#contents

Ruby on Rails 2.1 - O que tem de novo?

ActiveRecord

ActiveSupport

ActiveResource

ActionPack

TimeZone

Definindo um fuso-horário padrão

Uma nova opção foi acrescentada ao método **time_zone_select**, agora você pode indicar um valor padrão para os casos em que o seu usuário ainda não tenha selecionado nenhum **TimeZone**, ou quando a coluna no banco de dados for nula. Para isto foi criada a opção **:default**, então você poderá usar o método das seguintes maneiras:

```
time_zone_select("user", "time_zone", nil, :include_blank => true)
time_zone_select("user", "time_zone", nil, :default => "Pacific Time (US & Canada)" )
time_zone_select( "user", 'time_zone', TimeZone.us_zones, :default => "Pacific Time (US & Canada)")
```

Nos casos onde usamos a opção **:default** deve aparecer com o **TimeZone** informado já selecionado.

Auto Link

Para quem não conhece, o método **auto_link** recebe um texto qualquer como parâmetro e se o texto tiver algum endereço de email ou de um site ele retorna o mesmo texto com hyperlinks.

Por exemplo:

```
auto_link("Acesse este endereço: http://www.rubyonrails.com")
# => Acesse este endereço: http://www.rubyonrails.com
```

Acontece que alguns sites como o Amazon estão usando também o sinal de "=" (igual) em suas URLs, e este método não reconhece este sinal. Veja como o método se comporta neste caso:

```
auto_link("Acesse este endereço: http://www.amazon.com/Testing-Equal-Sign-In-Path/ref=pd_bbs_sr_1?ie=UTF8&s=books
# => Acesse este endereço: http://www.amazon.com/Testing-Equal-Sign-In-Path/ref=pd_bbs_sr_1?ie=UTF8&s=books&qid=1
```

Note que o método terminou o hyperlink exatamente antes do sinal de "=", pois ele não suporta este sinal. Quer dizer, não suportava.
Nesta nova versão do Rails já temos este problema resolvido.

Rótulos

Ao criar um novo formulário usando **scaffold** ele será criado com o seguinte código:

```
<% form_for(@post) do |f| %>
  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit "Update" %>
  </p>
<% end %>
```

Desta forma faz muito mais sentido. O método **label** foi incluído. Este método retorna uma *string* com o título da coluna dentro de uma tag HTML **<label>**.

```
>> f.label :title
=> <label for="post_title">Title</label>

>> f.label :title, "A short title"
=> <label for="post_title">A short title</label>
```

```
>> label :title, "A short title", :class => "title_label"
=> <label for="post_title" class="title_label">A short title</label>
```

Percebeu o parâmetro **for** dentro da tag? O "post_title" é o nome da caixa de texto que contém o título do nosso post. A tag **<label>** é na verdade um rótulo associado ao objeto **post_title**. Quando se clica no rótulo (ele não é um link) o controle associado à ele recebe o foco.

Robby Russell escreveu um artigo interessante em seu blog sobre este assunto. Você pode lê-lo no endereço:
http://www.robbyonrails.com/articles/2007/12/02/that-checkbox-needs-a-label

Também foi incluído o método **label_tag** no **FormTagHelper**. Este método funciona exatamente como o label mas de uma forma mais simplista:

```
>> label_tag 'nome'
=> <label for="nome">Nome</label>

>> label_tag 'nome', 'Seu nome'
=> <label for="nome">Seu Name</label>

>> label_tag 'nome', nil, :class => 'small_label'
=> <label for="nome" class="small_label">Nome</label>
```

Uma nova forma de usar partials

Algo muito normal no desenvolvimento de softwares em Rails é o uso de partials para evitar a repetição de código. Vejamos um exemplo de seu uso:

```
<% form_for :user, :url => users_path do %>
    <%= render :partial => 'form' %>
    <%= submit_tag 'Create' %>
<% end %>
```

Partial é um fragmento de código (um template). A vantagem de se usar uma partial é evitar a repetição desnecessária de código. Para usar uma partial é muito simples, você pode começar com algo mais ou menos assim: **render :partial => "name"**. Depois deve criar um arquivo com o mesmo nome da partial, mas com um underscore na frente, só isso.

O código acima é a forma como estamos acostumados a fazer hoje, mas nesta nova versão do Rails, faremos a mesma coisa de uma forma um pouco diferente, assim:

```
<% form_for(@user) do |f| %>
    <%= render :partial => f %>
    <%= submit_tag 'Create' %>
<% end %>
```

Neste exemplo nós vamos renderizar a partial "users/_form", que receberá uma variável chamada form com as referências criadas pelo **FormBuilder**.

A forma antiga também vai continuar funcionando.

Novos namespaces no Atom Feed

Conhece o método **atom_feed**? Ele é uma novidade no Rails 2.0, que facilitou muito a criação de feeds Atom. Veja um exemplo de uso:

Em um arquivo *index.atom.builder*:

```
atom_feed do |feed|
  feed.title("Nome do Jogo")
  feed.updated((@posts.first.created_at))

  for post in @posts
    feed.entry(post) do |entry|
      entry.title(post.title)
      entry.content(post.body, :type => 'html')
```

```
      entry.author do |author|
        author.name("Carlos Brando")
      end
    end
  end
end
```

O que é um atom feed? Atom é o nome de um estilo baseado em XML e meta data. Em outras palavras é um protocolo quer serve para publicar conteúdo na internet que é sempre atualizado, como um blog, por exemplo. Os feeds sempre são publicados em XML e no caso do Atom Feed ele é identificado como application/atom+xml media type.

Nas primeiras versões do Rails 2.0 este método aceitava como parâmetros as opções **:language**, **:root_url** e **:url**, você pode obter mais informações sobre estes métodos na documentação do Rails. Mas com a alteração realizada, agora podemos incluir novos namespaces ao elemento root do feed. Por exemplo, se fizermos assim:

```
atom_feed('xmlns:app' => 'http://www.w3.org/2007/app') do |feed|
```

Ele retornará isto:

```
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom" xmlns:app="http://www.w3.org/2007/app">
```

Adaptando o exemplo anterior, poderíamos usá-lo assim:

```
atom_feed({'xmlns:app' => 'http://www.w3.org/2007/app',
           'xmlns:openSearch' => 'http://a9.com/-/spec/opensearch/1.1/'}) do |feed|

  feed.title("Nome do Jogo")
  feed.updated((@posts.first.created_at))
  feed.tag!(openSearch:totalResults, 10)

  for post in @posts
```

```
    feed.entry(post) do |entry|
      entry.title(post.title)
      entry.content(post.body, :type => 'html')
      entry.tag!('app:edited', Time.now)

      entry.author do |author|
        author.name("Carlos Brando")
      end
    end
  end
end
```

Cache

Todos os métodos **fragment_cache_key** agora retornam por padrão o namespace 'view/' como prefixo.

Todos os caching stores foram retirados de **ActionController::Caching::Fragments::** *e agora estão em* ***ActiveSupport::Cache::***. Neste caso se você faz referência a um store, como **ActionController::Caching::Fragments::MemoryStore**, por exemplo, será necessário alterar sua referência para **ActiveSupport::Cache::MemoryStore**.

ActionController::Base.fragment_cache_store deixa de existir e dá lugar à **ActionController::Base.cache_store**.

Foi incluído no **ActiveRecord::Base** o método **cache_key** para facilitar o armazenamento em cache de Active Records pelas novas bibliotecas **ActiveSupport::Cache::***. Este método funciona assim:

```
>> Product.new.cache_key
=> "products/new"

>> Product.find(5).cache_key
=> "products/5"
```

```
>> Person.find(5).cache_key
=> "people/5-20071224150000"
```

Foi incluído o **ActiveSupport::Gzip.decompress/compress** para facilitar o wrapper para o **Zlib**.

Agora você pode usar entre as opções de environment o **config.cache_store** para informar o local padrão de armazenamento do cache. Vale lembrar que se o diretório **tmp/cache** existir o padrão é o **FileStore**, caso contrário o **MemoryStore** é usado. Você pode configurar das seguintes formas:

```
config.cache_store = :memory_store
config.cache_store = :file_store, "/path/to/cache/directory"
config.cache_store = :drb_store, "druby://localhost:9192"
config.cache_store = :mem_cache_store, "localhost"
config.cache_store = MyOwnStore.new("parameter")
```

Para facilitar as coisas, foi incluído o comentário abaixo no arquivo *environments/production.rb*, afim de lembrá-lo desta opção.

```
# Use a different cache store in production
# config.cache_store = :mem_cache_store
```

Railties

Rake Tasks

Ruby 1.9

Detalhes

O principal foco das alterações do Rails foi o Ruby 1.9, mesmo os menores detalhes foram analisados para deixar o Rails o mais compatível possível com a nova versão do Ruby. Detalhes como alterar de **File.exists?** para **File.exist?** não foram deixados de fora.

Também, no Ruby 1.9, o módulo **Base64** (base64.rb) foi removido, por isto todas as referencias a ele foram substituídas por **ActiveSupport::Base64**.

Novos métodos para a classe DateTime

Outra alteração interessante para a nova versão. Para manter a compatibilidade (duck-typing) com a classe **Time**, três métodos novos foram adicionados à classe **DateTime**. Os métodos são **#utc**, **#utc?** e **#utc_offset**. Vamos ver um exemplo de uso de cada um:

```
>> date = DateTime.civil(2005, 2, 21, 10, 11, 12, Rational(-6, 24))
#=> Mon, 21 Feb 2005 10:11:12 -0600

>> date.utc
#=> Mon, 21 Feb 2005 16:11:12 +0000

>> DateTime.civil(2005, 2, 21, 10, 11, 12, Rational(-6, 24)).utc?
#=> false

>> DateTime.civil(2005, 2, 21, 10, 11, 12, 0).utc?
#=> true
```

```
>> DateTime.civil(2005, 2, 21, 10, 11, 12, Rational(-6, 24)).utc_offset
#=> -21600
```

Prototype e script.aculo.us

Atualizações

O Rails passa a usar a partir de agora a versão 1.6.0.1 do Prototype. Isto serve como um preparatório para a versão 1.8.1 do script.aculo.us.

Debug

Ruby-debug nativo

Foi habilitada novamente a opção de usar o **ruby-debug** nos testes do Rails.

Se antes você desejasse usá-lo nos testes seria necessário incluir um **require 'ruby-debug'** na classe e logo em seguida usar o método **debugger** no local desejado. Agora só se preocupe com o método **debugger**, o resto é nativo, desde que você já tenha o gem instalado.

Bugs e Correções

Adicionar colunas no PostgreSQL

Havia um bug ao se usar o banco de dados **PostgreSQL**. O bug ocorria quando se criava uma migration para adicionar uma coluna em uma tabela já existente, veja um exemplo:

Arquivo: *db/migrate/002_add_cost.rb*

```
class AddCost < ActiveRecord::Migration
  def self.up
    add_column :items, :cost, :decimal, :precision => 6,
   :scale => 2
  end

  def self.down
    remove_column :items, :cost
  end
end
```

Note que estou criando uma coluna com **:precision => 6** e **:scale => 2**. Agora é hora de rodar o **rake db:migrate** e vamos ver como ficou nossa tabela no banco:

Column	Type	Modifiers
id	integer	not null
descr	character varying(255)	

price	numeric(5,2)	
cost	numeric	

Veja a coluna "cost" que acabamos de criar. Ela é um **numeric** comum, mas deveria ser uma coluna como a "price", logo acima dela, mais precisamente um **numeric(6,2)**. Nesta versão este erro não existe mais, a coluna será criada da forma correta neste banco de dados.

Informações Adicionais

Protegendo-se de Cross Site Scripting

No Rails 2.0 o arquivo application.rb ficou desta maneira:

```
class ApplicationController < ActionController::Base
  helper :all

  protect_from_forgery
end
```

Note a chamada para o método **protect_from_forgery**.

Já ouviu falar de Cross Site Scripting? Este é o nome de uma falha de segurança encontrada facilmente em grande parte dos websites e aplicações web que permite à pessoas maldosas (aqui estou me referindo à adolescentes sem nada para fazer e sem vida social) alterarem o conteúdo de páginas web, incluírem conteúdo hostil, executarem ataques de phishing, obterem o controle do navegador através de códigos JavaScript e na maioria dos casos forçarem o usuário a executar algum comando que eles desejem. Este último tipo de ataque se chama crosssite request forgeries.

O Cross Site Request Forgeries é um tipo de ataque que consiste em obrigar usuários legítimos a executarem uma série de comandos sem nem mesmo saberem disto. E agora com o aumento do uso de Ajax, a coisa tem ficado ainda pior.

Na verdade, este método serve para nos assegurar de que todos os formulários que sua aplicação está recebendo estão vindo dela mesma, e não de um link perdido de algum outro site. Ele consegue isto incluindo um token baseado na sessão em todos os formulários e requisições Ajax geradas pelo Rails, e depois verifica a autenticidade deste token no controller.

Lembre-se que requisições via GET não são protegidas. Mas isto não será um problema se somente à usarmos para nos trazer dados, e nunca para alterar ou gravar algo em nosso banco de dados.

Se quiser aprender mais sobre CSRF(Cross-Site Request Forgery) use os endereços abaixo:

- http://www.nomedojogo.com/2008/01/14/como-um-garoto-chamado-samy-pode-derrubar-seu-site/isc.sans.org/diary.html?storyid=1750

- http://www.nomedojogo.com/2008/01/14/como-um-garoto-chamado-samy-pode-derrubar-seu-site/isc.sans.org/diary.html?storyid=1750

Mas lembre-se que isto não é uma solução definitiva para nosso problema, ou como costumamos dizer, não é uma bala de prata.