ActiveRecord

O ActiveRecord é uma camada de mapeamento objeto-relacional (object-relational mapping layer), responsável pela interoperabilidade entre a aplicação e o banco de dados e pela abstração dos dados. (wikipedia)

Nova opção para associações, :validate

Foi adicionado ao Rails uma nova opção para associações. Se incluirmos a opção `:validate => false` na associação o **ActiveRecord** salvará os dados do objeto pai, sem validar os objetos associados. Exemplo:

```
class AuditLog < ActiveRecord::Base
  belongs_to :developer, :validate => false
end

log = AuditLog.create(:developer_id => 0 , :message => "")
log.developer = Developer.new

puts log.developer.valid?
# => false

puts log.valid?
# => true

puts log.save
# => true
```

Note que mesmo com a associação não sendo valida, o objeto **log** foi salvo.

O valor padrão é `false`, ou seja, todas as validações em associações `belongs_to` estarão desligadas como padrão e para ligarmos devemos usar a expressão `:validate => true`.

Uma nova forma de especificar conditions usando Hash

Ao realizar buscas no banco de dados, por vezes temos de fazer uso da opção `:joins` a fim de melhorar a performance de nosso aplicativo, em outros casos precisamos simplesmente recuperar algum tipo de informação que depende do resultado de duas tabelas.

Por exemplo, se desejássemos recuperar todos os usuários do sistema que compraram itens da cor vermelha, faríamos algo assim:

```
User.all :joins => :items, :conditions => ["items.color = ?", 'red']
```

Este tipo de sintaxe parece incomodar já que você precisa incluir o nome da tabela (no caso **items**) dentro de uma **string**. O código parece estranho.

No Rails 2.2 encontraremos uma novidade nesta questão, nos permitindo fazer a mesma coisa de uma forma um pouco diferente, usando uma chave dentro do **hash** para identificar a tabela:

```
User.all :joins => :items, :conditions => {
  :age => 10,
  :items => { :color => 'red' }
}

# um outro exemplo que talvez deixe o código mais claro
User.all :joins => :items, :conditions => {
  :users => { :age => 10 },
  :items => { :color => 'red' }
}
```

Na minha opinião, desta forma o código fica muito mais claro, principalmente se temos de condicionar muitos campos de várias tabelas.

Só tenha em mente que a chave usada é o nome da tabela (você percebe pelo nome pluralizado) ou um alias caso você o tenha especificado na query.

Nova opção :from para métodos de cálculo do ActiveRecord

Uma nova opção foi incluída aos métodos de cálculos do **ActiveRecord** (`count`, `sum`, `average`, `minimum` e `maximum`).

Ao fazer uso da opção `:from`, podemos sobrecarregar o nome da tabela na query gerada pelo **ActiveRecord**, o que não parece muito útil em um primeiro momento. Mas algo interessante que esta opção nos permite fazer é forçar o MySQL a usar um índice especifico ao realizar o cálculo desejado.

Veja alguns exemplos de uso:

```
# Forçando o MySQL a usar um índice para realizar o cálculo
Edge.count :all, :from => 'edges USE INDEX(unique_edge_index)',
           :conditions => 'sink_id < 5')

# Realizando o cálculo em uma tabela diferente da classe associada
Company.count :all, :from => 'accounts'
```

Método merge_conditions do ActiveRecord agora é público

O método `merge_conditions` do **ActiveRecord** agora é um método público, o que significa que ele estará presente em todas os seus **Models**.

Este método faz exatamente o que o nome diz. Você pode informar várias **conditions** separadas em seus parâmetros e ele junta tudo em uma condition só. Por exemplo:

```
class Post < ActiveRecord::Base
end

a = { :author => 'Carlos Brando' }
b = [ 'title = ?', 'Edge Rails' ]

Post.merge_conditions(a, b)
# => "(\"posts\".\"author\" = 'Carlos Brando') AND (title = 'Edge Rails')"
```

Note que ele une as **conditions** com um **AND**, sempre.

Definindo como o método validates_length_of deve funcionar

O método `validates_length_of` faz parte dos muitos métodos de validação contidos no **ActiveRecord**. Este método em particular serve para garantir que o valor gravado em uma determinada coluna no banco de dados terá um tamanho máximo, mínimo, exato, ou até mesmo se está em um intervalo de valores.

Mas o termo "tamanho" é relativo. Hoje quando dizemos "tamanho" estamos nos referindo a quantidade de caracteres no texto.

Mas imagine um caso onde eu tenha um campo em um formulário onde a limitação não seja definida pela quantidade de caracteres e sim pela quantidade de palavras, algo como "escreva um texto com no mínimo 100 palavras". Imagine uma página onde o usuário tenha de redigir uma redação, por exemplo.

Hoje, para validar isto não teríamos outra escolha senão criarmos um novo método que faça esta validação. Mas à partir do Rails 2.2 poderemos personalizar o método `validates_length_of` para funcionar da forma como desejamos usando a opção `:tokenizer`.

Veja um exemplo que resolveria o problema citado acima:

```
validates_length_of :essay,
                    :minimum => 100,
                    :too_short => "Sua redação deve ter no mínimo %d palavras."),
                    :tokenizer => lambda {|str| str.scan(/\w+/) }
```

Este é apenas um exemplo do que podemos fazer com esta nova opção. Além disso podemos usa-lá para contar apenas a quantidade de dígitos, menções de uma única palavra, etc..

Tratando a opção :limit como bytes

À partir desta versão do Rails quando usarmos a opção `:limit` para colunas com números inteiros, em nossas migrations, estaremos nos referindo ao número de bytes, no MySQL e no PostgreSQL (no sqlite sempre foi assim).

O tipo da coluna no banco de dados dependerá da quantidade de bytes especificida. Veja o trecho de código que identifica o tipo da coluna para o MySQL:

```
when 1; 'tinyint'
when 2; 'smallint'
when 3; 'mediumint'
when nil, 4, 11; 'int(11)' # compatibility with MySQL default
when 5..8; 'bigint'
else raise(ActiveRecordError, "No integer type has byte size #{limit}")
```

E para o PostgreSQL:

```
when 1..2; 'smallint'
when 3..4, nil; 'integer'
when 5..8; 'bigint'
```

Usando a opção :accessible para fazer atribuições em massa no ActiveRecord

Todos os métodos de associações do **ActiveRecord** (`belongs_to`, `has_one`, `has_many` e `has_and_belongs_to_many`) receberam um nova opção que permite a atribuição de valores para suas associações usando **hashes** aninhados.

Vamos pegar como exemplo um modelo chamado **Post**:

```
class Post < ActiveRecord::Base
  belongs_to :author, :accessible => true
  has_many :comments, :accessible => true
end
```

Note que logo após associar este modelo com **author** e **comments** estou definindo a nova opção `:accessible` como `true` (verdadeiro). Com esta opção "ligada" eu posso fazer algo assim:

```
post = Post.create({
  :title => 'Accessible Attributes',
  :author => { :name => 'David Dollar' },
  :comments => [
    { :body => 'First Post!' },
    { :body => 'Nested Hashes are great!' }
  ]
})
```

Veja que estou criando todas as associações usando apenas um único **hash** com sub-níveis para cada associação. Isto não era possível de forma automática no Rails.

Eu também poderia fazer algo assim, para adicionar um novo comentário:

```
post.comments << { :body => 'Another Comment' }
```

Informando outro primary_key em associações

Uma nova opção foi acrescentado aos métodos `has_many` e `has_one`: a opção `:primary_key`.

Fazendo uso desta opção podemos definir qual método do modelo associado retornará a chave primária que será usada na associação. Obviamente o método padrão é o **id**.

Veja um exemplo de uso:

```
has_many :clients_using_primary_key, :class_name => 'Client',
  :primary_key => 'name', :foreign_key => 'firm_name'
```

O método `has_one` funciona exatamente como no exemplo acima.

Novo helper para testes (assert_sql)

Talvez você já conheça o método `assert_queries` que ajuda a validar nos testes a quantidade de queries executadas. Por exemplo:

```
assert_queries(Firm.partial_updates? ? 0 : 1) { firm.save! }
```

No teste acima estou afirmando que se houver **partial_updates** uma query deve ser executada no banco de dados, caso contrário nenhuma deve ser executada.

Agora ganhamos mais um helper para ajudar a testar o tipo de query executada, o método `assert_sql`. Um exemplo:

```
def test_empty_with_counter_sql
  company = Firm.find(:first)
  assert_sql /COUNT/i do
    assert_equal false, company.clients.empty?
  end
end
```

No exemplo acima estou confirmando que no bloco informado ao método pelo menos uma query deve conter a palavra **COUNT**. Obviamente você pode ser mais especifico na expressão regular que estiver usando. Vamos pegar um outro exemplo:

```
assert_sql(/\(\"companies\".\"id\" IN \(1\)\)/) do
  Account.find(1, :include => :firm)
end
```

Cuidado para não abusar deste recurso.

ActiveSupport

Active Support é uma coleção de várias classes úteis e extensões de bibliotecas padrões, que foram considerados úteis para aplicações em Ruby on Rails. (wikipedia)

Array#second até Array#tenth

No objeto Array já tínhamos o método `first` e `last`, então porque não ter também os métodos `second`, `third`, `fourth` e assim por diante? É isso mesmo, foram acrescentados ao objeto **Array** os métodos que vão do `second` (segundo) até o `tenth` (décimo), que servem para retornar o objeto especifico dentro do **Array** (o terceiro objeto do array, por exemplo).

Vamos aos exemplos:

```
array = (1..10).to_a

array.second  # => array[1]
array.third   # => array[2]
array.fourth  # => array[3]
array.fifth   # => array[4]
array.sixth   # => array[5]
array.seventh # => array[6]
array.eighth  # => array[7]
array.ninth   # => array[8]
array.tenth   # => array[9]
```

Novo método Enumerable#many?

Um novo método foi adicionado ao módulo `Enumerable`: `many?`. E como o nome mesmo diz, ele verifica se a coleção possui mais de um objeto, ou em outras palavras se tem muitos objetos associados.

Este método é um alias para `collection.size > 1`. Vamos ver alguns exemplos:

```
>> [].many?
# => false

>> [ 1 ].many?
# => false

>> [ 1, 2 ].many?
# => true
```

Além deste formato dado nos exemplos, este método também recebeu uma nova implementação permitindo que ele aceite blocos, que funciona exatamente como o método `any?`.

Vamos aos exemplos:

```
>> x = %w{ a b c b c }
# => ["a", "b", "c", "b", "c"]

>> x.many?
# => true

>> x.many? { |y| y == 'a' }
# => false

>> x.many? { |y| y == 'b' }
# => true


# um outro exemplo...
people.many? { |p| p.age > 26 }
```

Apenas para relembrar e reforçar, este método só retornará `true` se mais de um objeto passar nas condições quando usado o bloco, e quando a coleção tiver mais de um objeto quando usado sem condicionais.

Só uma curiosidade, o método inicialmente se chamaria `several?`, mas foi alterado para `many?` depois.

Crie regras para o String#humanize

Já faz um certo tempo que Pratik Naik estava tentando colocar este patch no Rails e parece que finalmente conseguiu.

No arquivo **config/initializers/inflections.rb** você tem a opção de acrescentar novas inflexões para pluralização, singularização e outros:

```
Inflector.inflections do |inflect|
  inflect.plural /^(ox)$/i, '\1en'
  inflect.singular /^(ox)en/i, '\1'
  inflect.irregular 'person', 'people'
  inflect.uncountable %w( fish sheep )
end
```

No Rails 2.2 você também pode incluir inflexões para o método `humanize` da classe `String`. Vamos aos famosos exemplos:

```
'jargon_cnt'.humanize # => "Jargon cnt"
'nomedojogo'.humanize # => "Nomedojogo"

ActiveSupport::Inflector.inflections do |inflect|
  inflect.human(/_cnt$/i, '\1_count')
  inflect.human('nomedojogo', 'Nome do Jogo')
end

'jargon_cnt'.humanize # => "Jargon count"
'nomedojogo'.humanize # => "Nome do jogo"
```

Introduzindo Memoizable para cache de atributos

Performance é coisa séria, e um dos métodos mais usados para aumentar a velocidade de execução em códigos é o uso de cache. Quem nunca fez algo assim?

```
class Person < ActiveRecord::Base
  def age
    @age ||= um_calculo_muito_complexo
  end
end
```

Nesta versão do Rails temos uma forma mais elegante de fazer isto usando o método `memoize` (é `memoize` mesmo e não **memorize**). Vamos alterar o exemplo acima para funcionar com esta nova funcionalidade:

```
class Person < ActiveRecord::Base
  def age
    um_calculo_muito_complexo
  end
  memoize :age
end
```

O método **age** será executado apenas uma vez e o seu retorno será armazenado e retornado em futuras chamadas ao método.

Só existe uma diferença entre os dois códigos acima. No primeiro, como o método é executado todas as vezes, se o valor armazenado na variável **@age** for `nil` ou `false` o cálculo (muito complexo) será executado novamente até termos a idade da pessoa.

No segundo exemplo, o método **age** só será executado uma vez e o valor retornado será sempre devolvido nas próximas chamadas, mesmo que seja `nil` ou `false`.

Novo método Object#present?

Um novo método foi acrescentado à classe `Object`. O método `present?` é o equivalente a `!Object#blank?`.

Em outras palavras um objeto está presente se ele não for vazio. Mas o que é um objeto vazio?

```
class EmptyTrue
  def empty?() true; end
end

a = EmptyTrue.new
b = nil
c = false
d = ''
e = '   '
g = "  \n\t  \r "
g = []
h = {}

a.present? # => false
b.present? # => false
c.present? # => false
d.present? # => false
e.present? # => false
f.present? # => false
```

```
g.present? # => false
h.present? # => false
```

Todos estes objetos são vazios ou não estão presentes.

Mas, muito cuidado, algumas pessoas tem confundido as coisas. Veja alguns exemplos de objetos que NÃO estão vazios, ou seja, estão presentes:

```
class EmptyFalse
  def empty?() false; end
end

a = EmptyFalse.new
b = Object.new
c = true
d = 0
e = 1
f = 'a'
g = [nil]
h = { nil => 0 }

a.present? # => true
b.present? # => true
c.present? # => true
d.present? # => true
e.present? # => true
f.present? # => true
g.present? # => true
h.present? # => true
```

Qualquer objeto que contenha um valor, está presente, isto vale até mesmo para um `Array` preenchido com um `nil`, porque o `Array` não está vazio.

StringInquirer

Uma nova classe foi incluída ao Rails, a classe `StringInquirer`.

Para entender como funciona, vou ter de explicar usando alguns exemplos. Vamos criar uma classe chamada **Cliente** que contém um método que retorna o **status** do cliente:

```
class Cliente
  def status
    "ativo"
  end
```

```
end

c = Cliente.new
c.status
# => "ativo"

c.status == "ativo"
# => true

c.status == "inativo"
# => false
```

Ok, até aqui tudo normal. Agora vou modificar a implementação do método status usando a classe `StringInquirer`, sempre lembrando que o retorno do método **status** pode vir de uma coluna do banco de dados (claro), isto é apenas um exemplo.

```
class Cliente
  def status
    ActiveSupport::StringInquirer.new("ativo")
  end
end

c = Cliente.new
c.status
# => "ativo"

# Agora vem a grande diferença:
c.status.ativo?
# => true

c.status.inativo?
# => false
```

Para verificar se o **status** do cliente é o esperado, ao invés de comparar `Strings`, eu uso um método com o valor do status e o sinal de interrogação.

Claro que isto já começou a ser usado no próprio Rails. Por exemplo, caso você precise verificar se o Rails foi carregado em ambiente de produção, você pode substituir o velho `Rails.env == "production"`, por:

```
Rails.env.production?
```

Nova sintaxe para testes

Uma nova forma de se declarar testes foi adicionada ao Rails, usando declarações **test/do**. Veja:

```
test "verify something" do
  # ...
end
```

Este é o novo padrão para testes do Rails, veja como ficou um arquivo de teste unitário recém criado nesta versão:

```
require 'test_helper'

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

A forma convencional, usando métodos, também continuará funcionando, então nossos testes antigos não quebrarão.

ActiveResource

O ActiveResource é uma camada de mapeamento responsável pela implementação do lado cliente de sistemas RESTful. Através do ActiveResource é possível consumir serviços RESTful através do uso de objetos que funcionam como um proxy para serviços remotos.

ActionPack

Compreende o Action View (geração de visualização de usuário, como HTML, XML, JavaScript, entre outros) e o Action Controller (controle de fluxo de negócio). (wikipedia)

Nova opção :layout no método caches_action

Foi acrescentado a opção `:layout` no método `caches_action`.

```
class ListsController < ApplicationController
  ...

  caches_action :index, :layout => false

  ...
end
```

No exemplo acima eu especifiquei `:layout => false`, isto significa que o layout não será armazenado no cache, apenas o conteúdo da action será. Isto é muito útil quando temos conteúdo dinâmico no layout (o que acontece na maioria dos casos).

Se você não especificar nada ele assumirá o padrão atual que é `true`.

Templates em cache

Para melhorar a performance do Rails as páginas de template serão armazenadas em cache automaticamente em ambiente de produção.

Em outras palavras, se você alterar algum template em produção terá de reiniciar o servidor para que isto tenha efeito.

Alteração no método concat

Se você tem o costume de evitar repetições em suas views criando helpers, com certeza já usou o método `concat`. Se você nunca usou este método, saiba que ele é como o `puts` para uma view.

A implementação atual do método recebe dois parâmetros, uma `String` com o texto que será exibido na view e um segundo chamado `binding`. Acontece que devido a melhorias no código, embora ele ainda espere estes dois parâmetros, o `binding` não é mais necessário, na verdade o método simplesmente não o usa mais.

Então este segundo parâmetro foi deprecado, ou seja, se você estiver informando ele à chamada do método e rodando o seu projeto sob o Rails 2.2, receberá a seguinte mensagem ao subir o seu servidor:

The binding argument of #concat is no longer needed. Please remove it from your views and helpers.

Em uma futura versão do Rails, este segundo parâmetro será removido.

Método link_to com blocos

O método `link_to` recebeu uma atualização que permite seu uso com blocos. Isto é interessante para os casos onde temos textos muito longos no hyperlink. Por exemplo, se hoje fazemos assim:

```
<%= link_to "<strong>#{@profile.name}</strong> -- <span>Check it out!!</span>", @profile
```

Agora podemos fazer assim, que teremos o mesmo resultado:

```
<% link_to(@profile) do %>
  <strong><%= @profile.name %></strong> -- <span>Check it out!!</span>
<% end %>
```

Não é uma mudança significativa em funcionalidade, mas permite deixar o código mais legivel, e isto também é importante.

RJS#page.reload

O método `reload` foi incluído ao `ActionView::Helpers::PrototypeHelper` para ser usado em templates **.rjs** ou blocos `render(:update)`. Este método força a recarga da página atual no browser usando javascript. Em outras palavras é um atalho para o já muito usado `window.location.reload();`.

Veja como usar:

```
respond_to do |format|
  format.js do
    render(:update) { |page| page.reload }
  end
end
```

Dando um nome para a variável local de uma coleção de partials

No código abaixo estamos usando uma **partial** com uma coleção de dados:

```
render :partial => "admin_person", :collection => @winners
```

Dentro da **partial** podemos usar então a variável **admin_person** para acessar os itens da coleção. Mas temos de concordar que este nome de variável é meio ruim.

Agora temos a opção de personalizar o nome desta variável usando a opção `:as`. Vamos alterar o exemplo acima:

```
render :partial => "admin_person", :collection => @winners, :as => :person
```

Agora podemos acessar cada item da coleção usando a variável **person** que tem um nome mais intuitivo.

polymorphic_url agora é capaz de lidar com recursos singleton

Para mais detalhes sobre o que são rotas singulares veja o capítulo "Informações Adicionais" no fim deste livro.

Até agora o helper `polymorphic_url` não estava tratando singleton resources corretamente.

Um novo patch foi incluído no Rails para permitir que especifiquemos um singular resource usando símbolos, assim como fazemos com namespaces. Exemplo:

```
# este código
polymorphic_url([:admin, @user, :blog, @post])

# é a mesma coisa que
admin_user_blog_post_url(@user, @post)
```

Suporte a expressões regulares no time_zone_select

Na classe `TimeZone` do `ActiveSupport` existe o método `us_zones` que convenientemente retorna uma lista dinâmica com todos os fusos-horários americanos.

O problema é que em alguns casos vamos desenvolver software para pessoas em outros países, mas não existe um método tão conveniente assim que liste os fusos-horários destes países.

Ouve uma longa discussão sobre criar ou não métodos como `african_zones`, `american_zones`, etc.. No fim prevaleceu o seguinte:

Foi implantado no objeto `TimeZone` o suporte para `=~` afim de ajudar a montar uma lista de fusos-horários a partir de uma expressão regular. Além disso o método `time_zone_select` foi alterado para trabalhar em conjunto com esta novidade.

Agora podemos fazer isto:

```
<%= time_zone_select( "user", 'time_zone', /Brasilia/) %>
```

Para aprender mais sobre `TimeZones` aconselho assistir ao episódio 160 do RailsCasts (http://railscasts.com/episodes/106) e dar uma olhada no livro antecessor a este.

Fazendo uso da opção accessible em formulários

No capítulo sobre `ActiveRecord` comentei sobre a nova forma de atribuir valores à associações usando a opção `:accessible`.

Agora vou mostrar uma das grandes vantagens dessa novidade. Fazendo uso deste novo recurso podemos melhorar a forma como usamos o método `fields_for`. Vamos pegar um exemplo, baseado nos códigos dados antes:

```
<% form_for @post do |f| %>
  <%= f.text_field :body %>
  <% fields_for :author do |a_f| %>
    <%= a_f.text_field :name %>
  <% end %>
  <%= submit_tag %>
<% end %>
```

Estou usando o método `fields_for` para acrescentar o nome do autor na página de criação de posts. Agora vem a melhor parte, veja como ficaria a criação do post e do autor usando este novo recurso:

```
class PostController < ApplicationController

  def new
    @post = Post.create(params[:post])
    respond_to do |wants|
      ...
    end
  end
end
```

Veja que estou usando apenas o método `create` do objeto `Post`. Como foi definido a opção `:accessible` do relacionamento deste objeto com o objeto `Author` como verdadeiro (`true`), a criação do autor é automática e transparente.

render :template agora aceita :locals

Os métodos `render :action` e `render :partial` permitem que passemos um **hash** através da opção `:locals` com dados para serem processados por eles, mas o `render :template` não permitia.

Nesta versão do Rails isto irá funcionar também:

```
render :template => "weblog/show", :locals => {:customer => Customer.new}
```

:use_full_path do método render não existirá mais

```
render :file => "some/template", :use_full_path => true
```

A opção `:use_full_path` do método render não existe mais no Rails 2.2. Não é algo sério já que ela nem mesmo era necessária.

define_javascript_functions

Mais um método que deixa de existir no Rails: `define_javascript_functions`.

Faz sentido já que métodos como `javascript_include_tag` e outros realizam a mesma tarefa de uma forma melhor.

Desabilitando o cabeçalho Accept em requisições HTTP

Quando usamos o método `respond_to` para fazer algo assim:

```
def index
  @people = Person.find(:all)

  respond_to do |format|
    format.html
    format.xml { render :xml => @people.to_xml }
  end
end
```

O Rails tem duas formas de identificar qual é o formato que deve ser usado, a primeira e mais comum é através do formato informado na URL (/index.xml, por exemplo) e a segunda forma para o caso de o formato não ter sido especificado é examinando o cabeçalho **Accept** da requisição HTTP.

Para quem não sabe o cabeçalho **Accept** é aquele que informa o tipo do documento desejado em **strings** mais ou menos assim:

```
Accept: text/plain, text/html
Accept: text/x-dvi; q=.8; mxb=100000; mxt=5.0, text/x-c

# recuperando esta informação via código
@request.env["HTTP_ACCEPT"] = "text/javascript"
```

Para ver uma lista dos tipos mais comuns acesse a endereço: http://www.developershome.com/wap/detectio

Acontece que este cabeçalho é porcamente implementado na maioria dos browsers. E quando ele é usado em sites públicos algumas vezes estranhos erros acontecem quando robôs indexadores fazem seus requests.

Assim, tomou-se a decisão de desativar este cabeçalho por padrão, depois voltou-se atrás e resolveu-se que era melhor deixar apenas a opção de desligar se desejado. Para isto basta incluir a seguinte linha no arquivo **environment.rb**:

```
config.action_controller.use_accept_header = false
```

Quando desligado, caso o formato não seja informado na URL o Rails assumirá que deve usar o format.html.

Existe um caso especial quando fazemos requisições usando ajax se o cabeçalho **X-Requested-With** estiver presente. Neste caso o formato **format.js** ainda será usado mesmo que o formato não tenha sido especificado (/people/1, por exemplo).

button_to_remote

Se você já usa Rails há um tempo, com certeza conhece o método `submit_to_remote`, certo? Este método retorna um botão HTML preparado para enviar o formulário via **XMLHttpRequest**.

No Rails 2.2 este método foi renomeado para `button_to_remote`, para manter uma certa consistência com o nome do seu método irmão o `link_to_remote`.

Aqueles que irão migrar seus projetos não precisam se preocupar já que o nome antigo (`submit_to_remote`) será um alias para novo.

Recebendo alertas para melhoria de desempenho

O Rails recebeu mais um parâmetro de configuração que faz com que ele emita um alerta caso esteja renderizando um template fora do diretório especificado para views. Isto é muito bom já que arquivos fora dos diretórios especificados para views não são armazenados em cache o que resulta em mais operações no disco.

Para começar a receber os avisos basta incluir a seguinte linha no arquivo **environment.rb** do projeto:

```
config.action_view.warn_cache_misses = true
```

Fazendo isto a seguinte mensagem aparecerá caso algum arquivo fora dos diretórios configurados seja renderizado:

```
[PERFORMANCE] Rendering a template that was not found in view path. Templates outside th
```

Esta configuração está desligada como padrão.

Opção :index funcionando no fields_for

Muitas vezes a opção `:index` do método `select` pode ser útil, como por exemplo quando se precisa criar diversos dropdowns dinamicamente em uma página.

Até agora o método `fields_for` não repassava esta opção para métodos como `select`, `collection_select`, `country_select` e `time_zone_select`, o que limitava o seu uso em determinados casos.

Isto já foi corrigido nesta versão do Rails. Por exemplo, apenas para efeito de teste veja o código abaixo e o seu retorno:

```
fields_for :post, @post, :index => 108 do |f|
  concat f.select(:category, %w( abe <mus> hest))
end
```

```
<select id=\"post_108_category\" name=\"post[108][category]\">
  <option value=\"abe\">abe</option>
  <option value=\"&lt;mus&gt;\" selected=\"selected\">&lt;mus&gt;</option>
  <option value=\"hest\">hest</option>
</select>
```

Melhorando a performance usando ETags

Antes de começar a falar sobre isto, deixa eu tentar explicar o que são ETags (Entity Tags). Etags seriam de uma forma grosseira identificadores associados a cada recurso para determinar se o arquivo que está no servidor é o mesmo que está no cache do browser. No caso, o recurso seria uma página em HTML, mas também poderia ser um XML ou JSON.

Cabe ao servidor a responsabilidade de verificar se o recurso solicitado é igual dos dois lados. Caso o servidor confirme que o recurso armazenado no cache do browser do usuário é exatamente o mesmo que seria enviado de volta para ele, ao invés de devolver todo o conteúdo do recurso novamente ele apenas retorna um status **304** (Not Modified) e o browser usará o que está em seu cache.

Servidores Web como o Apache e o IIS já sabem fazer isto para página estáticas. Mas quando o conteúdo é dinâmico, como na maioria das páginas de um projeto Ruby on Rails, a responsabilidade é todo nossa.

O objeto `response` ganhou dois novos métodos, o `last_modified=` e o `etag=` (note o sinal de igualdade "="). Quando você atribuir valores a estes métodos eles serão repassados aos cabeçalhos **HTTP_IF_MODIFIED_SINCE** e **HTTP_IF_NONE_MATCH** respectivamente. Quando uma nova requisição (`request`) deste recurso for feita ela retornará com estes cabeçalhos permitindo que você possa comparar com o novo valor informado no `response` e decidir se quer enviar todo o conteúdo do recurso novamente ou apenas um status 304.

Vamos ao código (roubando o exemplo dado por Ryan Daigle em seu blog):

```
class ArticlesController < ApplicationController
  def show
    @article = Article.find(params[:id])

    # informado o identificador
    response.last_modified = @article.published_at.utc
    response.etag = @article

    # verificando se houve mudanças
    if request.fresh?(response)
      head :not_modified
    else
      respond_to do |wants|
        # retornando conteúdo normalmente
      end
    end
end
```

Note que estou atribuindo valores para `response.last_modified` e `response.etag`, e logo em seguinda usando o método `request.fresh?(response)` para determinar se houve alguma mudança ou não. Caso os dois cabeçalhos continuem iguais eu apenas retorno um status 304 e o browser do usuário usará o conteúdo que está armazenado no seu cache.

Quanto ao método `etag`, também podemos passar um **array** se for o caso:

```
response.etag = [@article, current_user]
```

Uma forma mais simples de usar ETags

Também ganhamos mais dois novos métodos que devem facilitar a forma como vamos implementar ETags. Os métodos `etag!` e `last_modified!`.

Veja a implementação destes métodos direto do código fonte do Rails:

```
# Sets the Last-Modified response header. Returns 304 Not Modified if the
# If-Modified-Since request header is <= last modified.
def last_modified!(utc_time)
  head(:not_modified) if response.last_modified!(utc_time)
end

# Sets the ETag response header. Returns 304 Not Modified if the
# If-None-Match request header matches.
```

```
def etag!(etag)
  head(:not_modified) if response.etag!(etag)
end
```

Isto facilita muito as coisas. Veja como ficaria o exemplo dado no tópico anterior usando estes novos métodos:

```
class ArticlesController < ApplicationController
  def show
    @article = Article.find(params[:id])

    etag! @article
    last_modified! @article.published_at.utc
  end
end
```

Bem mais simples!

Alteração na assinatura do método number_with_delimiter

O método `number_with_delimiter` recebeu uma nova implementação. Além de uma melhora no código para que fique mais limpo a assinatura do método também mudou. Veja a antiga:

```
def number_with_delimiter(number, delimiter=",", separator=".")

# Exemplos de uso
number_with_delimiter(12345678) # => 12,345,678
number_with_delimiter(12345678.05) # => 12,345,678.05
number_with_delimiter(12345678, ".") # => 12.345.678
number_with_delimiter(98765432.98, " ", ",")
```

E a nova:

```
def number_with_delimiter(number, *args)

# Exemplos de uso
number_with_delimiter(12345678) # => 12,345,678
number_with_delimiter(12345678.05) # => 12,345,678.05
number_with_delimiter(12345678, :delimiter => ".") # => 12.345.678
number_with_delimiter(12345678, :seperator => ",") # => 12,345,678
number_with_delimiter(98765432.98, :delimiter => " ", :separator => ",")
```

Então atenção, agora ao usarmos o método `number_with_delimiter` devemos informar as opções na chamada do método.

Executando múltiplas instâncias de um projeto em subdiretórios

Às vezes você tem de rodar múltiplas cópias do mesmo projeto. Talvez você tenha um produto que será usado por vários clientes, ou talvez você apenas deseje rodar uma versão de teste e produção do seu software ao mesmo tempo.

A forma mais simples de se fazer isto é ter múltiplos (sub)domínios com uma instância do aplicativo em cada uma. Mas se isto não for possível você pode colocar seu aplicativo em um subdiretório e usar um prefixo na sua URL para distinguir as instâncias do seu software. Por exemplo, você poderia rodar vários blogs de usuários diferentes usando URLs como:

- http://www.nomedojogo.com/fulano/blog
- http://www.nomedojogo.com/sicrano/blog
- http://www.nomedojogo.com/beltrano/blog

Nestes casos, os prefixos **fulano**, **sicrano** e **beltrano** identificarão as instâncias do aplicativo rodando em subdiretórios com os mesmos nomes. O roteamento do aplicativo começa depois disto. Você pode dizer ao Rails para ignorar esta parte das URLs quando uma requisição for feita, mas coloca-la nas URLs geradas por ele, configurando isto através da constante **RAILS_RELATIVE_URL_ROOT** ou do método `AbstractRequest.relative_url_root`.

Porém se seu projeto Rails estiver rodando sob o Apache, esse recurso já é ativado automaticamente, por isto na maioria dos casos não temos de nos preocupar em configurar isto hoje. Isto se você estiver usando Apache.

Porém, no Rails 2.2 o `relative_url_root` não será mais configurado automaticamente pelo HTTP header. Teremos de fazer isto manualmente, colocando uma linha mais ou menos assim no arquivo **environment.rb** de cada um dos aplicativos:

```
config.action_controller.relative_url_root = "/fulano"
```

Feito isto, seu aplicativo irá ignorar o prefixo "fulano" logo depois do domínio. Mas ao gerar URLs ele sempre colocará este prefixo para garantir que você estará acessando o projeto no subdiretório correto.

Alteração no método error_message_on

O método `error_message_on` é extremamente útil. Com ele podemos exibir mensagens de erro retornadas por determinados métodos em um objeto de uma forma bem simples.

```
<%= error_message_on "post", "title" %>

<!-- ou -->

<%= error_message_on @post, "title" %>
```

Isto fará com que uma mensagem de erro seja exibida na sua página dentro de uma tag DIV, caso um erro esteja associado ao campo title do modelo post.

Mas o mais interessante do método `error_message_on` é que podemos personaliza-lo para que exibida mensagens mais amigáveis. E é aqui que entra a alteração para o Rails 2.2.

Na versão atual os parâmetros de personalização são passadas diretamente para o método, mas no Rails 2.2 serão passadas via um **Hash** de opções:

```
<%= error_message_on "post", "title",
      :prepend_text => "Title simply ",
      :append_text => " (or it won't work).",
      :css_class => "inputError" %>
```

Fique tranqüilo quanto a uma possível migração de seus projetos atuais, pois o código está preparado para funcionar também da forma antiga (pelo menos por um tempo), mas emitindo um aviso de alerta para que o código seja atualizado.

Mais métodos atualizados para receber Hashes de opções

truncate

```
truncate("Once upon a time in a world far far away")
# => Once upon a time in a world f...

truncate("Once upon a time in a world far far away", :length => 14)
# => Once upon a...

truncate("And they found that many people were sleeping better.", :omission => "... (con
# => And they found... (continued)
```

highlight

```
highlight('You searched for: rails', ['for', 'rails'], :highlighter => '<em>\1</em>')
# => You searched <em>for</em>: <em>rails</em>

highlight('You searched for: rails', 'rails', :highlighter => '<a href="search?q=\1">\1<
# => You searched for: <a href="search?q=rails">rails</a>
```

excerpt

```
excerpt('This is an example', 'an', :radius => 5)
# => ...s is an exam...

excerpt('This is an example', 'is', :radius => 5)
# => This is a...

excerpt('This next thing is an example', 'ex', :radius => 2)
# => ...next...

excerpt('This is also an example', 'an', :radius => 8, :omission => '<chop> ')
# => <chop> is also an example
```

word_wrap

```
word_wrap('Once upon a time', :line_width => 8)
# => Once upon\na time

word_wrap('Once upon a time', :line_width => 1)
# => Once\nupon\na\ntime
```

auto_link

```
post_body = "Welcome to my new blog at http://www.myblog.com/. Please e-mail me at me@ema

auto_link(post_body, :urls)
# => "Welcome to my new blog at
    <a href=\"http://www.myblog.com/\">http://www.myblog.com</a>.
    Please e-mail me at me@email.com."

auto_link(post_body, :all, :target => "_blank")
# => "Welcome to my new blog at
    <a href=\"http://www.myblog.com/\" target=\"_blank\">http://www.myblog.com</a>.
    Please e-mail me at <a href=\"mailto:me@email.com\">me@email.com</a>."
```

Todos os métodos continuam funcionando da forma antiga por enquanto, mas alertas serão emitidos no terminal para lembra-lo de atualizar seu código o mais rápido possível.

ActionController

O ActionController é a camada responsável por receber as requisições web e de tomar as decisões do quê será executado e renderizado ou de redirecionar para outra ação. Uma ação é definido como métodos públicos nos controladores que são automaticamente disponíveis através das rotas.

ActionView

O ActionView é a camada responsável pela geração da interface visível ao usuário através da conversão dos templates ERB.

Railties

Está chegando o fim dos plugins?

No Rails 2.1, gems passaram a poder ser usadas como plugins em nossos projetos. Para isto bastava criar uma pasta chamada **rails** dentro do projeto do gem e incluir um arquivo **init.rb**.

Isto acrescentou um leque de novidades como `config.gem` e `rake:gems`. Mas isto nos faz pensar, já que agora eu posso carregar gems dentro da minha aplicação Rails, seria apenas uma questão de tempo até que plugins deixassem de existir.

E parece que isto realmente pode acontecer. Para esta versão do Rails, por exemplo, foi incluída uma alteração que permite inicializar plugins tanto pelo arquivo **init.rb** na raiz do plugin, como em um arquivo em um diretório **rails/init.rb** (da mesma forma como fazemos com os gems), sendo esta segunda opção a prioritária.

Assim, eu poderia por exemplo criar um gem (que funcionaria como um plugin) e instalar de duas maneiras:

```
./script/plugin install git://github.com/user/plugin.git
```

```
sudo gem install user-plugin --source=http://gems.github.com
```

Isto sem precisar manter dois arquivos **init.rb** (um na raiz e outro no diretório rails).

Suporte ao Thin melhorado no Rails

O `script/server` agora verifica a disponibilidade do **Thin** e o usa. Muito conveniente se vocês estiver usando **Thin** no seu ambiente de produção (e quiser rodar o mesmo em desenvolvimento). Você deve acrescentar a seguinte linha no seu arquivo **environment.rb** para que isto funcione.

```
config.gem 'thin'
```

Testando apenas arquivos uncommitteds no Git

Existe uma tarefa **rake** no Rails que pouca gente conhece mas que é muito útil:

```
rake test:uncommitted
```

Como o nome já diz esta tarefa roda os testes apenas dos arquivos que ainda não foram enviados (commit) para o subversion, ao invés de rodar todos os testes do projeto.

Eu costumava usar isto muito, mas quando mudei para Git ela não funcionou mais, o suporte era apenas para o SVN. Mas um patch foi aplicado no Rails garantindo que daqui em diante teremos o mesmo suporte também para o Git.

Rails.initialized?

Novo método adicionado ao Rails:

```
Rails.initialized?
```

Ele informa se todos os processos de inicialização já foram finalizados.

cache_classes agora estará ligado por padrão

Nos arquivos de configuração do seu projeto provavelmente deve haver uma linha assim:

```
config.cache_classes = false
```

Esta linha informa ao Rails que ele não deve fazer cache do código de seu projeto, ou seja, para cada requisição feita ele carregará o código novamente. Embora isto torne a execução de seu código mais lenta, é ótimo para o ambiente de

desenvolvimento, assim você não precisa ficar recarregando o seu servidor web a cada alteração.

Em produção é importantíssimo que você deixe isto ligado.

Em projetos Rails 2.1, caso a linha acima não se encontre em seus arquivos de configuração, o Rails assumirá que não deve fazer o cache, esta era a condição padrão.

No Rails 2.2 isto foi invertido, caso nenhuma configuração seja encontrada ele assumirá que deve fazer o cache. Isto ajudará os inexperientes que colocam seus projetos em produção sem configurá-lo corretamente.

Rake Tasks, Plugins e Scripts

Internacionalização (I18n)

Tudo começou em setembro de 2007 quando um grupo de desenvolvedores começou a construção de um plugin para o Rails chamado **rails-I18n**, que visava eliminar a necessidade de monkey patching no Rails para internacionalizar uma aplicação.

O que é I18n?

Primeiro, para entender o porque deste nome, precisamos ter um conhecimento profundo de cálculos matemáticos. Conte comigo, quantas letras temos entre o I e o n na palavra Internationalization? 18. Muito bom, I18n.

O mesmo vale para Localization e L10n.

Já viu quando um site tem aquelas bandeirinhas no topo, permitindo que você escolha em que língua quer navegar? Quando você clica em uma delas, todos os textos do site mudam para a língua correspondente daquele país. Isto se chama Internationalization, ou como acabamos de aprender I18n.

Claro que estou sendo muito simplista, na maioria das vezes não é só os textos que mudam de um país para outro. Não podemos nos esquecer do formato da data, fuso-horário, padrões de peso e medida. E talvez o mais importante a moeda.

I18n e L10n, qual a diferença?

Internacionalização é preparar seu software para que pessoas de outros países e línguas possam usá-lo.

Localização (L10n) é adaptar o seu produto as necessidades de um país. É como pegar um site americano que só aceita pagamento via PayPal e adaptá-lo para aceitar boleto bancário, por exemplo.

O que tem de novo no Rails?

No Rails 2.2 este plugin de internacionalização será integrado ao seu código fonte.

Isto não significa que o Rails sofreu grandes alterações. Na verdade quase nada mudou, ele continua sendo o mesmo framework de sempre, com todas as suas mensagens de validações em inglês e tudo mais.

A diferença é que se quiséssemos estas mensagem em português, ou em outro idioma, teríamos de criar um monkey patch para isto. Não posso deixar de citar como exemplo o famoso Brazilian Rails, que faz exatamente isto para traduzir as mensagens do Active Record.

A novidade é que com o lançamento do Rails 2.2 temos uma forma padronizada e mais simples de fazer isto, usando uma interface comum.

Como isto funciona?

Básicamente este gem é dividido em duas partes:

- A primeira acrescenta à API do Rails uma coleção de novos métodos, estes métodos basicamente servirão para acessar a segunda parte do gem.
- A segunda parte é um backend simples que implementa toda a lógica para fazer o Rails funcionar exatamente como funcionava antes, usando a localização en-US.

A grande diferença é que esta segunda parte poderá ser substituída por uma que dê suporte à internacionalização que você deseja. Melhor ainda, uma série de plugins que serão lançados irão fazer exatamente isto.

O alicerce desta implementação é um módulo chamado `I18n` que vem através do gem incorporado ao `ActiveSupport`. Este módulo acrescenta as seguintes funcionalidades ao Rails:

- O método `translate`, que será usado para retornar traduções.
- O método `localize`, que usaremos para "traduzir" objetos `Date`, `DateTime` e `Time` para a localização atual.

Além destes métodos este módulo traz todo o código necessário para armazenar e carregar os backends de "localização". E já vem com um manipulador de exceções padrão que captura exceções levantadas no backend.

Tanto o backend como o manipulador de exceções podem (devem) ser substituídos. Além disso para facilitar, os métodos `#translate` e `#localize` também poderão ser executados usando os métodos `#t` e `#l` respectivamente.

Exemplos práticos

A melhor forma de enteder como usar este suporte a internacionalização no Rails é vendo seu funcionamento na prática. Eu aconselho então acessar o projeto criado por Sven Fuchs e outros no endereço: **http://i18n-demo.phusion.nl/**.

Ruby 1.9

Performace

Melhorando a performace do Rails

Jeremy Kemper andou trabalhando em melhorias de performace no Rails.Uma das coisas que ele andou melhorando foi o **Erb**, tornando-o mais rápido. Além disso ele tem atacado alguns métodos especiais como o `concat` e `capture` que são usados por muitos **helpers** do Rails.

Jeremy também atacou o processo de inicialização de **partials** e otimizou diversos helpers que geravam código em **Javascript**.

A classe `RecordIdentifier` também foi melhorada através do uso de caches. O `RecordIdentifier` incorpora uma série de convenções para lidar com registros **ActiveRecords** e **ActiveResources** ou praticamente qualquer outro tipo de modelo que tenha um id.

É interessante ver este tipo de ação, o Rails já está ficando grande e pesado demais, e processos de otimização devem se tornar uma contante daqui para frente.

Criando testes de performace

No Rails 2.2 ganhamos um novo **generator** para testes de performace.

Ao executar no terminal o seguinte comando:

```
[carlosbrando:edge]$ ./script/generate performance_test Login
      exists  test/performance/
      create  test/performance/login_test.rb
```

Será criado um arquivo chamado **test/performance/login_test.rb**. Veja o código gerado:

```
require 'performance/test_helper'

class LoginTest < ActionController::PerformanceTest
  # Replace this with your real tests.
  def test_homepage
    get '/'
  end
end
```

Neste arquivo podemos colocar todos os testes que desejarmos e ao executá-lo teremos informações sobre cada um dos testes, como tempo de processamento, uso de memória e outros. Para realizar o teste executamos no terminal:

```
[carlosbrando:edge]$ ruby test/performance/login_test.rb
Loaded suite test/performance/login_test
Started
LoginTest#test_homepage (32 ms warmup)
        process_time: 11 ms
              memory: unsupported
             objects: unsupported
.
Finished in 0.870842 seconds.
```

Bugs e Correções

ActiveRecord

Correção de uma colisão entre named_scope e :joins.

Quando se usava `with_scope` junto com `:joins` todos os atributos da tabelas secundárias eram adicionados ao modelo da tabela principal.

Partial updates não atualizavam o lock_version se nada foi alterado.

Quando usávamos optimistic locking com partial updates, tinhamos queries extras quando na verdade elas não eram necessárias.

end_of_quarter

Nem bem havia saído o Rails 2.1 e já foi encontrado um erro sério. Se você ainda tiver um projeto criado nesta versão entre no **irb** e tente rodar isto:

```
Date.new(2008, 5, 31).end_of_quarter
```

ERRO!

Por que? A implementação do método `end_of_quarter` foi feita da maneira errada, ele avança até o último mês do trimestre e depois pega último dia. O problema é que ele apenas avança o mês, e como estou partindo do dia 31 de maio, ele tentar criar uma nova instância do objeto `Date` para 31 de junho, que não existe. Com o objeto `Time` não é disparado uma exceção, mas ele retorna a data errada: 31 de julho.

Nesta versão este erro já foi corrigido, mas caso você ainda esteja usando a versão 2.1 em algum projeto, muito cuidado, porque este erro só ocorrerá se usarmos o método `end_of_quarter` nos dias 31 de maio, julho e agosto.

Correção nas tarefas db:migrate:down e :up

Quando se usava o comando `rake db:migrate:down VERSION=alguma_versão`, os registros na tabela **schema_migrations** não eram atualizados.

Isto significa que após usar o comando `rake db:migrate:down` ou `up` se você rodar o comando `rake db:migrate` algumas **migrations** podem não ser executadas. Vamos simular isto para ficar fácil de entender o problema:

```
$ ./script/generate migration test_migration
      create  db/migrate
      create  db/migrate/20080608082216_test_migration.rb

$ rake db:migrate
(in /Users/erik/projects/railstest)
== 20080608082216 TestMigration: migrating ====================================
-- create_table("foo")
   -> 0.0137s
== 20080608082216 TestMigration: migrated (0.0140s) ===========================

$ rake db:migrate:down VERSION=20080608082216
(in /Users/erik/projects/railstest)
== 20080608082216 TestMigration: reverting ====================================
-- drop_table("foo")
   -> 0.0139s
== 20080608082216 TestMigration: reverted (0.0144s) ===========================
```

```
$ rake db:migrate
(in /Users/erik/projects/railstest)

$
```

Este problema foi corrigido ao se certificar de atualizar a tabela **schema_migrations** após a execução destas tarefas.

PostgreSQL

No PostgreSQL, a sintaxe dele de **typecast** é a seguinte: :: O problema é que quando se usava essa sintaxe, o `ActiveRecord` achava que o na verdade era um named bind e reclamava que o valor para ele não estava sendo passado no **hash**. Agora este problema está corrigido, permitindo que façamos algo assim:

```
:conditions => [':foo::integer', { :foo => 1 }]
```

Solução de bug no método rename_column

Esta alteração trata-se na verdade de uma correção de um bug no método `rename_column`. Para entender qual era o problema precisamos de um cenário como exemplo. Primeiro criamos um **migration**:

```
create_table "users", :force => true do |t|
  t.column :name, :string, :default => ''
end
```

Ok, agora criamos um segundo **migration** onde vamos renomear a coluna **name** da tabela:

```
rename_column :users, :name, :first_name
```

Se você fizer o teste em sua máquina, notará que ao usar o método `rename_column` a "nova" coluna `first_name` não terá mais o valor padrão definido no primeiro **migration**.

Este bug já está resolvido para esta versão do Rails.

Informações Adicionais

O que é uma Rota Singular?

Além do `map.resources`, há também uma forma singular (ou "singleton") de rotear recursos: `map.resource`. Esta forma é usada para representar um recurso que só aparece uma vez no contexto.

Faz muito sentido usar uma rota singular quando temos um recurso que será único dentro da aplicação ou da sessão do usuário corrente.

Por exemplo, em um projeto de uma agenda cada usuário registrado tem seu próprio catálogo de endereços, então poderíamos cria nossa rota assim:

```
map.resource :address_book
```

Com isto podemos usar todo o conjunto de recursos disponibilizados pelo Rails, tais como:

```
GET/PUT address_book_url
GET     edit_address_book_url
PUT     update_address_book_url
```

Note que tudo está no singular. Estamos assumindo que no contexto atual não precisamos especificar qual catálogo de endereços desejamos, ao invés disso podemos simplesmente dizer "o catálogo de endereços", já que o usuário atual só tem um.

O relacionamento entre a tabela de catálogos e o usuário corrente não é automático, você deve autenticar o usuário e retornar o catálogo dele. Não existe mágica aqui, é apenas uma outra técnica de roteamento a nossa disposição, se precisarmos.

CHANGELOG