RUBY ON RAILS 2.1

WHAT'S NEW ?

CARLOS BRANDO REVIEW: MARCOS TAPAJÓS - COVER: DANIEL LOPES

Ruby on Rails 2.1

WHAT'S NEW

Second Edition

Ruby on Rails 2.1

WHAT'S NEW

Second Edition

Carlos Brando
Marcos Tapajós

Second edition: June 2008

Carlos Brando
Website: www.nomedojogo.com

Marcos Tapajós
Website: www.improveit.com.br/en/company/tapajos

Chapter 1

Introdução

Por volta do mês de julho de 2004 David Heinemeier Hansson lançou publicamente o framework Ruby on Rails, que havia sido extraído de um software chamado Basecamp. Mais de três anos depois, no dia 7 de dezembro de 2007 o Ruby on Rails chegou a sua versão 2.0 com diversas alterações importantes.

De lá para cá se passaram seis meses, e neste tempo mais de **1400 programadores** do mundo todo contribuiram criando **1600 patches**. E hoje, 1 de junho de 2008 o Ruby on Rails chega à sua versão 2.1.

De acordo com David as principais novidades nesta versão são:

- Timezones
- Dirty tracking
- Gem Dependencies
- Named scope
- UTC-based migrations

- Better caching

Para atualizar ou instalar a nova versão, é o de sempre:

```
gem install rails
```

AGRADECIMENTOS

Ao Marcos Tapajós que é o co-autor deste livro. Se não fosse por ele acho que você não estaria lendo isto.

Ao Daniel Lopes que fez uma linda capa para esta edição.

A toda a comunidade brasileira de Ruby on Rails que colaborou direta ou indiretamente com este livro, comentando os textos no blog e dando sugestões. É como sempre costumo dizer, o melhor do Rails é a comunidade! Continuem criando, inventando e principalmente compartilhando...

Chapter 2

ActiveRecord

O Active Record 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** afim 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 diferentes.

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 associada a classe
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..

Chapter 3

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, 'en'
```

```
  inflect.singular /^(ox)en/i, ''
  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, '_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.

Chapter 4

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.

Chapter 5

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**.

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)
```

Chapter 6

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.

Chapter 7

ActionView

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

Chapter 8

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'
```

Chapter 9

Rake Tasks, Plugins e Scripts

Chapter 10

Prototype e script.aculo.us

Chapter 11

Ruby 1.9

Chapter 12

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.
```

Chapter 13

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.

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.

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.

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.

Chapter 14

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.

Chapter 15

CHANGELOG

RUBY ON RAILS 2.1

WHAT'S NEW?

CARLOS BRANDO REVIEW: MARCOS TAPAJÓS - COVER: DANIEL LOPES