02

Adicionando comentários aos jobs através de associações entre modelos

Associações

Um aspecto importante de qualquer aplicação são associações. No Youtube, um vídeo possui diversos comentários que, por sua vez, pertencem a um usuário. Em uma rede social, um usuário pode publicar as suas fotos ao mesmo tempo em que aparece em fotos de amigos, além de enviar mensagens, participar em grupos e outros.

Neste capítulo nós vamos iniciar o suporte a comentários na nossa aplicação, possibilitando que candidatos interajam e comentem em vagas de empregos.

Gerando o modelo

Para adicionarmos comentários à nossa aplicação, nós precisamos gerar um novo modelo comentário. Este modelo deve mapear para uma tabela do banco de dados chamada *comments* que possuirá uma coluna *name* de tipo *string* para armazenar o nome de quem adicionou o comentário, e uma coluna *body* de tipo *text* para armazenar a mensagem em si.

Além do mais, nós sabemos que cada comentário se associa a um *job*. De forma geral, nós podemos falar que um *job* possui diversos comentários (em inglês dizemos que *a job* has many *comments*) e que um comentário pertence a um *job* (em inglês dizemos que *a comment* belongs to *a job*). Para modelar tal associação no banco de dados, nós também adicionaremos uma coluna *job_id* de tipo *integer* (ou seja, de tipo inteiro) à tabela. Essa coluna armazenará o id do *job* com o qual o comentário se relaciona.

Nós já usamos os geradores de código do Rails para gerar *migrations*, *controllers* e o *scaffold*. O *scaffold*, como vimos, gera código para todos as camadas do *MVC*, criando *models*, *controllers* e *views*.

Agora nós vamos usar o gerador do Rails para gerar apenas o modelo comentário. Nós adicionaremos o *controller* e as *views* quando precisarmos posteriormente. Para isso, entre no diretório da sua aplicação e digite:

```
$ rails generate model comment name:string body:text job_id:integer
```

O comando acima vai gerar um modelo em *app/models/comment.rb* com os campos descritos acima. Abra o arquivo gerado, verifique o seu conteúdo e também adicione validações de presença para os campos gerados:

```
class Comment < ActiveRecord::Base
  attr_accessible :body, :job_id, :name
  validates_presence_of :body, :job_id, :name
end
```

O gerador também criou uma migração para atualizarmos o banco de dados, então execute:

```
$ rake db:migrate
```

Agora, podemos começar um novo *console* do Rails e verificar que podemos criar comentários:

```
$ rails console
```

Primeiro, vamos acessar um *job*:

```
>> job = Job.first
=> #<Job ...>
```

Nota: se você não tiver nenhum *job* no seu banco de dados, você pode criar um novo pela linha de comando ou pela interface web.

Agora vamos criar o nosso primeiro comentário:

```
>> Comment.create(name: "Fulano", body: "Primeiro comentário", job_id: job.id)
=> #<Comment id: 1, name: "Fulano", body: "Primeiro comentário", ...>
```

E com isso adicionamos o nosso primeiro comentário! Observe que passamos manualmente o *id* do *job* ao qual o comentário se relaciona. Como associações e essa troca de informação é bastante comum, o Rails possui algumas conveniências para trabalhar com associações. É isso que vamos ver em seguida.

Associações

Na seção anterior dizemos que um *job* possui diversos comentários (ou em inglês, *a job* **has many** *comments*) e que um comentário pertence a um *job* (ou em inglês, *a comment* **belongs to** *a job*). É exatamente com os métodos `has_many` e `belong_to` que modelamos associações no Rails.

Abra o arquivo *app/models/job.rb* e adicione:

```
class Job < ActiveRecord::Base
  has_many :comments
  attr_accessible :description, :title, :premium
  validates_presence_of :description, :title
end
```

Em seguida, abra o arquivo *app/models/comment.rb* e adicione:

```
class Comment < ActiveRecord::Base
  belongs_to :job
  attr_accessible :body, :job_id, :name
  validates_presence_of :body, :job_id, :name
end
```

Agora, de volta ao *console* do Rails, digite `reload!` para forçar o Rails recarregar os nossos modelos agora com as associações:

```
>> reload!
Reloading...
=> true
```

Agora, vamos pegar novamente o primeiro *job* disponível e em seguida todos os comentários associados ao *job*:

```
>> job = Job.first
=> #<Job ...>
>> job.comments
=> [#<Comment id: 1, name: "Fulano", body: "Primeiro comentário", ...>]
```

Observe agora que simplesmente digitamos `job.comments` e o Rails retornou o comentário que criamos associado ao *job* a poucos minutos atrás. O método `comments` é automaticamente adicionado ao nosso modelo no momento que escrevemos `has_many :comments` no modelo `Job`. Note também que o Rails retornou a informação entre colchetes `[` e `]`. Isso ocorre porque `jobs.comments` retorna um array, ou seja, uma lista com todos os comentários associados. Vamos adicionar um novo comentário:

```
>> comment = job.comments.create(name: "Fulano", body: "Outro comentário")
=> #<Comment id: 2, name: "Fulano", body: "Outro comentário", ...>
>> comment.job_id == job.id
true
```

Agora usamos `job.comments.create` invés de `Comment.create` para criar o comentário. O comando `job.comments` funciona como um escopo e permite que a gente crie comentários que estão relacionados ao *job*. O Rails também setou automaticamente a coluna `job_id` para o valor de `job.id`, algo que haviamos feito manualmente anteriormente.

Note também que ao adicionar `belongs_to :job` ao nosso modelo `Comment`, em nenhum momento precisamos dizer ao Rails para usar a coluna `job_id` para armazenar a informação do *job* associado, graças ao *Convention over Configuration*.

Nós também podemos verificar que `job.comments` agora vai retornar um *array* com dois elementos:

```
>> job.comments
=> [#<Comment id: 1, name: "Fulano", body: "Primeiro comentário", ...>,
#<Comment id: 2, name: "Fulano", body: "Outro comentário", ...>]
```

E podemos checar qual o tamanho do array usando o método `size`:

```
>> job.comments.size
2
```

É possível também iterar cada elemento do array através do método `each`:

```
>> job.comments.each do |comment|
?>  puts comment.body
?> end
Primeiro comentário
Outro comentário
```

No exemplo acima, nós executamos o método `each` passando um bloco de código que vai imprimir o campo *body* para cada comentário.

E da mesma forma que podemos acessar os comentários de um *job*, também podemos verificar a qual *job* um comentário pertence:

```
>> comment.job
=> #<Job ...>
```

Neste caso, como um comentário pertence apenas a **um único** *job*, o Rails não retorna um array, mas o *job* diretamente.

Contador de comentários

Agora que sabemos como manipular os comentários associados a um *job*, vamos modificar a *partial* que criamos no capítulo anterior para mostrar um contador com quantos comentários existem em cada *job*. Para isso, vamos abrir o arquivo *app/views/jobs/_job.html.erb* para adicionar a seguinte linha:

```
<%= pluralize(job.comments.size, "comment") %> |
```

O resultado final deve ser:

```
<h3>
  <%= job.title %>
  <% if job.premium %>
   <strong>(premium)</strong>
  <% end %>
</h3>

<%= simple_format job.description %>

<p>
  <%= pluralize(job.comments.size, "comment") %> |
  <%= link_to 'Show', job %> |
  <%= link_to 'Edit', edit_job_path(job) %> |
  <%= link_to 'Destroy', job, method: :delete, data: { confirm: 'Are you sure?' } %>
</p>
```

Ao recarregar a página inicial da nossa aplicação, podemos ver que um contador aparece junto de cada *job* com o número de comentários! Tudo isso aconteceu com a ajuda do método `pluralize` disponibilizado pelo Rails. Caso o número de comentários seja 1, o Rails irá mostrar `1 comment`, caso contrário, o Rails mostrará `2 comments`, `3 comments`, etc.

Agora, podemos partir para o último passo deste capítulo, que é listar todos os comentários associados a um *job* na página de visualização do *job*.

Exibindo comentários

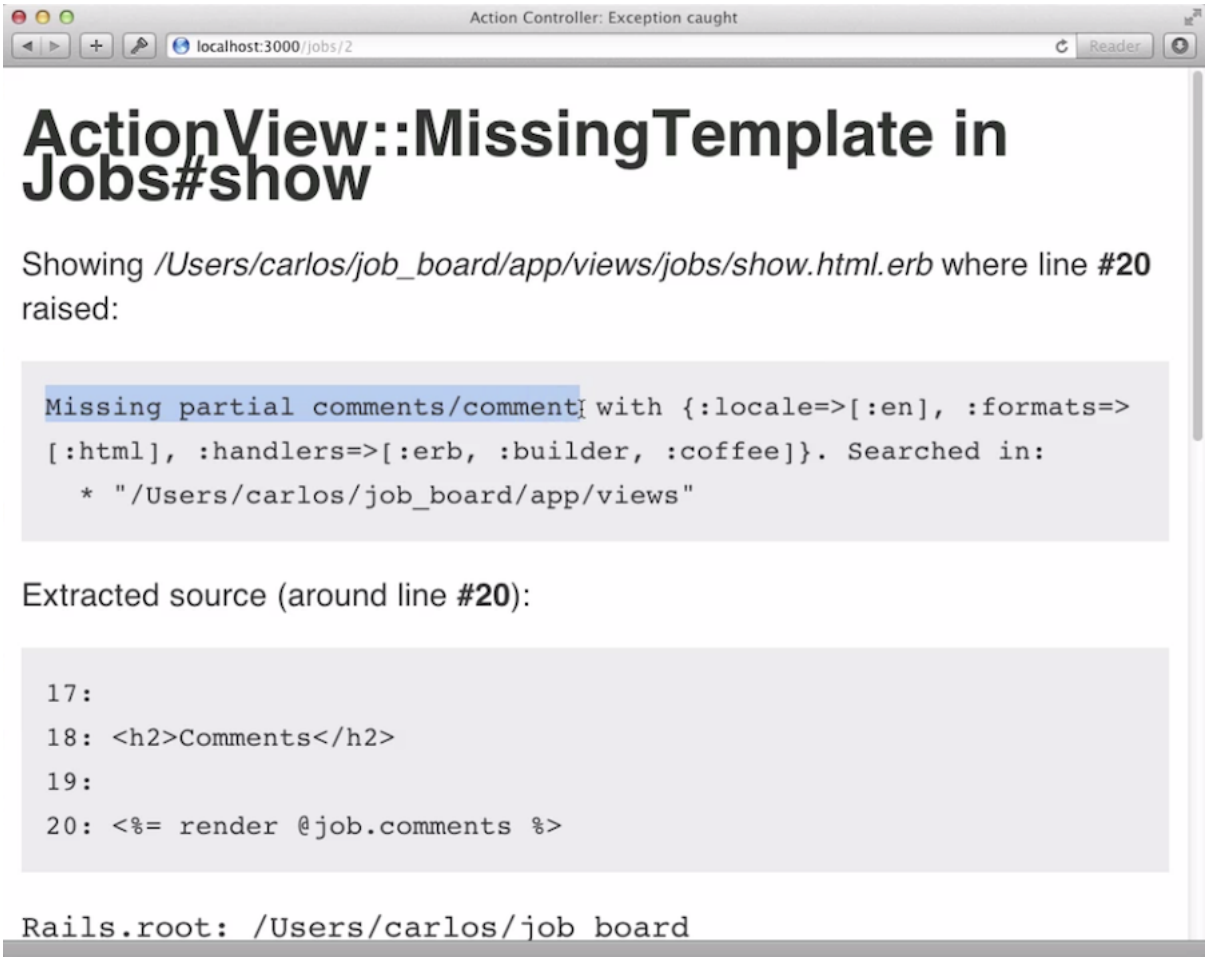
Para exibir os comentários, vamos modificar a *view* que exibe um *job*. Logo, abra o arquivo *app/views/jobs/show.html.erb* e modifique-o para o seguinte:

```
<p id="notice"><%= notice %></p>
```

```
<h1>
  <%= @job.title %>
  <% if @job.premium %>
   <strong>(premium)</strong>
  <% end %>
</h1>

<%= simple_format @job.description %>

<%= pluralize(@job.comments.size, "comment") %> |
<%= link_to 'Edit', edit_job_path(@job) %> |
<%= link_to 'Back', jobs_path %>

<h2>Comments</h2>

<%= render @job.comments %>
```

A nova *view* é parecida com a partial em `jobs/_job.html.erb` que criamos no capítulo anterior, exceto que utilizamos `h1` invés de `h3` e modificamos os links de navegação. Nós também adicionamos uma seção específica para comentários, onde executamos `render @job.comments` .

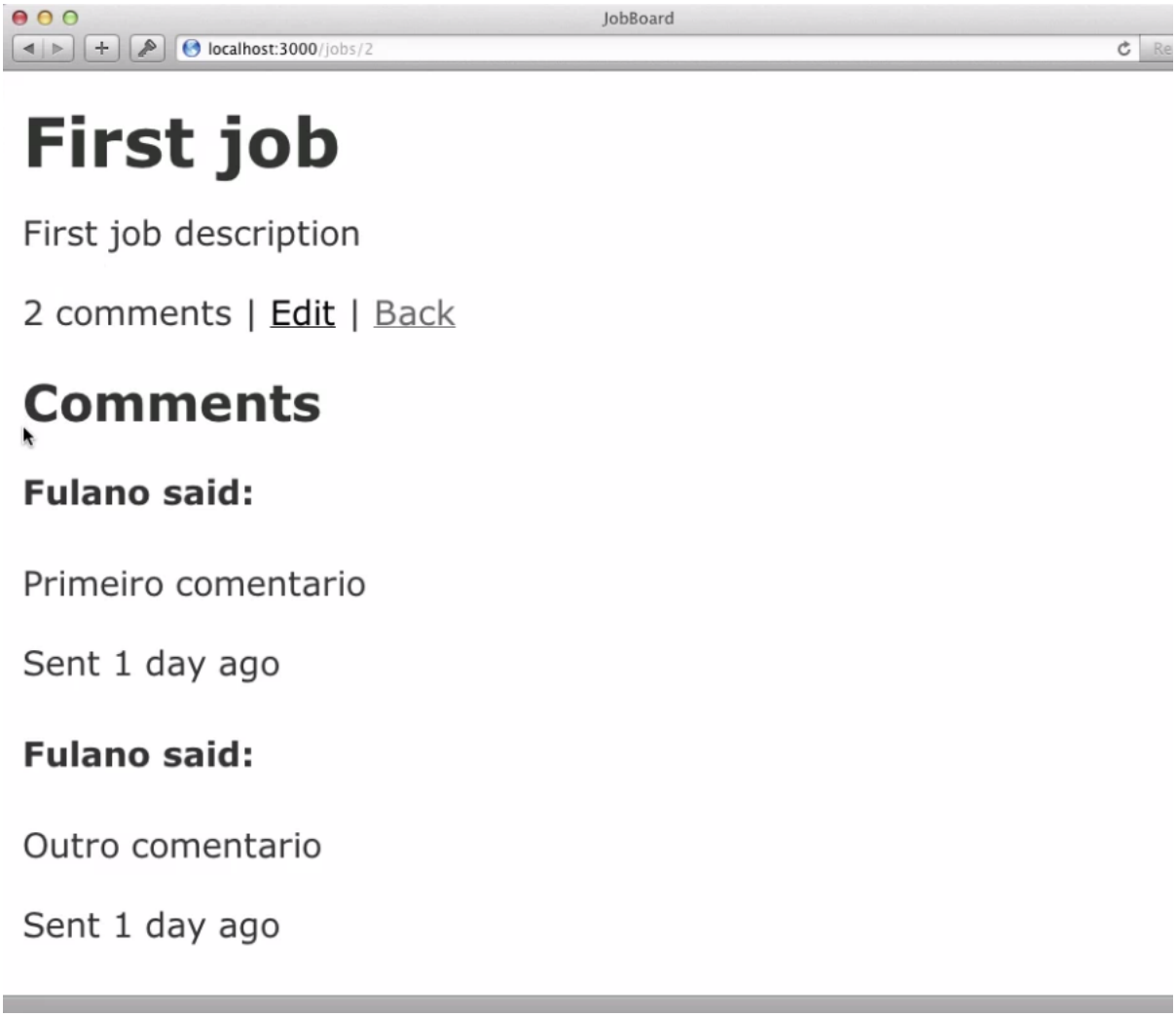
Porém, se tentarmos acessar a página de exibição de algum *job* no nosso navegador, nós vamos ver uma página de erro do Rails:



O erro ocorre porque, como vimos no capítulo anterior, ao executarmos `render @job.comments` , o Rails tentará renderizar uma *partial*. Neste caso, o Rails espera uma *partial* de comentário em *app/views/comments/_comment.html.erb* que ainda não criamos. Logo, para corrigir esse erro, crie esse arquivo com o seguinte conteúdo:

```
<h4><%= comment.name %> said:</h4>
<%= simple_format comment.body %>
Sent <%= time_ago_in_words comment.created_at %> ago
```

A nossa *partial* é bem simples, ela exibe o nome de quem criou o comentário, o seu conteúdo utilizando `simple_format` e utiliza um outro método disponibilizado pelo Rails chamado `time_ago_in_words` que mostrará de forma aproximada há quanto tempo o comentário foi criado. Agora, se abrirmos a página de exibição de um *job*, veremos que ela funcionou com sucesso:



Excelente! Utilizamos novamente as convenções do Rails para criar páginas e *partials* de modo rápido e elegante!

Helpers

Já que estamos falando em "elegância", existe uma parte da nossa aplicação que poderia ser melhorada. Observe que tanto a view em *jobs/show.html.erb* quanto a partial *jobs/_job.html.erb* possuem exatamente o mesmo código que gera o título de um *job*:

```
<%= @job.title %>
<% if @job.premium %>
  <strong>(premium)</strong>
<% end %>
```

Como vimos no capítulo anterior, desenvolvedores Rails sempre procuram evitar duplicação de código para facilitar a manutenção. No caso de views, existem duas formas para evitar tal duplicação: *partials*, que já conhecemos, e *helpers*. Neste caso, como o conteúdo que queremos evitar duplicar é pequeno, criaremos um *helper*. Abra o arquivo *app/helpers/jobs_helper.rb* e adicione um método chamado `job_title` :

```
module JobsHelper
  def job_title(job)
    title = h(job.title)

    if job.premium
      title + content_tag(:strong, " (premium)")
    else
      title
    end
  end
end
```

O método `job_title` que definimos acima é chamado de **helper** e está automaticamente disponível nas nossas *views*. Abra novamente o arquivo *app/views/jobs/show.html.erb* e altera o conteúdo de:

```
<h1>
  <%= @job.title %>
  <% if @job.premium %>
   <strong>(premium)</strong>
  <% end %>
</h1>
```

Para:

```
<h1><%= job_title(@job) %></h1>
```

Recarregue a página e verifique que tudo continua funcionando como esperado! Fica como exercício para o leitor alterar a partial em *app/views/jobs/_job.html.erb* para também utilizar o nosso novo *helper*.

De forma geral, os *helpers* são divididos em diversos arquivos de acordo com o seu contexto. No caso atual, como o nosso *helper* gera o título para *jobs*, ele foi incluído em *app/helpers/jobs_helper.rb*. Caso algum *helper* esteja relacionado ao comportamento geral da aplicação, ele poderia ser definido em *application_helper.rb*.

Com isso, encerramos o capítulo atual que contém os primeiros passos para adicionar suporte a comentários em nossa aplicação. Neste capítulo, também reforçamos o nosso conhecimento de *partials* e aprendemos o conceito de *helpers*. As duas técnicas são bastante importantes para evitarmos duplicar código nas nossas *views*. No próximo capítulo, possibilitaremos que usuários enviem comentários através da interface web, não perca!

Para saber mais

-
- Neste capítulo utilizamos o método `pluralize` , você pode aprender mais sobre ele na documentação do Rails (http://view-source:http://api.rubyonrails.org/classes/ActionView/Helpers/TextHelper.html#method-i-pluralize)