02

Acessando e estendendo jobs no banco de dados

Conversando com o banco de dados

No capítulo anterior, estudamos o código gerado para exibirmos o "Hello World", envolvendo a *rota*, o *controller*, a *view*, e a maneira como eles são interligados para que o Rails consiga reconhecer uma url e exibir a página relacionada. Agora continuaremos o desenvolvimento do nosso "Classificado de Empregos", entendendo um pouco melhor como o Rails interage com o banco de dados. Em seguida, vamos gerar instruções para o Rails atualizar a nossa tabela jobs com um coluna extra chamada **premium**, possibilitando marcar as ofertas de jobs mais interessantes como **premium**.

CRUD e o Active Record

O comando que executamos para criar o *CRUD* de *jobs* no primeiro capítulo foi este:

```
$ rails generate scaffold job title:string description:text
```

O *scaffold generator* criou uma série de arquivos, como um *controller* similar ao *HelloController* que vimos anteriormente, e também várias *views*. Além disso, especificamos também os atributos que o *job* possui: *title* e *description*, que foram utilizados para criar uma tabela no banco de dados, chamada *jobs*, onde ficam armazenados os registros de *jobs* que criamos pela aplicação.

Porém, é necessário que as informações salvas nesta tabela dentro do banco de dados possam ser extraídas para podermos exibí-las na *view* quando acessamos */jobs*. Da mesma forma, também é necessário que consigamos salvar novas informações dentro desta tabela, como um novo registro de *job*, ou alterar um *job* existente - todas as operações que já executamos pela interface da aplicação. Ou seja, precisamos de alguma maneira de mapear a tabela do banco de dados para Ruby, para que tenhamos maior controle para manipular a tabela e as informações que ela contém.

Para isto o Rails conta com um componente chamado **Active Record** que é responsável por mapear as nossas classes em Ruby para tabelas no banco de dados e o conteúdo dessas tabelas para objetos.

Nós já passamos pelo modelo `Job` no primeiro capítulo, quando adicionamos a validação de presença. Vamos dar uma olhada no código dele novamente, abra o arquivo *app/models/job.rb*:

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

Parecido com a forma em que os nossos *controllers* herdam de `ActionController::Base`, o nosso modelo `Job` **herda** comportamento de `ActiveRecord::Base`, que possui a funcionalidade básica para conversarmos diretamente com o banco de dados.

Para entendermos um pouco melhor como a classe `Job` funciona e o que recebemos do **Active Record**, vamos conhecer uma versão "especial" do *irb* que o Rails fornece.

O Rails console

O Rails possui um console próprio, que funciona como o *irb* que vimos no capítulo anterior, mas com algumas funcionalidades adicionais. Para acessar o console do Rails, digite em seu terminal (estando no diretório da aplicação):

```
$ rails console
```

Você deve ver algo como:

```
$ rails console
Loading development environment (Rails 3.2.6)
>>
```

O console do Rails funciona exatamente como o *irb*. Para comprovar isto, digite o mesmo comando que utilizamos anteriormente, que converte seu nome para maiúsculo:

```
>> "carlos".upcase
=> "CARLOS"
```

O resultado é o mesmo. O que torna o console do Rails "especial" é que, através dele, você tem acesso à algumas funcionalidades só existentes na aplicação, como por exemplo aos modelos existentes. Vamos acessar a classe `Job`, digite:

```
>> Job
=> Job(id: integer, title: string, description: text, created_at: datetime, updated_at: datetime
```

O Rails exibirá o nome da classe `Job` e todos os atributos relacionados à ela, incluindo `title` e `description` que fomos nós que definimos. Os demais atributos - `id`, `created_at` e `updated_at` - são gerados automaticamente pelo Rails: o `id` é uma chave que identifica cada *job* dessa tabela, um número sequencial; e os outros dois são campos que o Rails gerencia para nós populando com a data/hora atual, ao criar um registro (`created_at`), ou ao atualizar um registro (`updated_at`).

Note que em nenhum momento tivemos que dizer ao Rails *aonde* procurar informações sobre os *jobs*. Ele automaticamente busca a tabela de nome *jobs*, relacionando-a com a classe `Job`. Eis o que acontece: o Rails utiliza o nome do modelo `Job`, em minúsculo e no plural, para procurar uma tabela no banco de dados - *jobs* - que corresponda ao modelo. Digite no console:

```
>> Job.table_name
=> "jobs"
```

E você verá que o Rails devolve o nome da tabela *jobs*. Esse é um dos pontos mais fortes do Rails: ele não pede em nenhum momento que você como desenvolvedor *configure* qual a tabela relacionada a classe `Job`, ele possui uma **convenção** bem definida para isso - nome da classe em minúsculo e no plural. Por isso dizemos que o Rails segue uma filosofia de **Convenção sobre Configuração (Convention over Configuration)**, onde ele define vários padrões ao invés de pedir ao desenvolvedor uma série de configurações.

Agora vamos buscar o primeiro elemento do banco de dados. Se nós queremos o primeiro (first), natural que usemos o comando `Job.first` :

```
>> Job.first
  Job Load (0.3ms)  SELECT "jobs".* FROM "jobs" LIMIT 1
=> #<Job id: 2, title: "", description: "", created_at: "2012-07-12 17:19:00", updated_at: "201
```

Nota: Caso a sua aplicação não mostre nenhum job na página de listagem de jobs, crie ao menos um job antes de digitar `Job.first` no console.

Neste caso, o primeiro registro disponível na minha máquina possui o `id: 2`, e os atributos `title` e `description` em branco - o registro inválido que criamos no primeiro capítulo. Ao executar o mesmo comando na sua máquina, você poderá receber um registro diferente baseado no que você adicionou ou removeu do seu banco de dados.

Nota: perceba que o **Active Record** executa comandos SQL (exibido acima como `Job Load`) para obter as informações do banco de dados, e retorna uma **instância** da classe `Job`.

Para obter todos os *jobs* salvos no banco de dados, utilizamos o método `all`:

```
>> Job.all
  Job Load (0.3ms)  SELECT "jobs".* FROM "jobs"
=> [#<Job id: 2, title: "", description: "", created_at: "2012-07-12 17:19:00", updated_at: "20
```

E para criar um novo *job* e salvá-lo no banco de dados, podemos utilizar o método `save`:

```
>> job = Job.new
=> #<Job id: nil, title: nil, description: nil, created_at: nil, updated_at: nil>
>> job.title = "Novo job criado pelo console"
=> "Novo job criado pelo console"
>> job.description = "Novo alo console"
=> "Novo alo console"
>> job.save
   (0.1ms)  begin transaction
  SQL (0.6ms)  INSERT INTO "jobs" ("created_at", "description", "title", "updated_at") VALUES (
   (2.0ms)  commit transaction
=> true
>> job.id
=> 5
```

Ao executar o `save`, o **Active Record** vai salvar os atributos que você setou na tabela *jobs*, adicionando um novo registro, e retornando o `id` deste novo registro - neste caso, `5`. Este exemplo é bastante semelhante ao da classe `Person` que criamos no capítulo 2, pois estamos lidando diretamente com Ruby, objetos e atributos.

Agora que sabemos o `id` deste *job* em específico, podemos utilizar esta informação para procurar por ele na tabela, usando o método `find`, como abaixo:

```
>> Job.find(5)
  Job Load (4.6ms)  SELECT "jobs".* FROM "jobs" WHERE "jobs"."id" = ? LIMIT 1  [["id", 5]]
=> #<Job id: 5, title: "Novo job criado pelo console", description: "Novo alo console", created_
```

Perceba que ele retorna o mesmo *job* que acabamos de criar.

Porém, nem sempre conseguimos criar um *job* com sucesso. No primeiro capítulo, nós adicionamos algumas validações ao nosso modelo, no caso, um *job* tem que possuir os campos *title* e *description* para ser válido. O que será que acontece se não preenchermos esses campos e tentarmos salvar o nosso job de qualquer forma usando `save` ? Vamos verificar:

```
>> job = Job.new
=> #<Job id: nil, title: nil, description: nil, created_at: nil, updated_at: nil>
>> job.save
   (0.1ms)  begin transaction
   (0.1ms)  rollback transaction
=> false
```

Observe que nesse caso, nenhum `INSERT` foi feito no banco de dados, como no exemplo anterior. Note também que `job.save` retornou `false` como resultado, indicando que o objeto não foi salvo com sucesso no banco de dados, enquanto no exemplo anterior `job.save` retornou true.

É possível checar se qualquer *job* é válido ou não, sem tentar salvá-lo no banco de dados, usando o método `valid?` , que deve retornar `true` ou `false` :

```
>> job = Job.new
=> #<Job id: nil, title: nil, description: nil, created_at: nil, updated_at: nil>
>> job.title = "Novo job!"
=> "Novo job!"
>> job.valid?
=> false
>> job.description = "Descricao do novo job"
=> "Descricao do novo job"
>> job.valid?
=> true
```

Note que após setar ambos os atributos, `job.valid?` passa a retornar `true` . Neste momento, se executarmos `job.save` , o registro será salvo com sucesso na tabela *jobs*, e retornará `true` :

```
>> job.save
   (0.1ms)  begin transaction
  SQL (103.6ms)  INSERT INTO "jobs" ("created_at", "description", "title", "updated_at") VALUES
   (2.4ms)  commit transaction
=> true
```

Os métodos que vimos acima são todos implementados pelo **Active Record**, e que **herdamos** na nossa classe `Job` , permitindo a manipulação dos registros no banco de dados de forma simplificada.

Migrações

Quanto geramos o scaffold no primeiro capítulo, nós comentamos que ele gerou instruções para adicionar a tabela *jobs* ao banco de dados. Para executar tal instruções, usamos o comando `rake db:migrate` .

Essas instruções são conhecidas no Rails com o nome de **migrations** (migrações), e possibilitam que a gente **migre** o nosso banco de dados de uma versão para outra.

Para entender melhor como migrações funcionam, vamos gerar uma nova migração que vai adicionar uma coluna **premium** à tabela *jobs*. No futuro, nós marcaremos os jobs mais interessantes como **premium** e eles terão uma área dedicada apenas para eles no website.

Sem mais demoras, execute o comando abaixo dentro da diretório da sua aplicação para gerar uma nova migração:

```
$ rails generate migration add_premium_to_jobs premium:boolean
```

Observe que o comando é bastante similar ao scaffold. Nós estamos dizendo ao Rails que queremos gerar (*generate*) uma migração (*migration*) chamada *add_premium_to_jobs* (adicionar premium a jobs) com um atributo **premium** que é um *booleano* - ou seja, o atributo pode ter apenas dois valores: `true` ou `false`.

Ao executar o comando, você deve ver um output similar ao abaixo:

```
$ rails generate migration add_premium_to_jobs premium:boolean
invoke  active_record
 create    db/migrate/20120723094404_add_premium_to_jobs.rb
```

Observe que o comando criou um arquivo dentro do diretório `db/migrate`. O nome do arquivo começa com o **timestamp** em que o arquivo foi criado (ou seja, o ano, seguido do mês, dia, hora, minutos e segundos), mais o nome dado à migração.

Abra o arquivo no seu editor e você verá o seguinte conteúdo:

```
class AddPremiumToJobs < ActiveRecord::Migration
  def change
    add_column :jobs, :premium, :boolean
  end
end
```

A nossa migração é uma classe que **herda** de `ActiveRecord::Migration` e possui um método chamado `change`, que traduz para **alterar**. O método `change` descreve as instruções necessárias para atualizar o banco de dados, neste caso, devemos adicionar uma coluna (`add_column`) a tabela `:jobs` chamada `:premium` com tipo `:boolean`.

Já que criamos uma nova migração, temos que pedir ao Rails para atualizar o banco de dados com as instruções contidas nessa nova migração. Para isso, execute novamente o comando abaixo para atualizar o banco de dados:

```
$ rake db:migrate
```

Executando o comando dentro do diretório da sua aplicação deve retornar:

```
$ rake db:migrate
==  AddPremiumToJobs: migrating ===============================================
-- add_column(:jobs, :premium, :boolean)
```

```
    -> 0.0006s
==  AddPremiumToJobs: migrated (0.0007s) ======================================
```

Ou seja, as nossas novas intruções foram executadas. Para verificar que a nossa tabela *jobs* foi atualizada, vamos retornar ao console do Rails:

```
$ rails console
```

Nota: Se o console do Rails ainda estiver aberto na sua máquina, você não precisa iniciar outro. Simplesmente execute o método `reload!` dentro do console para forçar o Rails para carregar a versão mais recente do seu modelo.

```
>> Job
=> Job(id: integer, title: string, description: text, created_at: datetime, updated_at: datetime
```

Podemos ver que o nosso modelo foi atualizado e contém uma coluna chamada `premium` com tipo `boolean`. Sucesso!

Antes de finalizarmos o capítulo, vamos para o último passo: adicionar um *check box* no formulário de criação e edição de jobs para que possamos marcar um job como premium.

Formulários

Quando executamos o scaffold, o Rails gerou views tanto para a criação quanto a edição de jobs. Essas views estão localizadas em **app/views/jobs/new.html.erb** e **app/views/jobs/edit.html.erb**.

Se navegarmos a nossa aplicação, podemos notar que as views de criação e edição de jobs são bastante similares, isso acontece porque na prática elas compartilham o mesmo formulário. Este conteúdo compartilhado pode ser encontrado em **app/views/jobs/_form.html.erb**.

Abra o arquivo **app/views/jobs/_form.html.erb** no seu editor e você verá o seguinte conteúdo:

```
<%= form_for(@job) do |f| %>
  <% if @job.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@job.errors.count, "error") %> prohibited this job from being saved:</h2

      <ul>
      <% @job.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_area :description %>
```

```
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

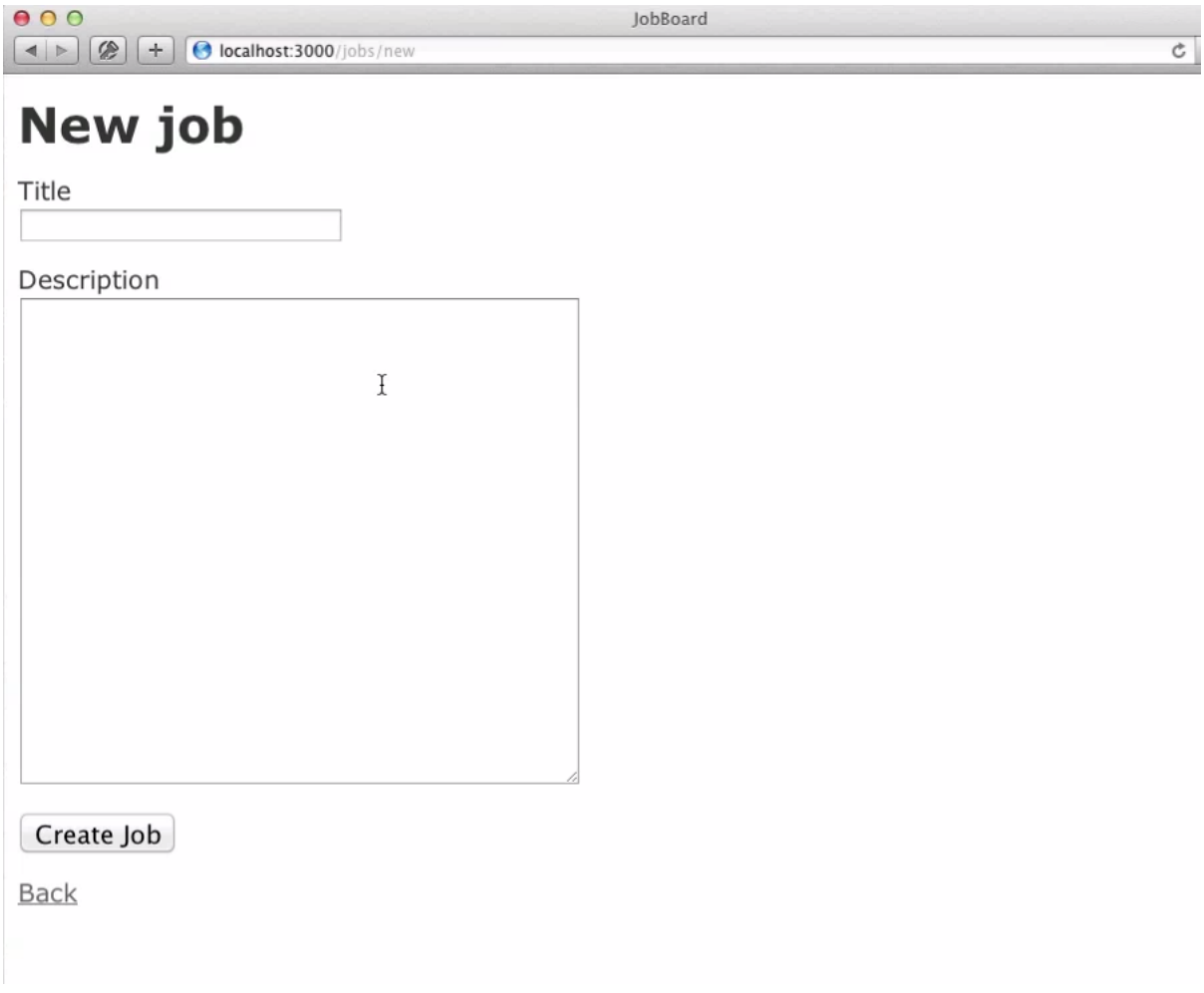
Observe que a página contém conteúdo **ERB**, da mesma forma que a **app/views/hello/world.html.erb**. Por agora essa *view* parece ser um pouco complexa, por tanto não se preocupe em entender todos os detalhes. A linha mais importante para nós é a primeira:

```
<%= form_for(@job) do |f| %>
```

A linha acima está criando um formulário para um `@job` e dando o nome `f` para esse formulário. Nós podemos ver esse formulário `f` sendo usado algumas linhas abaixo na mesma view:

```
<div class="field">
  <%= f.label :title %><br />
  <%= f.text_field :title %>
</div>
```

Aqui, nós estamos dizendo ao formulário que queremos uma etiqueta (`f.label`) para o atributo *title* do job, seguido de um campo de texto (`f.text_field`), que é exatamente o que vemos na página de criação:



Ou seja, se quisermos adicionar um campo **premium** ao nosso formulário, provavelmente precisamos usar instruções similares às instruções acima, exceto que queremos um `check_box` ao invés de um `text_field` , algo como:

```
<div class="field">
  <%= f.label :premium %><br />
  <%= f.check_box :premium %>
</div>
```

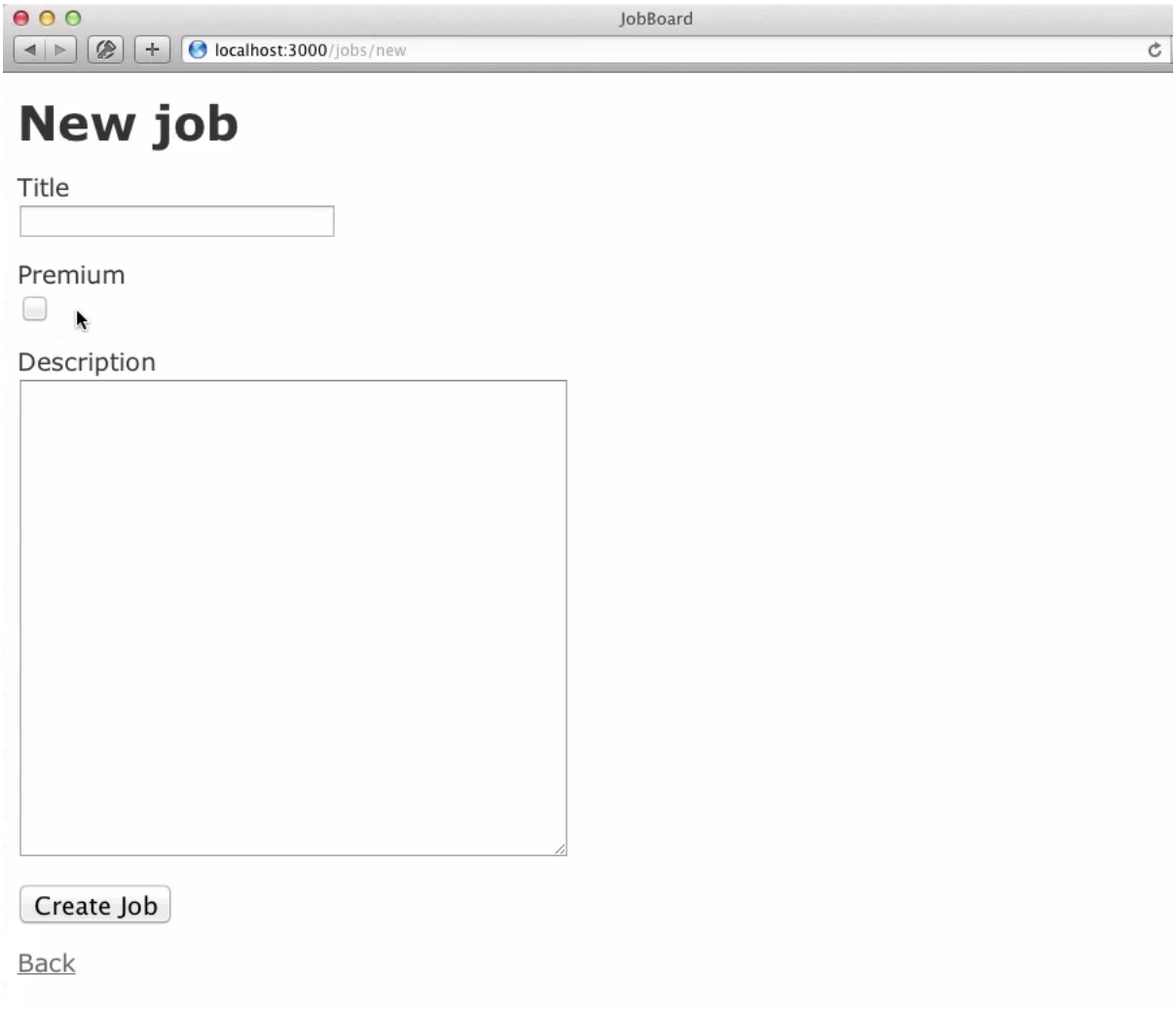
Vamos adicionar o código acima ao nosso arquivo, logo após o campo para *title*. O arquivo final deve ser igual abaixo:

```
<%= form_for(@job) do |f| %>
  <% if @job.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@job.errors.count, "error") %> prohibited this job from being saved:</h2

      <ul>
      <% @job.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :premium %><br />
    <%= f.check_box :premium %>
  </div>
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

Se acessarmos novamente a página de criação de jobs no nosso navegador, podemos ver que existe um *check box*!

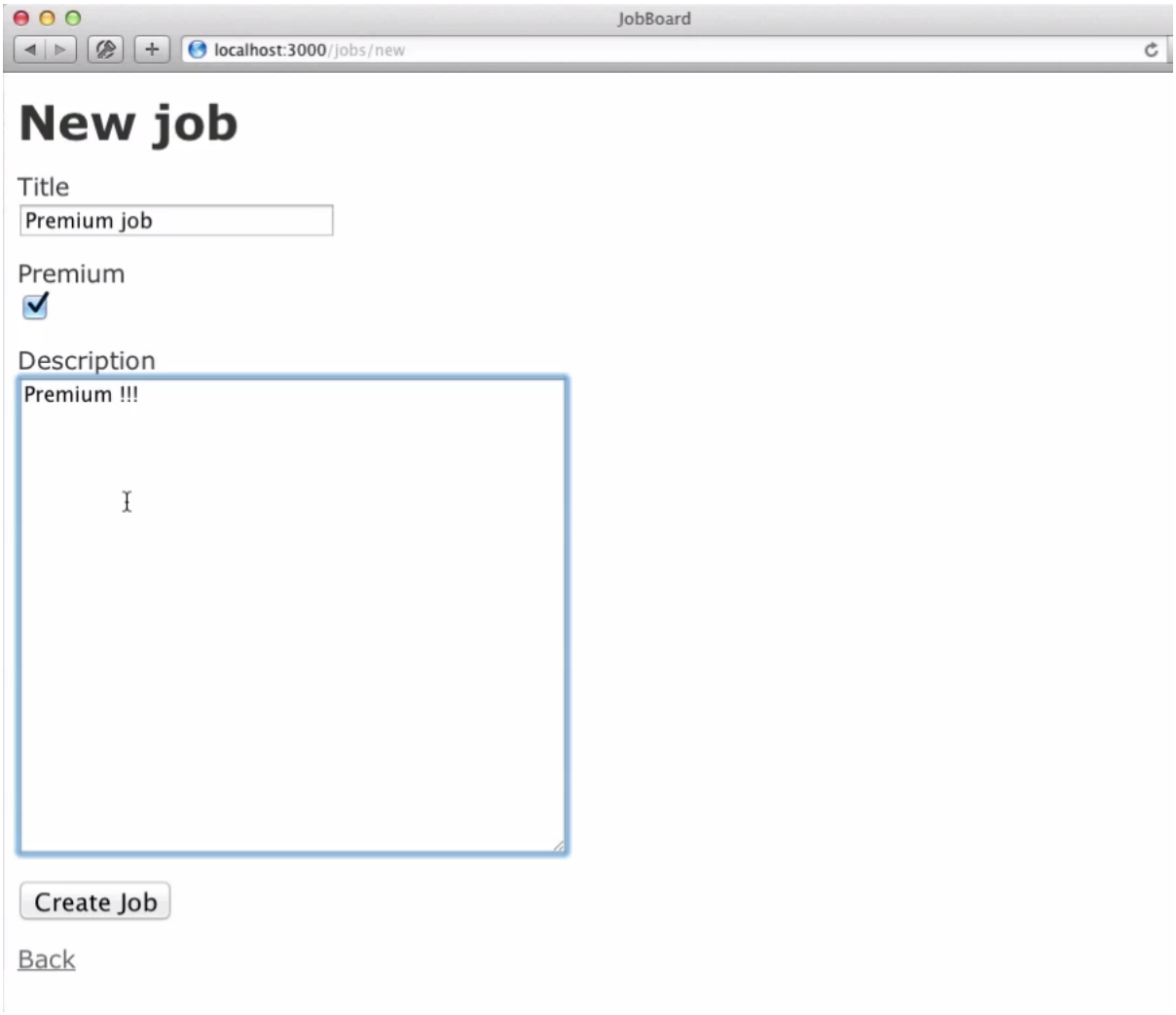


Ótimo! Existe um último passo para o nosso campo funcionar como esperado.

Nós temos que dizer ao nosso modelo `Job` que o atributo *premium* pode ser acessado através de um formulário. Para isso, abra **app/models/job.rb** e adicione `:premium` à lista de atributos em `attr_accessible` , conforme o resultado final abaixo:

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

Agora vamos criar um novo job no nosso navegador e marcá-lo como premium:



Podemos ver que o novo job foi criado como **premium** com sucesso abrindo novamente o console do Rails:

```
$ rails console
```

E dentro do console digitamos `Job.last`:

```
>> job = Job.last
  Job Load (0.3ms)  SELECT "jobs".* FROM "jobs" ORDER BY id DESC LIMIT 1
=> #<Job id: 6, title: "Premium", description: "Premium", created_at: "2012-07-23 17:19:00", up
>> job.premium
=> true
```

Note que utilizamos o comando `Job.last` invés de `Job.first`, já que queríamos o último job criado invés do primeiro. Observe também que `job.premium` retornou `true`, como esperado!

Com isso finalizamos o terceiro capítulo. Nós aprendemos como o Rails conversa com o nosso banco de dados, utilizando o Active Record, e como atualizar os nossos modelos e banco de dados. No próximo capítulo vamos criar uma página específica para listar apenas jobs premium, concluíndo o trabalho que começamos hoje.