



Ruby on Rails do Básico ao Avançado

Aula 19

Cache, Formatos e Internacionalização

Essa é a nossa décima nona aula teórica, avançando um pouco no Rails. Esperamos que você continue a se divertir com o curso!

II. Rails

H. Linha de Chegada

ii. Indo Além

Dois assuntos interessantes relacionados à implementação de aplicações mais sofisticadas em Rails são o uso de *cache* e a possibilidade de servir outros formatos em uma aplicação—como PDF, por exemplo—a partir de uma mesma ação em que outros formatos estão sendo retornados. Nesta aula, veremos um pouco de como esses dois assuntos podem ser implementados em uma aplicação.

A implementação de *caches* em Rails é razoavelmente simples em termos de uso geral, mas ainda está evoluindo. Nesta aula, trataremos as opções existentes para isso na versão atual em Rails, lembrando somente que essa é uma área em que a pesquisa antes da implementação é interessante para verificar se algum *plugin* ou novidade no desenvolvimento do Rails não oferece alguma solução mais atualizada.

Como qualquer otimização em uma aplicação Rails, o uso de *cache* é algo que deve ser implementado no ciclo final de desenvolvimento já que abrange uma classe de alterações potencialmente complicadas na aplicação em termos de novos testes e práticas. Nesse sentido, há um pequeno paradoxo: *caches* são fáceis de serem usados no que tange aos métodos a serem chamados, mas conseguir o balanço certo pode ser um pouco complicado.

Para testarmos a parte de *cache* do Rails, vamos precisar de pequenas alterações na aplicação e na configuração da mesma. Inicialmente, o modo de *cache* só é habilitado no ambiente de produção e para estudá-lo precisamos fazer com o mesmo funcione também no modo de desenvolvimento. Para que isso aconteça, precisamos modificar o arquivo *development.rb* que está no diretório *config/environment*:

```
# Settings specified here will take precedence over those in
config/environment.rb

# In the development environment your application's code is reloaded on
# every request. This slows down response time but is perfect for
development

# since you don't have to restart the webserver when you make code changes.

config.cache_classes = false

# Log error messages when you accidentally call methods on nil.

config.whiny_nils = true

# Show full error reports and disable caching

config.action_controller.consider_all_requests_local = true

config.action_view.debug_rjs = true

config.action_controller.perform_caching = true

# Don't care if the mailer can't send

config.action_mailer.raise_delivery_errors = false
```

Depois da modificação acima, você precisa reiniciar o servidor.

Essa opção habilita o *cache* para o modo de desenvolvimento e permite o mesmo uso que aconteceria em produção. Obviamente, essa não deve ser a prática comum de uma aplicação. Novamente, esse tipo de comportamento só deve ser implementado como uma fase de otimização.

Agora podemos modificar uma *view* qualquer para começar o nosso teste de *caches*. Digamos que uma de nossas páginas faz operações complexas mas que não mudam com tanta frequência ao longo de um determinado período. Uma página assim seria uma candidata ideal para uso de *cache*.

Como não temos nada similar aqui, vamos simular usando a página de listagem de clientes. O uso do *cache* seria feito da seguinte forma:

```
<h1>Listing customers</h1>

<% cache do %>
  <table>
    <tr>
      <th>Name</th>
      <th>Description</th>
      <th>Phone</th>
      <th>Rating</th>
    </tr>

    <% for customer in @customers %>
      <tr>
        <td><%=h customer.name %></td>
        <td><%=h customer.description %></td>
        <td><%=h customer.phone %></td>
        <td><%=h customer.rating %></td>
        <td><%= link_to 'Show', customer %></td>
        <td><%= link_to 'Edit', edit_customer_path(customer) %></td>
        <td><%= link_to 'Destroy', customer, :confirm => 'Are you sure?',
:method => :delete %></td>
      </tr>
    <% end %>
  </table>
<% end %>

<br />

<%= link_to 'New customer', new_customer_path %>
```

Se observamos o *log* de desenvolvimento, veremos a seguinte situação:

```
Processing CustomersController#index (for 127.0.0.1 at 2008-07-19 17:59:51)
[GET]

  Session ID: BAh7CToMY3NyZl9pZCilODg2ZzJA0ZThmNTA5MTI2YzNmNjgwNmE3MGQ0NjEy
MGY6CXVzZXJpBjoOcmV0dXJuX3RvIg8vY3VzdG9tZXJzIgpmbGFzaElDOidB

Y3Rpb25Db250cm9sbGVyOjpbGFzaDo6Rmxhc2hIYXNoewAGOgpAdXNlZHSA--
5f8107939e6d7d0f8b2005845242b1c709f8a61f

Parameters: {"action"=>"index", "controller"=>"customers"}

Customer Load (0.001790)  SELECT * FROM "customers"
```

```

Rendering template within layouts/application
Rendering customers/index

Cached fragment hit: views/localhost:3000/customers (0.00003)

Cached fragment miss: views/localhost:3000/customers (0.00003)

Completed in 0.03136 (31 reqs/sec) | Rendering: 0.01586 (50%) | DB: 0.00179
(5%) | 200 OK [http://localhost/customers]

```

Isso indica que o fragmento da página em questão está sendo lido do *cache*.

Se adicionarmos agora um novo registro em nossa listagem de cliente, veremos que a listagem obviamente não será atualizada porque ainda está sendo lida do *cache*. No Rails, algo que entra no *cache* fica lá até que seja explicitamente expirado. Para resolver essa situação, precisamos de algumas mudanças em nosso *controller*:

```

class CustomersController < ApplicationController
  # GET /customers
  # GET /customers.xml
  def index
    @customers = Customer.find(:all)

    respond_to do |format|
      format.html # index.html.erb
      format.xml { render :xml => @customers }
    end
  end

  # GET /customers/1
  # GET /customers/1.xml
  def show
    @customer = Customer.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml { render :xml => @customer }
    end
  end

  # GET /customers/new
  # GET /customers/new.xml

```

```

def new
  @customer = Customer.new

  respond_to do |format|
    format.html # new.html.erb
    format.xml { render :xml => @customer }
  end
end

# GET /customers/1/edit
def edit
  @customer = Customer.find(params[:id])
end

# POST /customers
# POST /customers.xml
def create
  @customer = Customer.new(params[:customer])

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

# PUT /customers/1
# PUT /customers/1.xml
def update
  @customer = Customer.find(params[:id])

  respond_to do |format|
    if @customer.update_attributes(params[:customer])
      expire_fragment :action => "index"
      flash[:notice] = 'Customer was successfully updated.'
      format.html { redirect_to(@customer) }
      format.xml { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml { render :xml => @customer.errors, :status =>
:unprocessable_entity }
    end
  end
end

```

```

    end
  end

  # DELETE /customers/1
  # DELETE /customers/1.xml
  def destroy
    @customer = Customer.find(params[:id])
    @customer.destroy
    expire_fragment :action => "index"

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

```

Essas chamadas forçam a renovação do *cache* após mudanças efetuadas no banco de dados.

O controle de *cache* no Rails pode ser bastante preciso em relação aos fragmentos que se deseja salvar. Embora a situação descreva um caso em que uma listagem completa entra no *cache* é possível especificar várias partes de uma página e controlá-las de maneira específica. É possível também especificar diferentes formas de *cache* para diferentes usuários. Isso é feito através do próprio método *cache* que recebe parâmetros da mesma forma que uma ação.

Se tivéssemos, por exemplo, a situação em que usuários diferentes vêem dados diferentes nessa página, poderíamos usar algo assim:

```

<h1>Listing customers</h1>

<% cache(:action => "index", :id => session[:user]) do %>
  <table>
    <tr>
      <th>Name</th>
      <th>Description</th>
      <th>Phone</th>
      <th>Rating</th>
    </tr>

```

```

    <% for customer in @customers %>
      <tr>
        <td><%=h customer.name %></td>
        <td><%=h customer.description %></td>
        <td><%=h customer.phone %></td>
        <td><%=h customer.rating %></td>
        <td><%= link_to 'Show', customer %></td>
        <td><%= link_to 'Edit', edit_customer_path(customer) %></td>
        <td><%= link_to 'Destroy', customer, :confirm => 'Are you sure?',
:method => :delete %></td>
      </tr>
    <% end %>
  </table>
<% end %>

<br />

<%= link_to 'New customer', new_customer_path %>

```

Isso faria com que cada usuário recebesse um fragmento específico da página de acordo com o que tivesse sido gerado para ele. O código de expiração seria então modificado da mesma forma:

```

class CustomersController < ApplicationController
  # GET /customers
  # GET /customers.xml
  def index
    @customers = Customer.find(:all)

    respond_to do |format|
      format.html # index.html.erb
      format.xml { render :xml => @customers }
    end
  end

  # GET /customers/1
  # GET /customers/1.xml
  def show
    @customer = Customer.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml { render :xml => @customer }
    end
  end
end

```



```

# GET /customers/new
# GET /customers/new.xml
def new
  @customer = Customer.new

  respond_to do |format|
    format.html # new.html.erb
    format.xml { render :xml => @customer }
  end
end

# GET /customers/1/edit
def edit
  @customer = Customer.find(params[:id])
end

# POST /customers
# POST /customers.xml
def create
  @customer = Customer.new(params[:customer])

  respond_to do |format|
    if @customer.save
      expire_fragment :action => "index", :id => session[:user]
      flash[:notice] = 'Customer was successfully created.'
      format.html { redirect_to(@customer) }
      format.xml { render :xml => @customer, :status => :created,
:location => @customer }
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @customer.errors, :status =>
:unprocessable_entity }
    end
  end
end

# PUT /customers/1
# PUT /customers/1.xml
def update
  @customer = Customer.find(params[:id])

  respond_to do |format|
    if @customer.update_attributes(params[:customer])
      expire_fragment :action => "index", :id => session[:user]
      flash[:notice] = 'Customer was successfully updated.'
      format.html { redirect_to(@customer) }
      format.xml { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml { render :xml => @customer.errors, :status =>
:unprocessable_entity }
    end
  end
end

```

```

        end
      end
    end

    # DELETE /customers/1
    # DELETE /customers/1.xml
    def destroy
      @customer = Customer.find(params[:id])
      @customer.destroy
      expire_fragment :action => "index", :id => session[:user]

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

```

Note que esse tipo de uso só é válido para situações em que os dados realmente são individualizados por usuário. Se vários usuários compartilham os mesmos objetos, você poderia usar um *cache* por usuário, mas sua expiração teria que ser global, ou seja, todos os *caches* seriam expirados independente de qual usuário fez a modificação. No caso acima, seria apenas uma questão de manter a forma de expiração como ela estava anteriormente.

Existem mais dois métodos de *cache* para o Rails que fazem o *cache* de páginas completas. Esses métodos são *caches_page* e *caches_action* e funcionam como uma espécie de filtro que intercepta o processamento de uma página completa e retorna o *cache* se o mesmo existir.

A diferença entre esses dois métodos é que o primeiro, *caches_page*, nem passa pelo Rails se o *cache* for usado. Isso acontece porque, logo após o processamento da página, esse método gera uma versão final da mesma em HTML (ou qualquer outro formato usado) e como geralmente o Rails é configurado para delegar arquivos estáticos ao servidor, a requisição é servida diretamente do arquivo, sem passar pela ação.

Obviamente, esse é o tipo de *cache* mais eficiente mas também o mais estático já que nenhuma parte da página será processada pelo Rails nesse caso.

Não entraremos em detalhes desse tipo de *cache* porque seguem basicamente o mesmo padrão visto acima mudando somente a forma de aplicação.

O nosso segundo assunto para esta aula é o uso de formatos diferentes daqueles que o Rails tem a capacidade de retornar por padrão.

Não existe absolutamente nada no Rails que impeça o uso de quaisquer formatos que o desenvolver deseja retornar. Um exemplo comum é o retorno de documentos PDF que são gerados quando a ação é chamada de acordo com as necessidades do usuário. De fato, existem vários *plugins* que fazem esse tipo de trabalho, adicionando a capacidade de gerar esses formatos diferentes.

Existem duas maneiras possíveis de enviar arquivos em formatos diferentes pelo Rails, especialmente quando há a necessidade de proteger a ação que faz o envio contra acesso não autorizados.

A primeira maneira é usar os métodos *send_file* e *send_data*. Esses dois métodos retornam dados binários e só diferem no sentido em que o primeiro retorna esses dados de um arquivo existente em disco e o segundo retorna os dados diretamente da memória. Por exemplo:

```
class ReportsController < ApplicationController
  def list_immediate_action_required
    @tasks = Task.find_high(:order => "title")
  end

  def send_from_disk
    # Process request and create file in disk
    send_file @filename, :filename => "#{@article.title}.pdf",
      :type => "application/pdf", :disposition => "attachment"
  end

  def send_from_memory
    # Process request and create file in disk
  end
end
```

```

    send_data PDFGenerator.generate,
      :filename => "#{@article.title}.pdf",
      :type => "application/pdf", :disposition => "attachment"
  end
end

```

Na primeira ação acima, *@filename* contém um nome de arquivo qualquer e o Rails se encarrega de ser e enviar o arquivo sem necessidade de processamento adicional. No segundo caso, digamos que temos um objeto chamado *PDFGenerator* que possui um método chamado *generate* que retorna os dados em questão. O método *send_data* pegaria esses dados e faria um *streaming* direto para o navegador.

A segunda maneira é utilizar um *plugin* que insira um novo formato. É o caso, por exemplo, do *plugin* Prawn e o Prawn View. O Prawn View é uma melhoria do antigo RailsPDFPlugin, e atualizado para usar o engine Prawn. Esse *plugin* introduz um novo tipo de *view* cuja extensão é *.prawn* e que pode ser usada da mesma forma que os *.rjs* e *.builder* que vimos anteriormente. Esse tipo de uso é o ideal dentro do Rails porque permite uma boa flexibilidade de controle da geração e se integra automaticamente com as demais facilidades que o Rails oferece.

Para instalar, devemos colocar tanto o “prawn” quando o “prawn_view” no diretório *vendor/plugins*. Acesse <http://github.com/sandal/prawn> e http://github.com/thorny-sun/prawn_view para baixar cada um deles.

Um exemplo do uso do Prawn seria o seguinte:

```

class ReportsController < ApplicationController
  def list_immediate_action_required
    @tasks = Task.find_high(:order => "title")
  end

  def index
    @customers = Customer.find :all
    respond_to do |format|
      format.pdf
    end
  end
end

```

```
end
```

E:

```
# View: app/views/reports/index.pdf.prawn
pdf.fill_color "0000ff"
pdf.text "Clients", :at => [200,720], :size => 32

pdf.font "Times-Roman"
pdf.fill_color "000000"
@customers.each do |customer|
  pdf.text customer.name
end

pdf.text "Powered by Rails", :at => [288,50]
```

Isso torna o uso de um novo formato bem simples.

Com isso terminamos a nossa aula sobre *caches* e formatos. Como podemos perceber, o Rails continua fácil mesmo depois de todas as novas características do mesmo que estamos usando.

Internacionalização (I18n)

O Ruby on Rails 2.2 acabou de sair e com toda a certeza esta é uma das novidades que mais nos interessa.

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?

Não é possível explicar sobre I18n sem explicar o que isto significa. 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 é Internationalization, ou I18n.

Claro que esta abordagem é 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 foi integrado ao seu código fonte.

Isto não significa que o Rails sofreu grandes alterações. Na verdade quase nada foi alterado, ele continuará 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 mensagens em português, ou em outro idioma, antes teríamos de criar um monkey patch para isto. Não podemos deixar de citar como exemplo o famoso Brazilian Rails, que faz exatamente isto para traduzir as mensagens do ActiveRecord.

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

Como isto funciona?

Basicamente 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 funciona hoje, usando a localização em (inglês).

A grande novidade 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. E é claro que teremos um em português.

O alicerce desta implementação é um módulo chamado I18n que vem através de um 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 primeira coisa a fazer é criar um arquivo chamado `pt_br.yml` no diretório `config/locales`. É neste arquivo que colocaremos todas as nossas traduções e variáveis de internacionalização. Vamos iniciar com algo bem simples:

```
pt_br:

  hello: "Olá Turma"
```

Agora vamos testar o resultado disso no console do rails (`ruby script/console`):

```
>> I18n.t :hello

=> "Hello world"
```

De onde veio este "Hello world"? Veja o arquivo `en.yml` no diretório `config/locales` (estou considerando que você está trabalhando em um projeto Rails 2.2). Isto se dá porque por padrão o Rails está configurado para usar este arquivo por padrão. Para alterar para que ele use nosso arquivo, insira a seguinte linha no arquivo `config/environment.rb` do seu projeto:

```
config.i18n.default_locale = :pt_br
```

Reinicie o seu console, e vamos tentar recuperar o valor da variável `:hello` agora:

```
>> I18n.t :hello

=> "Olá Turma"
```

Agora sim, temos o valor que configuramos! Vamos aproveitar e internacionalizar uma página de nosso aplicativo. Que tal a página de login? Abra o arquivo `app/views/sessions/new.html.erb`:


```

<h1>Login</h1>

<% form_tag sessions_url, :method => :post do %>
  <%= content_tag(:p, flash[:notice], :style => "color:red") if
flash[:notice] %>
  <p>
    <b>Login</b><br />
    <%= text_field_tag :login, params[:login] %>
  </p>
  <p>
    <b>Password</b><br />
    <%= password_field_tag :password, "" %>
  </p>
  <p>
    <%= submit_tag "Enter" %>
  </p>
<% end %>

```

Vamos inserir em nosso arquivo pt_br.yml alguns textos que substituam estes trechos:

```

pt_br:
  sessions:
    title: "Acessar o site"
  form:
    login: "Usuário"
    password: "Senha"
    button: "Entrar"

```

Agora vamos preparar a nossa view para usar este arquivo:

```

<h1><%= t 'sessions.title' %></h1>
<% form_tag sessions_url, :method => :post do %>
  <%= content_tag(:p, flash[:notice], :style => "color:red") if
flash[:notice] %>
  <p>
    <b><%= t 'form.login' %></b><br />

    <%= text_field_tag :login, params[:login] %>
  </p>
  <p>
    <b><%= t 'form.password' %></b><br />

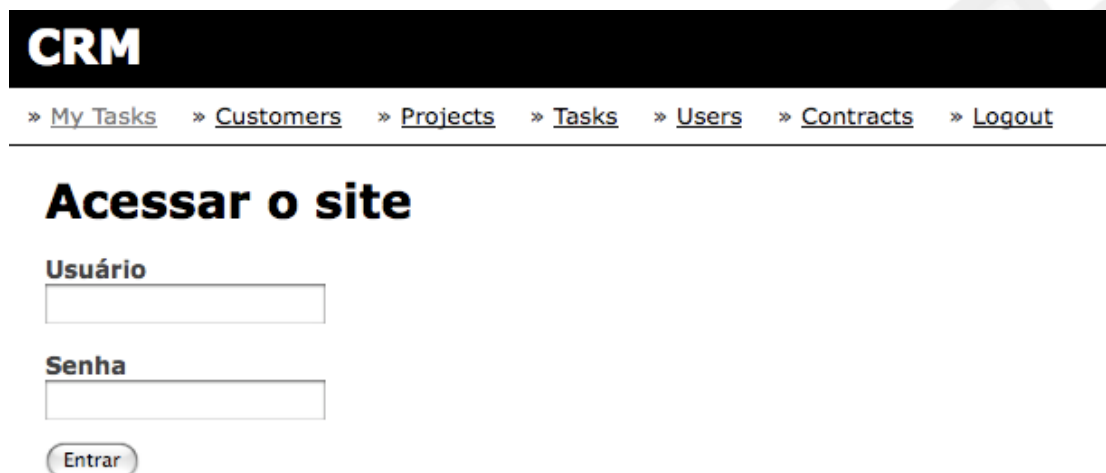
```

```

    <%= password_field_tag :password, "" %>
  </p>
  <p>
    <%= submit_tag t('form.button') %>
  </p>
<% end %>

```

Vejamos como ficou a nossa página:



CRM

» [My Tasks](#) » [Customers](#) » [Projects](#) » [Tasks](#) » [Users](#) » [Contracts](#) » [Logout](#)

Acessar o site

Usuário

Senha

Seguindo o mesmo padrão, você pode criar vários arquivos, traduzindo os campos do seu site para qualquer língua. Neste caso, seria interessante criar uma nova coluna na tabela do usuário onde você armazenará a língua escolhida de cada um. E para exibir o seu site na língua correta basta alterar o arquivo `app/controllers/application.rb` incluindo um filtro para configurar a língua escolhida. Para isto, após criar uma coluna chamada `language` na tabela `users`, insira o seguinte código no arquivo citado:

```

before_filter :set_user_language

def set_user_language
  I18n.locale = current_user.language if session[:user]
end

```

Seria prático alterar a tela de cadastro de novos usuários com uma caixa de seleção onde possa-se escolher em que língua o usuário deseja ver o site.

Além de permitir alterar “traduzir” seu site, o novo sistema de internacionalização do Rails 2.2 também permite que você possa internacionalizar todas as mensagens internas do Rails, como datas, horas, valores numéricos, monetários e até mesmo mensagens de erro do ActiveRecord.

Como exemplo, veja abaixo o arquivo `pt_br.yml` devidamente configurado para o formato usado por nós brasileiros:

```
pt_br:
  date:
    formats:
      default: "%d/%m/%Y"
      short: "%d de %B"
      long: "%d de %B de %Y"
      only_day: "%d"
    day_names: [Domingo, Segunda, Terça, Quarta, Quinta, Sexta, Sábado]
    abbr_day_names: [Dom, Seg, Ter, Qua, Qui, Sex, Sáb]
    month_names: [~, Janeiro, Fevereiro, Março, Abril, Maio, Junho, Julho,
Agosto, Setembro, Outubro, Novembro, Dezembro]
    abbr_month_names: [~, Jan, Fev, Mar, Abr, Mai, Jun, Jul, Ago, Set, Out,
Nov, Dez]
    order: [:day, :month, :year]
  time:
    formats:
      default: "%A, %d de %B de %Y, %H:%Mh"
      time: "%H:%M hs"
      short: "%d/%m, %H:%M hs"
      long: "%A, %d de %B de %Y, %H:%Mh"
      only_second: "%S"

  am: ''
  pm: ''
  datetime:
    distance_in_words:
```

```

half_a_minute: "meio minuto"
less_than_x_seconds:
  one: "menos de 1 segundo"
  other: "menos de {{count}} segundos"
x_seconds:
  one: "1 segundo"
  other: "{{count}} segundos"
less_than_x_minutes:
  one: "menos de um minuto"
  other: "menos de {{count}} minutos"
x_minutes:
  one: "1 minuto"
  other: "{{count}} minutos"
about_x_hours:
  one: "aproximadamente 1 hora"
  other: "aproximadamente {{count}} horas"
x_days:
  one: "1 dia"
  other: "{{count}} dias"
about_x_months:
  one: "aproximadamente 1 mês"
  other: "aproximadamente {{count}} meses"
x_months:
  one: "1 mês"
  other: "{{count}} meses"
about_x_years:
  one: "aproximadamente 1 ano"
  other: "aproximadamente {{count}} anos"
over_x_years:
  one: "mais de 1 ano"
  other: "mais de {{count}} anos"
number:
  format:
    precision: 3
    separator: ','
    delimiter: '.'
  currency:
    format:

    unit: 'R$'
    precision: 2
    format: "%u %n"
    separator: ','
    delimiter: '.'

```

```

percentage:
  format:
    delimiter: '.'
precision:
  format:
    delimiter: '.'
human:
  format:
    precision: 1
    delimiter: '.'
support:
  array:
    sentence_connector: "e"
    skip_last_comma: true
activerecord:
  errors:
    template:
      header:
        one: "Não foi possível gravar {{model}}: 1 erro"
        other: "Não foi possível gravar {{model}}: {{count}} erros"
      body: "Por favor, verifique os seguintes campos:"
  messages:
    inclusion: "não está incluído na lista"
    exclusion: "não está disponível"
    invalid: "não é válido"
    confirmation: "não está de acordo com a confirmação"
    accepted: "precisa de ser aceite"
    empty: "não pode estar vazio"
    blank: "não pode estar vazio"
    too_long: "é muito longo (não mais do que {{count}} caracteres)"
    too_short: "é muito curto (não menos do que {{count}} caracteres)"
    wrong_length: "não é do tamanho correto (precisa ter {{count}}
caracteres)"
    taken: "não está disponível"
    not_a_number: "não é um número"
    greater_than: "precisa ser maior do que {{count}}"
    greater_than_or_equal_to: "precisa ser maior ou igual a {{count}}"

    equal_to: "precisa ser igual a {{count}}"
    less_than: "precisa ser menor do que {{count}}"
    less_than_or_equal_to: "precisa ser menor ou igual a {{count}}"
    odd: "precisa ser ímpar"
    even: "precisa ser par"

```

Como exercício, tente internacionalizar todo o seu projeto. Você pode encontrar dezenas de arquivos pré-configurados para várias línguas no seguinte endereço:

<http://github.com/svenfuchs/rails-i18n/tree/master/rails%2FIocale>

Espero que seu próximo site possa fazer muito sucesso não só no Brasil, mas também no mundo todo através do novo sistema de internacionalização (I18n) do Ruby on Rails 2.2.

