Introdução

O conceito REST (Representational State Transfer) foi criado por Roy Fielding em sua dissertação de Ph.D. [1] e representa um novo paradigma na arquitetura de aplicações web, que utiliza os métodos HTTP (GET, POST, PUT, DELETE) para manipular as resources (recursos) da web.

REST on Rails

1. Resources

Na arquitetura REST a web é considerada uma coleção de resources ao invés de páginas. E estas podem ser representadas em HTML, XML, ou RSS dependendo da requisição do cliente.

Na visão do paradigma REST, para que você visualizasse este artigo você simplismente requisitou a representação HTML da resource identificada por '*http://wiki.cercomp.ufg.br/Equipe_Web/RoR/REST*'. O servidor retornou esta representação HTML, pois o seu navegador especificou, na requisição GET, que o formato desejado da resource era HTML.

No contexto de uma aplicação Rails as resources são uma combinação de um controle e um modelo, como em '*www.cercomp.ufg.br/project/1*', onde '*1*' seria uma instância do modelo (ActiveRecord) e '*project*' o controle responsável por manipular as instâncias.

2. Métodos HTTP

Em aplicações Rails uma URL padrão seria *'/projects/show/1'* onde se defini um controlador, uma ação a ser tomada (métodos do controlador) e uma instância específica da entidade em questão no banco de dados (id).

Ao se utilizar o padrão REST você apenas defini qual resource está manipulando e seu id: *'/projects/1*'. A ação a ser tomada com esta resource é definida pelo método HTTP.

HTTP Verb	REST-URL	Action	URL without REST
GET	/projects/1	show	GET /projects/show/1
DELETE	/projects/1	destroy	GET /projects/destroy/1
PUT	/projects/1	update	POST /projects/update/1
POST	/projects	create	POST /projects/create

Lembrando que "projects" é um recurso, que poderia ser qualquer outro recurso, dependendo de cada sistema.

Os navegadores utilizam apenas os métodos GET e POST, por isso para se utilizar os métodos PUT e DELETE temos que fazer um tratamento especial utilizando hidden fields.

A eliminação da ação na URL torna os endereços mais compatíveis com o paradgima DRY.

3. Respond_to

Como vimos, as aplicações Rails renderizam a resources de diferentes formas, dependendo do formato que o cliente pede ao servidor. O responsável por gerenciar isso, é o método *respond_to.*

```
# GET /projects/1
# GET /projects/1.xml
def show
    @project = Project.find(params[:id])
    respond_to do |format|
        format.html # show.rhtml
        format.xml { render :xml => @project.to_xml }
    end
end
```

No código acima, caso a requisição do cliente seja um formato xml, o rails converterá o conteúdo a ser renderizado para XML através da função *@project.to_xml*.

Para se definir o formato requisitado ao servidor existem duas formas: através do *Accept field* do cabeçalho HTTP, ou através da URL.

4. Definindo links REST nas Views (métodos Path)

Tradicionalmente, para se criar os links com os quais os usuários interagem com a aplicação, os desenvolvedores utilizam o helper *link_to.* O método *link_to* recebe, por padrão, um hash contendo um controlador e uma ação, além de outros parâmetros opcionais.

```
link_to :controller => "projects", :action => "show", :id => project
=>
<a href="/projects/show/1">Show</a>
```

Claramente esta utilização do *link_to* não corresponde com a filosofia REST. Para resolver este problema o próprio Rails tem a solução: O *link_to* é utilizado com um método Path ao invés do hash "controlador/ação/id". Este método Path cria um link que é colocado no href gerado pelo *link_to*. O código abaixo chama a ação Show do controlador ProjectsController:

```
link_to "Show", project_path(project)
=>
<a href="/projects/1">Show</a>
```

O link gerado atende ao padrão REST contendo um controlador e um id, e como o navegador envia o verbo HTTP GET por padrão, a aplicação retorna a ação Show. Para cada resource, o Rails gera automaticamente 7 métodos Path (tabela 1.2).

Table 1.2: Standard Path methods

Path Method	HTTP Verb	Path	Requested Action
projects_path	GET	/projects	index
project_path(1)	GET	/projects/1	show
new_project_path	GET	/projects/new	new
edit_project_path(1)	GET	/projects/1;edit	edit
projects_path	POST	/projects	create
project_path(1)	PUT	/projects/1	update
project_path(1)	DELETE	/projects/1	destroy

Para algumas ações (show, create), por padrão, são transmitidos os métodos HTTP corretos (GET, POST). Em outros casos (update, delete) será nescessário um tratamento especial como veremos a seguir.

5. Formulários para 'create' e 'update' com métodos Path

Um formulário é tradicionalmente criado especificando uma ação:

```
<% form_for :project, @project, :url => { :action => "create" } do |f| %>
...
<% end %>
```

Já em uma aplicação REST são utilizados os métodos Path.

projects path -> new form
project path(:id) -> edit form

5.1 Formulário New

Um formulário é transmitido, por padrão, com o método POST. Ao se definir o project_path sem um id é gerado um path '/project' que ao enviado com POST resulta na ação *create.*

```
form_for(:project, :url => projects_path) do |f| ...
=>
<form action="/projects" method="post">
```

5.2 Formulário Edit

Como já vimos os métodos PUT e DELETE não são suportados pelos browsers. Logo para se executar a ação update (PUT) colocamos uma chave *:method* no hash *:html* do *form_for*.

```
form_for(:project, :url => project_path(@project),
          :html => { :method => :put }) do |f| ...
```

```
=>
<form action="/projects/1" method="post">
<div style="margin:0;padding:0">
   <input name="_method" type="hidden" value="put" />
</div>
```

O Rails gera um hidden field com o nome '*_method*' indicando o método HTTP 'PUT', o que direciona para a ação update.

6. Destroy

```
link_to "Destroy", project_path(project), :method => :delete
```

Para se passar o método DELETE, utilizamos o parâmetro :method no link_to. Para driblarmos a incompatibilidade dos navegadores com os métodos PUT e DELETE, o Rails gera um código Javascript de forma a direcionar a aplicação para a ação Destroy.

```
link_to "Destroy", project_path(project), :method => :delete
=>
<a href="/projects/1"
   onclick="var f = document.createElement('form');
   f.style.display = 'none'; this.parentNode.appendChild(f);
   f.method = 'POST'; f.action = this.href;
   var m = document.createElement('input');
   m.setAttribute('type', 'hidden');
   m.setAttribute('name', '_method');
   m.setAttribute('value', 'delete'); f.appendChild(m);f.submit();
   return false;">Destroy</a>
```

7. Métodos URL nos Controladores

Nos controladores também precisamos tratar de forma diferente os redirects. Para isso o Rails nos dá os 'Métodos URL', gerando um método URL para cada método Path.

```
project_url for project_path
or
projects_url for projects_path.
```

Os métodos URL criam URLs completas, com protocolo, host, porta e path:

```
project_url(1)  =>  "http://localhost:3000/projects/1"
projects_url    =>  "http://localhost:3000/projects"
```

Nos controladores, cada *redirect_to* irá utilizar um método URL ao contrário do hash "controler/action/parâmetros".

Um exemplo de uma ação destroy onde o método URL sem parâmetros redireciona a apliação para a lista dos projetos após deletar um projeto específico:

```
def destroy
  @project = Project.find(params[:id])
  @project.destroy
  respond_to do |format|
    format.html { redirect_to projects_url }
    format.xml { head :ok }
  end
end
```

8. REST Routing

Vimos que o desenvolvimento no padrão REST se utiliza de vários métodos em links, formulários e controladores. Estes helpers e o tratamento adequado das requisições REST são graças a uma nova entrada no arquivo de rotas do Rails, config/routes.rb:

```
map.resources :projects
```

Esta entrada é automaticamente gerada quando utilizamos o "*scaffold_resource*":

```
ruby script/generate scaffold_resource project nome:string desc:text
```

Esta entrada gera todos os métodos que precisamos para trabalhar com REST

```
map.resources :projects
=>
Route         Generated Helper
------------------------------------------------------------
projects      projects_url, projects_path
project       project_url(id), project_path(id)
new_project   new_project_url, new_project_path
edit_project edit_project_url(id), edit_project_path(id)
```

9. Convenção

Para se programar em Rails utilizando os paradigmas REST é preciso seguir um padrão nos nomes dos métodos que tratam das ações CRUD. Isto fica claro no exemplo abaixo:

```
link_to "Show", project_path(project)
=>
<a href="/projects/1">Show</a>
```

Neste caso o controlador 'Project' precisa ter um método com o nome "Show". O mesmo vale para as ações index, update, create, destroy, new e edit.

10. Nested Resources (Resources aninhadas)

A aplicabilidade do paradigma REST e a importância das URLs limpas se tornam muito mais claras ao utilizarmos Resources Aninhadas. Resources aninhadas são resources intimamente ligadas, com relacionamento do tipo pai-filho. Em Rails podemos pensar em modelos (models) que representam a relação one-to-many (um para muitos), como em Projetos e Empregados, onde um Projeto possui vários empregados.

Considerando que já temos a resource 'Project' criada iremos agora criar a resource 'Empregado":

```
> ruby script/generate scaffold_resource empregado name:string \
       idade:integer telefone:string project_id:integer
> rake db:migrate
```

Como a relação é de 'um para muitos', modificamos os models:

```
/models/project.rb
  class Project < ActiveRecord::Base
     has_many :empregados
 end

/models/empregado.rb
  class Empregado < ActiveRecord::Base
     belongs_to :project
  end
```

Além dos controlers, models e views, o scaffold_resource inseriu no routes.rb a entrada:

```
map.resources :empregados
```

O que, assim como para os projetos, gerou as rotas e os helpers para manipular a resource Empregado. Porém existir um empregado, pelo menos no nosso contexto, só faz sentido quando este está relacionado a um projeto, assim o helper "new_empregado_path" cria o path "empregado/new" sem nenhuma informação sobre o projeto.

Resources aninhadas consistem exatamente em criar uma resource filha que não pode existir fora do contexto da resource pai. Para que isso aconteça adequadamente, temos que fazer as seguintes modificações:

```
  map.resources :empregados
=>
```

```
map.resources :projects do |projects|
   projects.resources :empregados
end
```

Esta entrada, trata "Empregado" como uma resource aninhada à resource "Project", gerando as rotas apropriadas com o seguinte formato:

```
/project/:project_id/empregados
/project/:project_id/empregados/:id
```

Para listar todas os empregados de um determinado projeto a URL seria :

```
http://localhost:3000/projects/1/empregados
```

10.1 Adaptando os Controladores

O controlador EmpregadosController não sabe que é responsável por uma resource aninhada, logo a ação index, por exemplo, retornará todas os empregados cadastrados, independentemente do projeto trazido pela URL.

```
def index
  @empregados = Empregado.find(:all)
  respond_to do |format|
    format.html # index.rhtml
    format.xml { render :xml => @empregados.to_xml }
  end
end
```

Para retornar apenas os empregados do projeto escolhido, o código será:

```
def index
  project = Project.find(params[:project_id])
  @empregados = project.empregados.find(:all)
  ...
end
```

Obs.: As ações 'create' e 'update' também precisam ser ajustadas.

10.2 Parâmetros para os helpers Path e URL

Com as resources aninhadas, são gerados também helpers diferenciados. Estes helpers possuem os mesmos nomes dos helpers de uma resource não aninhada, porém agora devem possuir como primeiro parâmetro uma id da resource 'pai', no nosso caso um id do projeto. Por exemplo, para se listar os empregados de um projeto utilizaremos o helper 'empregados_path' com a seguinte sintaxe:

```
link_to "Empregados", empregados_path(project)
=>
<a href="/projects/1/empregados">Empregados</a>
```

Um exemplo de view para o index do ProjectController:

```
...
<% for project in @projects %>
    <tr>
        <td><%=h project.name %></td>
        <td><%=h project.desc %></td>
        <td><%= link_to "Empregados", empregados_path(project) %></td>
        <td><%= link_to "Show", project_path(project) %></td>
        <td><%= link_to "Edit", edit_project_path(project) %></td>
        <td><%= link_to "Destroy", project_path(project),
                :confirm => "Are you sure?", :method => :delete %></td>
    </tr>
<% end %>
...
```

As views do EmpregadoController também devem ser ajustadas, já que as ações show, edit e destroy devem possuir o id do projeto como primeiro parâmetro.

```
/empregados/index.rhtml
...
<% for empregado in @empregados %>
    <tr>
        <td><%=h empregado.name %></td>
        <td><%=h empregado.idade %></td>
        <td><%=h empregado.telefone %></td>
        <td><%= link_to "Show", empregado_path(empregado.project,
empregado) %></td>
        <td><%= link_to "Edit", edit_empregado_path(empregado.project,
empregado) %></td>
        <td><%= link_to "Destroy", empregado_path(empregado.project,
empregado),
            :confirm => "Are you sure?", :method => :delete %></td>
    </tr>
<% end %>
....
```

10.3 New

Para se adicionar um novo empregado, simplismente adicionamos:

```
<%= link_to "Novo Empregado", new_empregado_path(project) %>
```

O Formulário para a criação de um novo empregado possui uma rota aninhada no parâmetro action:

```
 /views/empregados/new.rhtml

<% form_for(:empregado, :url => empregados_path(params[:project_id])) do
|f| %>
...
<% end %>
=>
<form action="/projects/1/empregados" method="post">
```

Porém o uso de "params[:project_id]" é opcional já que o Rails coloca automaticamente o project_id no parâmetro action do formulário. Logo o form_for pode ser:

```
form_for(:empregado, :url => empregados_path)
```

O método 'create' no controlador EmpregadosController ficaria:

```
1 def create
2     @empregado = Empregado.new(params[:empregado])
3     @empregado.project = Project.find(params[:project_id])
4
5     respond_to do |format|
6         if @empregado.save
7             flash[:notice] = "Empregado foi criado com sucesso."
8             format.html { redirect_to empregado_url(@empregado.project,

9                                 @empregado) }
10             format.xml { head :created, :location =>
11                                 empregado_url(@empregado.project,
@empregado) }
12         else
13             format.html { render :action => "new" }
14             format.xml { render :xml => @empregado.errors.to_xml }
15         end
16     end
17 end
```

Falta ajustarmos os links para Edit e Back da view "Show":

```
views/empregados/show.rhtml
...
<%= link_to "Edit", edit_empregado_path(@empregado.project, @empregado) %>
<%= link_to "Back", empregados_path(@empregado.project) %>
```

10.4 Editando

Para editarmos um Empregado, incialmente precisamos modificar o form_for da view "edit":

```
form_for(:empregado,
    :url => empregado_path(params[:project_id], @empregado),
    :html => { :method => :put }) do |f|
```

A ação "update" do Controlador "EmpregadosController", também precisa ser modificada de forma que o project_id seja passado no método "redirect_to empregado_url()" na linha 7;

```
1 def update
2     @empregado = Empregado.find(params[:id])
3
4     respond_to do |format|
5         if @empregado.update_attributes(params[:empregado])
6             flash[:notice] = "Empregado Atualizado."
7             format.html { redirect_to empregado_url(@empregado) }
8             format.xml { head :ok }
9         else
10            format.html { render :action => "edit" }
11            format.xml { render :xml => @empregado.errors.to_xml }
12        end
13     end
14 end
```

O resultado na linha 7 é:

```
format.html { redirect_to empregado_url(@empregado.project, @empregado) }
```

Referências

[0] http://en.wikipedia.org/wiki/Represe...State_Transfer

[1] Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures, http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm

[2] http://www.rubyrailways.com/great-ru...est-resources/

[3] http://www.b-simple.de/download/restful_rails_en.pdf

[4] http://www.softiesonrails.com/2008/1...eets-rails-2-0