02

Criando e removendo comentários através da interface web

Relembrando

No capítulo anterior, nós associamos comentários a *jobs* de forma que possamos criar e buscar todos os comentários associados a um *job* de forma simples. Neste capítulo, daremos o próximo passo permitindo que os usuários da nossa aplicação criem e removam comentários através da interface web.

Criando comentários

Nós vamos adicionar a funcionalidade de criar comentários em três passos:

1.
2. Adicionar uma nova rota que vai apontar para um controller e action responsáveis por criar novos comentários;
3.
4. Alterar a view de visualização de *jobs* para incluir o formulário de envio de comentários;
5.
6. Criar o controller e action que vão receber os dados do formulário.

Para adicionar a nova rota, abra o arquivo *config/routes.rb* e modifique a linha:

```
resources :jobs
```

Para:

```
resources :jobs do
  post "comments", to: "comments#create"
end
```

Observe que nós definimos a nossa nova rota dentro de `resources :jobs`. O Rails chama tais rotas de *rotas aninhadas* e elas são bastante úteis para modelar associações. Se executarmos `rake routes`, podemos ver a seguinte linha:

```
job_comments POST   /jobs/:job_id/comments(.:format) comments#create
```

Note que a rota de comentários está aninhada dentro da rota de *jobs*. Isso mapeia bem para a estrutura que temos no banco de dados onde um *job* possui vários comentários. A contra-partida de usar rotas aninhadas é que, para gerarmos a rota para um comentário, precisamos do *job* ao qual o comentário irá pertencer.

Nós também apontamos essa rota para um *controller* chamado "comments" e *action* "create", que vamos criar mais tarde.

Com a rota em mãos, podemos adicionar o formulário de criação de comentários. De forma a manter a mesma convenção do Rails, nós vamos criar o formulário em uma partial e depois usaremos essa partial dentro de *jobs/show.html.erb*. Logo, crie um arquivo em *app/views/comments/_form.html.erb* com o seguinte conteúdo:

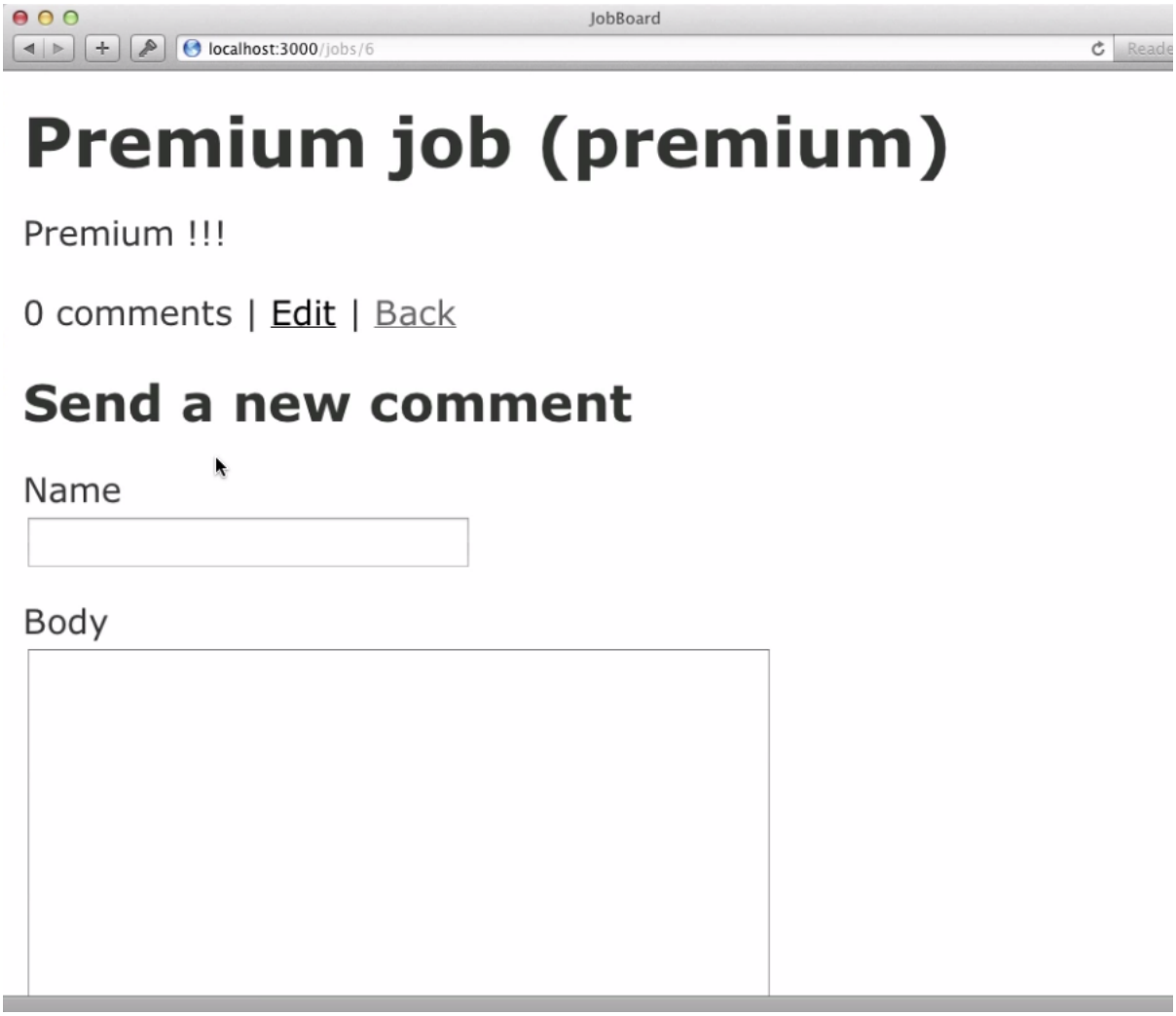
```
<%= form_for [@job, Comment.new] do |f| %>
  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </div>
  <div class="actions">
    <%= f.submit "Send" %>
  </div>
<% end %>
```

O nosso formulário é bem simples. Ele usa o helper `form_for` do Rails para gerar o formulário, juntamente com `f.label`, `f.text_field` e `f.text_area`, de forma similar ao form de *jobs*. Observe que passamos como argumento para o `form_for` um array com o job atual (`@job`) e um novo comentário (`Comment.new`). Com essa informação, o Rails vai saber usar a rota que acabamos de adicionar.

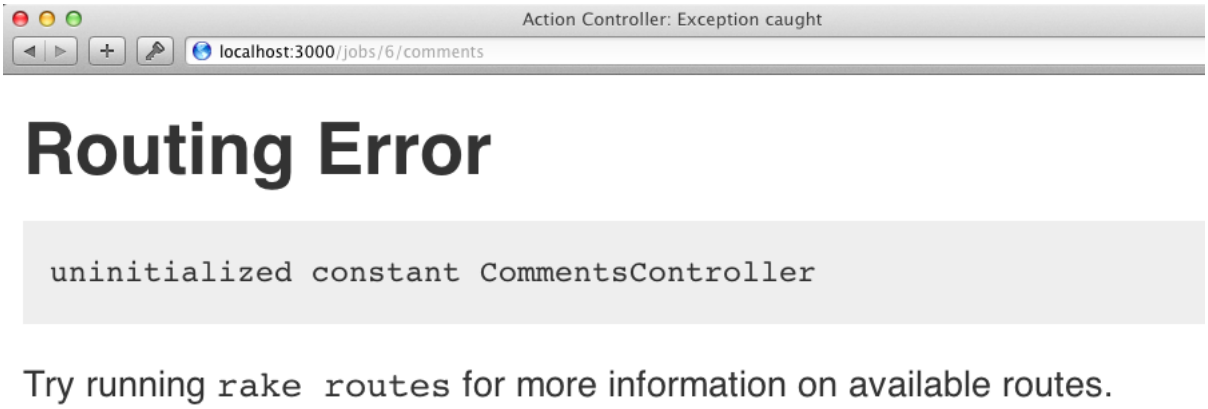
Após criarmos a partial, vamos modificar a *view* em *app/views/jobs/show.html.erb* e adicionar o seguinte conteúdo antes de renderizar os comentários:

```
<h2>Send a new comment</h2>
<%= render "comments/form" %>
```

Agora, a página de exibição de *jobs* vai ter uma seção para enviar um novo comentário ao renderizar a *partial* que acabamos de criar. Abra a página de exibição de jobs no browser para verificar:



Porém, se preenchermos os campos e pressionarmos enviar, veremos uma página de erro:



Observe a uri da página de erro: *http://localhost:3000/jobs/ID/comments (http://localhost:3000/jobs/ID/comments)*, onde **ID** é o *id* do *job* ao qual queremos adicionar um comentário. Essa uri também é compatível com a rota que definimos e verificamos via `rake routes` :

```
job_comments POST   /jobs/:job_id/comments(.:format) comments#create
```

Logo, o motivo do erro não é a ausência da rota, mas sim porque ainda não definimos o *CommentsController* usado pela rota. Para isso, crie um arquivo em *app/controllers/comments_controller.rb* com o seguinte conteúdo:

```
class CommentsController < ApplicationController
  def create
    @job = Job.find(params[:job_id])
    @comment = @job.comments.build(params[:comment])
    @comment.save
    redirect_to @job
  end
end
```

O nosso controller possui uma action `create` que será responsável por criar um novo comentário. A primera ação que fizemos nessa *action* é buscar um *job* no banco de dados o qual o *id* do *job* é igual ao valor de `params[:job_id]`, onde `params` contém os parâmetros (`params`) da requisição HTTP. No caso de `params[:job_id]`, o parâmetro `:job_id` foi extraído da rota que acessamos.

Na segunda linha, construímos um comentário associado ao `@job` usando a estrutura `@job.comments` que aprendemos no capítulo anterior e passando os parâmetros em `params[:comments]`. Neste caso, o parâmetro `:comment` vêm direto do formulário enviado pelo browser e vai conter os valores preenchidos nos campos *Name* e *Body*.

Em seguida, salvamos o comentário no banco de dados e redicionamos para a página do *job*. Abra novamente a página de um *job* no browser, preencha todos os campos de comentário e clique em "Send". Dessa vez, tudo deve funcionar, e você deve ver o seu comentário criado com sucesso!

Se abrirmos a aba do terminal em que executamos `rails server`, podemos ver que o Rails está **logando** todos os acessos que fazemos na aplicação. Se voltarmos um pouco nesse log de informações, podemos achar um acesso

começando com `Started POST "/jobs/1/comments"` , copiado abaixo:

```
Started POST "/jobs/1/comments" for 127.0.0.1 at 2012-08-15 20:22:14 -0300
Processing by CommentsController#create as HTML
  Parameters: {"utf8"=>"", "authenticity_token"=>"74PANCvyFOQ7gU70Wit6sdit8/j4f9yUsvommz1ClZc="
  Job Load (0.1ms)  SELECT "jobs".* FROM "jobs" WHERE "jobs"."id" = ? LIMIT 1  [["id", "1"]]
Redirected to http://localhost:3000/jobs/1
Completed 302 Found in 3ms (ActiveRecord: 0.2ms)
```

O *log* do Rails mostra a rota que acessamos, qual o IP que está acessando a rota e o horário. Além do mais, ele mostra o *controller* e *action* na segunda linha seguido dos parâmetros (*Parameters*) enviados do navegador, onde podemos verificar os valores de `params[:job_id]` e `params[:comment]` com os dados do formulário. Essa informação no *log* é importante quando precisamos identificar o que está sendo enviado para a nossa aplicação ou precisamos apurar algo que não está funcionando corretamente.

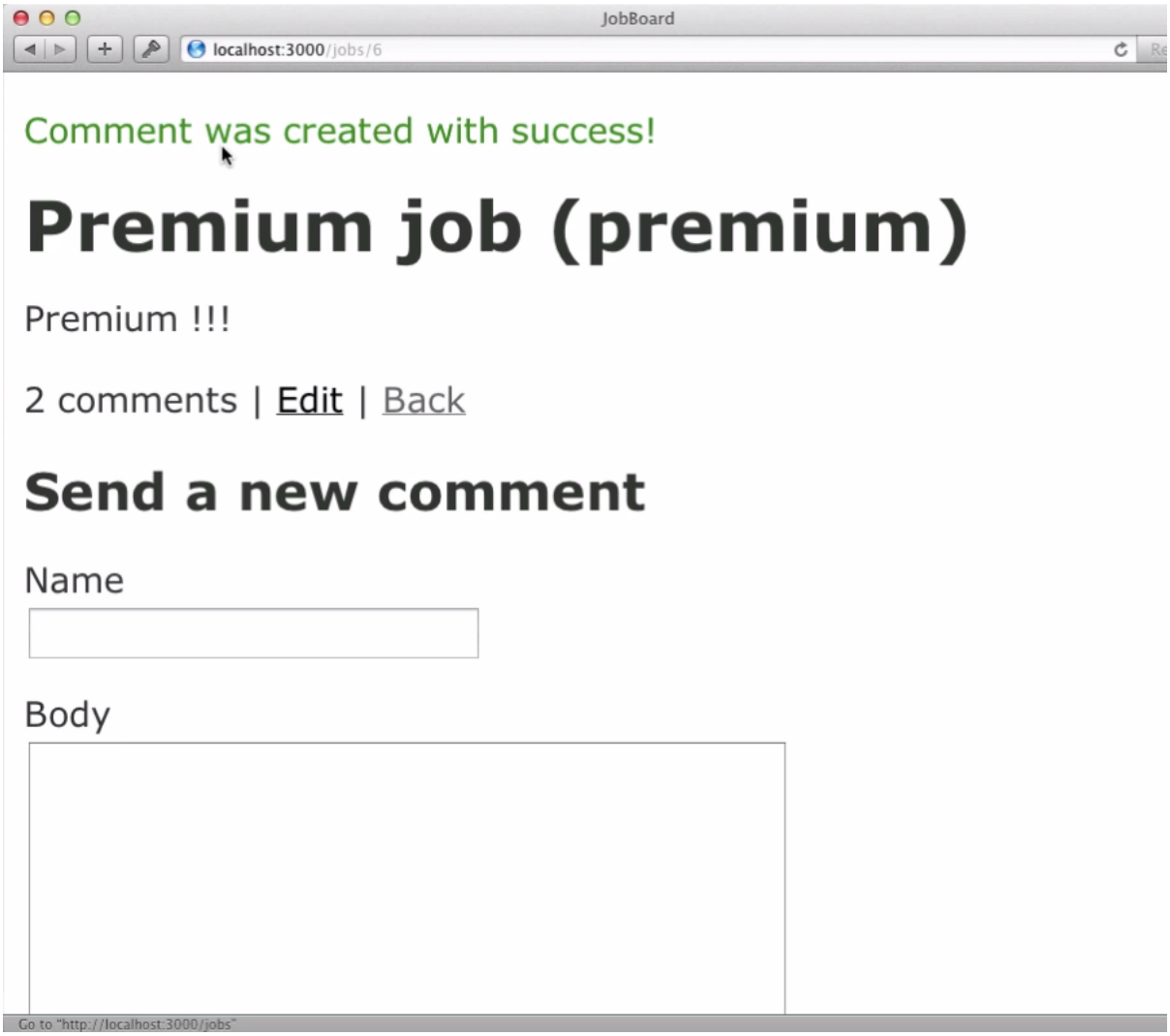
Mensagens de sucesso

Já que agora podemos criar comentários, seria interessante se pudessemos mostrar ao usuário uma mensagem dizendo que o comentário foi criado com sucesso. Para tal, podemos modificar a nossa *action* `create` para o seguinte:

```
class CommentsController < ApplicationController
  def create
    @job = Job.find(params[:job_id])
    @comment = @job.comments.build(params[:comment])
    @comment.save
    flash[:notice] = "Comment was created with success!"
    redirect_to @job
  end
end
```

Observe que adicionamos uma linha que seta o valor de `flash[:notice]` para uma mensagem de sucesso. No Rails, essas mensagens são chamadas de **mensagens flash**, exatamente porque elas irão aparecer uma única vez na próxima página que for exibida.

Crie um novo comentário e você pode verificar que a mensagem agora aparece no topo da página:



Você pode estar imaginando qual parte do Rails exatamente é responsável por mostrar essa mensagem de sucesso? Abra a view que estamos renderizando para mostrar o *job*, ou seja *app/views/jobs/show.hmtl.erb*, e verifique que a primeira linha contém:

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

Logo, essa linha é a responsável por gerar um parágrafo no HTML com o conteúdo do `notice`. Neste caso, o helper `notice` é simplesmente um atalho para `flash[:notice]`. Para verificar isso, troque a linha acima por:

```
<p id="notice"><%= flash[:notice] %></p>
```

E verifique que a mensagem aparece corretamente ao criar um novo comentário. Como setar mensagens de `notice` é algo bastante comum, o Rails possui o helper `notice` como uma conveniência.

Casos de erro

Embora possamos criar comentários com sucesso, existe um cenário que não contemplamos na nossa aplicação: o caso em que um comentário não pode ser criado por questões de validação. Por exemplo, se deixamos o campo *Name* em branco, o Rails não vai criar o comentário e mesmo assim a mensagem de sucesso continua aparecendo.

Logo, precisamos verificar se o comentário pôde ser criado com sucesso ou não. No capítulo 3, nós comentamos que o método `save`, que estamos usando no nosso `CommentsController`, retorna `true` se o modelo pôde ser salvo com

sucesso e, caso contrário, `false`. Com isso em mente, podemos modificar a nossa action de `create` para considerar estes dois cenários:

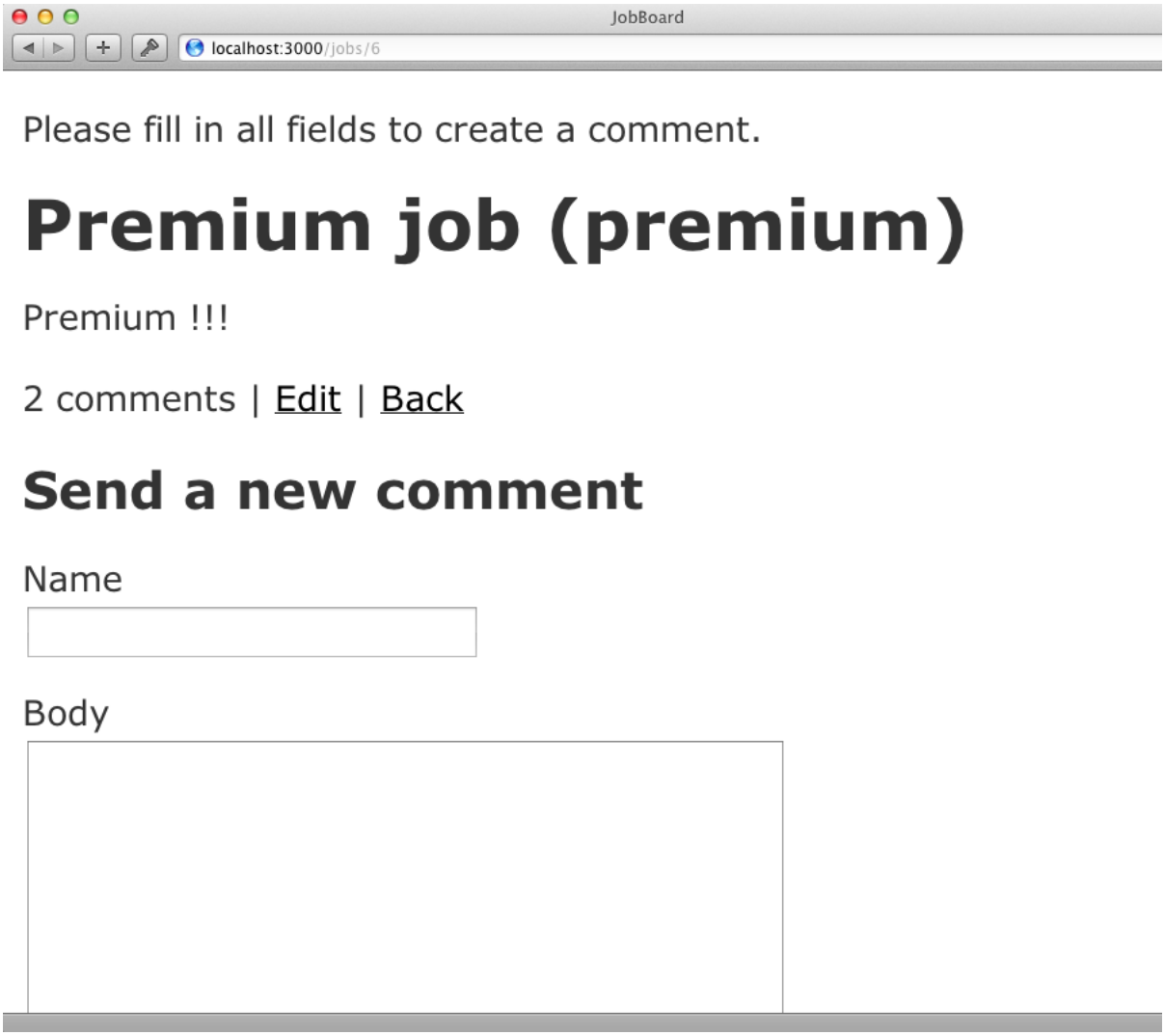
```
class CommentsController < ApplicationController
  def create
    @job = Job.find(params[:job_id])
    @comment = @job.comments.build(params[:comment])
    if @comment.save
      flash[:notice] = "Comment was created with success!"
    else
      flash[:alert] = "Please fill in all fields to create a comment."
    end
    redirect_to @job
  end
end
```

Observe que para o caso de erro, utilizamos `alert` invés de `notice`. Nós podemos armazenar o que quisermos como mensagens *flash* e elas ficarão disponíveis para nós até a próxima requisição, neste caso, a próxima requisição é a exibição do *job* que comentamos. Antes de verificarmos que a nossa mensagem de erro aparece, temos que modificar a view de exibição do *job* para mostrar a mensagem de *alert*. Abra novamente a view *jobs/show.html.erb* e adicione ao topo do arquivo o seguinte:

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

Nós usamos `alert` invés de `flash[:alert]`. Como é muito comum usarmos mensagens de sucesso (*notice*) e de erro (*alert*), o Rails possibilita que a gente use simplesmente `alert`, por conveniência, da mesma forma que podemos usar apenas `notice`.

Abra a página de exibição de *job* no seu navegador e agora tente enviar um comentário sem preencher nenhum dos campos. A nossa mensagem de erro aparecerá no topo da página:

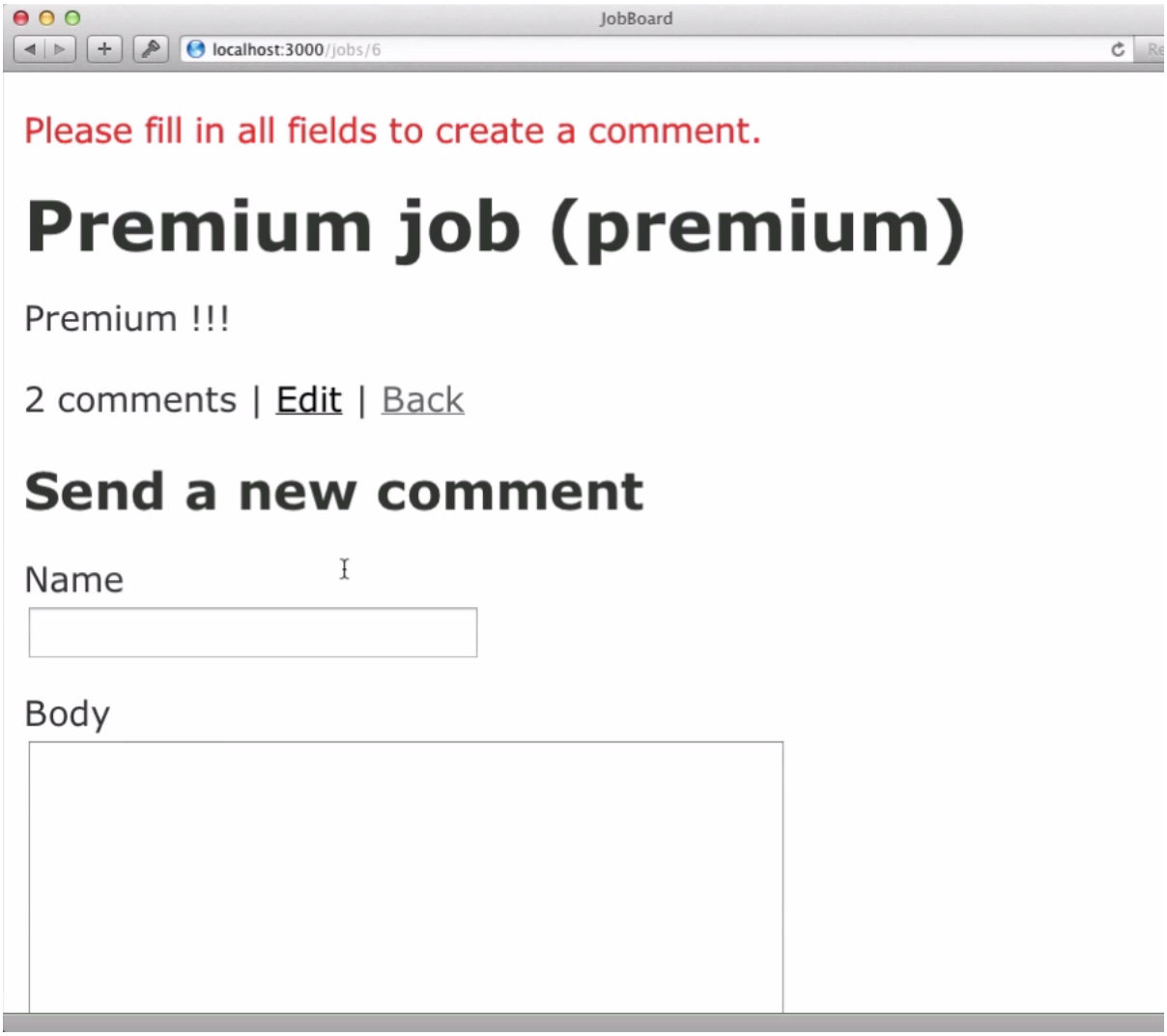


Porém, a nossa mensagem de erro aparece em fonte preta. Isso acontece porque, como o Rails gerou somente a mensagem de *notice* no *scaffold*, ele não criou um estilo CSS para a mensagem de *alert*. Nós podemos corrigir esse problema customizando o CSS enviado para o navegador.

O Rails mantém todos os *assets* da nossa aplicação, ou seja, imagens, arquivos javascript e CSS dentro do diretório *app/assets*. Abra o arquivo *app/assets/stylesheets/application.css*, que contém o CSS geral da nossa aplicação, e adicione a seguinte linha ao final:

```
#alert {
  color: red;
}
```

Agora tente enviar um comentário em branco através do navegador e observe que agora a nossa mensagem aparece em cor vermelha:



Excelente! Com isso concluímos o código responsável por criar comentários. Agora estamos prontos para adicionar o código que possibilitará usuários remover comentários enviados.

Apagando comentários

Para adicionar a funcionalidade de apagar comentários, temos que seguir os seguintes passos:

1.
2. Adicionar uma nova rota que vai apontar para um controller e action responsáveis por apagar comentários;
3.
4. Alterar a *partial* de comentários para incluir o link de *delete*;
5.
6. Criar o controller e action que vão finalmente apagar o comentário

Já que para remover um comentário, a única informação que precisamos é o *id* do comentário, que é um identificador único, vamos adicionar uma simples rota à nossa aplicação ao invés de uma rota aninhada dentro de *jobs*. Logo, abra o arquivo *config/routes.rb* e adicione a seguinte rota:

```
delete "comments/:id", to: "comments#destroy", as: :comment
```

A nossa rota estará disponível em */comments/:id* e apontará para a *action* de `destroy`. Observe que escolhemos `destroy` como os nome da *action* para seguir o padrão do Rails que gera essas mesmas actions no *scaffold*. Nós

também passamos um parâmetro extra chamado `as: :comment` que vai ser usado como nome da rota nomeada, que podemos verificar através do `rake routes`:

```
comment DELETE /comments/:id(.:format)          comments#destroy
```

Agora, abra a *partial* de comentários em *app/views/comments/_comment.html.erb* e vamos adicionar um link para a nova rota:

```
<h4><%= comment.name %> said:</h4>
<%= simple_format comment.body %>
Sent <%= time_ago_in_words comment.created_at %> ago
<%= link_to "Destroy", comment_path(comment), method: :delete, confirm: "Are you sure?" %>
```

O link que adicionamos é bastante similar ao que vimos anteriormente na listagem de *jobs*. Se abrirmos o nosso navegador e clicarmos no link, ele não funcionará, já que ainda não criamos o *action* no `CommentsController` responsável por tratar a requisição. Logo, abra o `CommentsController` e adicione a seguinte *action*:

```
def destroy
  @comment = Comment.find(params[:id])
  @comment.destroy
  redirect_to @comment.job, notice: "Comment destroyed with success."
end
```

A nossa *action* busca um *comment* com o *ID* enviado, usa o método *destroy* para remove-lo do banco de dados e depois redireciona para a página do *job* associado ao comentário. Observe que passamos a mensagem de *notice* direto ao método `redirect_to`, que seria exatamente o mesmo se tivessemos adicionado o seguinte à linha anterior:

```
flash[:notice] = "Comment destroyed with success"
```

Como dizemos anteriormente, já que setar mensagens de sucesso (*notice*) e erro (*alert*) é comum, o Rails possui algumas conveniências, e essa é uma outra conveniência.

Abra a página de exibição de algum *job* no navegador e clique no link de `Destroy` próximo ao comentário e podemos ver que o comentário pode ser removido com sucesso! Com isso, concluímos o que nos propomos a implementar neste capítulo. Porém, antes de encerrarmos, existe um tópico bastante importante que precisamos discutir, os métodos do protocolo HTTP.

Os métodos do protocolo HTTP

Quando o seu navegador (ou seja, um programa) acessa uma aplicação web (um outro programa), eles precisam de algum protocolo que especifique como eles vão trocar informação entre eles. Na nossa aplicação web, nós estamos usando o protocolo HTTP (Hypertext Transfer Protocol).

O protocol HTTP especifica como uma requisição deve ser enviada e como uma resposta deve ser retornada pela aplicação. Um exemplo de uma requisição é:

```
GET /jobs HTTP/1.1
Host: localhost:3000
```

O **GET** acima representa **o método do protocolo HTTP**, seguido pelo caminho */jobs* e a versão do protocolo HTTP sendo usada, neste caso *1.1*. Nas próximas linhas, temos os cabeçalhos da requisição, neste caso, um cabeçalho *Host* especificando o endereço web o qual a aplicação está hospedada e finalmente temos o corpo da requisição, que neste caso está vazio.

Dependendo do tipo de ação que queremos desempenhar na aplicação web, é recomendado que usemos um método diferente do HTTP e é exatamente isso que o Rails faz. Se executarmos `rake routes` e analizarmos as rotas para *jobs*, podemos ver que o Rails usa 4 diferentes métodos do HTTP:

```
jobs GET    /jobs(.:format)                 jobs#index
     POST   /jobs(.:format)                 jobs#create
     PUT    /jobs/:id(.:format)             jobs#update
     DELETE /jobs/:id(.:format)             jobs#destroy
```

Observe que, mesmo em que alguns casos o caminho das rotas seja o mesmo, o Rails consegue diferenciar entre uma *action* e outra baseada no método do HTTP. É exatamente por este motivo que, ao criarmos o *link* para remover o comentário, nós passamos um argumento `method: :delete`, indicando que queremos que o método **DELETE** seja usado:

```
<%= link_to "Destroy", comment_path(comment), method: :delete, confirm: "Are you sure?" %>
```

No geral, o comportamento de cada método pode ser resumido da seguinte forma:

-
- GET - Usado quando queremos simplesmente visualizar uma página. Ele não causa nenhuma mudança no servidor (não cria, edita ou remove informações).
-
- POST - Usado quando queremos adicionar algo ao nosso servidor, como criar um novo *job*.
-
- PUT - Usado quando queremos atualizar algum estado no nosso servidor.
-
- DELETE - Usado quando queremos remover algo do nosso servidor, como remover um comentário.

Um bom desenvolvedor Rails procura usar os métodos corretos do HTTP para cada uma de suas rotas, de forma a melhor usar as convenções do Rails.

Com isso, encerramos o nosso capítulo. Agora nós podemos criar e remover comentários na nossa aplicação, utilizar as mensagens *flash* do Rails ao mesmo que tempo que aprendemos mais sobre o protocolo HTTP.