02

Conhecendo o ecossistema: paginação e deploy para produção

Passos finais

Neste último capítulo, nós faremos pequenos ajustes na aplicação e finalmente vamos colocá-la em produção, para que todos possam acessá-la.

Ordenação

Um problema com a nossa aplicação é que nós sempre listamos os *jobs* mais antigos primeiro que os *jobs* mais recentes, e na prática gostaríamos que este comportamento fosse diferente. As oportunidades de emprego mais recentes deveriam vir primeiro, já que as oportunidades antigas provavelmente já foram preenchidas.

Felizmente, mudar a ordenação em que os *jobs* são mostrados pode ser feito de forma muito simples. Abra o arquivo *app/controllers/jobs_controller.rb* e modifique a *action premium* para o seguinte:

```
def premium
  @jobs = Job.where(premium: true).order("created_at DESC").all
end
```

Com isso, dizemos ao Active Record que queremos ordernar os *jobs* pelo campo `created_at` e por ordem decrescente (`DESC`). Se quisermos ordernar por ordem crescente, poderíamos usar `ASC` . Abra a página inicial da aplicação e verifique que os *jobs* agora são exibidos em ordem decrescente.

Nós poderíamos fazer a mesma modificação na *action index* para também mostrar ali os *jobs* mais recentes. Porém, se simplesmente adicionarmos `order("created_at DESC")` na *action index*, estaríamos duplicando o código. Nós já sabemos que o Rails não vê duplicação de código com bons olhos e possui diversas técnicas para que a gente não se repita. Neste caso, o Active Record possui o conceito de escopos (`scope`) que possibilita que a gente defina lógicas sobre como os nossos modelos acessarão o banco de dados.

Para criar o nosso primeiro escopo, abra o arquivo *app/models/jobs.rb* e adicione a seguinte linha:

```
scope :most_recent, order("created_at DESC")
```

Neste ponto, o nosso modelo `Job` deve estar similar a:

```
class Job < ActiveRecord::Base
  has_many :comments
  scope :most_recent, order("created_at DESC")
  attr_accessible :description, :title, :premium
  validates_presence_of :description, :title
end

Com isso, vamos voltar à *action premium* e reescrevê-la como:

```ruby
def premium
```
```

```
  @jobs = Job.where(premium: true).most_recent.all
end
```

Observe como trocamos `order("created_at DESC")` por `most_recent`. Se você recarregar a página no seu navegador, verá que a ordenação continua a mesma. O escopo `most_recent` que criamos pode ser usado sempre que quisermos obter os registros de *jobs* com essa ordenação. Por exemplo, podemos também usá-lo na *action index*, evitando a duplicação que acabamos de comentar.

Nós podemos criar quantos escopos quisermos, usando a sintaxe de busca do Active Record, como `where` (para setar condições), `order` (para ordenar), `limit` (para limitar o número de objetos retornados) e outros.

Paginação

A medida que mais pessoas usam a nossa aplicação, nós teremos mais *jobs* cadastrados, ao ponto que mostrar todos os *jobs* em uma página só se torna inviável. Neste caso, é comum quebrar a listagem de *jobs* em diversas páginas.

Uma das grandes vantagens de desenvolver projetos com Ruby on Rails é que existe uma grande comunidade de desenvolvedores criando ferramentas que podemos usar em nossas aplicações para sermos mais produtivos.

Logo, não iremos implementar a funcionalidade de paginação manualmente mas sim utilizar uma dessas ferramentas. Para isso, abra o arquivo *Gemfile* que está na raiz da sua aplicação e adicione:

```
gem 'will_paginate'
```

Gems é a forma que a comunidade Ruby utiliza para distribuir código. Neste caso, simplesmente mudamos o arquivo *Gemfile* para adicionar a **gem** chamada **wilL_paginate** como dependência do nosso projeto. Após adicionar a **gem**, vá para a linha de comando e digite:

```
$ bundle install
```

Em aplicações Rails, o **Bundler** é a ferramenta responsável por gerenciar as nossas **gems** e dependências. Com o comando `bundle install`, estamos pedindo ao **Bundler** para instalar qualquer dependência que não esteja ainda disponível na nossa máquina, neste caso o **will_paginate** que acabamos de adicionar ao **Gemfile**.

Com a *gem* instalada, podemos acessar novamente o `JobsController` e modificar a *action premium* para incluir paginação:

```
def premium
  @jobs = Job.where(premium: true).most_recent.paginate(page: params[:page], per_page: 10)
end
```

Observe que trocamos `all` por `paginate(page: params[:page], per_page: 10)`. `paginate` é um método adicionado pela **gem will_paginate** que fará a paginação. Esse método recebe como opções qual a página atual que gostaríamos de exibir (`page`) e o número de *jobs* que gostaríamos de mostrar por página (`per_page`).

Agora que modificamos o *controller*, vamos alterar a *view* para mostrar os links de paginação. Abra a *view* *jobs/premium.html.erb* e modifique-a para:

```
<h1>Premium jobs</h1>

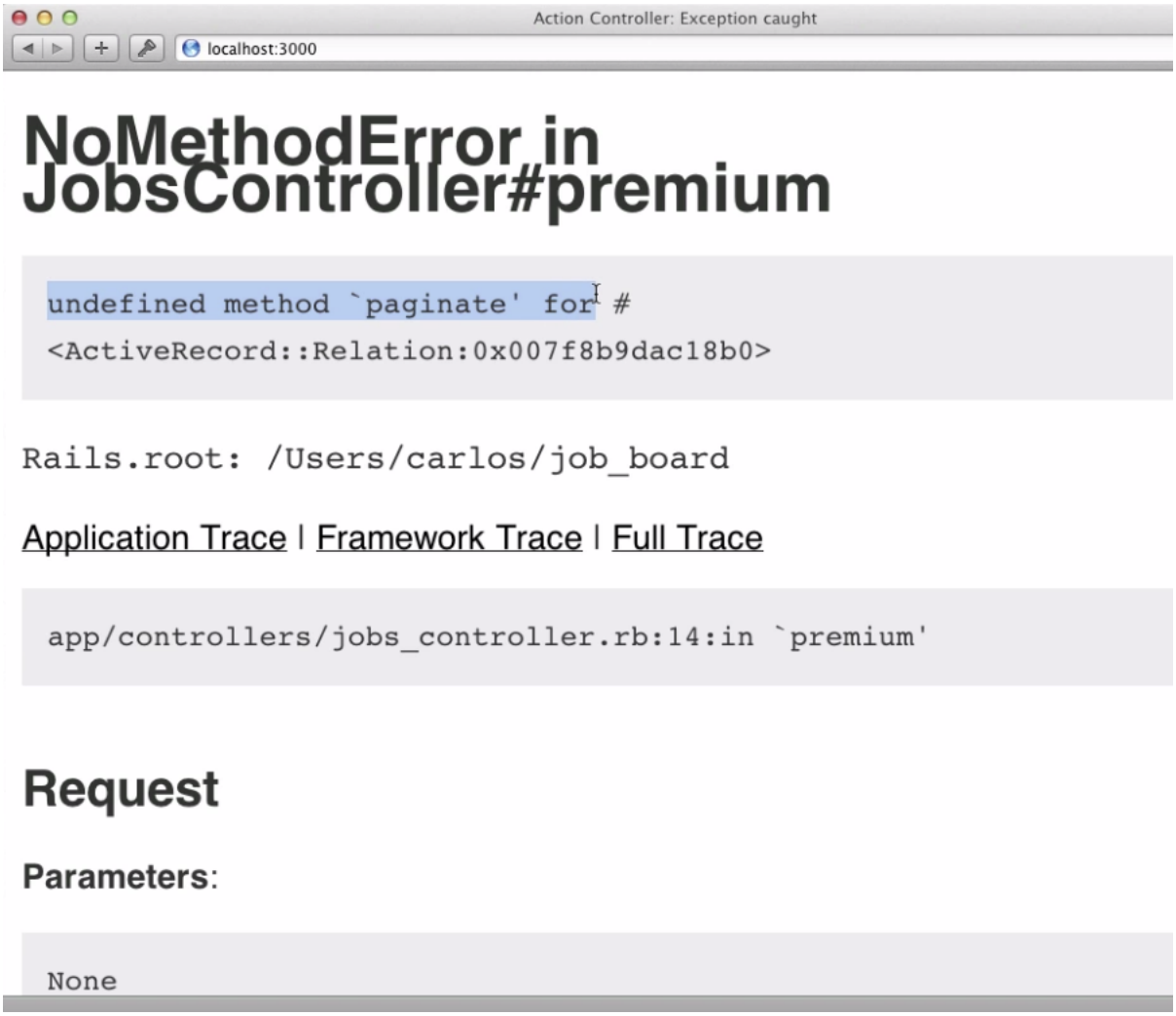
<%= render @jobs %>
<%= will_paginate @jobs %>

<br />
<%= link_to 'New Job', new_job_path %>
<br />
<%= link_to 'Hello World', hello_world_path %>
<%= link_to 'All jobs', jobs_path %>
```

E com isso, o *helper* `will_paginate` mostrará os *links* de paginação para nós sempre que tivermos mais de 10 *jobs premium* no nosso banco de dados. Para garantir que temos mais que 10 *jobs premium* para testarmos a paginação, inicie o console do Rails com `rails console` e digite:

```
>> 15.times do
>>   Job.create(title: "Novo job", description: "Descricao exemplo", premium: true)
>> end
```

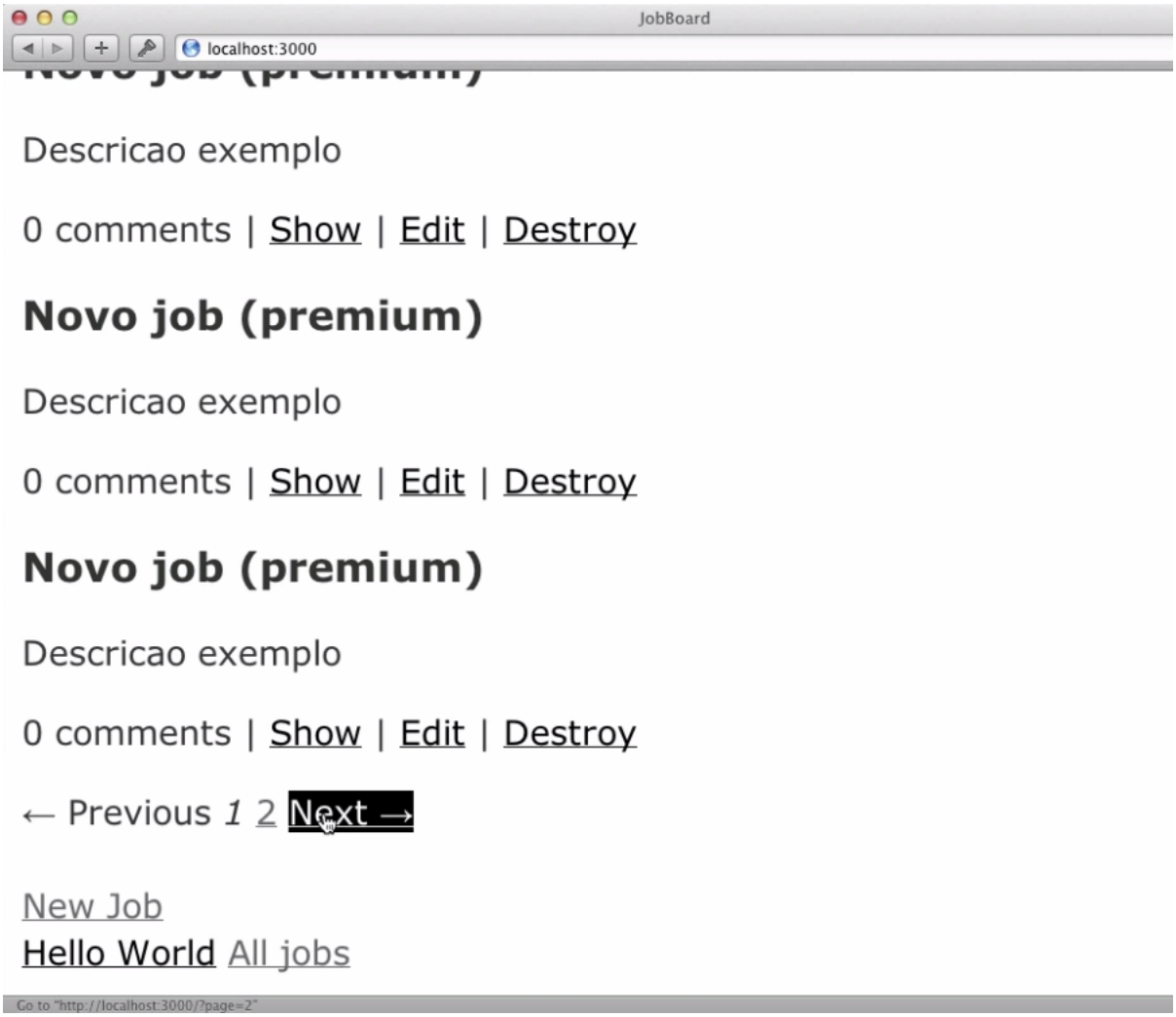
Com isso, podemos ver que 15 *jobs premium* foram criados no banco de dados. Para verificar que o **will_paginate** funciona como esperado, vamos acessar a página inicial da aplicação:



Opa, parece que algo errado aconteceu. A mensagem de erro diz que o método `paginate` , que utilizamos na *action premium*, não está disponível. Isso acontece porque, toda vez que adicionamos uma nova *gem* à nossa aplicação, nós

precisamos reiniciar o servidor do Rails. Para isso, termine o servidor atual utilizando `Ctrl+C` e comece um novo com `rails server .`

Agora sim, acesse a página inicial da aplicação e verifique que os *links* de paginação estão disponíveis:



Lembre-se que, na maioria dos casos, nós não precisamos reiniciar o servidor para que as nossas mudanças sejam recarregadas enquanto estamos desenvolvendo a aplicação. Porém, toda vez que adicionamos uma nova *gem* ou modificamos o arquivo *config/application.rb*, precisamos reiniciar o servidor para que o Rails carregue essas alterações.

Colocando a nossa aplicação em produção com o Heroku

Agora que nós fizemos os últimos ajustes na nossa aplicação, estamos prontos para colocá-la em produção (ou seja, fazermos **deploy**). Vamos utilizar o *heroku* (http://www.heroku.com), que é uma plataforma para deploy de aplicações. Crie uma nova conta através do botão **Sign Up**, usando seu email. O *heroku* vai enviar um email de confirmação, abra este email e clique no link existente nele, para voltar ao *heroku* e confirmar a validade do seu email. O próximo e último passo para e criar sua conta é informar uma senha, e pronto!

Com a conta criada, é exibida uma página com informações sobre como fazer o deploy da nossa primeira aplicação. O *heroku* disponibiliza um **toolbelt** (https://toolbelt.heroku.com), um pacote de ferramentas com tudo que precisamos para colocar nossa aplicação no ar:

- O cliente de linha de comando do *heroku*, para executarmos comandos via terminal.
- Foreman, uma opção para executarmos nossas aplicações localmente - que não vamos precisar no momento.

- Git (http://git-scm.com), o sistema de controle de versão necessário para enviarmos a aplicação para o *heroku*.

Baixe e instale o toolbelt de acordo com seu sistema operacional. Após isso você deve ser capaz de verificar a versão do git pelo terminal, executando

```
$ git --version
```

E também verificar que o cliente do *heroku* foi instalado, usando o comando:

```
$ heroku --help
```

Com as ferramentas que precisamos já instaladas, vamos ao deploy. A própria página do *heroku toolbelt* nos mostra o que fazer: devemos primeiro autenticar nosso usuário do *heroku* usando o cliente de linha de comando:

```
$ heroku login
```

Ao executar o comando `heroku login`, vai ser necessário informar o email e a senha que você cadastrou no *heroku*. Após isso, caso você não tenha uma chave SSH na sua máquina, ele vai perguntar se você deseja gerar uma e vai então enviar essa chave para o *heroku*. Caso sua máquina já possua uma chave SSH, o heroku vai utilizá-la. O SSH é uma forma segura de fazer a comunicação entre seu computador (máquina local) e o *heroku* (máquina remota), por isso é necessário uma chave para garantir a segurança dessa comunicação.

Bom, com a linha de comando `heroku` propriamente configurada, partimos para a única alteração que vamos precisar fazer na aplicação: a dependência do banco de dados. O Rails possui três ambientes de desenvolvimento por padrão: **development** (desenvolvimento), **test** (testes) e **production** (produção).

Durante o desenvolvimento da aplicação, utilizamos o banco de dados *SQLite*, mais simples e próprio para o desenvolvimento, e que normalmente já vem instalado em muitos sistemas operacionais (ou pode ser instalado de forma simples). Porém, o *heroku* utiliza o banco de dados *PostgreSQL*, mais robusto e mais apropriado para aplicações em produção. Desta forma, precisamos configurar uma *gem* como dependência da nossa aplicação para comunicar com esse banco de dados quando estivermos em produção, e manter o *SQLite* para quando estamos desenvolvendo localmente ou executando os testes. Abra o *Gemfile*, procure pela linha que diz `gem 'sqlite3'` e altere conforme abaixo:

```
gem 'sqlite3', group: [:development, :test]
gem 'pg', group: :production
```

Como podemos ver, modificamos a linha que adicionava a `gem sqlite3`, para ser utilizada apenas nos ambientes **development** e **test**, e adicionamos a gem `pg` para que possamos utilizar o banco de dados *PostgreSQL* no *heroku*, quando o ambiente for **production**.

Com isso configurado, vamos voltar para o console e rodar novamente o comando que garante que as dependências necessárias estão instaladas na nossa máquina, mas com uma pequena diferença desta vez:

```
$ bundle install --without production
```

Note que ao executarmos o `bundle install` agora passamos o argumento `--without production`, informando para o *Bundler* não se preocupar em tentar instalar as gems do ambiente de produção `production`, nesse caso a gem `pg`. Fazemos isto pois neste momento não queremos instalar o banco de dados *PostgreSQL* e nem essa gem `pg` na nossa máquina de desenvolvimento, pois vamos continuar com o *SQLite*.

Com isso resolvido, precisamos agora enviar o código da aplicação para o *heroku*. Vamos fazer uso da ferramenta `git` que é um sistema de controle de versão de arquivos. Primeiro inicie o controle de versão no diretório da aplicação:

```
$ git init .
```

Depois adicione todos os arquivos no controle de versão, para que o `git` saiba o que deve ser gerenciado por ele:

```
$ git add .
```

E finalmente, peça para o `git` gravar essas informações no controle de versão:

```
$ git commit -m 'Initial commit'
```

Perceba que ao executar o comando `git commit` para gravar a versão dos arquivos, passamos o argumento `-m` com um texto descrevendo o que estamos gravando: esse texto serve para facilitar a identificação no futuro do que estamos salvando no `git`.

Maravilha, com o controle de versão inicializado na aplicação, basta apenas enviarmos tudo para o *heroku*. Vamos então pedir ao *heroku* que crie uma nova aplicação para nós, executando:

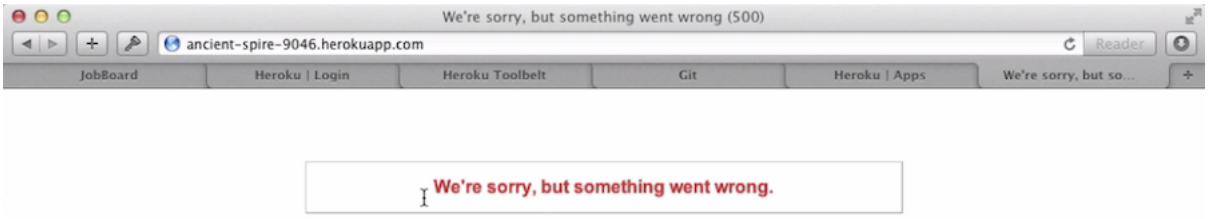
```
$ heroku create
```

Ao fazer isso, o *heroku* criar toda a estrutura da aplicação para nós, sem precisarmos nos preocupar com vários detalhes que envolvem o deploy. Ele vai nos devolver qual o nome e a url da nossa nova aplicação, que no meu caso foi (http://ancient-spire-9046.herokuapp.com)http://ancient-spire-9046.herokuapp.com (http://ancient-spire-9046.herokuapp.com). Você receberá um nome e url diferentes. Vamos então enviar nosso código para o *heroku* utilizando o `git`:

```
$ git push heroku master
```

Nota: o nome da aplicação foi gerado automaticamente pelo *heroku*, mas podemos também passar um nome para o comando `heroku create` se quisermos definir o nome da aplicação. Tenha em mente que este nome deve ser único, ou seja, se outra pessoa já estiver utilizando o nome que você deseja, você precisará escolher outro. Utilize `heroku create --help` para ver mais informações sobre este comando.

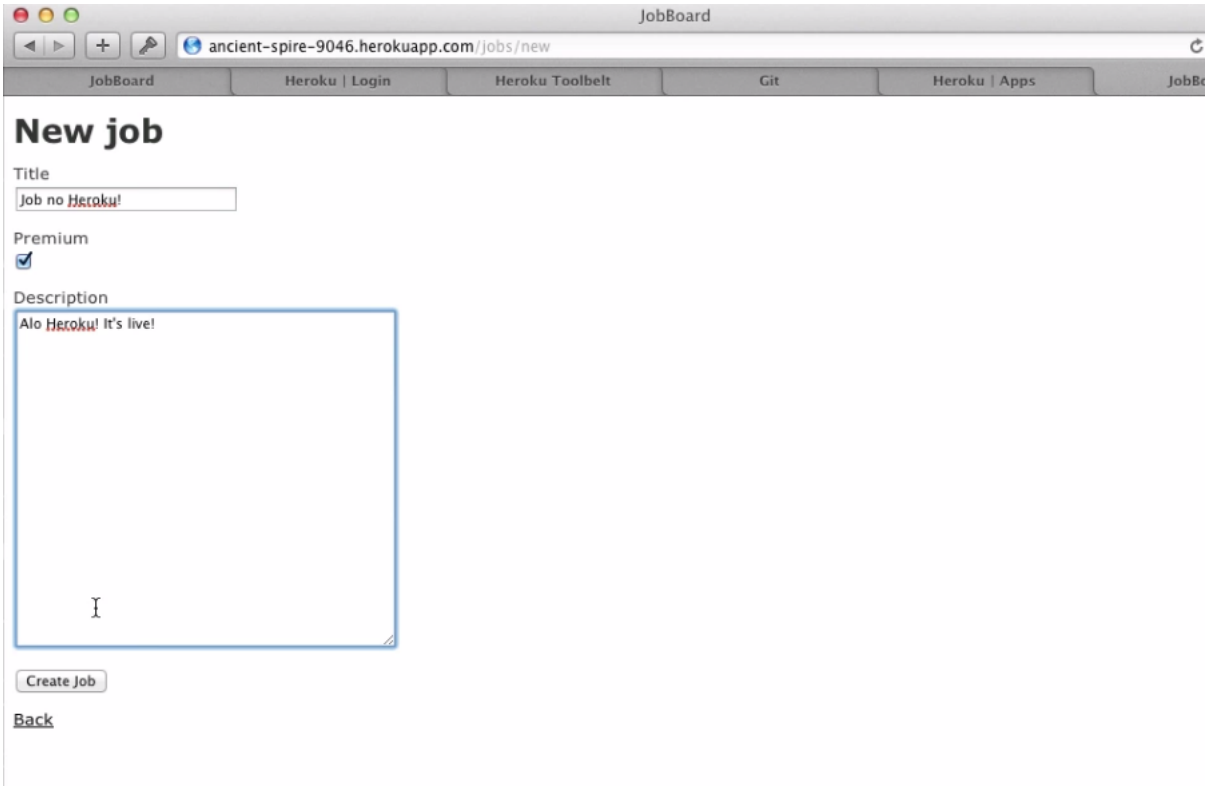
Tudo pronto! Hora de testar a aplicação no navegador, copie a url que você recebeu do *heroku* e utilize-a no navegador para acessar a aplicação.



Opa, algo deve ter saído errado, ou esquecemos de algo? Na verdade, esquecemos que é necessário rodar as migrações no novo banco de dados de produção, pois ele ainda não sabe da existência das nossas tabelas *jobs* e *comments*. Vamos utilizar o mesmo comando `rake db:migrate`, mas através do `heroku`, execute o comando abaixo:

```
$ heroku run rake db:migrate
```

Após executar as migrações no banco de dados do *heroku*, podemos voltar ao navegador e ver que nossa aplicação está funcionando, podemos listar e criar novos jobs, sucesso!



Neste capítulo aprendemos sobre ordenação de registros e escopos para evitarmos duplicação de lógica relacionada aos modelos. Também vimos como adicionar paginação de dados as nossas listas, e com isso como fazer uso de código escrito por outras pessoas da comunidade através de *gems*, para melhorar nossa produtividade. Ao final, utilizamos a plataforma *heroku* para colocar nossa aplicação em produção, para que outras pessoas possam acessá-la de qualquer lugar.

Com isso fechamos o capítulo 7 e este curso de Rails, obrigado.