CAPÍTULO 20

Apêndice: Deployment

"Há noites que eu não posso dormir de remorso por tudo o que eu deixei de cometer."
— Mario Quintana

Como construir ambientes de produção e deployment para aplicações Rails sempre foram alguns dos maiores desafios desta plataforma. Existem diversos detalhes a serem considerados e diversas opções disponíveis.

20.1 - WEBRICK

A forma mais simples de executar aplicações rails é usar o servidor que vem embutido em todas estas aplicações: **Webrick**.

É um servidor web muito simples, escrito em Ruby, que pode ser iniciado através do arquivo **script/server**, dentro do projeto:

```
cd projetorails
rails server
```

Por padrão, o Webrick inicia na porta 3000, porém isto pode ser mudado com a opção `-p`:

```
rails server -p 3002
```

Por ser muito simples, **não é recomendado** o uso do webrick em produção.

20.2 - CGI

Uma das primeiras alternativas de deployment para aplicações Rails foi o uso de servidores web famosos, como o Apache Httpd. Porém, como o httpd só serve conteúdo estático, precisar delegar as requisições dinâmicas para processos Ruby

que rodam o Rails, através do protocolo CGI.

Durante muito tempo, esta foi inclusive uma das formas mais comuns de servir conteúdo dinâmico na internet, com linguagens como Perl, PHP, C, entre outras.

O grande problema no uso do CGI, é que o servidor Web inicia um novo processo Ruby a cada requisição que chega. Processos são recursos caros para o sistema operacional e iniciar um novo processo a cada requisição acaba limitando bastante o tempo de resposta das requisições.

20.3 - FCGI - FastCGI

Para resolver o principal problema do CGI, surgiu o **FastCGI**. A grande diferença é que os processos que tratam requisições dinâmicas (*workers*) são iniciados junto ao processo principal do servidor Web.

Desta forma, não é mais necessário iniciar um novo processo a cada requisição, pois já foram iniciados. Os processos ficam disponíveis para todas as requisições, e cada nova requisição que chega usa um dos processos existentes.

Pool de processos

O conjunto de processos disponíveis para tratar requisições dinâmicas também é popularmente conhecido como **pool** do processos.

A implementação de FCGI para aplicações Rails, com o apache Httpd nunca foi

satisfatória. Diversos bugs traziam muita instabilidade para as aplicações que optavam esta alternativa.

Infelizmente, FCGI nunca chegou a ser uma opção viável para aplicações Rails.

20.4 - Lighttpd e Litespeed

Implementações parecidas com Fast CGI para outros servidores Web pareceram ser a solução para o problema de colocar aplicações Rails em produção. Duas alternativas ficaram famosas.

Uma delas é a implementação de Fast CGI e/ou SCGI do servidor web **Lighttpd**. É um servidor web escrito em C, bastante performático e muito leve. Muitos reportaram problemas de instabilidade ao usar o Lighttpd em aplicações com grandes cargas de requisições.

Litespeed é uma outra boa alternativa, usado por aplicações Rails em produção até hoje. Usa o protocolo proprietário conhecido como LSAPI. Por ser um produto pago, não foi amplamente difundido dentro da comunidade de desenvolvedores Rails.

http://www.litespeedtech.com/ruby-lsapi-module.html

20.5 - Mongrel

Paralelamente às alternativas que usam FCGI (e variações) através de servidores Web existentes, surgiu uma alternativa feita em Ruby para rodar aplicações Rails.

Mongrel é um servidor web escrito por Zed Shaw, em Ruby. É bastante performático e foi feito especificamente para servir aplicações Rails. Por esses motivos, ele rapidamente se tornou a principal alternativa para deployment destas aplicações. Hoje suporta outros tipos de aplicações web em Ruby.

de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

Faça sua pergunta.

20.6 - Proxies Reversos

O problema com o Mongrel é que uma instância do Rails não pode servir mais de uma requisição ao mesmo tempo. Em outras palavras, o Rails não é thread-safe. Possui um lock que não permite a execução de seu código apenas por uma thread de cada vez.

Por causa disso, para cada requisição simultânea que precisamos tratar, é necessário um novo processo Mongrel. O problema é que cada Mongrel roda em uma porta diferente. Não podemos fazer os usuários terem de se preocupar em qual porta deverá ser feita a requisição.

Por isto, é comum adicionar um **balanceador de carga** na frente de todos os Mongrels. É o balanceador que recebe as requisições, geralmente na porta 80, e despacha para as instâncias de Mongrel.

Como todas as requisições passam pelo balanceador, ele pode manipular o conteúdo delas, por exemplo adicionando informações de cache nos cabeçalhos HTTP. Neste caso, quando faz mais do que apenas distribuir as requisições, o balanceador passa a ser conhecido como **Proxy Reverso**.

Reverso?

Proxy é o nó de rede por onde passam todas as conexões que saem. O nome Proxy Reverso vem da ideia de que todas as conexões **que entram** passam por ele.

O principal ponto negativo no uso de vários Mongrels é o processo de deployment. A cada nova versão, precisaríamos instalar a aplicação em cada um dos Mongrels e reiniciar todos eles.

Para facilitar o controle (*start*, *stop*, *restart*) de vários Mongrels simultaneamente, existe o projeto **mongrel_cluster**.

20.7 - Phusion Passenger (mod_rails)

Ninh Bui, **Hongli Lai** e **Tinco Andringa** da empresa Phusion decidiram tentar novamente criar um módulo para rodar aplicações Rails usando o Apache Httpd.

Phusion **Passenger**, também conhecido como **mod_rails**, é um módulo para o Apache Httpd que adiciona suporte a aplicações Web escritas em Ruby. Uma de suas grandes vantagens é usar o protocolo **Rack** para enviar as requisições a processos Ruby.

Como o **Rack** foi criado especificamente para projetos Web em Ruby, praticamente todos os frameworks web Ruby suportam este protocolo, incluindo o Ruby on Rails, o Merb e o Sinatra. Só por serem baseados no protocolo Rack, são suportados pelo **Passenger**.

A outra grande vantagem do mod_rails é a facilidade de deployment. Uma vez que o módulo esteja instalado no Apache Httpd, bastam três linhas de configuração no arquivo **httpd.conf**:

```
<VirtualHost *:80>
    ServerName www.aplicacao.com.br
    DocumentRoot /webapps/aplicacoes/projetorails
</VirtualHost>
```

A partir daí, fazer deployment da aplicação Rails consiste apenas em copiar o código para a pasta configurada no Apache Httpd. O mod_rails detecta que é uma aplicação Rails automaticamente e cuida do resto.

A documentação do Passenger é uma ótima referência:

http://www.modrails.com/documentation/Users%20guide.html

20.8 - Ruby Enterprise Edition

Além do trabalho no mod_rails, os desenvolvedores da Phusion fizeram algumas modificações importantes no interpretador MRI. As mudanças podem trazer redução de até 30% no uso de memória em aplicações Rails.

O *patch* principalmente visa modificar um pouco o comportamento do Garbage Collector, fazendo com que ele não modifique o espaço de memória que guarda o código do Rails. Desta forma, os sistemas operacionais modernos conseguem usar o mesmo código Rails carregado na memória para todos os processos. Esta técnica

é conhecida como Copy on Write; suportada pela maioria dos sistemas operacionais modernos.

Outra mudança importante promovida pelos desenvolvedores da Phusion foi o uso de uma nova biblioteca para alocação de memória, **tcmalloc**, no lugar da original do sistema operacional. Esta biblioteca é uma criação do Google.

20.9 - Exercícios: Deploy com Apache e Passenger

1. Abra o FileBrowser e copie o projeto **restaurantes** para o Desktop. o projeto está em /rr910/Aptana Studio Workspace/restaurantes

2. Abra o terminal e digite:

```
install-httpd
```

Feche e abra o terminal novamente Esse comando baixa httpd e compila na pasta /home/apache. No nosso caso=> /home/rr910/apache

3. Ainda no terminal entre no diretório do projeto e rode a migration para atualizar o banco em produção:

```
cd Desktop/restaurante
rake db:migrate:reset RAILS_ENV=production
```

4. Abra o arquivo de configuração do Apache:

```
gedit /home/rr910/apache/conf/httpd.conf
```

Altere a linha abaixo:

```
Listen 8080
```

Adicione a linha a baixo em qualquer lugar do arquivo:

```
Servername http://localhost:8080
```

5. Suba o Apache, no terminal rode:

```
apachectl start
```

Acesse http://localhost:8080/ no browser e confira se o Apache subiu, deve aparecer a mensagem **It works!**.

6. vamos instalar o Passenger. no terminal rode:

```
gem install passenger
passenger-install-apache2-module
```

Quando o instalador surgir no terminal, pressione **enter**.

No fim, copie a instrução semelhante a essa:

```
LoadModule passenger_module
  /home/rr910/.gem/ruby/1.8/gems/passenger-2.2.5/ext/apache2/
mod_passenger.so
 PassengerRoot /home/rr910/.gem/ruby/1.8/gems/passenger-2.2.5

 PassengerRuby /usr/bin/ruby1.8
```

7. Abra o arquivo de configuração do Apache:

```
gedit /home/rr910/apache/conf/httpd.conf
```

Adicione essas linhas ao final do arquivo:

```
<VirtualHost *:8080>
     DocumentRoot /home/rr910/Desktop/restaurante/public

</VirtualHost>

<Directory />
   Options FollowSymLinks
   AllowOverride None
   Order allow,deny
   Allow from all
</Directory>
```

8. No terminal, de o restart no apache:

```
apachectl restart
```

Acesse **http://localhost:8080/restaurantes**. Nossa aplicação está rodando no

CAPÍTULO ANTERIOR:

Apêndice: Integrando Java e Ruby

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter