02

Traduzindo a aplicação com I18n

Traduzindo a aplicação com I18n

Bem vindo ao último capítulo deste curso. Vimos anteriormente como enviar notificações por email em uma aplicação Rails. Neste capítulo faremos a internacionalização da aplicação, permitindo que os usuários acessam a mesma tanto em português quanto em inglês.

Primeiro abrimos nosso `company_mailer.rb` e notamos que existe um comentário deixado pelo generator: "Subject can be set in your I18n file", isto é, "O assunto pode ser configurado em seu arquivo de internacionalização" e ele cita também qual a chave, qual a frase chave que será usada para traduzir o assunto do email para diversas línguas.

O arquivo citado no exemplo `config/locales/en.yml` deve ser aberto e encontramos uma versão inicial dele:

```
en:
  hello: "Hello World"
```

Através deste arquivo podemos configurar todas as chaves e valores para internacionalizar uma aplicação. Cada texto é identificado por uma chave, como neste caso "hello".

Para nosso email, usaremos a chave mencionada "en.company_mailer.new_comment.subject". Repare que os "pontos" que são usados para agrupar as chaves são na verdade estruturas dentro de nosso arquivo yml:

```
en:
  company_mailer:
    new_comment:
      subject : "New comment received"
```

Agora podemos remover o subject e o comentário que estava presente no CompanyMailer:

```
mail to: @company.email
```

Ao testarmos enviar um comentário verificamos no log que o assunto está vindo corretamente. O próximo passo é traduzir este assunto para o português, mas como? Criaremos um novo arquivo yml que representa as chaves em português. No padrão de locales, o "pt" é o que indica a língua portuguesa, mas queremos ser ainda mais específicos, português brasileiro: "pt-BR".

Portanto criamos um arquivo chamado "pt-BR.yml" no mesmo diretório (config/locales), adicionando a chave para a versão em português:

```
pt-BR:
  company_mailer:
    new_comment:
      subject : "Novo comentário recebido"
```

Para testar abriremos o console do rails e alteramos o locale padrão para o pt-BR:

```
ruby console
>> I18n.locale = :'pt-BR'
```

Buscamos agora um job que já tem um comentário:

```
job = Job.find 1
comment = job.comments.first
```

E usamos o `CompanyMailer` para enviar o email:

```
CompanyMailer.new_comment(job, comment).deliver
```

Veja na saída que o subject utilizado foi o do arquivo internacionalizado para o português: "Novo comentário recebido". Em breve veremos como permitir que o usuário final possa escolher a língua de sua aplicação, mas antes disso alteraremos nossa aplicação para internacionalizar todos textos.

No layout application, o link original para jobs premium será alterado para usar uma tradução internacionalada de uma chave, a chave será "menu.premium_jobs", e para traduzí-la faremos:

```
t("menu.premium_jobs") # devolve a tradução de acordo com a internacionalização
```

Portanto ficamos com o link:

```
<%= link_to t("menu.premium_jobs"), root_path %>
```

No nosso arquivo de mensagens de internacionalização colocamos a chave menu, premium_jobs:

```
en:
  company_mailer:
    new_comment:
      subject : "New comment received"

  menu:
    premium_jobs : "Premium Jobs"
```

Note que o método `t` é um atalho para `I18n.translate` .

Com o servidor rodando podemos atualizar a página com um refresh no navegador e verificamos que o link continua funcionando normalmente. Agora é traduzir todo o menu!

```
en:
  company_mailer:
    new_comment:
      subject : "New comment received"

  menu:
    premium_jobs : "Premium Jobs"
```

```
    all_jobs : "All Jobs"
    new_job : "New job"
    logged_as : "Logged as"
    new_company : "New company"
    login : "Login"
```

Atualizamos todos os nossos arquivos erb até que chegamos ao arquivo `_comment.html.erb`. Nesse caso temos que a mensagem envolve o nome do comentário então chamaremos o método `t` passando esse valor como parâmetro:

```
t ".said", name: comment.name
```

E em nosso arquivo de internacionalização interpolamos a string com o valor de `name`. O mesmo deve ser feito com o `timestamp`:

```
en:
  comments:
    comment:
      said: "%{name} said"
      sent: "Sent %{timestamp} ago"
```

Esse trabalho de atualizar tudo é chato e repetitivo, por isso recomendamos efetuá-lo desde o começo, da criação de sua aplicação.

Na listagem de jobs vamos adicionar a chave "jobs.index.listing" para "Listing jobs". Mas repare que o controller que acessa esse erb é o `JobsController`. O método é o `index`. Nesse caso existe um atalho do Rails com o i18n que permite escolher a chave: "nome_do_controller.acao.chave":

```
I18n.translate(".listing")
```

Como o controller se chama `jobs` e a ação `index` o resultado é a busca pela chave `jobs.index.listing`. Isso facilita a organização de nossas chaves.

Precisamos agora permitir que o usuário escolha o idioma dele (além de determinarmos o idioma principal). No arquivo `application.rb` encontramos o `config.i18n.default_locale` que configuraremos para ser `pt-BR`:

```
config.i18n.default_locale = :'pt-BR'
```

Reiniciamos o servidor e notamos que toda nossa aplicação está agora com algumas tags em inglês e outras tags com valores inválidos. O que aconteceu? O i18n é um pouco mais esperto do que vimos até agora. Caso ele não encontre a chave na língua desejada (português), ele cria um span onde o atributo title indica que faltou uma tradução. Mas como invocamos alguns helpers, como o pluralize, ele sanitiza e plurazila a tag span, causando uma confusão visual.

Adicionamos então todas as chaves no arquivo em português. E atualizamos a página: quase tudo está traduzido mas ainda temos um problema na seção de comentários, o `time_ago_in_words`. Essa é uma tradução interna do Rails e já existe uma maneira de traduzí-la uma vez que isso já foi feita pela comunidade.

No arquivo en.yml existe um link para a gem que nos ajudará neste passo: copie a URL e veja os locales que existem. No gemfile vamos adicionar:

```
gem 'rails-18n'
```

E executamos o `bundle install` . Antes de reinicializar o servidor vamos no `application.rb` e configuramos quais línguas estamos interessados:

```
config.i18n.avaliable_locales = [:en, :'pt-BR']
```

Atualizamos o sistema e agora as características internas de mensagens do Rails estão internacionalizadas. Mas as tags dos formulários ainda não estão!

Poderíamos abrir cada formulário e adicionar como segundo argumento para os métodos `label` , o valor da internacionalização que desejamos: `t("chave")` . Mas isso seria trabalhoso demais. Como cada formulário está ligado a um modelo, o Rails já facilita e utiliza chaves padrão:

```
pt-BR:
  activerecord:
    models:
      comment: "Comentário"
```

E devemos adicionar também os atributos:

```
pt-BR:
  activerecord:
    models:
      comment: "Comentário"
    attributes:
      comment:
        name: "Nome"
        body: "Mensagem"
```

Atualizando a página e entrando no formulário de comentário vemos que está tudo ok, exceto o botão de submit do formulário. Para isso vamos trocar a tradução do helper submit:

```
pt-BR:
  helpers:
    submit:
      comment:
        create: "Enviar"
```

Mas para o helper usar esse valor default, devemos remover o valor que especificamos no formulário _form.html.erb de comentário:

```
f.submit class: 'btn btn-primary'
```

Não devemos esquecer de adicionar essas traduções no arquivo em inglês. Agora chegamos ao momento de adicionar a opção para nosso usuário final: desejo usar outra língua. No nosso layout `application.html.erb` , logo após a definição do Job Board:

```
<ul class="nav">
  <%= link_to "English", locale: "en" %>
  <%= link_to "Português", locale: "pt-br" %>
</ul>
```

Repare que esses links simplesmente adicionam um parâmetro como `locale=en` na URI da requisição. Vamos então parsear esse parâmetro em um filtro. Abrindo nosso `ApplicationController` adicionamos um `before_filter :set_locale` .

Adicionamos então um método privado:

```
I18n.locale = params[:locale] if params[:locale].present?
```

Como o filtro está no `ApplicationController` ele sempre será executado. Podemos testar nosso link e a língua é alterada.

Mas note que se clicarmos em outro link, o sistema "esquece" qual era a língua. Não colocamos nem na sessão nem em um cookie qual a língua deste usuário. Vamos alterar então o método set_locale para trabalhar com cookies. Primeiro lemos o parâmetro anterior ou o cookie, caso ele exista:

```
locale = params[:locale] || cookies[:locale]
if locale.present?
  I18n.locale = locale
  cookies[:locale] = { value: locale, expires: 30.days.from_now}
end
```

Podemos navegar pela aplicação que agora sim o cookie garante que o idioma fica marcado. Para ter certeza podemos fechar e abrir o navegador. A língua continua a mesma.

Por fim, vamos traduzir nosso email. Para fazer isso de maneira simples, criamos uma nova view chamada `new_comment.pt-BR.text.erb` e traduzimos neste arquivo o conteúdo do email original em inglês. Jamais esqueça de colocar o locale no nome do arquivo: é ele que diz ao Rails que o arquivo deve ser usado quando o envio é feito em português.