/>).



Dica de Ruby

Este livro é voltado para o estudo do framework Ruby on Rails, mas ao longo dele você vai encontrar caixas como esta com dicas específicas da linguagem Ruby.

A quem se destina este livro

O objetivo deste livro é servir como um primeiro contato com o framework Ruby on Rails - não se trata de uma referência completa sobre o assunto. Trata-se de um material para desenvolvedores que se interessem por Ruby on Rails que queiram aprender os conceitos fundamentais na prática, criando uma aplicação real.

Ao completar este livro, você irá adquirir a base mínima necessária para começar a trabalhar e estudar mais à fundo a linguagem Ruby e o Framework Rails.

Configuração e instalação

Este material assume que você tenha familiaridade com programação Web em geral, e com pelo menos um banco de dados relacional. Na elaboração deste material foi utilizado o MySQL, por ele ser um banco de fácil instalação e por estar amplamente difundido na web, mas você pode usar o que mais se adaptar ao seu ambiente, principalmente considerando que o Rails esconde a maior parte da complexidade nessa área e que ele suporta os principais bancos de dados existentes no mercado.

Para começar, vamos assumir que você tenha um banco de dados instalado e que esteja pronto para instalar a linguagem Ruby e o Rails.



Dica de Ruby - GEM

Nas instalações para Windows, Mac e Linux será utilizado o GEM.

Gem é o gerenciador de pacotes do Ruby, capaz de baixar os arquivos necessários da Web e instalá-los no local apropriado dentro das bibliotecas do Ruby sem que o desenvolvedor tenha que se preocupar com os detalhes do processo.

INSTALAÇÃO EM WINDOWS

No Windows existe o Ruby One-Click Installer. Esse instalador pode ser baixado de <http://rubyforge.org/projects/rubyinstaller/> e basta rodá-lo para ter um ambiente Ruby para Windows (apenas Ruby, sem Rails por enquanto).

Vamos agora utilizar o gerenciador de pacotes GEM para instalar o próprio Rails e mais algumas outras GEMS que serão necessárias durante o desenvolvimento. Abra a linha de comando do seu Windows e digite.

```
gem install sqlite3-ruby  
gem install mysql  
gem install rails
```

Pronto, agora você tem um ambiente completo Ruby on Rails em Windows.

Alternativamente, ao invés de realizar manualmente a instalação você pode baixar o InstantRails, uma aplicação que contém tudo o que é necessário para começar o desenvolvimento: Ruby, Rails, servidores Apache e Mongrel além do MySQL. O InstantRails é tão simples que nem é necessário instalá-lo – basta descompactá-lo na pasta desejada e rodar a aplicação.

O InstantRails pode ser baixado em <http://instantrails.rubyforge.org>

INSTALAÇÃO EM MAC OS

No Mac OS 10.5 em diante, o ruby e o rails já vêm pré-instalados por padrão. O único problema é que a versão do Rails que já vêm instalada não é a versão mais recente. Para atualizar o seu sistema siga os seguintes procedimentos:

1-Instale o XCode Tools, presente no CD de instalação do MAC OS, na categoria “Optional Installs” (arquivoXcodeTools.mpkg)

2-Abra o terminal e digite:

```
sudo gem update --include-dependencies
```

3 - Selecione as opções padrões de instalação e após alguns instantes seu sistema estará atualizado.

INSTALAÇÃO EM LINUX

Em distribuições do Linux como o Ubuntu ou Debian, com facilidades para a instalação de aplicativos, uma maneira rápida é usar as ferramentas da própria distribuições para baixar os pacotes e dependências necessárias.

Assim, se você está utilizando o Linux, eu recomendo que você compile a sua própria versão do Ruby. A versão 1.8.6 é a estável atualmente. Para compilá-la, utilize o procedimento abaixo:

```
wget ftp://ftp.ruby-lang.org/pub/ruby/ruby-1.8.6.tar.gz
tar xvfz ruby-1.8.6.tar.gz
cd ruby-1.8.6
./configure --prefix=/usr
make
make install
```

Para verificar a versão instalada do Ruby, use o seguinte comando:

```
ruby -v
ruby 1.8.6 (2007-09-24) [i486-linux]
```

Uma vez que o Ruby esteja instalado, é hora de fazer o mesmo com o Rails. O procedimento é bem simples: basta executar o comando abaixo!:

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

Com os passos acima finalizados, você está pronto para começar o desenvolvimento em Rails.

Iniciando em Rails

DESENVOLVENDO UMA APLICAÇÃO COMPLETA

Para exemplificar os conceitos abordados neste material, vamos desenvolver uma aplicação completa de agendamento de eventos. Esta aplicação listará uma série de eventos como cursos, encontros e seminários com suas respectivas descrições e datas, e permitirá que se faça a gestão destas informações com o cadastramento, edição e exclusão de eventos.

A primeira coisa a fazer, então, é criar o diretório de trabalho da aplicação. Isso é feito digitando o comando abaixo no console (ou no prompt de comando, no windows):

```
rails eventos
```

O comando rails gera o esqueleto completo de uma aplicação, pronta para rodar. Esse comando foi criado em seu sistema quando você instalou as bibliotecas necessárias.

Após a digitação você verá o resultado do processo de criação, conforme demonstrado abaixo:

```
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
...
```

No esqueleto gerado, cada parte da aplicação tem um local específico para ser colocado. Isso deriva de uma das filosofias por trás do Rails que pode ser descrita pela frase “convenção ao invés de configuração”. Convenção, nesse caso, significa que há um acordo entre o *framework* e você, o desenvolvedor, de modo que você não precisa de preocupar com certos detalhes que, de outra forma, teriam que se descritos em um arquivo de configuração.

Não vamos entrar em detalhes sobre a estrutura de diretórios agora porque ela ficará evidente à medida que avançarmos no livro. Não se preocupe: ela é bem lógica e, na maior parte do tempo, automaticamente gerenciada pelo Rails.

Agora que temos uma aplicação básica, vamos rodá-la e ver o resultado.

Para facilitar a vida, o Rails vem com seu próprio servidor Web, utilizando as bibliotecas do próprio Ruby. Para rodar o servidor, basta usar um dos *scripts* utilitários presentes em cada aplicação gerada pelo Rails (veja o diretório `script` sob o esqueleto da aplicação).

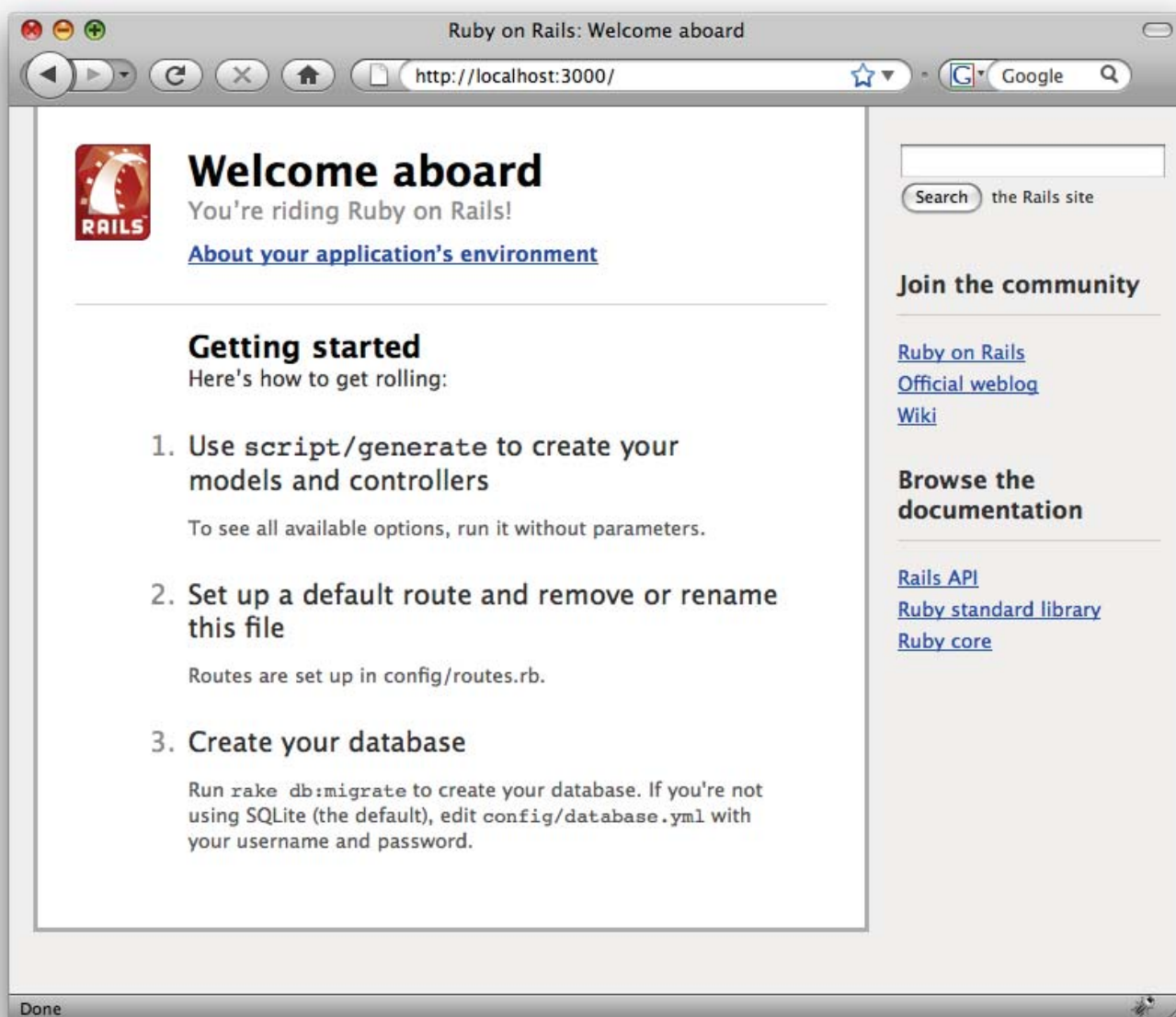
Entre no diretório recém-criado e digite:

```
script/server start

=> Booting Mongrel (use 'script/server webrick' to force
WEBrick)
=> Rails application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
** Starting Mongrel listening at :3000
** Starting Rails with development environment...
** Rails loaded.
** Loading any Rails specific GemPlugins
** Signals ready. TERM => stop. USR2 => restart. INT => stop
** Rails signals registered. HUP => reload (without restart).
** Mongrel available at 0.0.0.0:3000
** Use CTRL-C to stop.
```

Como você poder ver, o comando inicia uma instância do servidor Mongrel, capaz de servir aplicações Rails sem necessidade de qualquer configuração adicional. O servidor roda localmente e recebe requisições na porta 3000.

Lembre-se que em ambiente Windows, você deve digitar “`ruby script/server start`”.



Como mostrado acima, temos uma aplicação inicial rodando que, inclusive, mostra quais são os próximos passos para continuar o desenvolvimento da mesma.

CONFIGURANDO O BANCO DE DADOS

Vamos agora configurar o banco de dados da aplicação. Esse primeiro passo é muito importante porque o Rails usa as informações proveniente do *schema* do banco de dados para gerar automaticamente arquivos e configurações adicionais. Esse é mais um exemplo de convenção ao invés de configuração. Dessa forma, garantir que o banco está funcionando corretamente, configurado para uso da aplicação, é o passo inicial de qualquer desenvolvimento em Rails.

O arquivo de configuração de banco de dados se chamada `database.yml` e está localizado no diretório `config`, junto com os demais arquivos de configuração da aplicação.



Dica de Ruby - Arquivos YAML (*.yml)

Um arquivo do tipo YAML é um arquivo de dados como o XML, por exemplo, mas com uma sintaxe mais limpa e enxuta. Um documento YAML é bastante útil para guardar arquivos de configuração.

Abrindo o arquivo `database.yml`, repare que uma aplicação Rails geralmente utiliza três bancos de dados, um para cada ambiente de desenvolvimento padrão. Repare também que por padrão o arquivo já vem previamente configurado para o banco de dados SQLite. Na nossa aplicação utilizaremos o banco MySQL, portanto nossa configuração ficará assim:

```
development:
  adapter: mysql
  encoding: utf8
  database: eventos_development
  pool: 5
  username: root
  password:
  host: localhost

test:
  adapter: mysql
  encoding: utf8
  database: eventos_test
  pool: 5
  username: root
  password:
  host: localhost

production:
  adapter: mysql
  encoding: utf8
  database: eventos_production
  pool: 5
  username: root
  password:
  host: localhost
```

Um banco de dados é utilizado para o desenvolvimento, onde todas as mudanças são aplicadas. Esse banco tem seu correspondente em um banco de produção, onde modificações somente são aplicadas uma vez que estejam completas. O arquivo permite configurar, inclusive, um banco remoto para onde suas modificações finais serão redirecionadas—embora geralmente seja melhor utilizar um outro método de implantação, como veremos mais adiante.

O terceiro banco mostrado acima é um banco de testes, utilizado pelo Rails para a execução de *unit testing*. Esse banco **deve** ser mantido necessariamente à parte já que todos os dados e tabelas presentes no mesmo são excluídos e recriados a cada teste completo efetuado na aplicação.

Voltando à nossa aplicação, como um arquivo de configuração importante foi mudado, o servidor Web precisa ser reiniciado. Isso acontece porque ele somente lê essas configurações no início de execução. Esse é um dos raros casos em que um servidor de desenvolvimento precisa ser reiniciado já que o Rails recarrega praticamente qualquer modificação feita na aplicação quando está no modo de desenvolvimento.

Agora temos uma aplicação e um banco de dados configurado. Com isso já podemos iniciar o processo de desenvolvimento propriamente dito.

MVC – Models, Views e Controllers

O Rails utiliza uma estratégia de desenvolvimento chamada de MVC (*Model-View-Controller*). Essa estratégia separa os componentes da aplicação em 3 partes distintas:

Model

O Model é a parte da aplicação que faz ligação com o banco de dados. Tecnicamente, o model é implementado pela classe *ActiveRecord*.

View

O View é a interface com o usuário – um sistema de template que gera documentos HTML (entre outros) que são enviados para o usuário. As classes conhecidas como *ActionPack*, implementam o View e também o *Controller*.

Controller

Um *controller* é uma classe responsável por receber as requisições feitas pela aplicação e executar as ações necessárias para atender essas requisições. É no controller que definimos a lógica do funcionamento da nossa aplicação.

O controller é quem será efetivamente solicitado pelo navegador. Quando necessário, o controller utiliza um Model previamente definido para acessar o banco de dados e, finalmente, encaminha um arquivo de visualização (uma View) para o usuário.

Apesar de parecer complicado e trabalhoso demais à primeira vista, este modo de trabalhar traz uma série de vantagens em relação ao modelo tradicional de desenvolvimento em que uma única página mistura códigos responsáveis pelo seu funcionamento, pelo acesso ao banco de dados e por exibir dados ao usuário.

Algumas vantagens são:

- A manutenção fica mais fácil por haver uma clara distinção entre as partes da aplicação
- É possível manter um controle centralizado de todo o site num único (ou em poucos) arquivo, ao invés de espalhado em dezenas deles.

O PRIMEIRO CONTROLLER

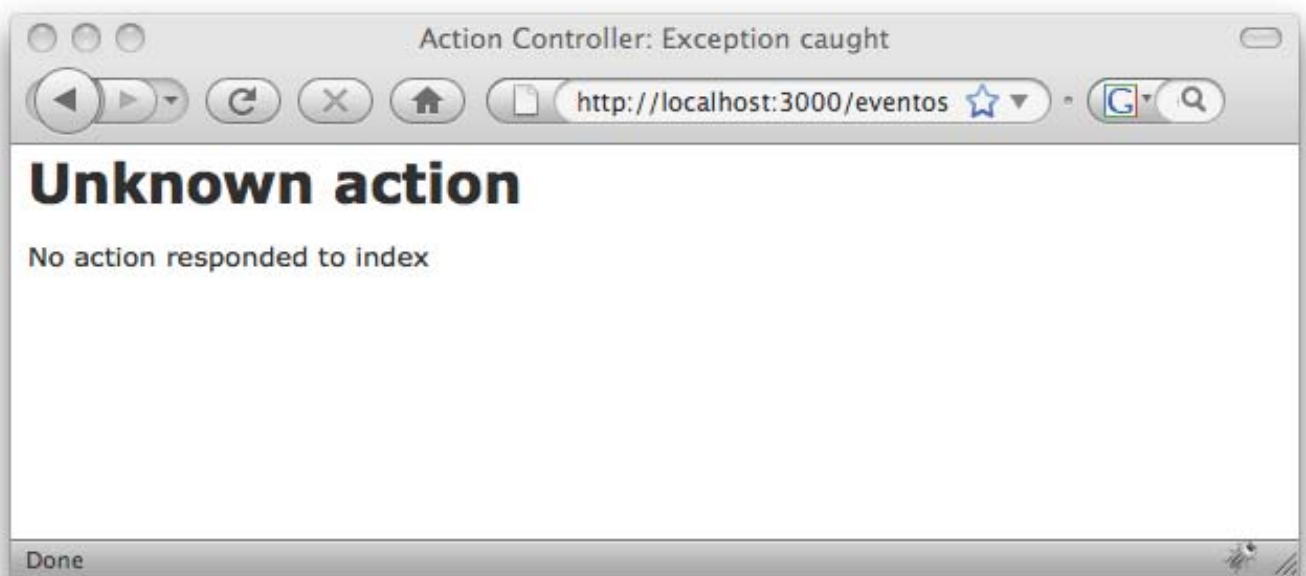
Vamos começar criando um *controller* para lidar com a página inicial de nossa aplicação. Para gerar um *controller*, usamos o comando abaixo:

```
script/generate controller eventos
```

```
exists  app/controllers/  
exists  app/helpers/  
create  app/views/eventos  
exists  test/functional/  
create  app/controllers/eventos_controller.rb  
create  test/functional/eventos_controller_test.rb  
create  app/helpers/eventos_helper.rb
```

Utilizaremos muito o comando `generate`, que gera documentos automaticamente. O nome passado para o comando é `eventos`, que será usado para compor todos os arquivos gerados pelo mesmo.

Se você acessar a URL desse *controller* agora, você vai receber a seguinte tela:



Repare que, nesse caso, eu informei somente `/controller/` como a URL, sem passar nem a parte correspondente a uma ação ou a um *id*. Nesse caso, o Rails assume que estamos invocando a ação *index* sem *id*. Esse é um mais exemplo de convenção ao invés de configuração. Ao invés de ter sempre que especificar uma ação padrão em algum lugar, o Rails convencionou que a ação padrão é *index*, poupando o desenvolvedor sem afetar a aplicação.

Vamos olhar o arquivo `app/controllers/eventos_controller.rb` gerado por nosso comando. Como já mencionamos, o Rails segue uma estrutura fixa de diretórios, poupando mais uma vez o desenvolvedor. Arquivos relacionados ao banco vão em no diretório `db`, arquivos de configuração em `config` e tudo relacionado a MVC vai em `app`. Dentro de `app` temos vários outros diretórios que contém mais partes específicas da aplicação. Em alguns casos, o Rails também adiciona um sufixo ao arquivo, evitando colisões de nomes e problemas estranhos na aplicação, como é o caso aqui. O arquivo do *controller* criado contém o seguinte:

```
class EventosController < ApplicationController
end
```

A classe `EventosController` define quem que irá responder a requisições padrão em `/home`. Ela herda da classe `ApplicationController`, que está definida no arquivo `application.rb`, no mesmo diretório. Sendo assim, qualquer método criado na classe `ApplicationController` estará automaticamente disponível nos *controllers* gerados para a aplicação.

Para criarmos a ação *index*, basta adicionarmos um método à classe:

```
class EventosController < ApplicationController  
  
  def index  
    end  
  
end
```

O princípio que usamos acima é comum a tudo em Rails. Basicamente tudo o que fazemos em uma aplicação usando o mesmo consiste em estender alguma classe por meio da adição de métodos customizados.

Recarregando a página no navegador, ficamos com o seguinte:



Vamos que, dessa vez, o Rails identificou a ação a ser executada, mas, como a aplicação não especificou nenhum retorno, houve um erro. Isso aconteceu porque o Rails tentou aplicar automaticamente uma *view* para aquela ação do *controller*. Como a *view* ainda não existe, temos o erro.

A PRIMEIRA VIEW

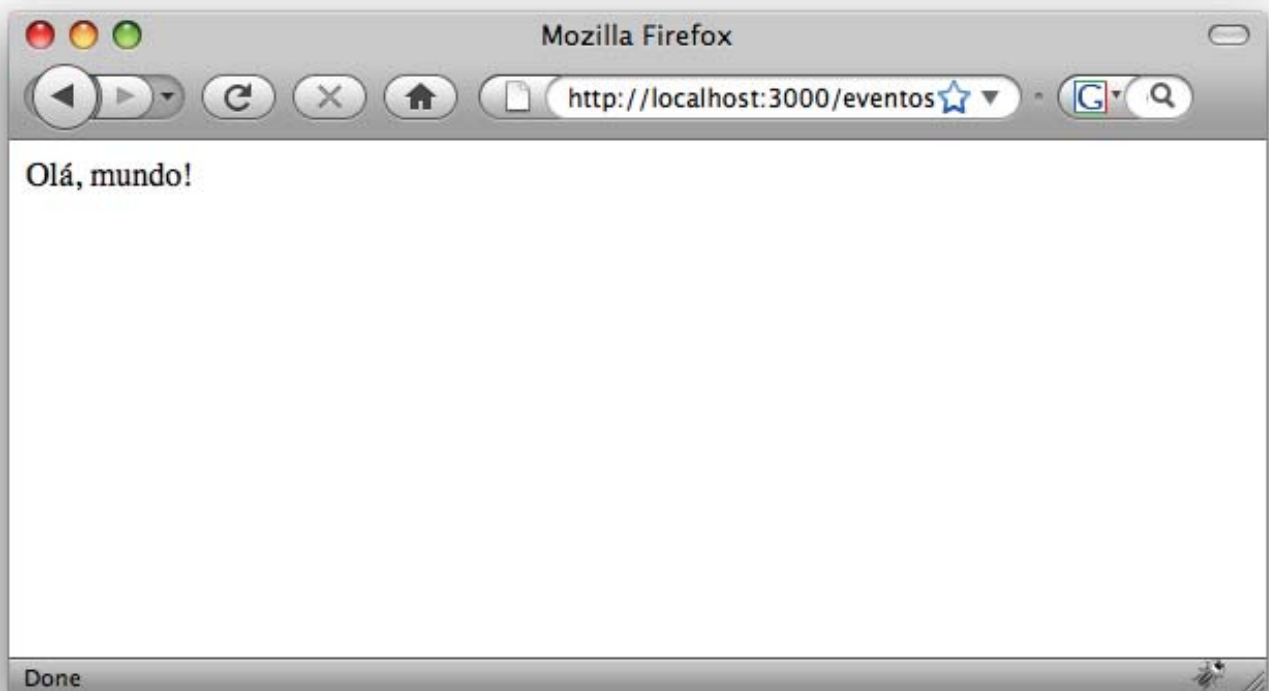
Vamos criar a nossa primeira *view*. O rails é automaticamente configurado para procurar pela view com o mesmo nome da ação. Nossa ação se chama index, portanto precisamos criar uma view com o mesmo nome “index”. As views são salvas numa pasta com o mesmo nome do controller, portanto no nosso caso criamos o arquivo `app/views/eventos/index.html.erb`

As extensões colocadas em sequencia `.html.erb` indicam que esse é um arquivo HTML com código Ruby embutido (Embed RuBy - `.erb`).

Vamos criar o seguinte arquivo, então:

```
<p>Olá, mundo!</p>
```

Agora, recarregando a página, temos:



Sem que haja necessidade de especificar qualquer coisa, o Rails está usando o arquivo adequado. Veja que não configuramos qualquer coisa. Criamos um *controller*, definimos um método no mesmo, e criamos um arquivo que contém o que queremos que seja retornado por aquele método. A simples existência desses arquivos representa uma cadeia de execução sem que precisamos nos preocupar com que parte da aplicação faz isso ou aquilo.

Qualquer outra ação que fosse criada dentro desse *controller* seguiria o mesmo padrão. O nome da ação seria associado a um nome de arquivo dentro de um diretório em *app/views* cujo nome seria o do próprio *controller*. O Rails também é inteligente ao ponto de responder a uma ação mesmo que o método não exista, desde que a *view* esteja presente no diretório correto.

INCREMENTANDO AS VIEWS COM USO DE LAYOUTS

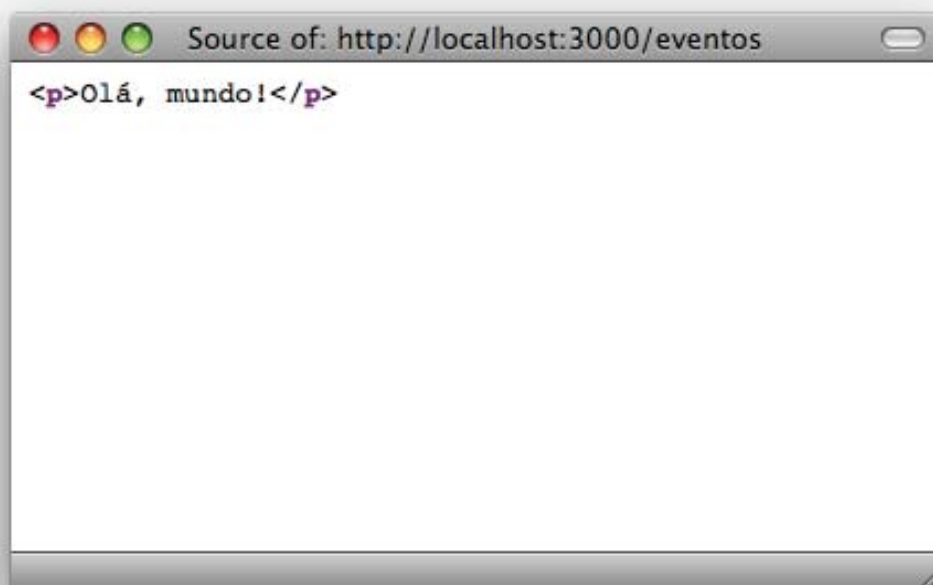
Para facilitar o desenvolvimento da parte visual de uma aplicação, o Rails possui um conceito denominado *layouts*.

Na maioria das aplicações Web, as páginas variam somente no seu conteúdo principal, possuindo cabeçalhos, rodapés e barras de navegação em comum. Obviamente, um *framework* cujo maior objetivo é aumentar a produtividade do desenvolvedor não exigiria que o código para esses elementos tivesse que ser repetido em cada *view*. Um *layout* funciona como um arquivo raiz, dentro do qual o resultado de uma *view* é inserido automaticamente.

Mais uma vez favorecendo convenção ao invés de configuração, o Rails define um arquivo padrão de *layout* que é usado automaticamente por qualquer *view* a não ser que haja especificação em contrário. O Rails também é inteligente o bastante para somente usar um *layout* em *views* com a mesma extensão.

O *layout* padrão para a aplicação fica em um arquivo chamado `application.html.erb`, dentro do diretório `app/views/layouts`.

Se olharmos agora o código gerado pela página que criamos no nosso primeiro exemplo de Controller e View, teremos o seguinte:



Como você pode notar, não estamos gerando nenhum dos elementos HTML geralmente vistos em uma página comum.

Vamos criar o arquivo `application.html.erb` no diretório especificado, com o seguinte conteúdo:

```
<html>
<head>
  <title>Central Eventos</title>
</head>

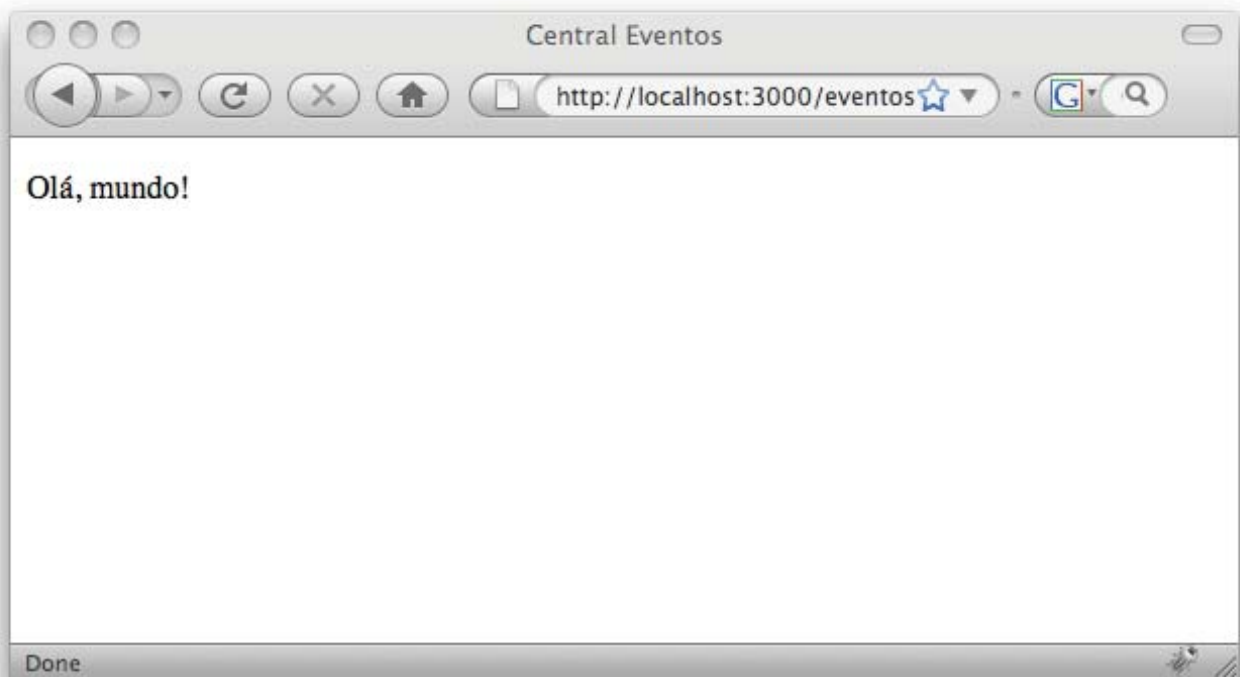
<body>
<div id="principal">
  <%= yield %>
</div>
</body>

</html>
```

Temos no arquivo acima, o primeiro exemplo de uso de código Ruby dentro de uma *view*, delimitado pelos marcadores `<%` e `%>`. Aqueles familiarizados com PHP e ASP reconhecerão o estilo de marcadores, com o uso de `<%= objeto %>` para retornar conteúdo.

No caso acima, o método especial `yield` retorna o conteúdo atual gerado pela ação, seja por meio de uma *view* ou usando `render` diretamente. O método `yield` tem uma conotação especial no Ruby, servindo para invocar o bloco associado ao contexto. Inserido em um *layout* do Rails, o bloco define a execução da ação, com seu conseqüente retorno de conteúdo.

A nossa página recarregada agora fica como mostrado abaixo:



Não parece muita coisa, mas, olhando o código, você verá o seguinte:

```
Source of: http://localhost:3000/eventos

<html>
<head>
  <title>Central Eventos</title>
</head>

<body>
<div id="principal">
  <p>Olá, mundo!</p>
</div>
</body>

</html>
```

O que precisamos fazer agora é estender o nosso *layout*. O nosso arquivo `application.html.erb` poderia ser mudado para o seguinte:

```
<html>
<head>
  <title>Central de Eventos</title>
  <%= stylesheet_link_tag "default" %>
</head>

<body>
  <div id="principal">
    <%= yield %>
  </div>
</body>

</html>
```

O método `stylesheet_link_tag` recebe o nome de uma *stylesheet* como parâmetro e gera um link para a mesma. O nosso código gerado agora ficou assim:



```
Source of: http://localhost:3000/eventos

<html>
<head>
  <title>Central Eventos</title>
  <link href="/stylesheets/default.css" media="screen" rel="stylesheet" type="text/css" />
</head>

<body>
  <div id="principal">
    <p>Olá, mundo!</p>
  </div>
</body>

</html>
```

Note que mais uma vez o Rails assumiu um caminho padrão. Nesse caso, o arquivo é servido diretamente da raiz da aplicação, que é o diretório `public`. Você pode criar um arquivo chamado `default.css` no diretório `public/stylesheets` para adicioná-lo à aplicação.

Um exemplo disso seria:

```
body{
  background-color:#763A3A;
}
h1{
  font-family:Verdana, Arial, Helvetica, sans-serif;
  font-variant:small-caps;
  font-size:30px;
  margin:0px;
  color:#572B2B;
}
h2{
  font-family:Arial, Helvetica, sans-serif;
  font-size:25px;
  margin:0px;
}
p{
  font-family:Arial, Helvetica, sans-serif;
  font-size:12px;
  color:#333333;
}

#principal{
  width:760px;
  height:600px;
  border:solid 2px #000000;
  background-color:#FFFFFF;
  margin:20px auto 0px auto;
  padding:15px;
}
```

Uma coisa a manter em mente é que o *layout* padrão da aplicação não é, de forma alguma, o único que pode ser gerado. Você pode criar tantos *layouts* quanto precisar, colocando-os no mesmo diretório, de onde estarão acessíveis a toda aplicação.

Para usar um *layout* diferente em um controler você poderia fazer algo assim:

```
class EventosController < ApplicationController  
  
  layout "home"  
  
  def index  
    end  
  
end
```

O *layout* cujo nome é *home* seria especificado no arquivo `app/views/layouts/home.html.erb`, com a mesma funcionalidade do arquivo `application.html.erb`.

MODELS

Models correspondem à camada de acesso ao banco de dados, implementada no Rails por um componente denominado *ActiveRecord*.

O Rails automaticamente entende que todas as tabelas criadas no banco para uso em models satisfarão a duas condições: terão um nome plural em inglês e terão uma chave primária auto-incrementada chamada *id*. É possível mudar ambas as condições, mas inicialmente ficaremos com elas para não ter que modificar o que o Rails gera e usa automaticamente.

Para gerar um modelo, utilizamos o seguinte comando:

```
script/generate model categoria

  exists  app/models/
  exists  test/unit/
  exists  test/fixtures/
  create  app/models/categoria.rb
  create  test/unit/categoria_test.rb
  create  test/fixtures/categorias.yml
  create  db/migrate
  create  db/migrate/20081011091001_create_categorias.rb
```

O comando acima gera toda estrutura de suporte ao modelo, incluindo testes e migrações para o mesmo (abordaremos estes assuntos adiante neste livro).

O arquivo responsável pela implementação do modelo está em `app/model/categoria.rb` e contém apenas o seguinte:

```
class Categoria < ActiveRecord::Base
end
```

Essas duas linhas, apoiadas pelo Rails, já providenciam uma riqueza de implementação que nos permite recuperar, inserir e atualizar dados no banco, e executar uma série de outras operações complexas sem a necessidade de qualquer comando SQL direto.

GERANDO MIGRAÇÕES

O Rails tem uma solução inovadora para criar as tabelas do seu banco de dados.: Ele criou o conceito de *migrations* (migrações).

Migrations são *scripts* em Ruby que descrevem operações e modificações de qualquer natureza no banco de dados. Migrações são a maneira mais fácil de acrescentar tabelas no banco.

No caso acima, estamos gerando uma migração, cujo nome é `create_categorias`.

```
class CreateCategorias < ActiveRecord::Migration
  def self.up
    create_table :categorias do |t|

      t.timestamps
    end
  end

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

O método `self.up` é utilizado para efetuar as modificações. O seu oposto, `self.down`, é usado para desfazer essas modificações caso você esteja revertendo para uma versão anterior do banco.

Vamos editar o arquivo agora para incluir a tabela que desejamos criar:

```
class CreateCategorias < ActiveRecord::Migration
  def self.up
    create_table :categorias do |t|
      t.string :nome_categoria
      t.timestamps
    end
  end

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

Para efetuar suas migrações, o Rails utiliza uma linguagem de domínio que permite especificar operações de banco de dados de forma abstrata, que não está presa a um servidor específico. É possível executar operações diretamente via SQL no banco, mas isso não é recomendado por quebrar essa abstração.

No caso acima, o método `create_table` dentro de `self.up` serve para especificar a tabela que será criada. Esse método recebe um bloco como parâmetro (indicado pela palavra-chave `do`) que especifica as colunas que serão adicionadas.

Voltando ao nosso código, como a tabela terá somente uma coluna inicialmente, que é o nome da categoria, somente precisamos de uma linha, identificando o tipo da coluna e seu nome. Mais à frente veremos outros tipos e opções.

RAKE

Agora é hora de aplicar essa migração ao banco, subindo sua versão. Para isso, temos o utilitário rake, que é parte do Ruby. O rake é capaz de realizar várias de suas tarefas diferentes, sendo grande parte delas destinadas à manipulação de bancos de dados.

A primeira tarefa que utilizaremos cria o banco de dados automaticamente para você. Basta digitar o comando `rake db:create`

Na sequência utilizaremos o `rake db:migrate`, que aplica as migrações ainda não efetuadas a um banco.

Rodamos o comando da seguinte forma:

```
rake db:migrate
== 20081011091001 CreateCategorias: migrating
=====
-- create_table(:categorias)
   -> 0.0021s
== 20081011091001 CreateCategorias: migrated (0.0022s)
=====
```

Se observamos o banco agora, teremos o seguinte *schema*:

<i>Tables in eventos</i>
categorias
schema_migrations

Duas tabelas foram criadas, *categorias* e *schema_migrations*. A primeira é a que especificamos. A segunda descreve informações do banco para o Rails, para controlar o versionamento do mesmo.

A tabela *categorias* ficou assim:

<i>Field</i>	<i>Type</i>	<i>Null</i>	<i>Key</i>	<i>Default</i>	<i>Extra</i>
id	int (11)	NO	PRI	NULL	auto_increment
nome_categoria	varchar (255)	YES		NULL	

Como você pode ver, não é preciso especificar a chave primária, que é automaticamente criada pelo Rails.

A tabela *schema_migrations* possui os seguintes dados:

<i>version</i>
200810110910 01

Esse número de versão é um timestamp, e cada nova migração terá uma versão.

Vamos criar agora um novo modelo correspondendo à tabela de eventos. O comando é:

```
script/generate model evento
```

```
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/evento.rb
create  test/unit/evento_test.rb
create  test/fixtures/eventos.yml
exists  db/migrate
create  db/migrate/20081011092126_create_eventos.rb
```

Você pode ver que o Rails gerou um *template* da migração, e basta editá-la:

```
class CreateEventos < ActiveRecord::Migration
  def self.up
    create_table :eventos do |t|

      t.timestamps
    end
  end

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

O arquivo final seria, com base no que pensamos até o momento:

```
class CreateEventos < ActiveRecord::Migration
  def self.up
    create_table :eventos do |t|
      t.string :titulo
      t.datetime :data
      t.text :descricao
      t.integer :categoria_id
      t.timestamps
    end
  end

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

Você não precisa se preocupar em criar todos os campos inicialmente. Você pode sempre usar outra migração para isso.

Rodando o comando rake para carregar as migrações mais uma vez, temos o seguinte:

```
rake db:migrate

== 20081011092126 CreateEventos: migrating
=====
-- create_table(:eventos)
   -> 0.0311s
== 20081011092126 CreateEventos: migrated (0.0314s)
=====
```

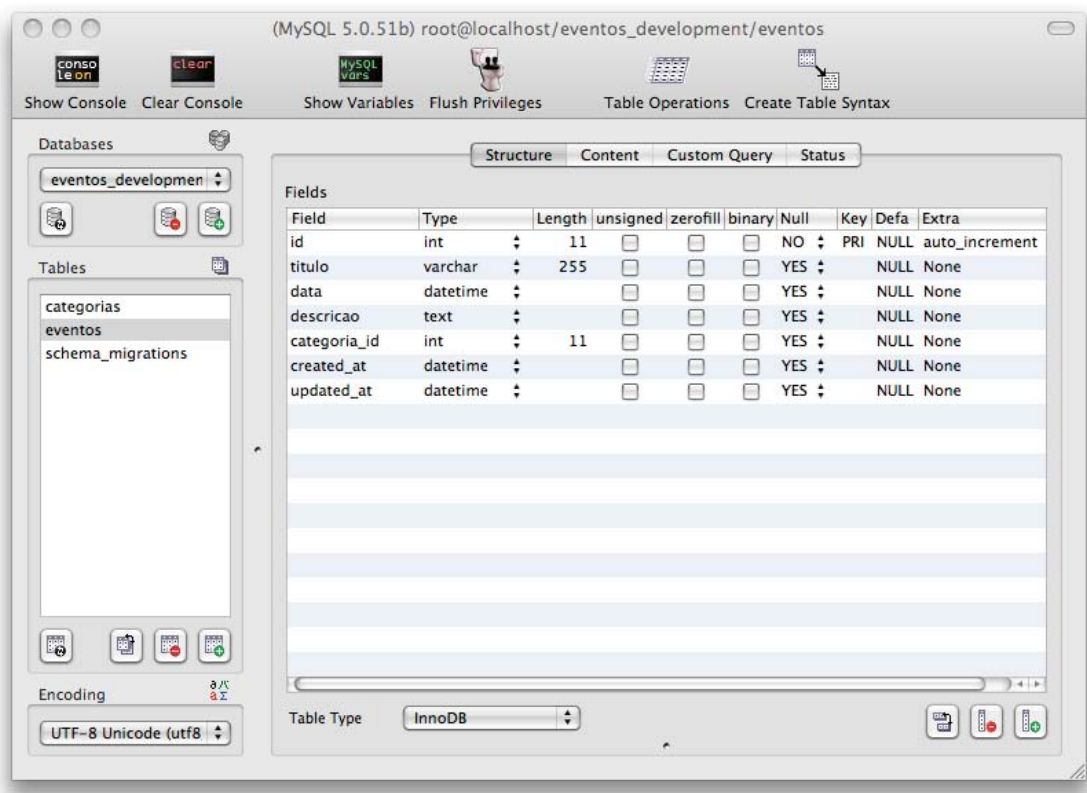
O resultado final do banco seria:

<i>Tables in Eventos</i>
categorias
eventos
schema_migrations

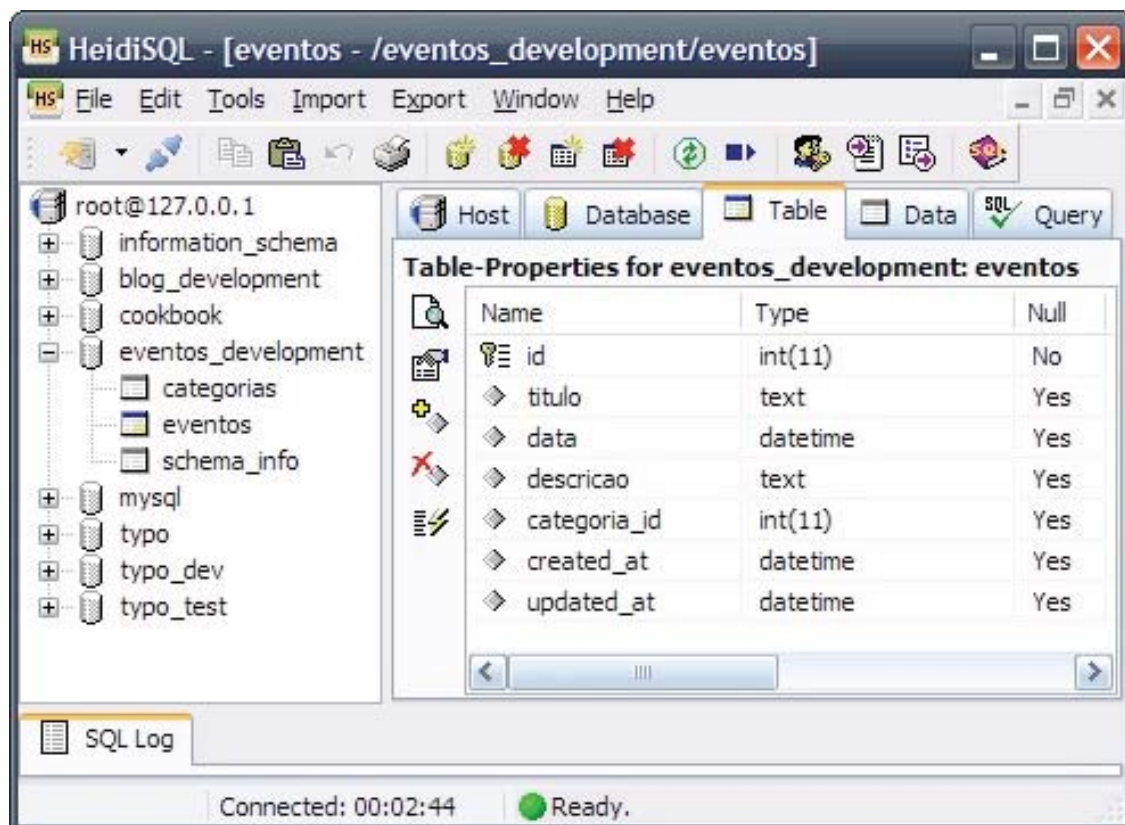
CADASTRO MANUAL NO BANCO DE DADOS

Quando criamos uma nova aplicação Rails, normalmente usamos as próprias funcionalidades desenvolvidas na aplicação ou fixtures para inserir dados de teste no banco de dados. Entretanto, como estamos aprendendo, precisaremos de cadastrar manualmente algumas informações em nosso banco de dados. O cadastramento manual de dados pode ser feito de duas formas – Utilizando um programa específico que se conecte ao banco de dados ou via códigos Rails:

- **Utilizando um programa específico:** Existem diversos softwares que se conectam a uma base de dados e mostram uma interface gráfica para você fazer manipulações. Alguns são até mesmo gratuitos, como o CocoaMySQL para mac ou o HeidiSQL para Windows.



Cocoa MySQL para Mac



HeidiSQL para Windows

- **Via código Rails através do console:** É possível também fazer a inclusão manual destes dados em linha de comandos através do console do Rails.

Rode o console do rails digitando:

```
script/console
Loading development environment (Rails 2.3.1)
```

Na sequencia, acrescente dados no banco diretamente pelo nome do model seguindo pela instrução create. Por exemplo, para acrescentar uma nova categoria à nossa tabela de categorias, digitamos:

```
>> Categoria.create(:nome_categoria => 'Palestra')
=> #<Categoria id: 1, nome_categoria: "Palestra">
>>
```

Para acrescentar um novo evento, digitamos:

```
>> Evento.create(  
  ?> :titulo => 'Desenvolvimento agil com Rails',  
  ?> :descricao => 'Palestra sobre desenvolvimento de  
    aplicacoes para web.',  
  ?> :data => '2008-03-01',  
  ?> :categoria_id => '1')  
=> #<Evento id: 1, titulo: "Desenvolvimento agil com  
Rails", data: "2008-03-01 00:00:00",  
descricao: "Palestra sobre desenvolvimento agil de  
aplicacoes p...", categoria_id: 1, created_at:  
"2008-02-09 11:17:07", updated_at: "2008-02-09  
11:17:07">  
>>
```

PRIMEIRA LISTAGEM DE DADOS DO BANCO

Agora que já sabemos configurar a aplicação e temos o banco de dados e as tabelas criadas com alguns dados cadastrados, vamos aprender a listar os dados do banco em páginas para o usuário.

Para isso, vamos alterar a ação `index` do nosso controller `Home` (até o presente momento, essa ação apenas responde “Olá Mundo”).

Agora, utilizaremos o método `find` para localizar as informações no banco de dados. O método `find` localiza um registro, ou um conjunto de registros, baseado nos argumentos que lhe são fornecidos. No exemplo abaixo, o método `find` do model `Evento` localizará apenas o registro cujo ID é 2 na tabela `eventos`:

```
class EventosController < ApplicationController

  def index
    Evento.find(2)
  end

end
```

Caso você queira listar todos os registros de uma tabela, o argumento `:all` pode ser fornecido. Como desejamos listar todos os registros do nosso banco, a ação *listagem* do nosso controller fica assim:

```
class EventosController < ApplicationController

  def index
    @eventos = Evento.find(:all)
  end

end
```



Dica de Ruby - Variáveis

A criação de variáveis é extremamente simples na linguagem Ruby, bastando utilizar o operador “=”, como no exemplo:

```
valor = 10
```

Escopo de variáveis em Ruby

Outro aspecto interessante da linguagem Ruby é que o escopo das variáveis pode ser determinado simplesmente acrescentando um caractere especial na frente do nome. O caractere \$ torna o escopo da variável público, já o caractere @ cria uma variável de instância - uma variável que pode ser posteriormente referenciada fora do controller (na view, por exemplo).

No exemplo de código acima pegamos todos os eventos e armazenamos na variável de instância @eventos, que utilizaremos posteriormente na view.

Precisamos agora de criar a View que será retornada para o usuário. Localize o arquivo de view que havíamos criado anteriormente na pasta app/views/eventos/index.html.erb

Neste documento, criaremos uma estrutura de loop do tipo for...in que permitirá repetir a mesma porção de códigos para cada um dos registros encontrados no banco de dados.

```
<h1>Listagem de Eventos</h1>
```

```
<% for evento in @eventos %>
```

```
<% end %>
```




Dica de Ruby - Loops

A linguagem Ruby trabalha com diversos tipos de Loops, sendo o mais comum deles o for.

Em ruby o for pode ser criado referenciando um intervalo numérico simples ou uma variável contendo múltiplos dados.

No primeiro caso, determinamos um intervalo numérico utilizando “..”, como no exemplo:

```
for i in 0..5  
...  
end
```

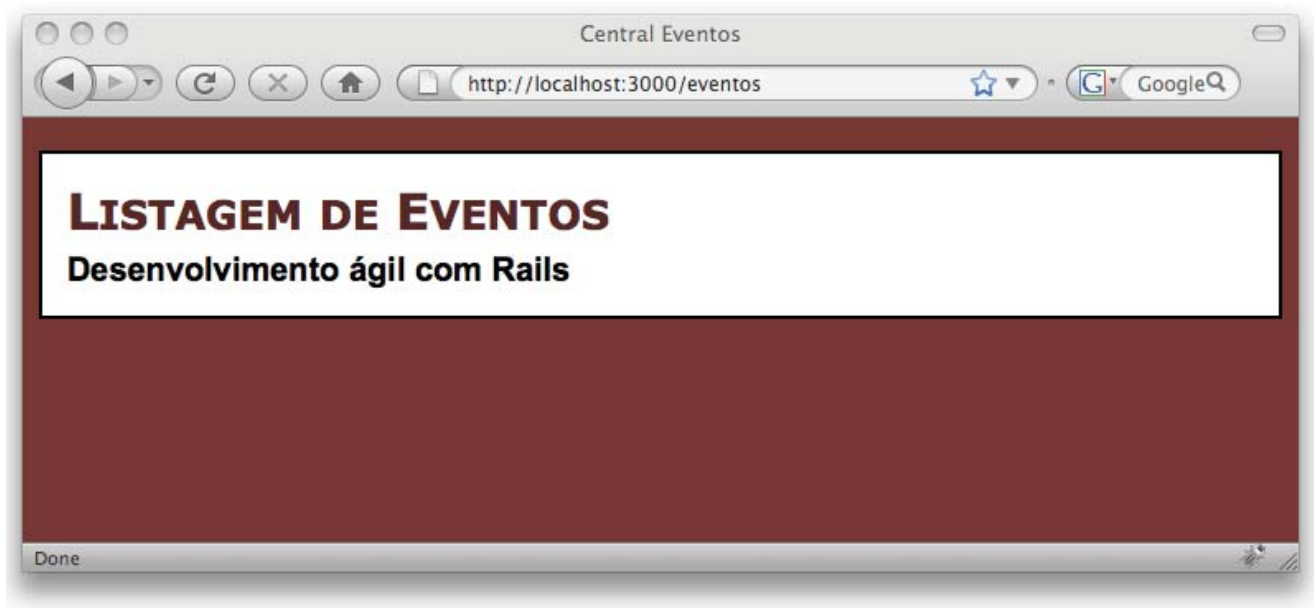
No segundo caso basta colocar a variável no lugar do intervalo

```
for i in @variavel  
...  
end
```

Dentro da área do loop inserimos o código para exibir o título dos eventos. O código completo fica assim:

```
<h1>Listagem de Eventos</h1>  
  
<% for evento in @eventos %>  
  
<h2><%= evento.titulo %></h2>  
  
<% end %>
```

Com o server iniciado e digitando o endereço localhost:3000/eventos/listagem temos o seguinte resultado final



Nosso próximo próximo passo é tornar o título de cada evento um link para uma nova página onde possamos ver os detalhes do evento clicado.

O Rails torna extremamente simples o processo de criar essa página de detalhes, acompanhe passo a passo:

Primeiro crie uma nova ação para o controller home. Vamos chamar esta ação de “show”.

```
class EventosController < ApplicationController

  def index
    @eventos = Evento.find(:all)
  end

  def show
    @evento = Evento.find(params[:id])
  end

end
```

Assim como na ação `index`, utilizamos o método `find` do model `Evento` para retornar um registro do banco de dados. Neste caso especificamente utilizamos o parâmetro `params[:id]` que faz com que o `find` retorne apenas o registro com o `id` passado como parâmetro através de um link.

Na sequência, edite o arquivo `index.html.erb`, nele vamos fazer com que o título do evento seja um link utilizando o método `link_to`. Veja o código:

```
<h1>Listagem de Eventos</h1>

<% for evento in @eventos %>

<h2><%= link_to evento.titulo, :action=>'show', :id=>evento.id
%></h2>

<% end %>
```

O primeiro parâmetro do `link_to` corresponde ao texto do link. Os outros dois parâmetros, como está claro no código, representam a ação para onde será direcionado o link e o `id` que será enviado junto.

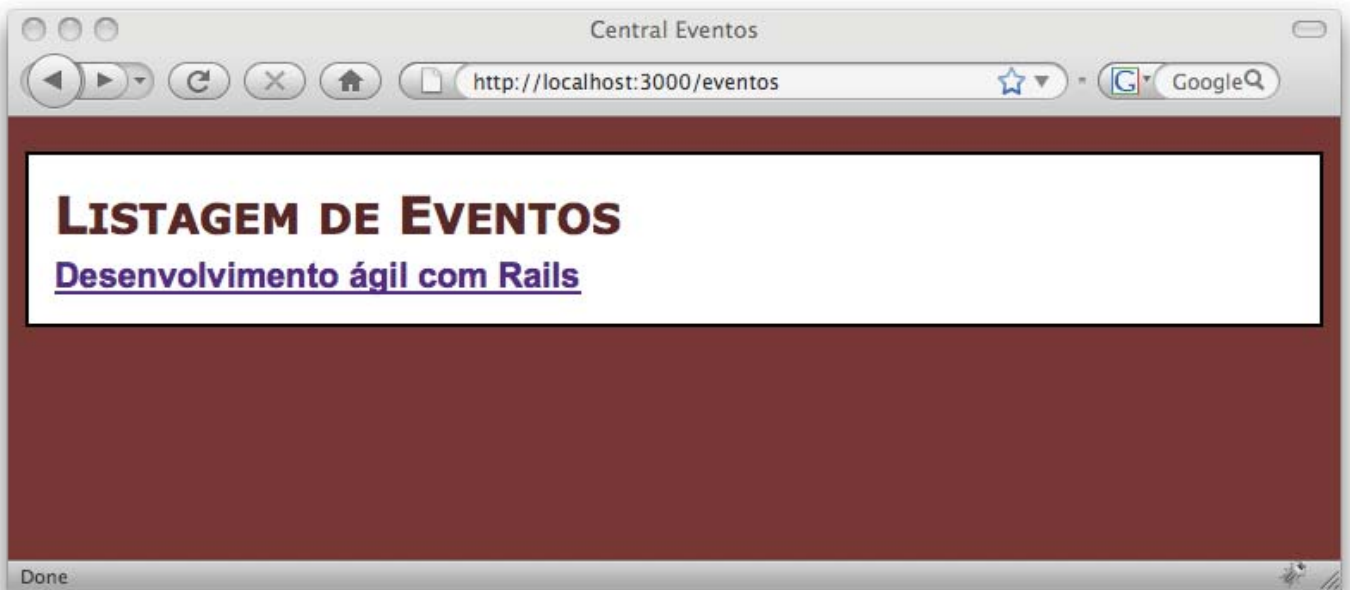
O método `link_to` possui várias outras capacidades entre as quais gerar confirmações em JavaScript e criar *forms* para submissão confiável de links que modificam destrutivamente seus dados, como veremos mais adiante.

Por fim, crie uma nova View chamada `show.html.erb`

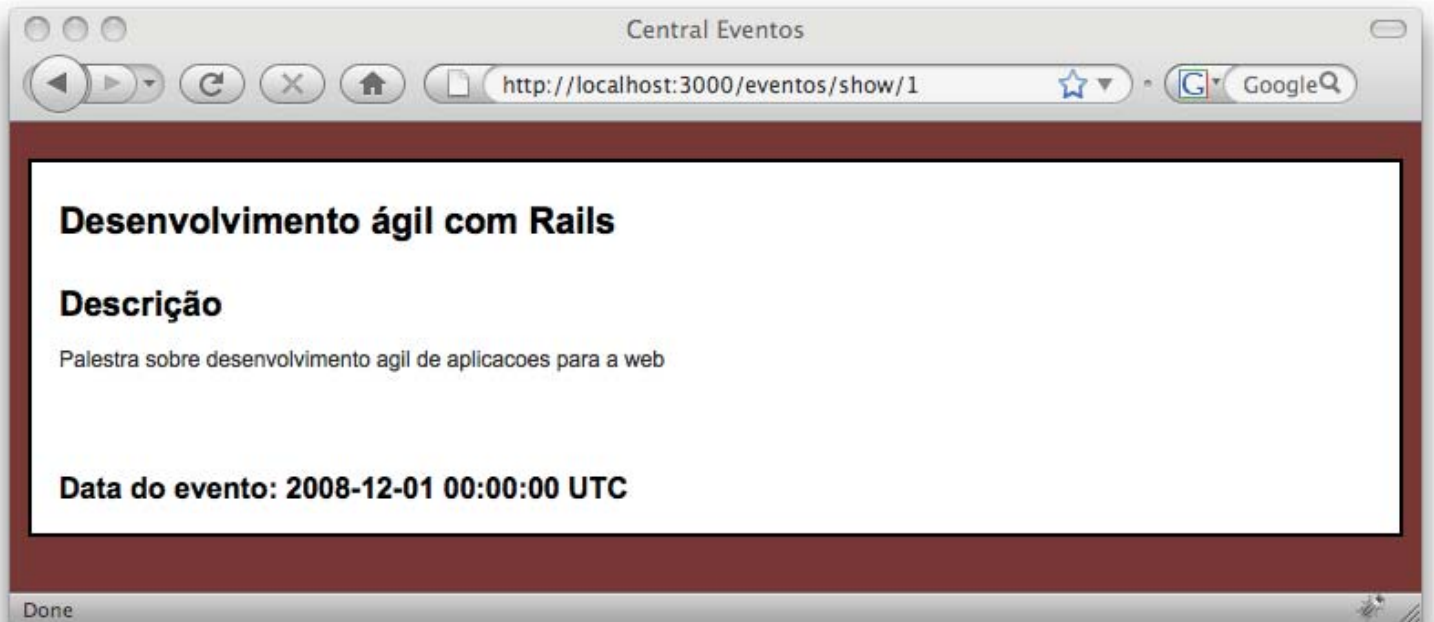
Esta view segue o mesmo padrão de funcionamento da View que lista todos os eventos, porém não terá uma estrutura de loop (uma vez que apenas um registro será exibido):

```
<h2><%= @evento.titulo %></h2>
<br />
<h3>Descrição</h3>
<p><%= @evento.descricao %></p>
<br /><br />
<h4>Data do evento: <%= @evento.data %></h4>
```

Veja a nossa página de listagem de eventos com os titulos transformados em links:

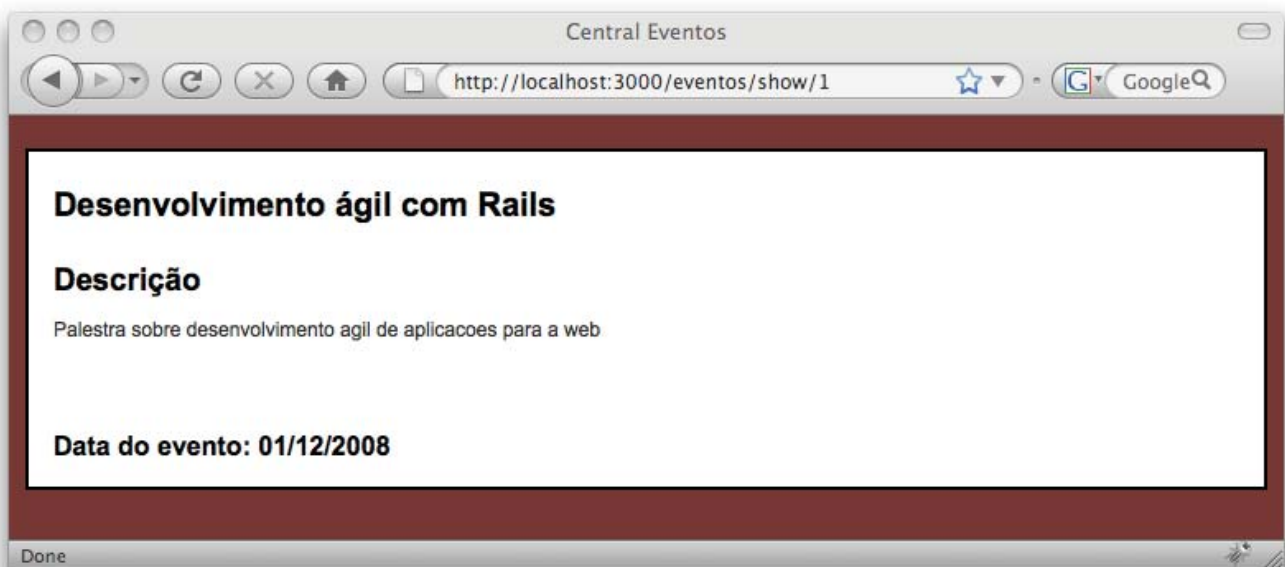


Clicando em um dos links, a página de detalhes é exibida



Um último detalhe: para modificar a forma como a data é exibida, podemos utilizar o método `strftime`:

```
<h2><%= @evento.titulo %></h2>
<br />
<h3>Descrição</h3>
<p><%= @evento.descricao %></p>
<br /><br />
<h4>Data do evento: <%= @evento.data.strftime("%d/%m/%Y") %></h4>
```



Mais adiante neste livro você aprenderá também como inserir novos registros, editar e apagar registros existentes.

REST

Até agora fizemos dois controllers diferentes que exibem duas views para os usuários: index e show.

De maneira geral, acessamos os controllers e views no navegador com a seguinte regra para a URL:

```
localhost:3000/nome_do_controller/nome_da_ação/id
```

Por exemplo, para acessar o controller index, que exibe a lista de eventos, usamos o seguinte endereço:

```
localhost:3000/eventos
```

Como o controller index é usado automaticamente se nenhuma outra ação for especificada, não precisamos preencher seu nome

Já a URL abaixo aponta para a nossa segunda ação, que mostra os dados específicos de um único evento:

```
localhost:3000/eventos/show/2
```

Além de especificar o nome do controller (home) e o nome da ação (show), um parâmetro é passado no final da URL. No exemplo acima o parâmetro “2” está sendo enviado.

Existe porém uma maneira melhor, mais organizada e enxuta de estruturar nossa aplicação, seus controllers e views, conhecida como REST.

O QUE É REST

O conceito principal do REST consiste em utilizar os métodos HTTP no lugar do nome das ações para rotear a ação correta dentro do controller. Isto é, ao invés de declarar explicitamente na URL o nome da ação (como “listagem” ou “detalhes”), o próprio Rails determina a ação correta dependendo do método de envio HTTP.

Qualquer desenvolvedor que já lidou com formulários conhece os métodos http como GET e POST. O primeiro envia dados para o servidor na própria URL (algo como "?nome=valor&nome2=valor2") e o último envia dados através do cabeçalho HTTP. O que talvez você não saiba é que todas as vezes que carrega uma página (exceto através do envio de um formulário com POST) você está fazendo uma solicitação GET.

Existem outros dois métodos pouco conhecidos: PUT e DELETE. O método DELETE é bastante óbvio - ele instrui o servidor para excluir algo. O método PUT é um pouco mais complicado - funciona de modo parecido ao POST no que diz respeito a enviar dados no cabeçalho HTTP, mas ele foi feito para modificar algo.

Como você deve ter percebido, o método GET tem muito a ver com leitura de dados, POST está relacionado à criação de um novo registro, PUT é o mesmo que atualizar e DELETE faz a exclusão de um registro.

É exatamente assim que o REST funciona no Rails: dependendo do método HTTP o Rails encaminha automaticamente para a ação correspondente dentro do controller: Solicitações com o método GET são encaminhadas para as ações "index" e "show", POST encaminha automaticamente para a ação "create", o método PUT encaminha para a ação "update" e o DELETE encaminha para "destroy"

MAP.RESOURCES

Para criar uma aplicação RESTful (nome que se dá a uma aplicação que utiliza os conceitos de REST), o Rails disponibiliza um recurso de roteamento chamado routes. O sistema de roteamento no Rails é o sistema que examina a URL digitada e determina a ação que deve ser executada.

Quando configurado no arquivo config/routes.rb o map.resources cria novas rotas e helpers configurados para trabalhar com sete ações com nomes padronizados dentro do seu controller: index, show, create, new, update, edit e destroy.

Para configurar nossa aplicação de eventos, abra o arquivo config/routes.rb. Excluindo todos os comentários, o arquivo deve estar assim neste momento:

```
ActionController::Routing::Routes.draw do |map|  
  map.connect ':controller/:action/:id'  
  map.connect ':controller/:action/:id.:format'  
  
end
```

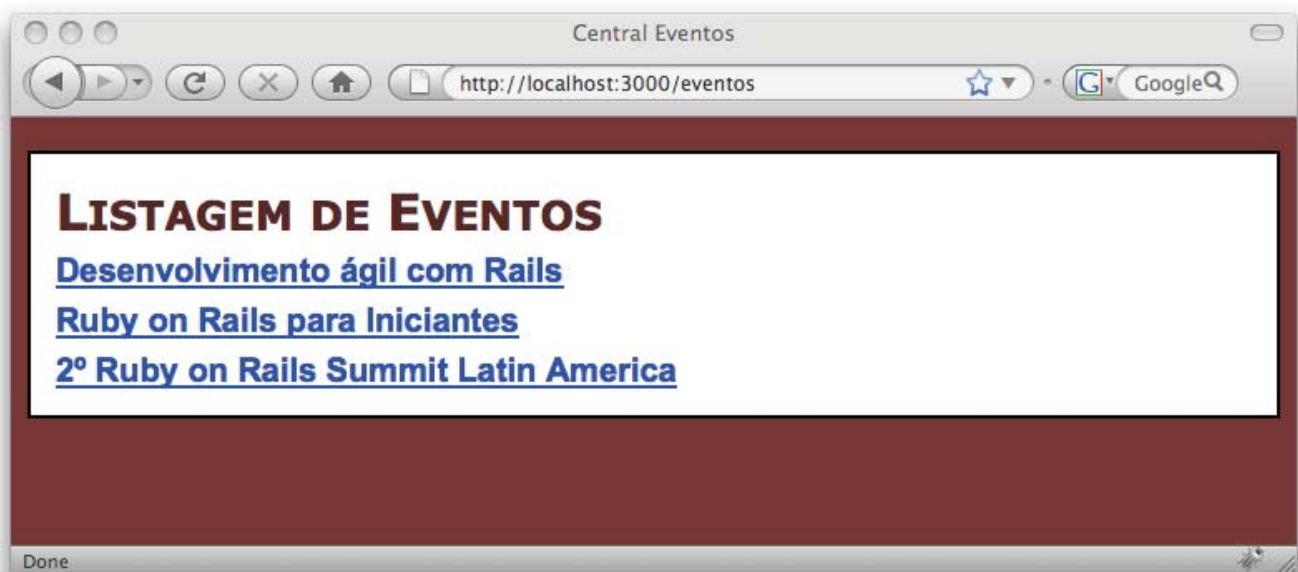
Como queremos que nosso controller “eventos” seja RESTfull, acrescentamos o map.resources para eventos. O arquivo ficará assim

```
ActionController::Routing::Routes.draw do |map|  
  map.resources :eventos  
  map.connect ':controller/:action/:id'  
  map.connect ':controller/:action/:id.:format'  
  
end
```

Como falamos anteriormente, uma aplicação RESTfull em Rails usa sete nomes padronizados de ações nos controllers. Estas sete ações são responsáveis por todas

as manipulações mais comuns que precisam ser feitas em um banco de dados, mas é claro que novas ações customizadas podem ser inseridas conforme a necessidade. Os dois primeiros nomes padronizados que nós já empregamos em nossa aplicação são “index” e “show”

Se você testar novamente a nossa aplicação na URL localhost:3000/eventos, verá a listagem de eventos executada pela ação index



À princípio nada mudou em nossa aplicação (exceto pelo fato de que eu cadastrei novos eventos).

A diferença acontece quando você clica em um dos links: Apenas pela passagem de um parâmetro (o número no final da URL) sem mencionar nenhum nome de ação, você verá os detalhes deste registro fornecidos pela ação show.

Isso acontece porque quando o parâmetro é enviado pela URL, o método HTTP em questão é o GET. Numa aplicação RESTfull parâmetros enviados com o método GET são automaticamente mapeados para a ação show.

Apesar do link continuar funcionando, em uma aplicação RESTful a criação do link entre uma página index e show pode ser facilitado: ao invés de configurar novamente action e id, utilizaremos um atalho que o map.resources criou: `evento_path(id)`. Este atalho aponta automaticamente para o endereço localhost:3000/evento/id.

O novo código completo da view index.html.erb fica assim:

```
<%= link_to evento.titulo, evento_path(evento) %>
```

Repare que dentro dos parênteses, ao invés de colocarmos o número do id, colocamos a variável `evento` inteira. A função `evento_path` localizará automaticamente o id e o link voltará a funcionar.



Relacionamento entre models

Se observarmos novamente os models que geramos para nossa aplicação, veremos que apesar deles não conterem praticamente nenhum código eles possibilitam a interação com os dados do nosso banco de forma muito fácil.

Se acrescentarmos algumas poucas instruções a mais em nossos models podemos expandir a sua funcionalidade de modo a tornar ainda mais prática a manipulação dos dados do banco. Um exemplo são as instruções que estabelecem o relacionamento entre models (e, conseqüentemente entre tabelas) `has_many` e `belongs_to`.

has_many

- Estabelece uma relação de um-para-muitos entre models. Por exemplo, para categoria que temos cadastrada, temos muitos eventos correspondentes.

belongs_to

- Estabelece uma relação de muitos-para-um, de modo inverso ao `has_many`. No nosso caso, cada um dos eventos pertence a uma categoria.

Para relacionar nossos models, abra primeiro o arquivo de modelo das categorias (`models/categoria.rb`) e acrescente a instrução `has_many` indicando para o model de eventos:

```
class Categoria < ActiveRecord::Base
  has_many :eventos
end
```

Faça o mesmo com o model Evento (`models/evento.rb`), que deve pertencer a uma categoria.

```
class Evento < ActiveRecord::Base
  belongs_to :categoria
end
```

Parece pouco, mas nossas tabelas estão relacionadas agora, sem necessidade de digitar nenhuma cláusula SQL nem nada mais complexo.



Dica de Ruby - Símbolos

À princípio o conceito de Símbolos pode parecer estranho e confuso pois trata-se de algo específico de Ruby que você não vai encontrar na maioria das outras linguagens.

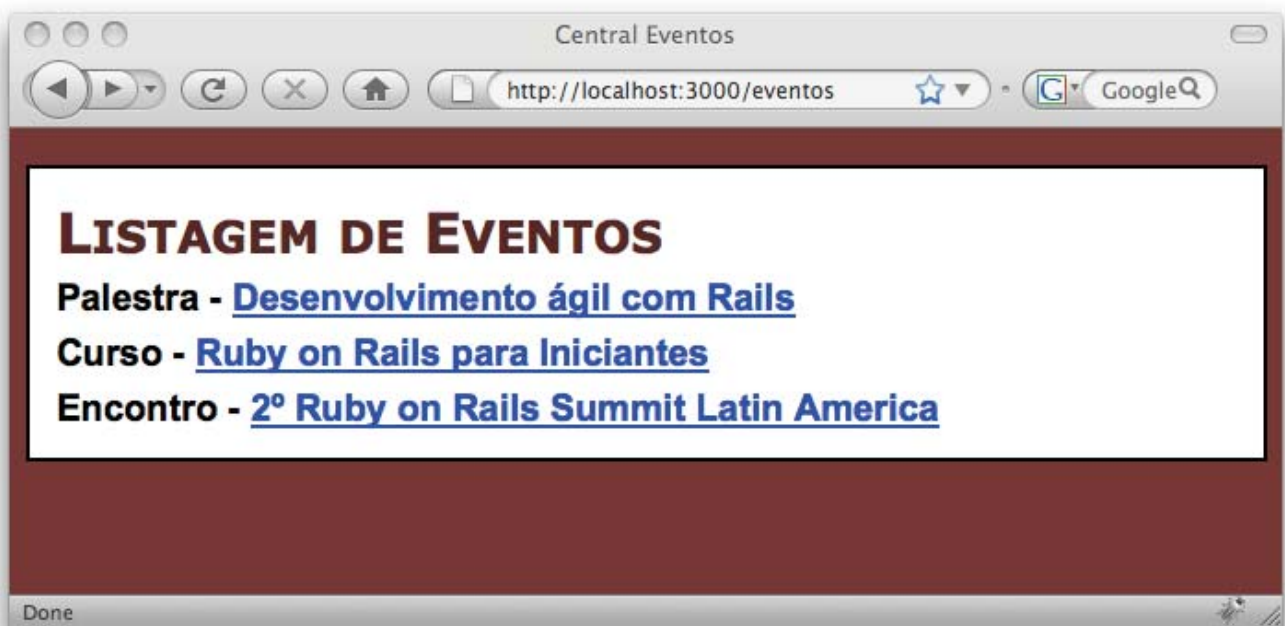
Símbolos são textos iniciados com o sinal de dois pontos “:” - Eles estão presentes por todo o framework Rails (como nos exemplos de relacionamento entre models que fizemos acima em que usamos os símbolos :eventos e :categoria)

Pense em símbolos como “String leves”: Como tudo na linguagem Ruby é um objeto, todas as vezes que você cria um string um novo objeto é instanciado na memória. Os símbolos representam nomes mas não são tão pesadas como strings.

Para testar o que podemos fazer com este relacionamento que acabamos de estabelecer, abra novamente o arquivo view de listagem (views/eventos/index.html.erb) e acrescente o seguinte código:

```
<h1>Listagem de Eventos</h1>
<% for evento in @eventos %>
  <h2>
    <%= evento.categoria.nome_categoria %> -
    <%= link_to evento.titulo, evento_path(evento) %>
  </h2>
<% end %>
```

Automaticamente ele consegue resgatar o nome da categoria atrelada a cada evento pelo campo categoria_id.



Inserção, edição e exclusão de registros

Até agora contruímos as páginas que exibem os eventos cadastrados em nosso sistema, mas não criamos um modo para manipular tais dados. Através de Ruby on Rails é fácil criar um sistema CRUD (**C**reate, **R**ead, **U**ppdate, **D**elete) completo.

Para fazer a inserção dos dados, vamos começar criando uma nova ação chamada “new” no controller eventos. Essa ação terá uma programação muito simples que cria uma nova instância do model “Evento”. Além disso, seu propósito será exibir o arquivo de visualização `new.html.erb` que conterá um formulário de cadastro. Nosso controller, portanto, ficará assim:

```
class EventosController < ApplicationController

  def index
    @eventos = Evento.find(:all)
  end

  def show
    @evento = Evento.find(params[:id])
  end

  def new
    @evento = Evento.new
  end

end
```

Crie agora um novo arquivo chamado `new.html.erb` na pasta `views/` `eventos`. Vamos acrescentar um formulário de inserção de dados neste documento através do helper `form_for`

HELPERS PARA FORMULÁRIOS (FORM_FOR)

Um dos recursos mais facilitadores do Rails são os Helpers. Como o próprio nome já diz, os helpers ajudam na cansativa tarefa de criar elementos html como formulários, por exemplo.

O helper "form_for" permite criar um formulário completo atrelado a uma instância de algum modelo. No nosso exemplo, criaremos um formulário atrelado à instância @evento do nosso model "Evento". O código ficará assim:

```
<h1>Cadastrar novo evento</h1>

<% form_for(@evento) do |f| %>

<% end %>
```

Esses comandos apenas criam um formulário, sem nenhum campo para pessoa preencher. Através dos métodos `text_field`, `text_area`, `check_box`, `radio_button`, `password_field`, `select`, `datetime_select`, `submit` e outros, podemos colocar os campos para a pessoa preencher:

```
<h1>Cadastrar novo Evento</h1>

<% form_for(@evento) do |f| %>
  <h4>Titulo</h4>
  <%= f.text_field :titulo %>

  <h4>Descrição</h4>
  <%= f.text_area :descricao %>

  <h4>Categoria</h4>
  <%= f.select :categoria_id, Categoria.find(:all).collect { |
item| [item.nome_categoria,item.id] } %>

  <h4>Data</h4>
  <%= f.date_select(:data, :order => [:day, :month, :year]) %>

  <br />
  <%= f.submit "Cadastrar" %>

<% end %>
```

Observe que via de regra utilizamos o método do formulário seguido de um símbolo (o nome iniciando com dois pontos), que corresponde ao nome utilizado no banco de dados.

As excessões são os campos do tipo `select` – `f.select` e `f.date_select`.

No primeiro caso, além de especificar o nome do campo, fazemos uma pesquisa de todos os registros do model `Categoria` e utilizamos o `collect` para estabelecer quais dados serão utilizados como `label` (valor exibido para o usuário) e `value` (valor enviado para o servidor) deste menu.

No caso do `date_select`, além do nome especificamos também a ordem em que os menus drop-down devem aparecer. Sem esta opção, os menus seriam exibidos no formato americano, com o ano primeiro.

Nosso formulário está pronto mas não existe nenhum link que leve até ele. Vamos acrescentar tal link na página de listagem. Até o presente momento o arquivo `index.html.erb` deverá estar assim:

```
<h1>Listagem de Eventos</h1>
<% for evento in @eventos %>
  <h2>
    <%= evento.categoria.nome_categoria %> -
    <%= link_to evento.titulo, evento_path(evento) %>

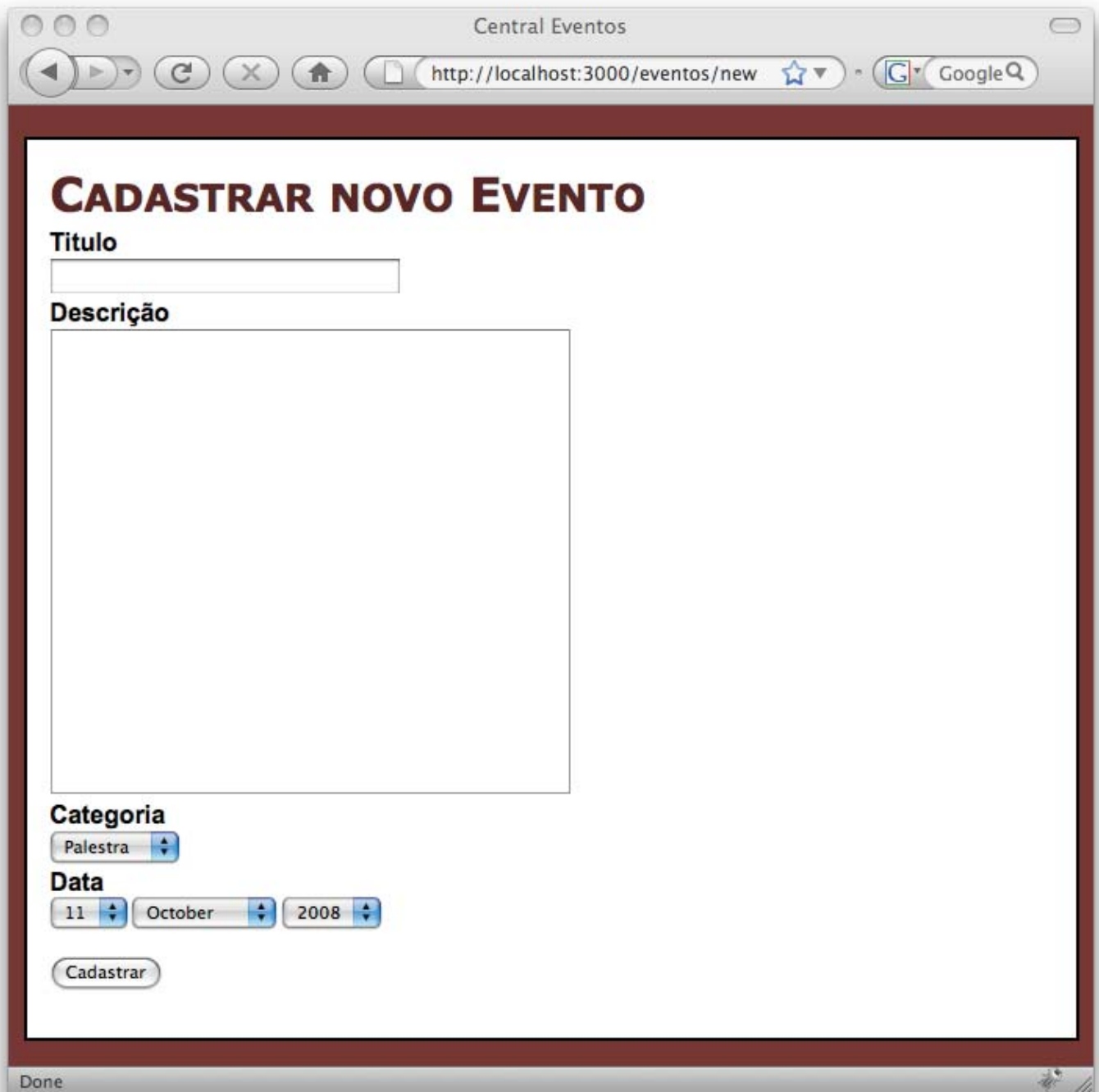
  </h2>
<% end %>
```

Vamos acrescentar um link na última linha utilizando o atalho `new_post_path` que foi gerado pelo routes.

```
<h1>Listagem de Eventos</h1>
<% for evento in @eventos %>
  <h2>
    <%= evento.categoria.nome_categoria %> -
    <%= link_to evento.titulo, evento_path(evento) %>
  </h2>
<% end %>

<br />
<p><%= link_to 'Cadastrar novo evento', new_evento_path %></p>
```

Se você testar agora e clicar no link para cadastrar um novo evento você verá o formulário pronto:



The screenshot shows a web browser window titled "Central Eventos". The address bar displays "http://localhost:3000/eventos/new". The page content is a form titled "CADASTRAR NOVO EVENTO". The form includes a "Titulo" field, a "Descrição" text area, a "Categoria" dropdown menu with "Palestra" selected, and a "Data" section with three dropdown menus for day, month, and year (11, October, 2008). A "Cadastrar" button is at the bottom.

CADASTRAR NOVO EVENTO

Titulo

Descrição

Categoria

Palestra

Data

11 October 2008

Cadastrar

INSERÇÃO DE DADOS

Ao clicar no botão de enviar, o formulário vai enviar as informações para o controller “eventos” através do método POST. Como nossa aplicação está sendo construída com REST, o Rails automaticamente mapeia este envio de dados por POST para uma ação “create”.

A ação create deve conter uma instrução `Evento.new` que recebe os parâmetros vindos do formulário.

Na sequencia, ele verifica se os dados foram registrados no banco e, em caso positivo, abre a página que mostra os dados do novo evento recém-criado.

```
class EventosController < ApplicationController
  def index
    @eventos = Evento.find(:all)
  end

  def show
    @evento = Evento.find(params[:id])
  end

  def new
    @evento = Evento.new
  end

  def create
    @evento = Evento.new(params[:evento])
    if @evento.save
      flash[:aviso] = 'Evento cadastrado com sucesso.'
      redirect_to(@evento)
    end
  end
end
```

Repare ainda que adicionamos uma linha `flash[:aviso] = 'Evento cadastrado com sucesso'`.

O flash armazena temporariamente uma informação. Por “temporariamente” entenda: Até que a página seja redirecionada, depois a informação se perde. O flash é usado para transmitir pequenas mensagens de status entre ações e views.

Para exibir os dados do flash, abra o arquivo de layout `application.html.erb` e acrescente os seguintes códigos:

```
<html>
<head>
  <title>Central Eventos</title>
  <%= stylesheet_link_tag "default" %>
</head>

<body>
<div id="principal">

  <% unless flash[:aviso].blank? %>
    <div id="aviso"><%= flash[:aviso] %></div>
  <% end %>

  <%= yield %>
</div>
</body>
</html>
```



Dica de Ruby - Estruturas condicionais

A linguagem Ruby trabalha com diversos tipos de estruturas condicionais - blocos de código que só são executados dependendo de certas condições.

A estrutura `if...end` é um exemplo disso. Na ação `Create`, utilizamos esta estrutura para verificar se os dados foram salvos, e somente em caso positivo a página é redirecionada.

A estrutura `unless...end` funciona mais ou menos como um `if` ao contrário, ou seja, o bloco de códigos sempre será executado, a não ser que a condição estipulada seja verdadeira. No arquivo de layout, a estrutura `unless` faz com que o `div` “aviso” seja criado sempre, a não ser que o flash não contenha nenhuma informação.

Vamos também acrescentar algumas instruções CSS para configurar o visual deste `div` “aviso”. Acrescente no arquivo `public/stylesheets/default.css` o seguinte:

```
#aviso{
  font-family:Arial, Helvetica, sans-serif;
  font-size:13px;
  font-weight:bold;
  color:#000000;
  border: 5px solid #990000;
  background-color: #ff9999;
  padding: 5px;
  margin: 10px 0;
}
```

Se você tentar inserir um novo evento você verá a mensagem do flash na parte superior da página:



EDIÇÃO DE DADOS

Assim como a inserção de novos registros, precisaremos criar duas ações no controller eventos para fazer a atualização de dados. A primeira ação apenas mostra o formulário, e a segunda efetivamente o processa fazendo as alterações cabíveis no banco.

Vamos começar criando a ação Edit no controller. Diferentemente da ação new que instanciava um novo objeto vazio do Model Evento, precisaremos localizar o registro específico que o usuário deseja alterar. Fazemos isso da mesma forma que a ação show:

```
class EventosController < ApplicationController
  def index
    @eventos = Evento.find(:all)
  end
  def show
    @evento = Evento.find(params[:id])
  end

  def new
    @evento = Evento.new
  end

  def create
    @evento = Evento.new(params[:evento])
    if @evento.save
      flash[:aviso] = 'Evento cadastrado com sucesso.'
      redirect_to(@evento)
    end
  end

  def edit
    @evento = Evento.find(params[:id])
  end
end
```

Crie também um novo arquivo de view chamado “edit.html.erb”. Ele conterá um formulário criado através do helper `form_for` idêntico ao formulário criado para a view `new.html.erb`:

```
<h1>Editar Evento</h1>
<% form_for(@evento) do |f| %>
  <h4>Titulo</h4>
  <%= f.text_field :titulo %>

  <h4>Descrição</h4>
  <%= f.text_area :descricao %>

  <h4>Categoria</h4>
  <%=
f.select :categoria_id,Categoria.find(:all).collect{ |item|
[item.nome_categoria,item.id] } %>

  <h4>Data</h4>
  <%= f.date_select(:data, :order =>
[:day, :month, :year]) %>
  <br />

  <%= f.submit "Editar" %>

<% end %>
```

Se queremos que o usuário possa editar os eventos, precisamos colocar um link à disposição para este propósito. Para tanto, abra o arquivo de view `views/eventos/show.html.erb` e acrescente o link através do `link_to`:

```
<h2><%= @evento.titulo %></h2>
<h3>Descrição</h3>
<p><%= @evento.descricao %></p>
<h4>Data do evento: <%= @evento.data.strftime("%d/%m/%Y") %></h4>

<p><%= link_to 'Editar', edit_evento_path %></p>
```

Teste escolhendo um dos eventos cadastrados e, na sequência, clicando em editar.

O resultado será:



The screenshot shows a web browser window titled 'Central Eventos'. The address bar displays 'http://localhost:3000/eventos/1/edit'. The page content is a form for editing an event, titled 'EDITAR EVENTO' in large, bold, dark red letters. Below the title, there are three main sections: 'Titulo', 'Descrição', and 'Categoria'. The 'Titulo' section has a text input field containing 'Desenvolvimento ágil com Rails'. The 'Descrição' section has a large text area containing the text 'Palestra sobre desenvolvimento agil de aplicacoes para a web'. The 'Categoria' section has a dropdown menu with 'Palestra' selected. Below these sections, there is a 'Data' section with three dropdown menus for day, month, and year, showing '1', 'December', and '2008' respectively. At the bottom of the form is an 'Editar' button. The browser's status bar at the bottom shows 'Done'.

Central Eventos

http://localhost:3000/eventos/1/edit

EDITAR EVENTO

Titulo

Desenvolvimento ágil com Rails

Descrição

Palestra sobre desenvolvimento agil de aplicacoes para a web

Categoria

Palestra

Data

1 December 2008

Editar

Done

O código fonte do formulário inclui o seguinte trecho:

```
<form action="/eventos/1" class="edit_evento"
id="edit_evento_1" method="post">
<input name="_method" type="hidden" value="put" />
```

O formulário está utilizando o método POST, porém veja que logo na sequência o Rails criou um campo invisível (“hidden”) onde passa a informação de método PUT. Ele fez isso para contornar uma limitação dos navegadores tradicionais que não suportam o método HTTP PUT. Como já havíamos visto na seção REST deste guia, sabemos que o método PUT está relacionado à edição de dados, e automaticamente o Rails encaminha dados enviados por PUT para a ação update. Vamos portanto criar esta ação:

```
class EventosController < ApplicationController
  def index
    @eventos = Evento.find(:all)
  end

  def show
    @evento = Evento.find(params[:id])
  end

  def new
    @evento = Evento.new
  end

  def create
    @evento = Evento.new(params[:evento])
    if @evento.save
      flash[:aviso] = 'Evento cadastrado com sucesso.'
      redirect_to(@evento)
    end
  end

  def edit
    @evento = Evento.find(params[:id])
  end
end
```

```

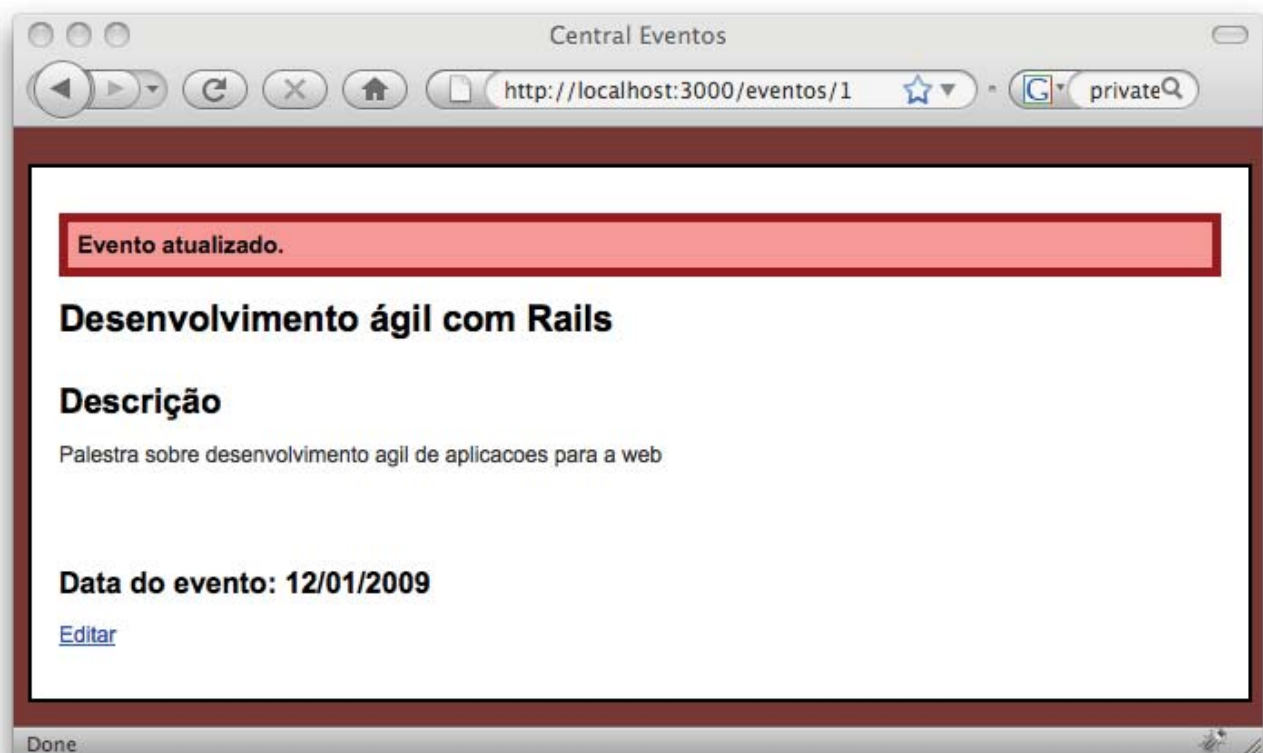
def update
  @evento = Evento.find(params[:id])
  if @evento.update_attributes(params[:evento])
    flash[:aviso] = 'Evento atualizado.'
    redirect_to(@evento)
  end
end
end
end

```

A ação update primeiro localiza o evento em questão através do id que foi enviado pelo formulário.

Na sequência ele faz a atualização e, se a atualização foi executada com sucesso, cria um flash e redireciona para a ação show para exibir o elemento recém-editado.

Se você tentar editar algum elemento, verá algo como:



EXCLUSÃO DE DADOS

Para a exclusão de dados, precisaremos de uma única ação no controller chamada “destroy”, além de um link na página de exibição. Vamos começar pelo link, acrescentando um novo link_to no arquivo views/eventos/show.html.erb

```
<h2><%= @evento.titulo %></h2>
<h3>Descrição</h3>
<p><%= @evento.descricao %></p>
<h4>Data do evento: <%= @evento.data.strftime("%d/%m/%Y") %></h4>
<p><%= link_to 'Editar', edit_evento_path %></p>

<p><%= link_to 'Excluir', evento_path, :confirm => 'Confirma exclusão?', :method => :delete %></p>
```

Assim como no link anterior, preenchemos o nosso novo link_to com dois parâmetros iniciais que representam o texto e a url do link. Nesta caso, entretanto, existem outros dois parâmetros na sequência, o primeiro (:confirm) exibe uma mensagem de confirmação antes de enviar o link, já o segundo estipula que o método HTTP deverá ser DELETE ao invés do GET que seria padrão.

Como vimos no nosso estudo sobre REST, o método HTTP DELETE está relacionado à exclusão de dados e é mapeado automaticamente para a ação “destroy”.

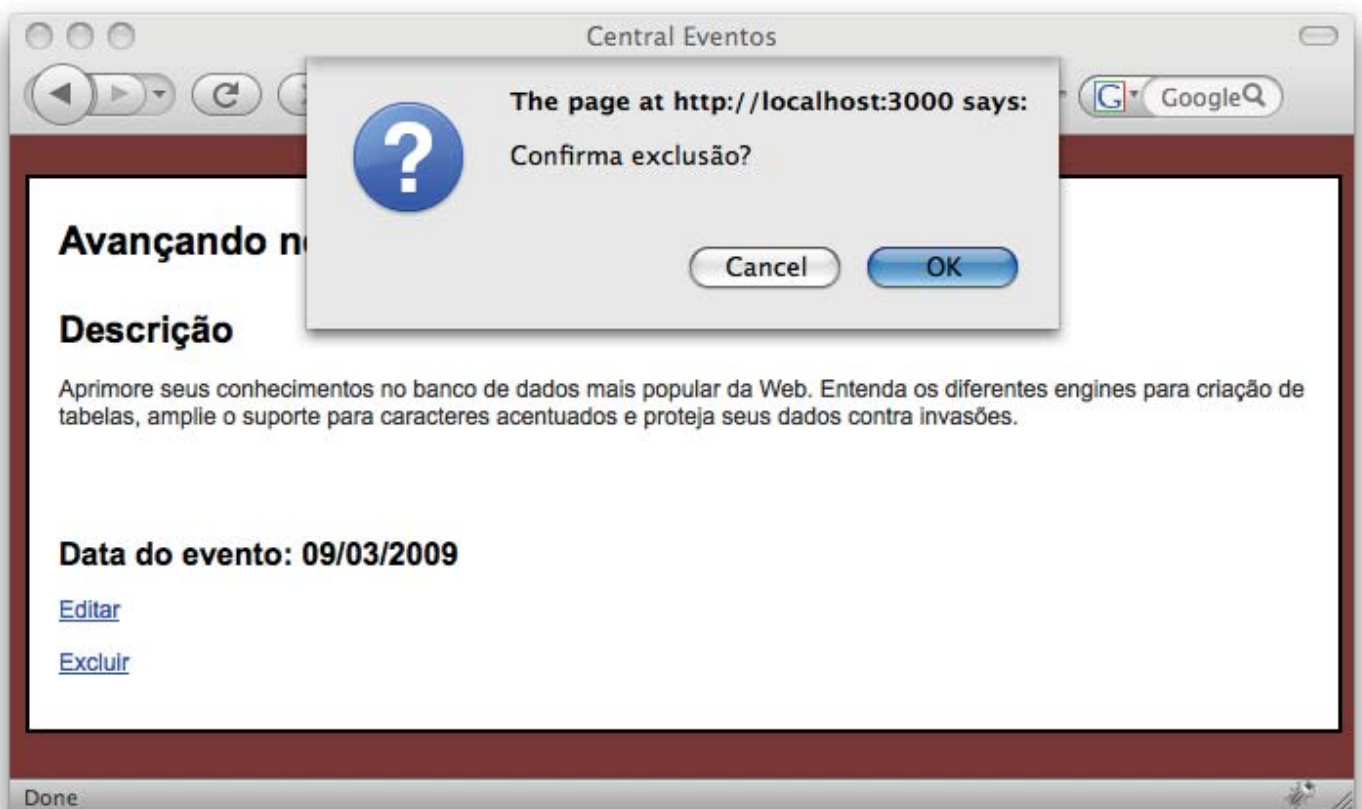
Vamos abrir novamente o nosso controller “eventos” para criar nossa ação “destroy” que irá excluir o elemento do banco:

```
class EventosController < ApplicationController
  def index
    @eventos = Evento.find(:all)
  end
  def show
    @evento = Evento.find(params[:id])
  end
  def new
    @evento = Evento.new
  end
  def create
    @evento = Evento.new(params[:evento])
    if @evento.save
      flash[:aviso] = 'Evento cadastrado com sucesso.'
      redirect_to(@evento)
    end
  end
  def edit
    @evento = Evento.find(params[:id])
  end
  def update
    @evento = Evento.find(params[:id])
    if @evento.update_attributes(params[:evento])
      flash[:aviso] = 'Evento atualizado.'
      redirect_to(@evento)
    end
  end
  def destroy
    @evento = Evento.find(params[:id])
    @evento.destroy
    flash[:aviso] = "Evento excluído com sucesso"
    redirect_to(eventos_path)
  end
end
```

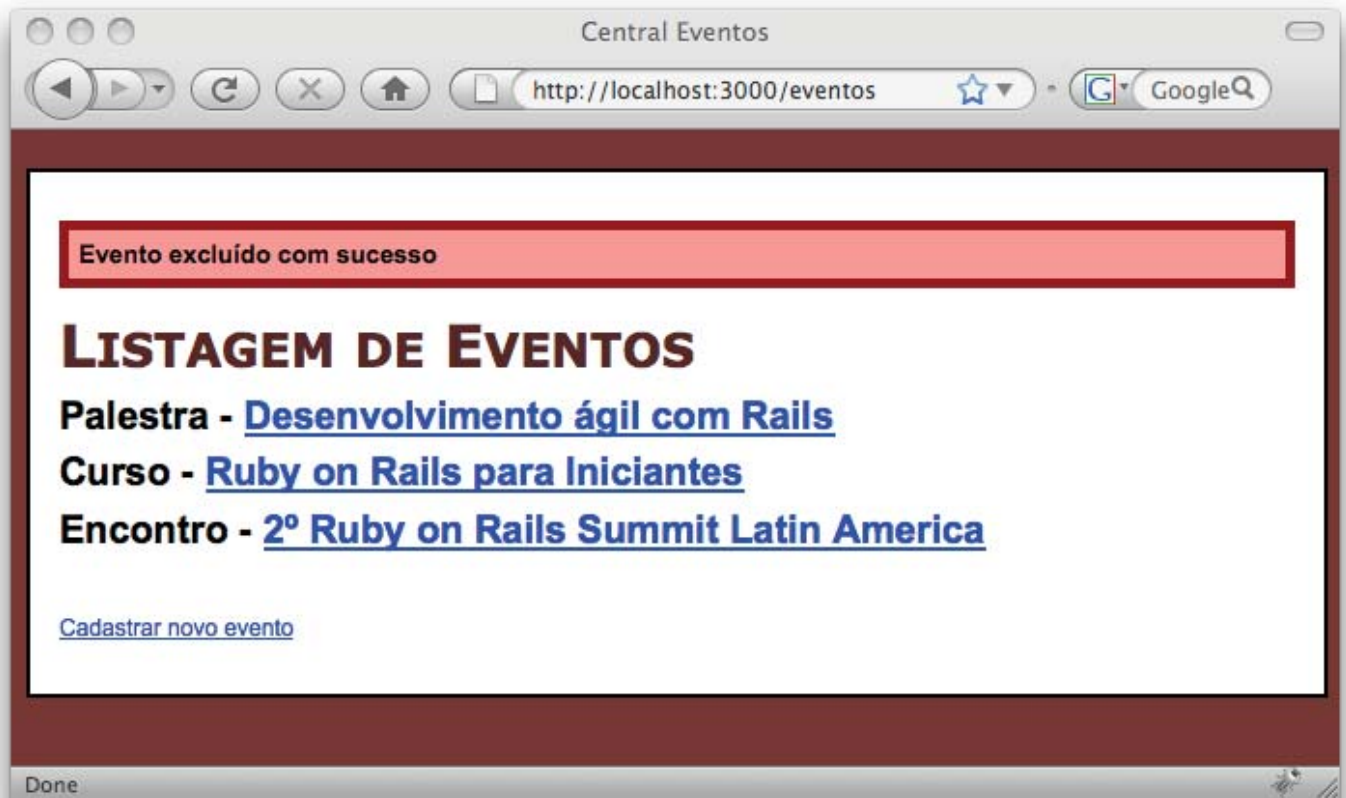
Em primeiro lugar, a ação destroy localiza o elemento a ser excluído através do parâmetro que foi enviado pelo link.

Na sequência ele efetivamente exclui o registro através do método @evento.destroy e, finalmente, cria uma nova mensagem temporária e redireciona para a listagem de eventos.

Ao tentar excluir algum evento você vai se deparar com uma mensagem de alerta como essa:



Após confirmar, o usuário é redirecionado para a página de listagem com a mensagem temporária de “evento excluído com sucesso”:



VALIDAÇÕES

Validações são um modo de garantir a integridade dos dados em uma aplicação.

O modelo de validações que o Rails fornece é bastante completo e pode ser facilmente expandido pelo desenvolvedor caso ele necessite.

No caso da nossa classe de dados inicial, precisamos validar pelo menos o fato do usuário ter informado o título do evento ao cadastrá-lo.

Para isso, vamos abrir o arquivo do model, que está em `model/evento.rb`, e editá-lo, inserindo uma validação simples:

```
class Evento < ActiveRecord::Base
  belongs_to :categoria
  validates_presence_of :titulo
end
```

Se você tentar inserir agora um novo evento sem título, verá o seguinte:



Central Eventos

http://localhost:3000/eventos/new

CADASTRAR NOVO EVENTO

Titulo

Descrição

Tentando cadastrar um novo evento sem titulo

Categoria

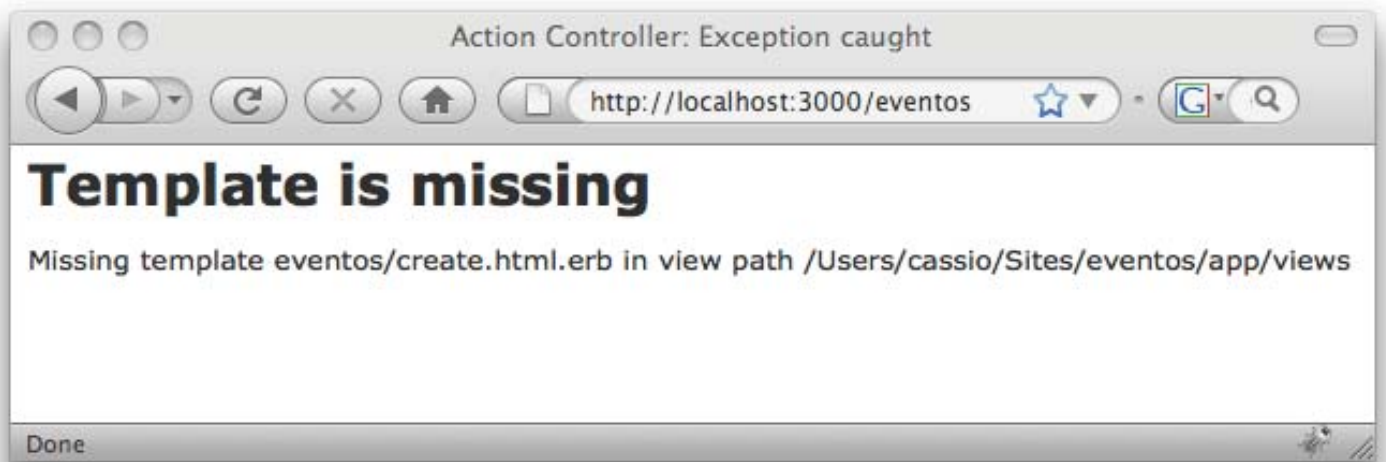
Curso

Data

9 December 2008

Cadastrar

Done



Para entender porque isso aconteceu, vamos olhar novamente a ação create do nosso controller eventos:

```
def create
  @evento = Evento.new(params[:evento])
  if @evento.save
    flash[:aviso] = 'Evento cadastrado com sucesso.'
    redirect_to(@evento)
  end
end
```

Perceba que ao não passar no critério de validação devido à falta de um título, o `@evento.save` não é executado. Como consequencia, a página não é redirecionada para exibir o evento.

Vamos resolver este problema acrescentando um else nesta estrutura:

```
def create
  @evento = Evento.new(params[:evento])
  if @evento.save
    flash[:aviso] = 'Evento cadastrado com sucesso.'
    redirect_to(@evento)
  else
    flash[:aviso] = 'O formulário contém erros. Verifique os campos digitados e envie novamente.'
    render :action => "new"
  end
end
```

Este else será executado caso o novo evento não possa ser salvo. Ele determina que neste caso seja criada uma informação temporária do tipo flash e redireciona de volta para a ação “new” (nosso formulário de cadastro).

Ao tentar cadastrar novamente um evento sem título a página retornada fica assim:

Central Eventos

http://localhost:3000/eventos/new

O formulário contém erros. Verifique os campos digitados e envie novamente.

CADASTRAR NOVO EVENTO

Titulo

Descrição

Tentando cadastrar um novo evento sem título

Categoria

Curso

Data

9 December 2008

Cadastrar

Done

Além da mensagem temporária, não parece que nada mudou. Mas houve alterações que você pode observar neste trecho código-fonte gerado:

```
<form action="/eventos" class="new_evento" id="new_evento"
method="post">
  <h4>Titulo</h4>
  <div class="fieldWithErrors">
    <input id="evento_titulo" name="evento[titulo]" size="30"
type="text" value="" />
  </div>

  <h4>Descrição</h4>
  <textarea cols="40" id="evento_descricao"
name="evento[descricao]" rows="20">
    Tentando cadastrar um evento sem título
  </textarea>
```

Repare que o Rails colocou automaticamente um novo div com a class “fieldWithErrors” ao redor do campo que falhou na validação.

Para darmos um retorno melhor para o usuário, basta acrescentar esta classe no nosso arquivo de estilo public/stylesheets/default.css, conforme o exemplo:

```
.fieldWithErrors{
  background-color:#ff0000;
  padding:3px;
  display:table;
}
```

Agora, além da mensagem temporária de erro, o usuário sabe claramente qual campo falhou na validação:

The screenshot shows a web browser window titled 'Central Eventos' with the address bar displaying 'http://localhost:3000/eventos'. A red error message banner at the top reads: 'O formulário contém erros. Verifique os campos digitados e envie novamente.' Below this, the form is titled 'CADASTRAR NOVO EVENTO'. It contains three main sections: 'Titulo' with a text input field that has a red border indicating an error; 'Descrição' with a text area containing the placeholder text 'Tentando cadastrar um novo evento sem titulo'; and 'Categoria' with a dropdown menu showing 'Palestra'. Below these is a 'Data' section with three dropdown menus for day, month, and year, showing '9', 'December', and '2008' respectively. At the bottom of the form is an 'Enviar' button. The browser's status bar at the bottom shows 'Done'.

Se você quiser, poderia ainda incluir um comando `<%= error_messages_for :evento %>` no início da view `new.html.erb` para que a lista de erros no formulário seja detalhada por escrito na página. Para este nosso exemplo não precisaremos detalhar os erros de preenchimento por texto, a mensagem temporária de erro e o feedback na borda colorida já são suficientes para alertar o usuário.

Múltiplas validações podem ser efetuadas em um mesmo campo. Por exemplo, para prevenir a inserção de títulos duplicados, a seguinte condição poderia ser colocada:

```
class Evento < ActiveRecord::Base
  belongs_to :categoria
  validates_presence_of :titulo
  validates_uniqueness_of :titulo
end
```

Scaffolding

Até agora, criamos todas as principais operações - Conseguimos listar, criar, editar e excluir informações de eventos. Apesar de ser interessante fazer toda essa programação pela primeira vez, este processo se repete para cada novo model de cada nova aplicação que você criar em Rails. Recriar as operações básicas de novo e de novo pode se tornar muito tedioso, e é exatamente por este motivo que existe o recurso de *scaffolding*.

O *scaffolding* cria um esqueleto com toda uma estrutura básica para operações CRUD (*Create, Retrieve, Update and Delete*), e que você pode posteriormente incrementar e modificar conforme a sua necessidade. Ou seja, todas as ações e views que programamos manualmente para os eventos serão criadas automaticamente.

A sintaxe para criação de um Scaffold é:

```
script/generate scaffold nome_do_modelo_de_dados  
campos_da_tabela
```

Na nova versão do Rails, o próprio Scaffold cria o model, o arquivo de migração, o controller e as views automaticamente.

Para criar o scaffold para as categorias, utilizaríamos o comando:

```
script/generate scaffold categoria nome_categoria:string
```

Entretanto, como nós já havíamos criado o modelo e o arquivo de migração das categorias anteriormente, para fins de aprendizado, precisamos instruir o Rails para criar apenas o controller e as views do scaffold - sem recriar o arquivo de migrations. Isso pode ser feito acrescentando a opção `--skip-migration`:

```
script/generate scaffold categoria nome_categoria:string --skip-migration
```

```
exists app/models/
exists app/controllers/
exists app/helpers/
create app/views/categorias
exists app/views/layouts/
exists test/functional/
exists test/unit/
exists public/stylesheets/
create app/views/categorias/index.html.erb
create app/views/categorias/show.html.erb
create app/views/categorias/new.html.erb
create app/views/categorias/edit.html.erb
create app/views/layouts/categorias.html.erb
create public/stylesheets/scaffold.css
create app/controllers/categorias_controller.rb
create test/functional/categorias_controller_test.rb
create app/helpers/categorias_helper.rb
route map.resources :categorias
dependency model
exists app/models/
exists test/unit/
exists test/fixtures/
identical app/models/categoria.rb
identical test/unit/categoria_test.rb
skip test/fixtures/categorias.yml
```

Note que, para a geração de um *scaffold* por esse comando, o nome do modelo de dados é que deve ser passado como parâmetro, e não o nome do *controller*. O Rails é capaz de derivar um do outro. Além disso, é necessário passar também como parâmetro os campos do banco de dados separados apenas por espaços. No nosso caso, apenas o `nome_categoria`, do tipo `string`.

Esse comando gera a classe do modelo de dados, o *controller* e *views* para cada ação CRUD necessária. No caso acima, como o model já existia, ele não foi criado, e foi necessário sobrescrever alguns arquivos.

O *scaffold* gerou o seu próprio *layout*, com sua própria *stylesheet*. Por causa disso, o layout original da aplicação foi perdido. Uma solução aqui será remover o arquivo `app/views/layouts/categorias.html.erb`, já que não precisamos dele, e adicionar a *stylesheet* `scaffold.css` ao arquivo `application.html.erb`.

ENTENDENDO OS ARQUIVOS GERADOS PELO SCAFFOLD:

O CONTROLLER CATEGORIAS

Como você poderá observar, o controller gerado pelo scaffold é muito parecido com o controller de eventos que criamos manualmente, à exceção de alguns comentários que ele insere automaticamente para cada ação e do `respond_to`, que permite retornar para o usuário documentos em formatos diversos além do html.

```
class CategoriasController < ApplicationController
  # GET /categorias
  # GET /categorias.xml
  def index
    @categorias = Categoria.find(:all)
    respond_to do |format|
      format.html # index.html.erb
      format.xml { render :xml => @categorias }
    end
  end

  # GET /categorias/1
  # GET /categorias/1.xml
  def show
    @categoria = Categoria.find(params[:id])
    respond_to do |format|
      format.html # show.html.erb
      format.xml { render :xml => @categoria }
    end
  end

  # GET /categorias/new
  # GET /categorias/new.xml
  def new
    @categoria = Categoria.new
    respond_to do |format|
      format.html # new.html.erb
      format.xml { render :xml => @categoria }
    end
  end
end
```

```

# GET /categorias/1/edit
def edit
  @categoria = Categoria.find(params[:id])
End

# POST /categorias
# POST /categorias.xml
def create
  @categoria = Categoria.new(params[:categoria])
  respond_to do |format|
    if @categoria.save
      flash[:notice] = 'Categoria was successfully created.'
      format.html { redirect_to(@categoria) }
      format.xml { render :xml => @categoria, :status
=> :created, :location => @categoria }
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @categoria.errors, :status
=> :unprocessable_entity }
    end
  end
end

# PUT /categorias/1
# PUT /categorias/1.xml
def update
  @categoria = Categoria.find(params[:id])
  respond_to do |format|
    if @categoria.update_attributes(params[:categoria])
      flash[:notice] = 'Categoria was successfully updated.'
      format.html { redirect_to(@categoria) }
      format.xml { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml { render :xml => @categoria.errors, :status
=> :unprocessable_entity }
    end
  end
end

```

```
# DELETE /categorias/1
# DELETE /categorias/1.xml
def destroy
  @categoria = Categoria.find(params[:id])
  @categoria.destroy
  respond_to do |format|
    format.html { redirect_to(categorias_url) }
    format.xml  { head :ok }
  end
end
end

end
```

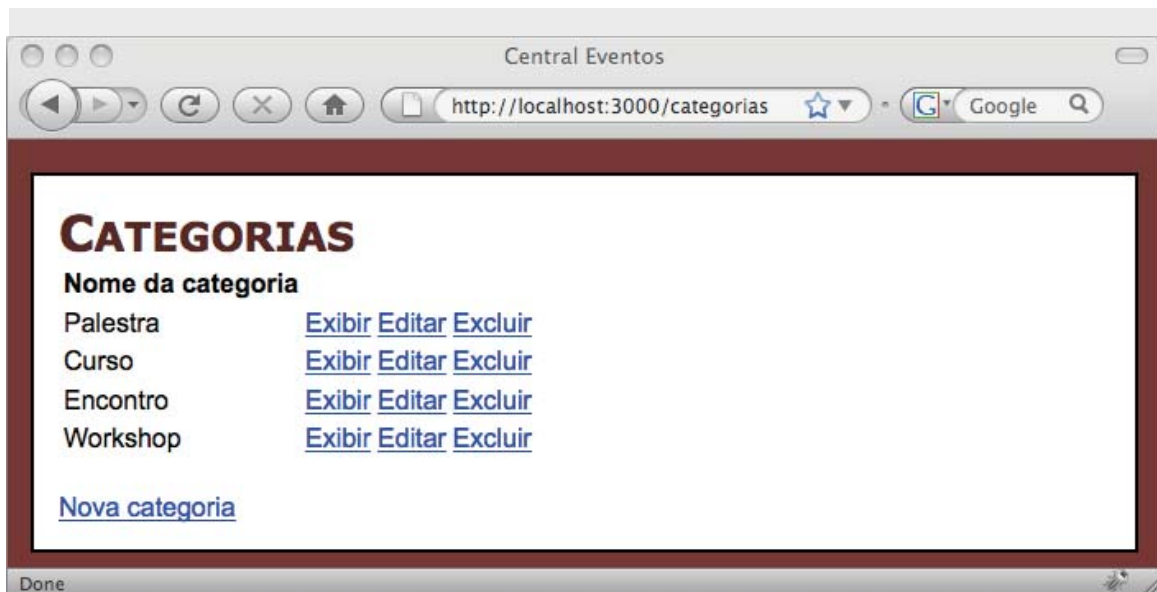
Como é possível observar, as ações deste controller foram criadas num modelo RESTfull. O Scaffold até gerou um roteamento para as categorias no arquivo config/routes.rb

ENTENDENDO OS ARQUIVOS GERADOS PELO SCAFFOLD:

As Views

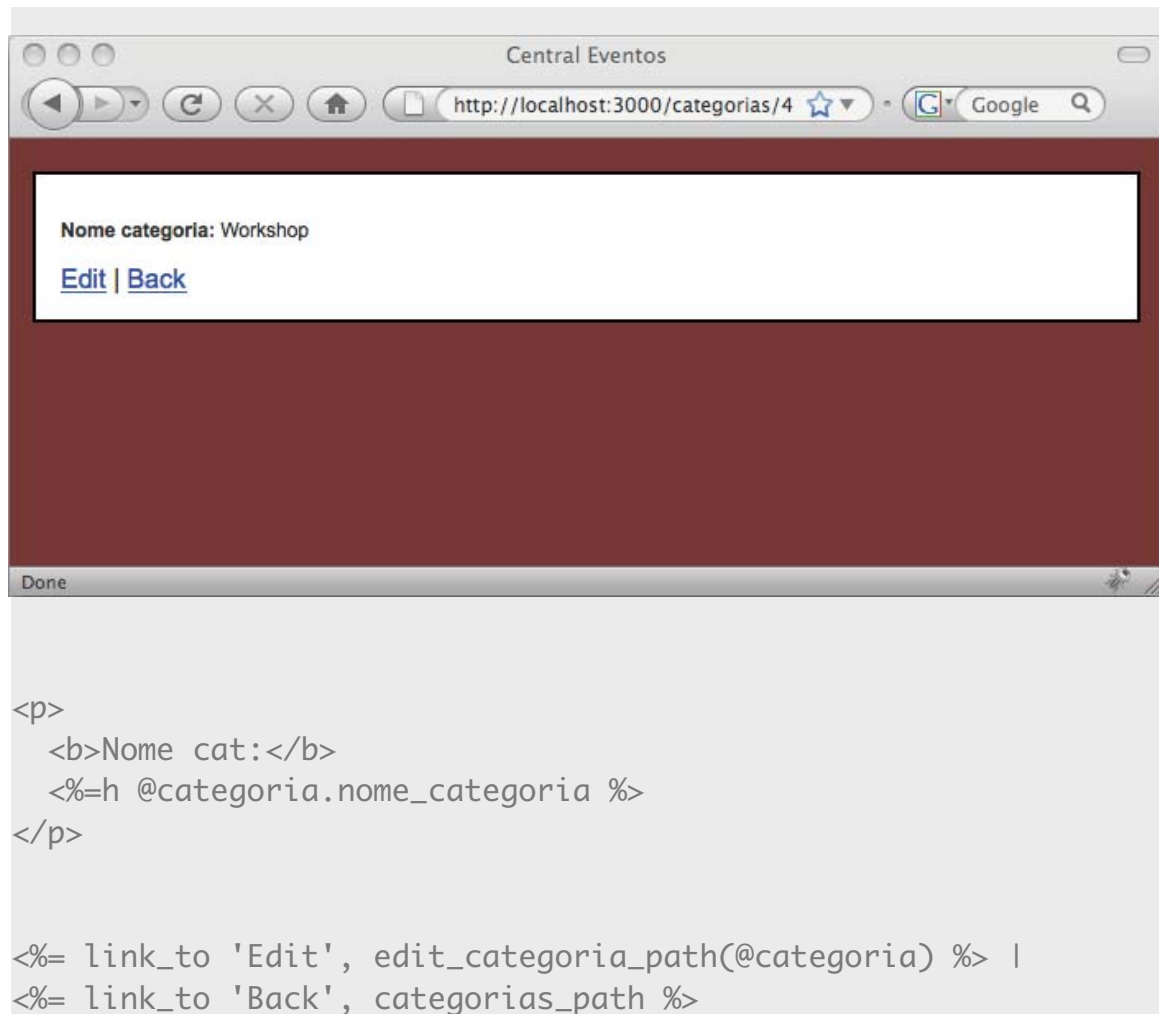
Vamos acompanhar cada uma das ações e suas respectivas views geradas automaticamente pelo scaffold.

Index

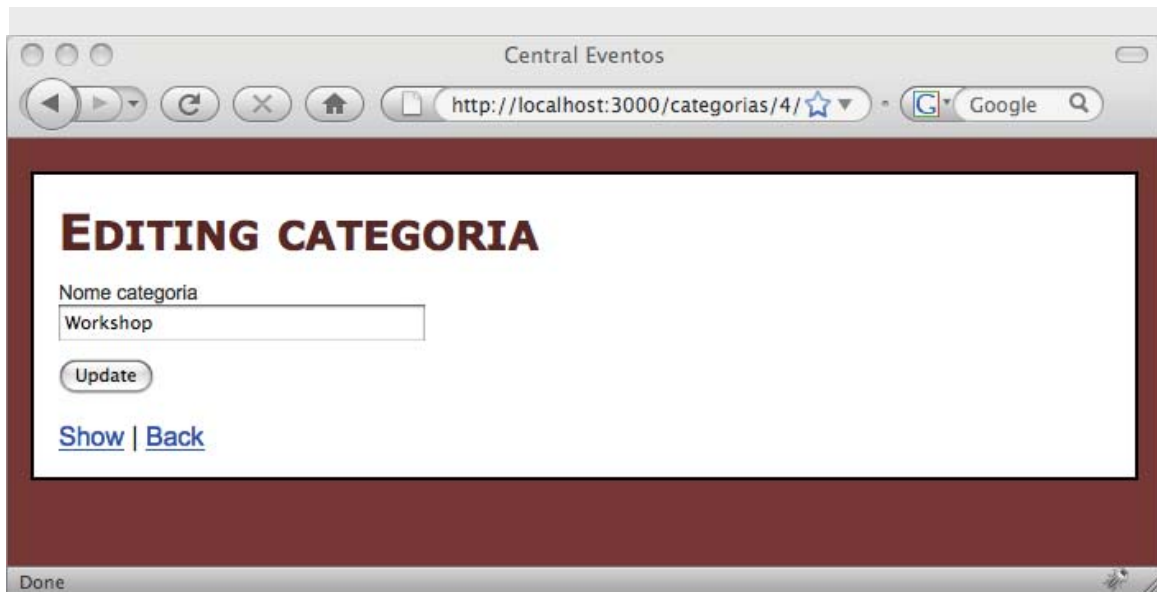


```
<h1>Listing categorias</h1>
<table>
  <tr>
    <th>Nome cat</th>
  </tr>
  <% for categoria in @categorias %>
    <tr>
      <td><%=h categoria.nome_categoria %></td>
      <td><%= link_to 'Show', categoria %></td>
      <td><%= link_to 'Edit', edit_categoria_path(categoria) %></td>
      <td><%= link_to 'Destroy', categoria, :confirm => 'Are you
sure?', :method => :delete %></td>
    </tr>
  <% end %>
</table>
<%= link_to 'New categoria', new_categoria_path %>
```

Show



Edit



```
<h1>Editing categoria</h1>

<%= error_messages_for :categoria %>

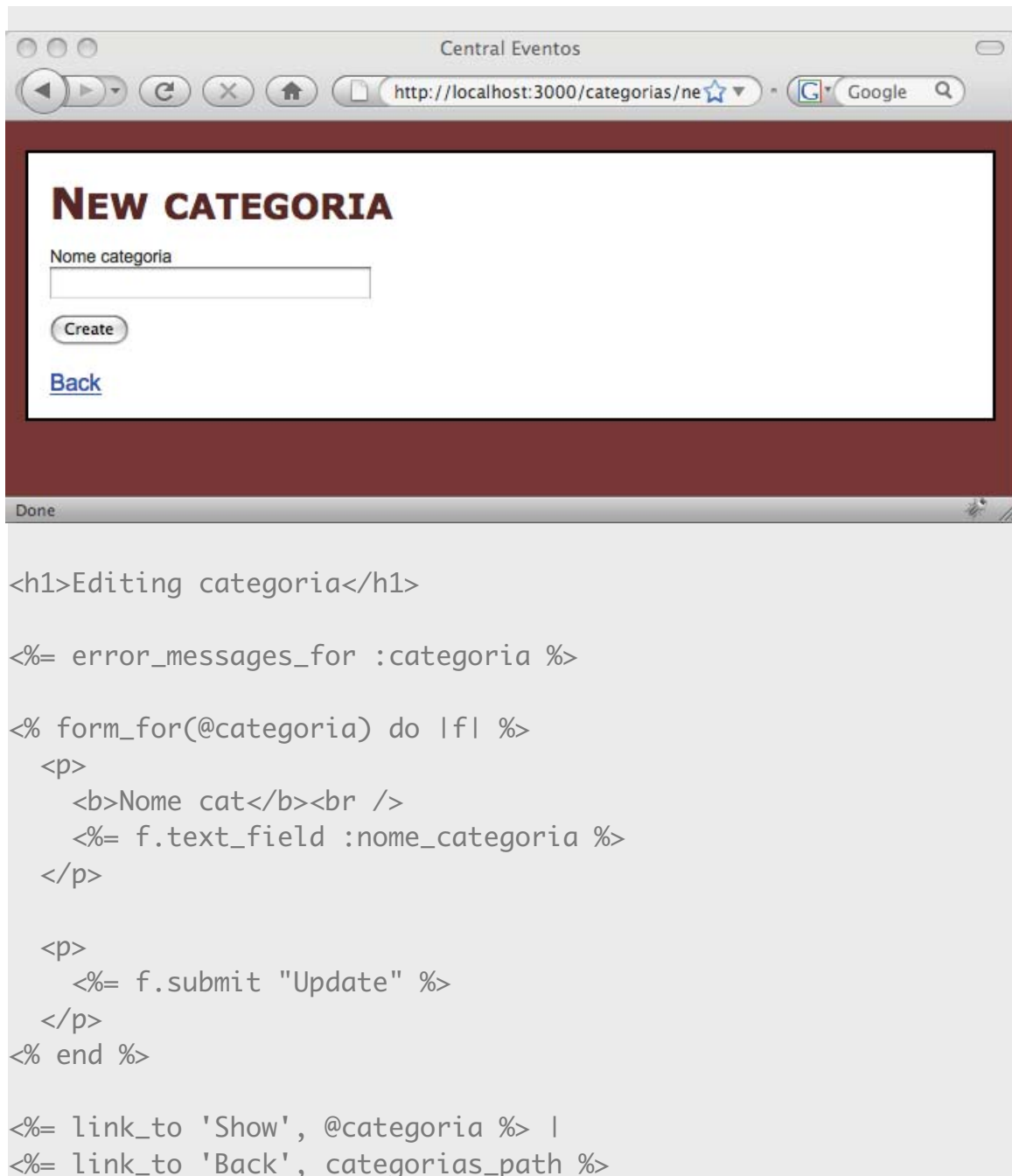
<% form_for(@categoria) do |f| %>
  <p>
    <b>Nome cat</b><br />
    <%= f.text_field :nome_categoria %>
  </p>

  <p>
    <%= f.submit "Update" %>
  </p>
<% end %>

<%= link_to 'Show', @categoria %> |
<%= link_to 'Back', categorias_path %>
```

Lembrando que após o envio, os dados do formulário são processados pela ação “update”

New



```
<h1>Editing categoria</h1>

<%= error_messages_for :categoria %>

<% form_for(@categoria) do |f| %>
  <p>
    <b>Nome cat</b><br />
    <%= f.text_field :nome_categoria %>
  </p>

  <p>
    <%= f.submit "Update" %>
  </p>
<% end %>

<%= link_to 'Show', @categoria %> |
<%= link_to 'Back', categorias_path %>
```

Lembrando que após o formulário ser enviado, seus dados serão processados pela ação “create”

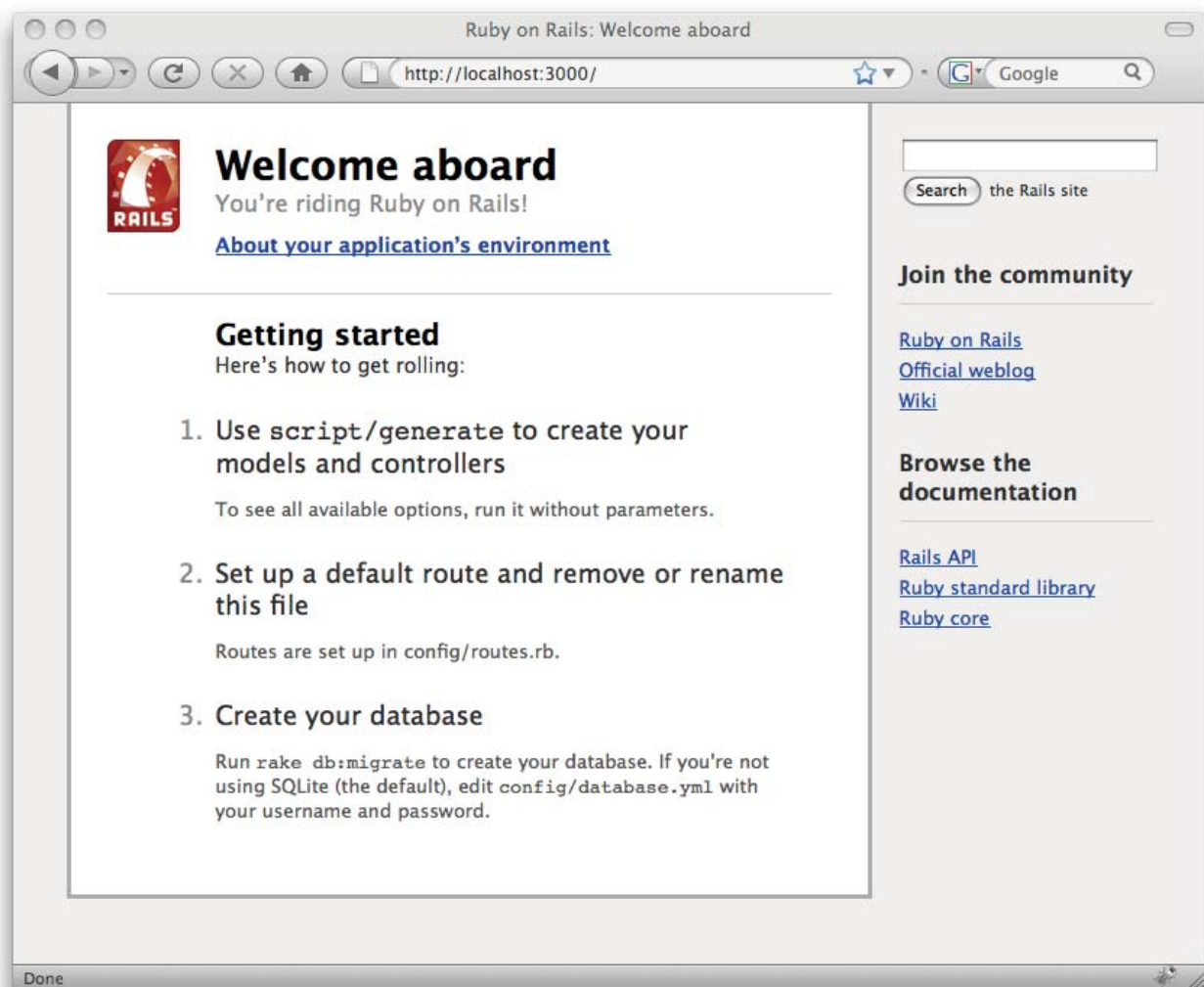
Claro que o scaffold faz apenas uma base, um esqueleto para sua aplicação que você deve editar e refinar de acordo com as suas necessidades. Nós poderíamos por exemplo traduzir todos os textos como “new “ e “show” para português. Com algumas alterações assim a página de listagem ficaria assim:



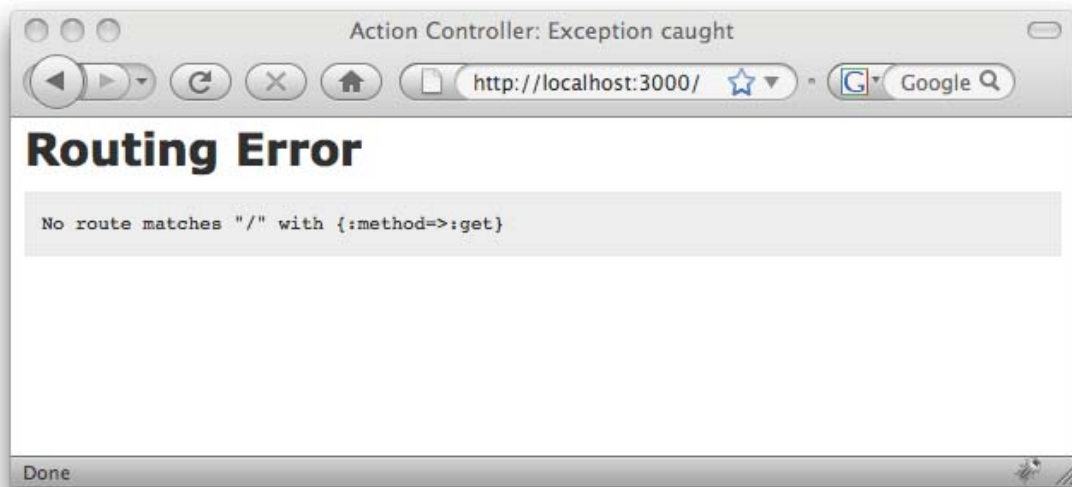
Avançando no Rails

ROTEAMENTO PADRÃO

Toda a nossa parte de eventos está pronta, mas se você acessar a URL raiz da aplicação, você vai notar que ela não mudou, e continua apresentando a página de boas-vindas do rails:



Isso acontece porque o Rails define um arquivo `index.html` que serve como padrão. Se removermos esse arquivo do diretório `public`, ficaremos com a seguinte página:



Esse erro indica que o Rails não consegue encontrar um roteamento que satisfaça a URL que especificamos.

O que é roteamento?

Para identificar qual *controller* será responsável por atender uma determinada solicitação da aplicação, o Rails utiliza um mecanismo de roteamento de requisições automático que não necessita de configurações complexas, mas que pode também ser customizado de acordo com as necessidades de cada aplicação. Você deve se lembrar que no capítulo sobre REST acrescentamos um roteamento a mais no arquivo `config/routes.rb`, que deverá ter um código semelhante a este (excluindo-se os comentários):

```
ActionController::Routing::Routes.draw do |map|

  map.resources :eventos
  map.connect ':controller/:action/:id'
  map.connect ':controller/:action/:id.:format'

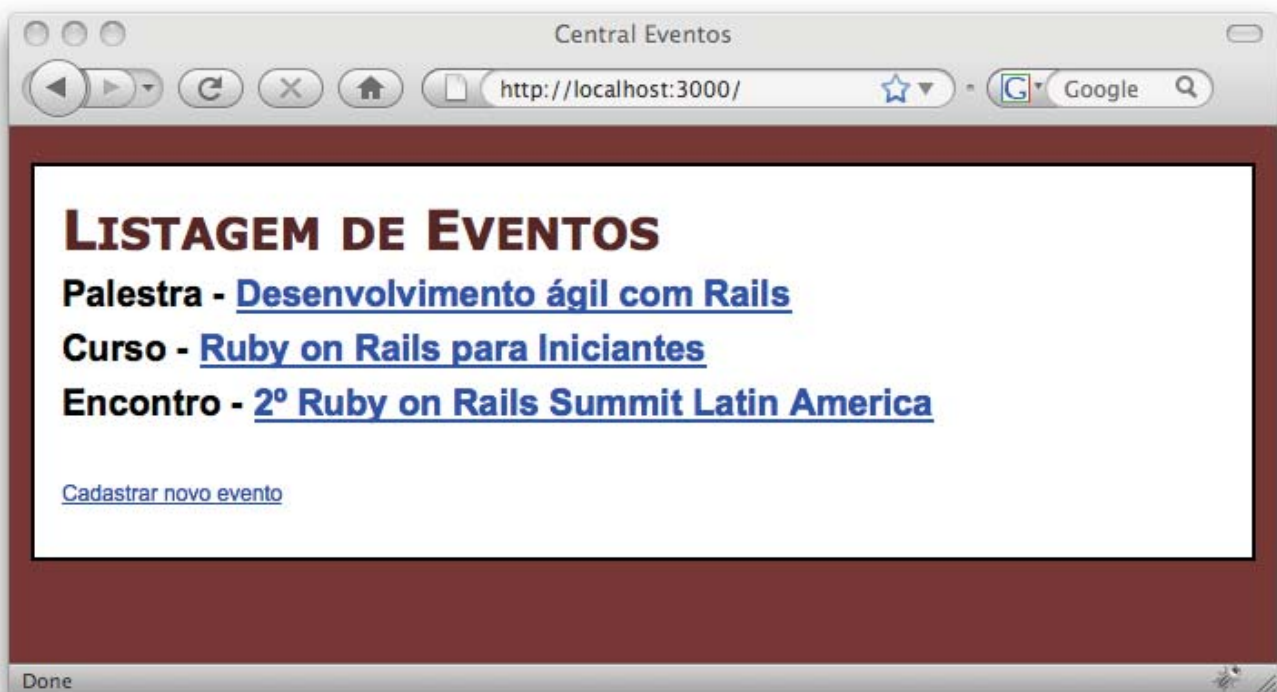
end
```

Para mapear o *controller* “eventos” como nossa página inicial, basta acrescentar a linha para dizer o seguinte:

```
map.root :controller => "eventos"
```

Estamos dizendo, com o código acima que se a rota raiz for invocada, a requisição deve ser servida pelo *controller* eventos, que havíamos criado anteriormente.

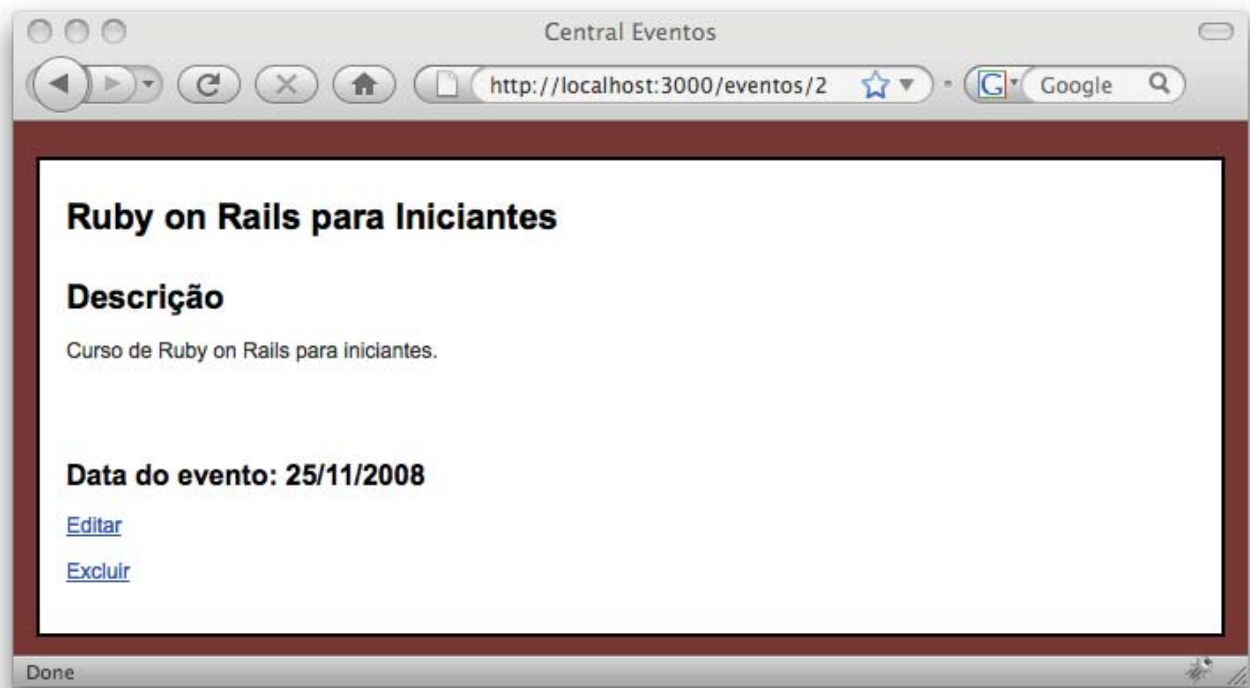
Agora, recarregando nossa página inicial temos:



Note que agora o mesmo *controller* serve dois roteamentos. Essa é apenas uma das possibilidades mais simples que estão disponíveis com o roteamento do Rails.

URLs AMIGÁVEIS

Para navegar entre as diversas views, muitas vezes precisamos passar o ID do evento através da URL. Se clicarmos em um dos eventos para ver seus detalhes a URL gerada fica assim:



Apesar de funcional, a URL não diz nada à respeito do conteúdo. Existe uma forma de construir URLs mais amigáveis, que são muito úteis não apenas porque são mais fáceis de serem lidas e entendidas pelos usuários como também porque fazem com que um sistema de busca como o Google entenda melhor do que se trata seu site e suas páginas.

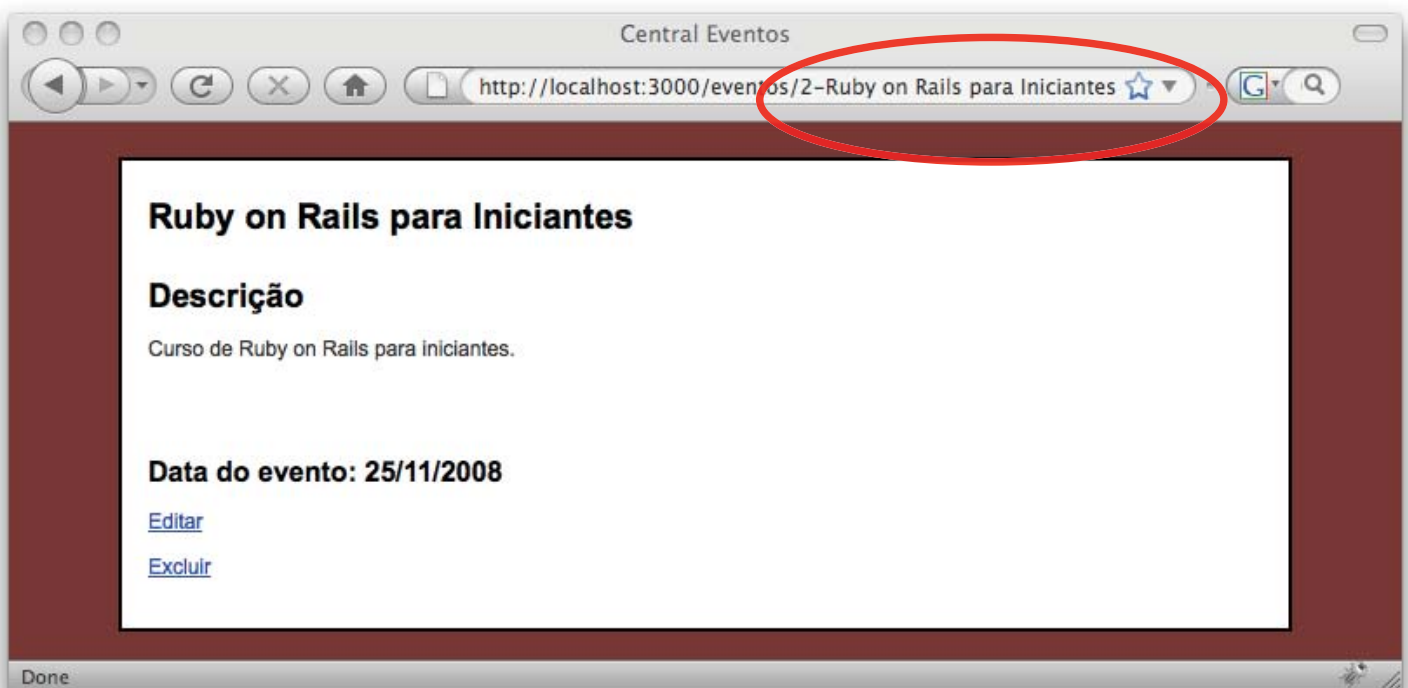
O Rails já tem por padrão suporte para URLs do tipo `/controller/action/1-meu_item`. Isto é possível de ser feito implementando o recurso `to_param` em nossos modelos. Abra o arquivo `models/evento.rb` e acrescento o código:

```
class Evento < ActiveRecord::Base
  belongs_to :categoria
  validates_presence_of :titulo

  def to_param
    "#{id}-#{titulo}"
  end
end
```

Estas novas URLs vão funcionar automaticamente, desde que o início da URL seja o ID. Isso acontece porque o Ruby vai converter uma URL do tipo `'123-ola-mundo'` automaticamente em `123`.

Se você tentar clicar novamente em um dos links da página de listagem verá os detalhes do evento com uma URL mais amigável:





Dica de Ruby - Concatenando strings e variáveis

No método `to_param` criamos um string concatenado com variáveis. Na linguagem Ruby, é possível concatenar um string com o valor de uma variável através do uso de substituições - bastando usar a construção `#{}` . Assim, no string abaixo:

```
fruta = "morangos"
"Adoro laranjas e #{fruta}"
```

O string final será “Adoro laranjas e morangos”

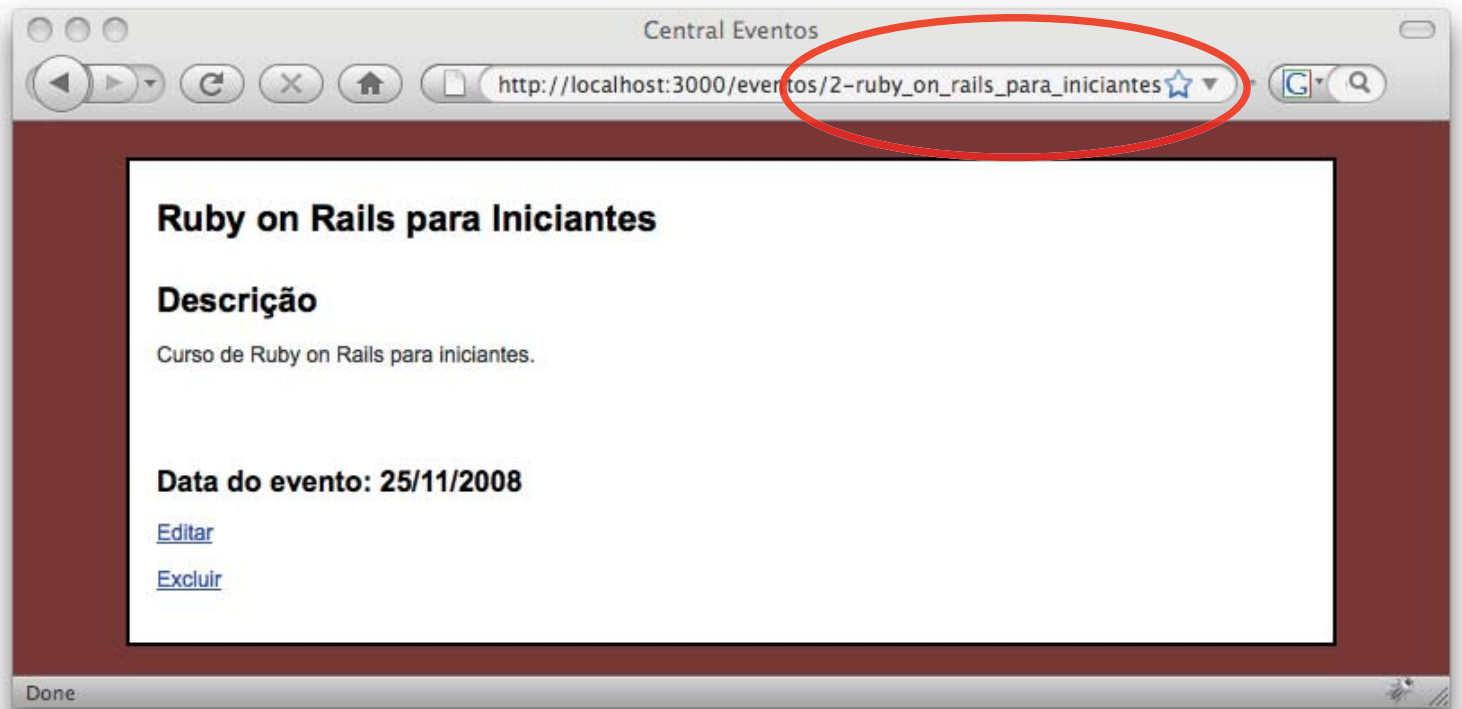
Já no caso do método `to_param`, o string final gerado será algo como “2-Ruby on Rails para iniciantes”

Apesar do Firefox 3 ter exibido corretamente a URL (incluindo até mesmo os espaços) outros navegadores substituirão os espaços, pontuação e acentos por caracteres estranhos (espaços, por exemplo, são exibidos como “%20”). Para retirar estes caracteres estranhos da URL, usamos o método `parameterize` no título. Esse método retira do texto tudo que não for letras ou números e substitui por hífens, além de substituir caracteres acentuados pelos caracteres correspondentes sem acentos (“é” para “e”, “ç” para “c” etc.):

```
class Evento < ActiveRecord::Base
  belongs_to :categoria
  validates_presence_of :titulo

  def to_param
    "#{id}-#{titulo.parameterize}"
  end
end
```

Se testarmos mais uma vez, clicando em um dos links da página principal, teremos agora:



Claro que você pode fazer URLs ainda mais amigáveis suprimindo completamente o ID e fazendo com que o Rails localize o registro no banco de dados através de outro campo. Existe um plugin que facilita muito o processo de criação de URLs amigáveis deste tipo chamado Permalink Fu (http://agilewebdevelopment.com/plugins/permalink_fu).

REAPROVEITAMENTO DE CÓDIGO COM PARTIALS

No começo deste guia falamos que um dos princípios do Rails é evitar a repetição desnecessária de códigos. Um dos recursos que auxiliam neste objetivo é o partial, que permite o reaproveitamento de fragmentos de página em diferentes views.

Um caso onde um partial poderia ser utilizado são os formulários de cadastro e edição de eventos. São dois formulários idênticos, que poderiam ser compartilhados uma única vez através do uso de partials. Veja os códigos:

views/eventos/new.html.erb

```
<h1>Cadastrar novo Evento</h1>

<% form_for(@evento) do |f| %>
  <h4>Titulo</h4>
  <%= f.text_field :titulo %>
<br />
  <h4>Descrição</h4>
  <%= f.text_area :descricao %>
<br />
  <h4>Categoria</h4>
  <%= f.select :categoria_id, Categoria.find(:all).collect { |
item| [item.nome_categoria,item.id] } %>
<br />
  <h4>Data</h4>
  <%= f.date_select(:data, :order => [:day, :month, :year]) %>
<br />
  <br />
  <%= f.submit "Cadastrar" %>
<% end %>
```

Views/eventos/edit.html.erb

```
<h1>Editar Evento</h1>
<% form_for(@evento) do |f| %>
  <h4>Titulo</h4>
  <%= f.text_field :titulo %>

  <h4>Descrição</h4>
  <%= f.text_area :descricao %>

  <h4>Categoria</h4>
  <%= f.select :categoria_id, Categoria.find(:all).collect
{ |item| [item.nome_categoria,item.id] } %>

  <h4>Data</h4>
  <%= f.date_select(:data, :order =>
[:day, :month, :year]) %>
  <br />

  <%= f.submit "Editar" %>

<% end %>
```

Vamos portanto transformar todo o formulário num partial. Crie um novo arquivo na pasta `views/eventos` chamado `_formulario.html.erb`. Por padrão, partials devem ser salvos com um nome de arquivo iniciado em underscore (`_`). Coloque neste novo documento todo o código do formulário.

O arquivo deverá ficar assim:

```
<% form_for(@evento) do |f| %>
  <h4>Titulo</h4>
  <%= f.text_field :titulo %>

  <h4>Descrição</h4>
  <%= f.text_area :descricao %>

  <h4>Categoria</h4>
  <%= f.select :categoria_id, Categoria.find(:all).collect
{ |item| [item.nome_categoria,item.id] } %>

  <h4>Data</h4>
  <%= f.date_select(:data, :order =>
[:day, :month, :year]) %>
  <br />
  <%= f.submit "Enviar" %>
<% end %>
```

Nos arquivos `views/eventos/new.html.erb` e `views/eventos/edit.html.erb`, remova todo o formulário e substitua por uma única instrução que indica que um partial deve ser utilizado:

views/eventos/new.html.erb

```
<h1>Cadastrar novo Evento</h1>

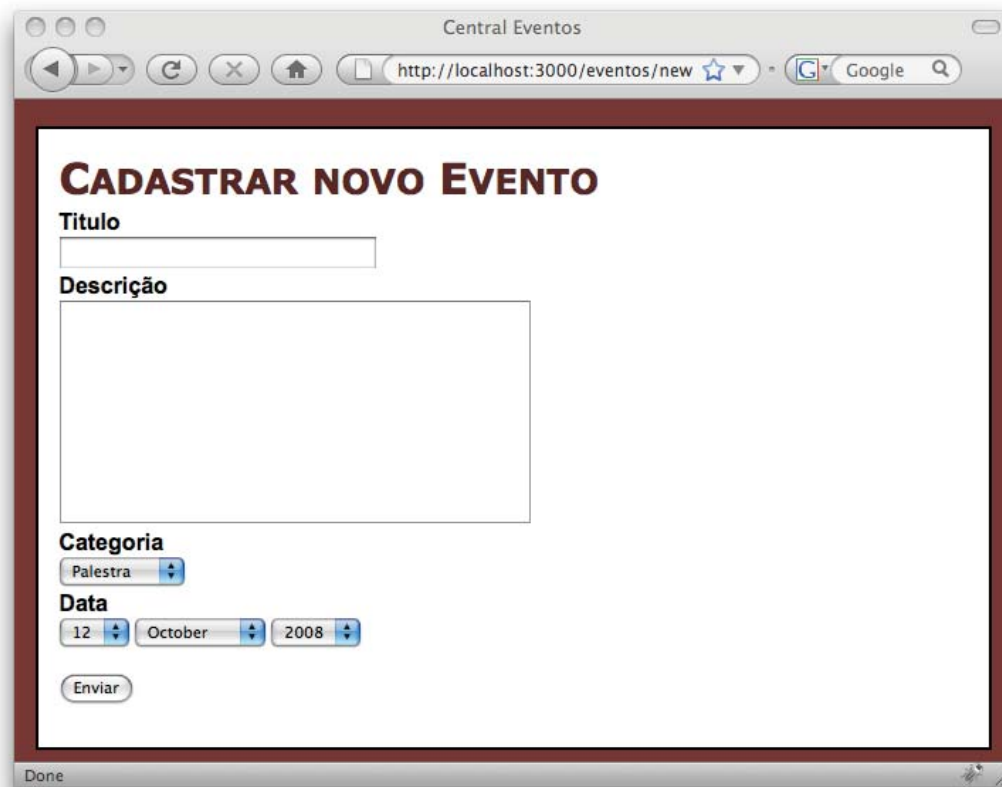
<%=render :partial => 'formulario'%>
```

views/eventos/edit.html.erb

```
<h1>Editar Evento</h1>
<%=render :partial => 'formulario'%>
```

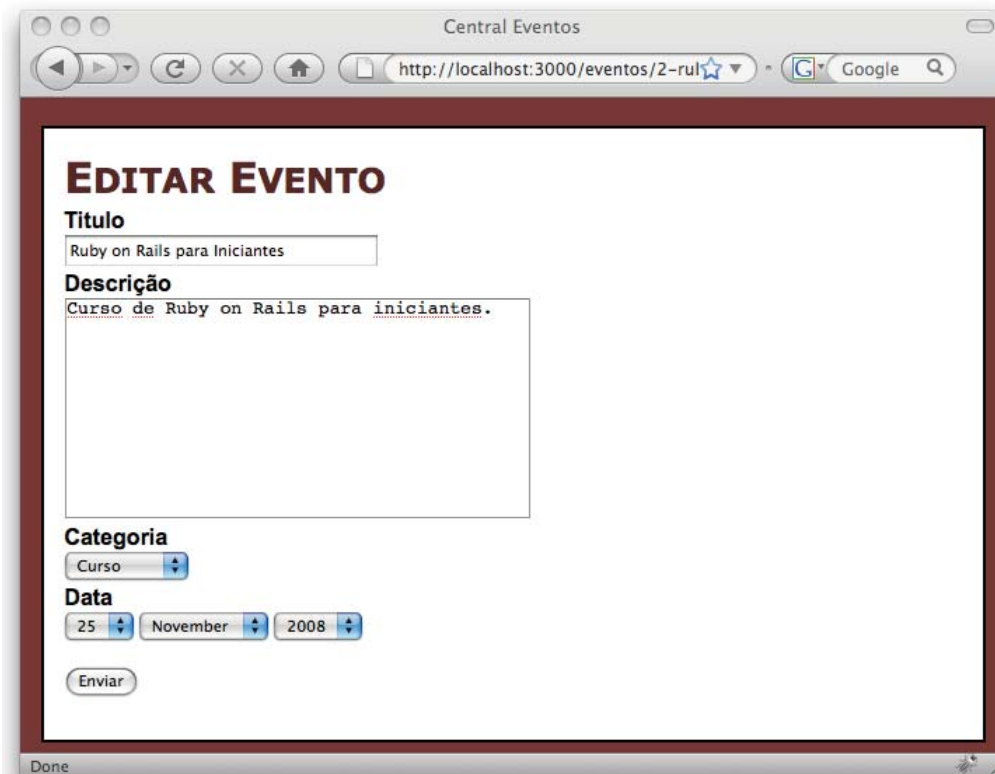
Repare que não é preciso indicar nem a pasta nem o nome completo do arquivo de partial – o Rails o localiza automaticamente e o resultado final pode ser visto na próxima página:

new



The screenshot shows a web browser window titled 'Central Eventos' with the address bar displaying 'http://localhost:3000/eventos/new'. The page has a dark red border and contains a form titled 'CADASTRAR NOVO EVENTO'. The form includes a 'Titulo' text input field, a 'Descrição' text area, a 'Categoria' dropdown menu with 'Palestra' selected, and a 'Data' section with three dropdowns for day (12), month (October), and year (2008). An 'Enviar' button is at the bottom of the form. The browser's status bar at the bottom shows 'Done'.

edit



The screenshot shows a web browser window titled 'Central Eventos' with the address bar displaying 'http://localhost:3000/eventos/2-rul'. The page has a dark red border and contains a form titled 'EDITAR EVENTO'. The form includes a 'Titulo' text input field with the value 'Ruby on Rails para Iniciantes', a 'Descrição' text area with the value 'Curso de Ruby on Rails para iniciantes.', a 'Categoria' dropdown menu with 'Curso' selected, and a 'Data' section with three dropdowns for day (25), month (November), and year (2008). An 'Enviar' button is at the bottom of the form. The browser's status bar at the bottom shows 'Done'.

Passando objetos para dentro dos partials

Em alguns casos, o fragmento de página que você usa dentro do partial precisa de exibir informações adicionais, muitas vezes provenientes de um Model. Você pode passar informações da view para um partial através do parâmetro :locals

Para entender melhor, vamos fazer um novo partial com a lista de categorias. Abra o arquivo views/categorias/index.html.erb:



O código deste arquivo deverá estar assim:

```
<h1>Listagem de categorias</h1>

<table id='teste'>
  <tr>
    <th>Categorias</th>
  </tr>

  <% for categoria in @categorias %>
    <tr>
      <td><%=h categoria.nome_categoria %></td>
      <td><%= link_to 'Editar',
edit_categoria_path(categoria) %></td>
      <td>
        <%= link_to 'Excluir', categoria, :confirm =>
'Confirma exclusão?', :method => :delete %>
      </td>
    </tr>

  <% end %>

</table>

<br />

<%= link_to 'Nova categoria', new_categoria_path %>
```

A porção de código em destaque é responsável por gerar cada uma das linhas da tabela listando as categorias. Nós vamos extrair todo este código e colocar num novo arquivo de partial chamado `views/categorias/_lista.html.erb`.

O novo arquivo ficará assim:

_lista.html.erb

```
<tr>
  <td><%=h categoria.nome_categoria %></td>
  <td><%= link_to 'Editar',
edit_categoria_path(categoria) %></td>
  <td>
    <%= link_to 'Excluir', categoria, :confirm =>
'Confirma exclusão?', :method => :delete%>
  </td>
</tr>
```

De volta ao arquivo de view em views/categorias/index.html.erb, carregamos a nova partial passando um parâmetro adicional: Além do nome do arquivo passaremos o objeto “categoria” que é utilizando dentro do partial usando o recurso :locals.

```
<h1>Listagem de categorias</h1>

<table id='teste'>
  <tr>
    <th>Categorias</th>
  </tr>

  <% for categoria in @categorias %>

    <%= render :partial =>"lista", :locals =>{:categoria
=> categoria} %>

  <% end %>

</table>

<br />
```

Partials são uma das grandes facilidades do Rails, sendo utilizados principalmente em aplicações Ajax para gerar somente os fragmentos do código necessários para atualizações de partes de uma página, conforme veremos no próximo capítulo.

Ajax

Uma das grandes tendências atuais no desenvolvimento de aplicações Web é o que se tornou conhecido como Web 2.0, que é uma tentativa de aproximar as aplicações Web de aplicações *desktop*. De modo prático, isso significa criar aplicações web mais responsivas, que possam atualizar partes do conteúdo da tela dinamicamente sem que uma nova página precise ser carregada todas as vezes que o usuário clicar em algum link ou botão.

Aplicações feitas no estilo da assim chamada Web 2.0 possuem características de integração e usabilidade muito superiores às aplicações que vinham sendo comumente desenvolvidas e estão se transformando em uma referência para o que deve ou não deve ser feito na Internet.

AJAX é o braço tecnologico da Internet. Não é um produto novo, apenas uma sigla para **A**synchronous **J**avascript **a**nd **X**ML, e é através desta tecnologia que conseguimos recuperar dados no servidor sem a necessidade de recarregamento de páginas, ou seja, de forma assíncrona.

Um dos grandes motivos da popularização do Rails nos últimos tempos tem sido o suporte integrado ao desenvolvimento de aplicações com recursos AJAX no *framework*, de modo fácil o suficiente para beneficiar mesmo o desenvolvedor iniciante.

PREPARANDO AS PÁGINAS PARA UTILIZAR AJAX

O Ruby on Rails utiliza duas bibliotecas de javascript para criação das interações e efeitos em AJAX: Prototype e Script.aculo.us.

Para fazer uso de AJAX em nossas páginas, precisamos instruí-las para carregar estas bibliotecas. Podemos fazer isso inserindo a instrução `javascript_include_tag :defaults` no layout padrão das nossas views (`views/layouts/applications.html.erb`):

```
<html>
<head>
  <title>Central Eventos</title>
  <%= stylesheet_link_tag "scaffold" %>
  <%= stylesheet_link_tag "default" %>

  <%= javascript_include_tag :defaults %>

</head>

<body>
<div id="principal">
<% unless flash[:aviso].blank? %>
<div id="aviso"><%= flash[:aviso] %></div>
<% end %>
  <%= yield %>
</div>
</body>

</html>
```

LINK_TO_REMOTE

O `link_to_remote` funciona de forma semelhante ao `link_to` que já utilizamos em diversos lugares na aplicação de eventos. Ambos fazem uma solicitação ao servidor, sendo que a única diferença é que no `link_to`, o servidor responde com uma nova página a ser carregada, enquanto no caso do `link_to_remote`, o servidor envia a resposta de forma assíncrona, sem que uma nova página precise ser carregada.

A diferença pode parecer sutil, mas o resultado final é bem diferente. Vamos utilizar o `link_to_remote` para manipular as categorias. Abra o partial responsável por listar as categorias na página de categorias (`views/categorias/_lista.html.erb`):

```
<tr>
  <td><%=h categoria.nome_categoria %></td>
  <td><%= link_to 'Editar', edit_categoria_path(categoria) %></td>
  <td>
    <%= link_to 'Excluir', categoria, :confirm => 'Confirma exclusão?', :method => :delete %>
  </td>
</tr>
```

Vamos começar alterando o helper `link_to` de exclusão para `link_to_remote`.

Além disso vamos acrescentar um parâmetro ID no tag `<tr>`, responsável por cada linha da tabela que lista as categorias. O parâmetro ID vai permitir que cada linha tenha uma identificação única na tabela – e isso será útil na hora em que aplicarmos efeitos especiais para eliminar a linha do item que o usuário excluiu. Para identificarmos cada linha, utilizamos o helper `dom_id`, que gera automaticamente um texto de identificação:

```
<tr id='<%= dom_id(categoria) %>'>
  <td ><%=h categoria.nome_categoria %></td>
  <td><%= link_to 'Editar', edit_categoria_path(categoria)
%></td>
  <td>
    <%= link_to_remote 'Excluir',
                      :url=>categoria_path(categoria),
                      :confirm => 'Confirma exclusão?',
                      :method => :delete

    %>

  </td>
</tr>
```

Se você quiser testar, vai ver que ao clicar o link o item é realmente excluído do banco de dados, mas ele permanece escrito na página. Para a aplicação de efeitos e interações mais avançadas, precisamos de trabalhar com templates do tipo rjs.

TEMPLATES JS.RJS

Assim como templates .html.erb são páginas escritas em Ruby que geram documentos HTML, templates.js.rjs são documentos escritos em Ruby que se transformam em uma série de instruções em javascript no navegador. No nosso caso, queremos que após clicar no link de exclusão e confirmar, o item excluído suma da lista através de um efeito de animação. Para fazer isso, crie um novo documento em views/categorias/destroy.js.rjs e escreva o código a seguir:

```
page.visual_effect :fade, dom_id(@categoria)
```

Para terminar, abra o controller de categorias (controller/categorias_controller.rb), e localize a ação destroy no final do arquivo.

A ação destroy é chamada quando o usuário clica no link excluir e, além de efetivamente excluir o registro, responde de modo diferente caso a solicitação tenha sido feita via html ou xml através do respond_to, como você pode ver abaixo:

```
# DELETE /categorias/1
# DELETE /categorias/1.xml
def destroy
  @categoria = Categoria.find(params[:id])
  @categoria.destroy

  respond_to do |format|
    format.html { redirect_to(categorias_url) }
    format.xml  { head :ok }
  end
end
```

O respond_to determina o que fazer quando a solicitação de exclusão vem a partir de um documento HTML ou XML, por exemplo. Como o link_to_remote é uma solicitação feita via Javascript, é necessária acrescentar a possibilidade da ação destroy responder também a javascript (utilizando um template js.rjs com mesmo nome da ação - destroy.js.rjs).

Na verdade, como não faremos solicitações a ação destroy de outras formas, manteremos apenas a resposta a javascript, e a ação destroy ficará assim:

```
# DELETE /categorias/1
# DELETE /categorias/1.xml
def destroy
  @categoria = Categoria.find(params[:id])
  @categoria.destroy

  respond_to do |format|
    format.js
  end
end
end
```

REMOTE_FORM_FOR

De modo análogo ao `link_to_remote`, o `remote_form_for` permite que você envie dados preenchidos em um formulário para o controller adequado sem troca de página, de modo assíncrono. Para testarmos o `remote_form_for`, vamos substituir o link “Nova categoria” por um formulário completo na página de listagem de categorias. Quando enviado, o controller `create` vai cadastrar a nova categoria e, através de efeitos de javascript, vamos fazer o novo item aparecer na lista sem que a página seja recarregada.

Começaremos criando um id para a tabela e um formulário de cadastro na página de listagem (`views/categorias/index.html.erb`)

```
<h1>Listagem de categorias</h1>

<table id='tabela'>
  <tr>
    <th>Categorias</th>
  </tr>

  <% for categoria in @categorias %>

    <%= render :partial =>"lista", :locals =>{:categoria =>
categoria} %>

  <% end %>

</table>

<br />

<% remote_form_for(@categoria) do |f| %>
  Categoria: <%= f.text_field :nome_categoria %>
  <%= f.submit "Criar" %>
<% end %>
```

O `remote_form_for`, assim como o `form_for` que empregamos anteriormente neste guia, precisa de uma instância do modelo de dados para funcionar. Vamos acrescentar esta nova instância na ação `index` do controller `controllers/categorias_controller.rb`:

```
class CategoriasController < ApplicationController
  # GET /categorias
  # GET /categorias.xml
  def index
    @categorias = Categoria.find(:all)
    @categoria = Categoria.new
    respond_to do |format|
      format.html # index.html.erb
      format.xml { render :xml => @categorias }
    end
  end
end
```

Se você testar, o formulário já funciona, enviando os dados para a ação `create` e cadastrando os dados no banco. Mas a página não é atualizada instantaneamente, você só vê o novo item se recarregar manualmente a página.

Agora vamos construir um template js.rjs para exibir o item novo com um efeito de animação, assim como fizemos nos exemplos anteriores de exclusão de categorias. Ainda no controller controllers/categorias_controller.rb, localise a ação create e acrescente o formato javascript no respond_to, da mesma maneira que fizemos com a ação destroy nos exemplos anteriores:

```
def create
  @categoria = Categoria.new(params[:categoria])

  respond_to do |format|
    if @categoria.save
      flash[:notice] = 'Categoria was successfully created.'
      format.html { redirect_to(@categoria) }
      format.xml { render :xml => @categoria, :status
=> :created, :location => @categoria }
      format.js
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @categoria.errors, :status
=> :unprocessable_entity }
      format.js
    end
  end
end
```

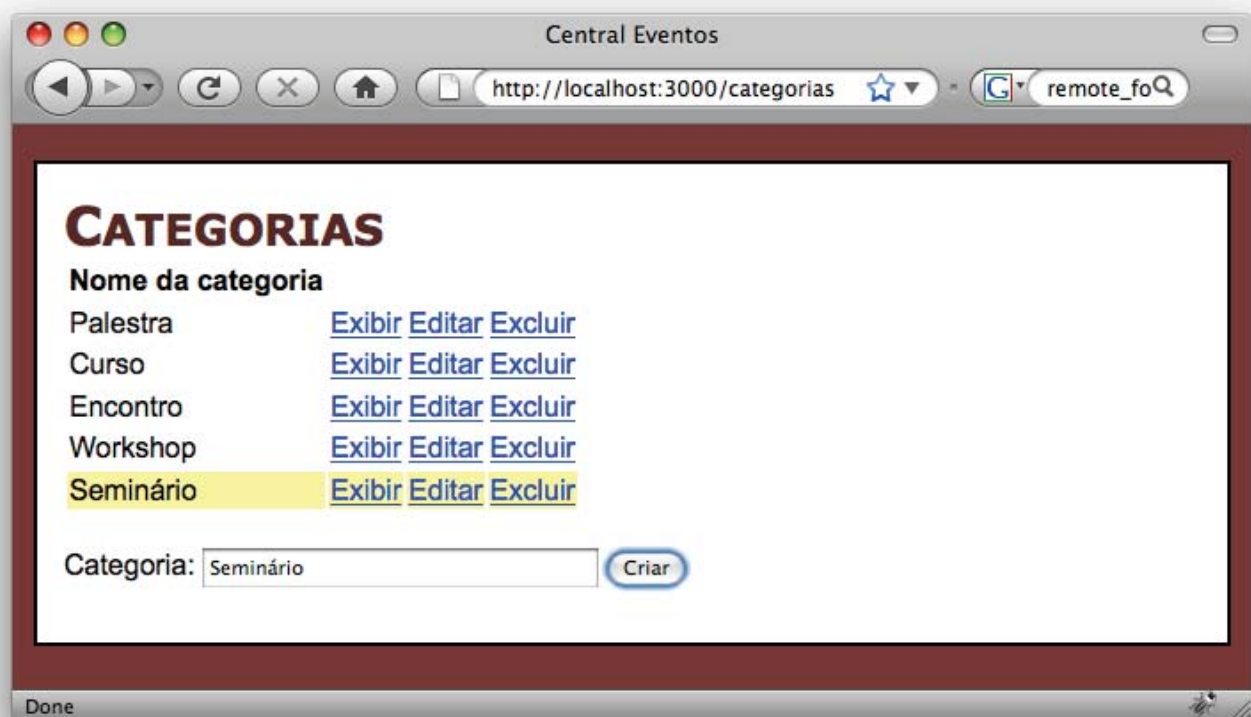
Atente para o detalhe que esta ação tem uma estrutura condicional (if...else). Você deve acrescentar a instrução format.js em ambos.

Finalmente, crie um novo arquivo chamado `views/categorias/create.js.rjs`, onde instruir o javascript para acrescentar o novo item ao final da tabela (cujo id é “tabela”) e fazer uma animação.

```
page.insert_html :bottom, 'tabela', :partial =>
"lista", :locals =>{:categoria => @categoria}
page.visual_effect :highlight, dom_id(@categoria)
```

A primeira linha manda acrescentar um partial ao final (:bottom) do elemento cujo ID é “tabela”. Este partial listará o mais recente item acrescentado no banco.

A segunda acrescenta um animação pintando momentaneamente o item recém-acrescentado na página de amarelo.



Desenvolvimento baseado em Testes

Embora tenhamos ignorado esse assunto durante todo livro, indiscutivelmente uma das partes mais importantes de qualquer aplicação Rails são os testes. Todo o framework Rails já vem com uma estrutura pronta para a criação de testes automatizados para seu site ou aplicação, e você perceberá que não é difícil começar a escrever seus testes.

Porque escrever testes?

Todo programador, mesmo iniciante, testa suas aplicações enquanto as desenvolvem, porém de modo manual: acessando individualmente cada uma das seções do site e agindo como se fossem usuários - listando e cadastrando alguns registros e até tentando provocar alguns erros para ver se a aplicação está preparada. O problema é que essa abordagem fica cada vez mais limitada quanto mais o site ou aplicação cresce em tamanho e recursos.

O desenvolvimento baseado em testes faz parte de um conceito mais amplo chamado metodologia Extreme Programming (XP).

Na metodologia XP, após o levantamento de quais funcionalidades uma aplicação precisa ter, o desenvolvimento sempre começa com a criação dos testes automatizados. Uma vez escritos, os testes servem como uma linha guia, definindo as condições segundo as quais o código a ser desenvolvido deve ser criado.

Em resumo: Somente após os testes terem sido escritos é que as funcionalidades da aplicação são desenvolvidas.

Apesar disso, para os propósitos deste livro, os testes foram ignorados até o momento porque introduzi-los significaria desviar a nossa atenção da demonstração do Rails em si.

TESTES DE UNIDADE E TESTES FUNCIONAIS

O Rails oferece duas formas principais de testes: testes de unidade e testes funcionais. Os primeiros testam os modelos de dados, garantindo que validações, associações etc funcionam corretamente. Os últimos testam se a lógica funcional da aplicação, presente em controllers e views, está correta.

CRIANDO NOSSO PRIMEIRO TESTE DE UNIDADE

Vamos começar criando alguns testes de unidade em nossa aplicação. Abra o arquivo `test/unit/evento_test.rb`

Este arquivo deverá estar assim.

```
require 'test_helper'

class EventoTest < ActiveSupport::TestCase
  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

Repare que essa classe já contém um teste (chamado “the truth”) que serve apenas como exemplo mas não faz absolutamente nada - você pode simplesmente apagá-lo.

Todos os blocos de teste são iniciados pela instrução `test` seguido por um nome ou uma descrição do que o teste faz. Algumas pessoas preferem nomear seus testes explicitando o que a aplicação **deveria** fazer em determinadas circunstâncias. Em inglês, podemos traduzir “deveria” como “should”, assim um bom nome para um teste que vai assegurar que é possível gravar dados no banco é:

“should create evento” (Deveria criar um evento)

Vamos criar este teste:

```
require 'test_helper'

class EventoTest < ActiveSupport::TestCase

  test "should create evento" do

    end

end
```

O mínimo que todo e qualquer teste deve ter é um (ou mais) “assert”.

Em português, a palavra “assert” significa afirmação, asserção - mas para compreender melhor o seu uso em testes uma tradução melhor seria “Assegure que...” ou “Verifique que...”

Vamos completar o código do primeiro teste para entender seu funcionamento:

```
require 'test_helper'

class EventoTest < ActiveSupport::TestCase

  test "should create evento" do
    evento = Evento.create(:titulo => "Aprendendo Rails")
    assert evento.valid?
  end

end
```

Na primeira linha do teste, utilizamos um comando Rails para inserir um novo registro na tabela de eventos, na sequencia utilizamos um assert para assegurar que o evento foi realmente salvo na tabela e é válido.

Quando criamos os testes para nossas models e controllers, é importante testar o lado positivo e também o lado negativo. Mas o que isso quer dizer? Quer dizer que além de cadastrar um registro e certificar de que ele foi inserido no banco, devemos também tentar inserir um registro inválido e ver se nossa aplicação o rejeita. Um exemplo disso seria criar um teste que tenta cadastrar um novo evento sem título. Você deve se lembrar que anteriormente neste livro nós acrescentamos uma validação no model Evento que requeria que os eventos tenham título. Vamos acrescentar este teste e ver se a aplicação rejeita o registro.

```
require 'test_helper'

class EventoTest < ActiveSupport::TestCase

  test "should create evento" do
    evento = Evento.create(:titulo => "Aprendendo Rails")
    assert evento.valid?
  end

  test "should require titulo" do
    evento = Evento.create(:titulo => nil)
    assert !evento.valid?
  end
end
```

Repare primeiro no nome que demos para o teste (should require titulo - deveria requerer o título). Neste teste tentamos inserir um novo evento com o título vazio (nil) e o assert verifica que o registro **não** é válido (a exclamação serve como negação.).

Mais adiante veremos que além do “assert” simples, que verifica se determinada condição é verdadeira, existem outros tipos de assert usados em situações específicas, como o assert_response, assert_select, assert_equal, assert_not_nil e muitos outros.

EXECUTANDO OS TESTES

O Ruby on Rails trabalha com um banco de dados exclusivo para testes, cujos dados são excluídos e recriados a cada novo teste. Antes de executar os testes, precisamos de garantir que todos os bancos de dados da sua aplicação foram criados (e não apenas o de desenvolvimento). Para isso, utilize o comando `rake db:create:all`

Executando os testes de Unidade

Para fazer os testes de unidade de uma aplicação em Rails, usamos o comando abaixo:

```
rake test:units
```

O resultado do comando abaixo, na aplicação que temos agora, é algo assim:

```
in /Users/cassio/Sites/eventos)
/System/Library/Frameworks/Ruby.framework/Versions/1.8/usr/bin/
ruby -Ilib:test "/Library/Ruby/Gems/1.8/gems/rake-0.8.1/lib/
rake/rake_test_loader.rb" "test/unit/categoria_test.rb" "test/
unit/evento_test.rb"
Loaded suite /Library/Ruby/Gems/1.8/gems/rake-0.8.1/lib/rake/
rake_test_loader
Started
..
Finished in 0.061371 seconds.

2 tests, 2 assertions, 0 failures, 0 errors
```

Como podemos ver, o que essa tarefa do rake faz é invocar a biblioteca `test/unit` para rodar cada um dos testes já criados até o momento, que neste caso passaram. Também é possível rodar apenas os testes funcionais, utilizando o comando `rake test:functionals`. Já o comando `rake:test` roda todos os testes. Faremos testes funcionais na sequência.

FIXTURES

Para realizar os testes um pouco mais sofisticados, é importante que o banco de dados de teste conheça alguns dados já cadastrados para manipularmos. Isso pode ser feito através do uso de fixtures.

Fixtures são arquivos yaml que contém dados que são carregados no banco de dados durante a execução dos testes.

Vamos abrir o fixture de eventos no diretório `test/fixtures/eventos.yml`, que deverá estar assim:

```
# Read about fixtures at http://ar.rubyonrails.org/classes/Fixtures.html

# one:
#   column: value
#
# two:
#   column: value
```

Vamos modificar o arquivo de fixtures acrescentando alguns dados. No final o arquivo ficará assim:

```
# Read about fixtures at http://ar.rubyonrails.org/classes/Fixtures.html

palestra_rails:
  id: 1
  titulo: Desenvolvimento ágil com Rails
  data: 2009-01-22
  descricao: Palestra sobre desenvolvimento agil de
aplicacoes para a web
  categoria_id: 1

curso_rails:
  id: 2
  titulo: Ruby on Rails para iniciantes
  data: 2009-03-15
  descricao: Curso de Ruby on Rails para iniciantes
  categoria_id: 2
```

Os dados que acabamos de introduzir serão automaticamente inseridos em nosso banco de testes quando precisarmos dos mesmos. Perceba que cada um dos dados tem um identificador, que neste caso são “palestra_rails” e “curso_rails”. Estes identificadores podem ser utilizados nos métodos de testes para se referir aos dados.

Vamos aproveitar para inserir alguns dados também na fixtures de categorias (test/fixtures/categorias.yml). O arquivo deverá ficar assim:

```
one:
  id: 1
  nome_categoria: Palestra

two:
  id: 2
  nome_categoria: Treinamento
```

Repare que nesta fixture vamos manter os índices “one” e “two”. O motivo é simples: Como usamos um scaffold para gerar o controller e as views de categorias, o scaffold já criou também uma série de testes funcionais para o controller categorias fazendo uso destes índices.

TESTES FUNCIONAIS

Vamos agora realizar alguns testes funcionais para aferir o funcionamento do nosso controller e das views de eventos. É importante lembrar que apesar de haver a distinção entre testes de unidade (que testam o modelo de dados) e testes funcionais (direcionados para testes no controller), é perfeitamente possível fazer testes funcionais que realizem manipulações no banco de dados.

Abra o arquivo de testes `test/functional/eventos_controller_test`, que deve estar assim:

```
require 'test_helper'

class EventosControllerTest < ActionController::TestCase
  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

Assim como no arquivo de testes de unidade, esse arquivo já contém um método de teste, porém este teste não faz absolutamente nada e pode ser excluído.

Testando as Exibições

Vamos começar inserindo um teste que testa simplesmente se a view index é encaminhada para o usuário:

```
require 'test_helper'

class EventosControllerTest < ActionController::TestCase

  test "should get index" do
    get :index
    assert_response :success
  end

end
```

O método http GET faz uma solicitação ao servidor pelo index e a nossa primeira asserção é feita não mais com o assert, mas com o assert_response, que verifica a resposta dada pelo servidor (neste caso testamos se a resposta é :success, ou seja, a view foi aberta)

Ainda complementando este teste, podemos ver se a lista de eventos foi carregada do banco - para isso utilizaremos o assigns. Assigns é um método auxiliar (um helper) disponível para os testes que pode ser utilizado para acessar todas as variáveis de instância definidas na ação solicitada. Com este acréscimo nosso teste fica assim:

```
require 'test_helper'

class EventosControllerTest < ActionController::TestCase

  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:eventos)
  end

end
```

Desta vez utilizamos o `assert_not_nil` para assegurar que a variável de instância que armazena os dados vindos do banco não está vazia.

No nosso próximo teste funcional, vamos verificar a ação `show` do controller de eventos.

```
require 'test_helper'

class EventosControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:eventos)
  end

  test "should show evento" do
    get :show, :id => eventos(:palestra_rails).id
    assert_response :success
  end
end
```

Mais uma vez usamos o método `get` para solicitar uma view, só que neste caso passamos um `id` como parâmetro. Repare que buscamos dinamicamente o `id` da fixture cuja identificação é `"palestra_rails"`.

Na sequência, usamos o `assert_response` para assegurar que a view foi encaminhada com sucesso.

Testando a inserção de dados

Vamos agora fazer alguns testes relacionados à inserção de dados. Primeiro, vamos escrever um teste simples para saber se a view “new” é encaminhada com sucesso:

Logo em seguida, vamos fazer um teste que tenta cadastrar um novo evento e verifica duas coisas: Se o novo dado foi registrado no banco e se o sistema redirecionou para a página que exibe o novo evento recém-criado.

Com os novos testes, o nosso arquivo deverá estar assim:

```
require 'test_helper'

class EventosControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:eventos)
  end

  test "should show evento" do
    get :show, :id => eventos(:palestra_rails).id
    assert_response :success
  end

  test "should get new" do
    get :new
    assert_response :success
  end

  test "should create evento" do
    assert_difference('Evento.count') do
      post :create, :evento => {:titulo => "Novo Evento"}
    end
    assert_redirected_to evento_path(assigns(:evento))
  end
end
```

Neste último exemplo empregamos dois novos tipos de asserções. Primeiro, utilizamos o “assert_difference”. O assert_difference cria um bloco dentro do qual realizamos manipulação no banco, e no final ele avalia se houve diferença no

número de registros no banco de dados antes e depois desta manipulação. Depois verificamos através do `assert_redirected_to` se a view show foi encaminhada para mostrar os dados do evento criado.

Testando a edição de dados

De modo similar aos testes de inserção de dados, onde primeiro verificamos se a view é enviada e depois verificamos se os dados foram gravados no banco, vamos agora testar se a view “edit” é encaminhada para o usuário e se o dado é atualizado no banco pela ação “update”. Nossos novos testes ficarão assim:

```
require 'test_helper'
class EventosControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:eventos)
  end

  test "should show evento" do
    get :show, :id => eventos(:palestra_rails).id
    assert_response :success
  end

  test "should get new" do
    get :new
    assert_response :success
  end

  test "should create evento" do
    assert_difference('Evento.count') do
      post :create, :evento => {:titulo => "Novo Evento"}
    end
    assert_redirected_to evento_path(assigns(:evento))
  end

  test "should get edit" do
    get :edit, :id => eventos(:palestra_rails).id
    assert_response :success
  end

  test "should update evento" do
    put :update, :id => eventos(:palestra_rails).id, :evento =>
{:titulo => "Titulo atualizado" }
    assert_redirected_to evento_path(assigns(:evento))
  end
end
```

O primeiro teste - `should get edit`, utiliza o método `get` passando como parâmetro o id do primeiro registro do banco de dados de teste (conforme havíamos configurado no arquivo de fixtures) para carregar solicitar a view `edit`. O `assert_response` verifica se esta view foi encaminhada corretamente.

O segundo teste - `should update evento`, utiliza o método `PUT` para enviar dados atualizados do evento para o banco de dados de teste(neste caso, tentamos atualizar o título do evento para “Título atualizado”). A asserção `assert_redirected_to` verifica se a view “`show`” com os dados atualizados do evento foi encaminhada.

Testando a exclusão de dados

Finalmente, vamos fazer um último teste que tenta excluir um registro do banco e verifica se o registro é de fato excluído e se a view de eventos é encaminhada.

Nosso teste ficará assim:

```
class EventosControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:eventos)
  end

  test "should show evento" do
    get :show, :id => eventos(:palestra_rails).id
    assert_response :success
  end

  test "should get new" do
    get :new
    assert_response :success
  end

  test "should create evento" do
    assert_difference('Evento.count') do
      post :create, :evento => {:titulo => "Novo Evento"}
    end
    assert_redirected_to evento_path(assigns(:evento))
  end
end
```

```

test "should get edit" do
  get :edit, :id => eventos(:palestra_rails).id
  assert_response :success
end

test "should update evento" do
  put :update, :id => eventos(:palestra_rails).id, :evento =>
{:titulo => "Titulo atualizado" }
  assert_redirected_to evento_path(assigns(:evento))
end

test "should destroy evento" do
  assert_difference('Evento.count', -1) do
    delete :destroy, :id => eventos(:palestra_rails).id
  end
  assert_redirected_to eventos_path
end

end

```

Como você pode notar, este teste faz a solicitação para o servidor utilizando o método http DELETE, enviando como parâmetro o id do primeiro registro do banco de dados de teste.

Utilizando um bloco `assert_difference`, primeiro verificamos se após a manipulação o banco de dados fica com um registro a menos que antes. Na sequência, verificamos se a view de eventos é encaminhada corretamente através do `assert_redirects_to`.